Trade-Off Exploration for Acceleration of Continuous Integration

by

Zhili Zeng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Abstract

Continuous Integration (CI) is a popular software development practice that allows developers to quickly verify modifications to their projects. To cope with the ever-increasing demand for faster software releases, CI acceleration approaches have been proposed to expedite the feedback that CI provides.

However, adoption of CI acceleration is not without cost. The trade-off in duration and trustworthiness of a CI acceleration approach determines the practicality of the CI acceleration process. Indeed, if a CI acceleration approach takes longer to prime than to run the accelerated build, the benefits of acceleration are unlikely to outweigh the costs. Moreover, CI acceleration techniques may mislabel change sets (e.g., a build labelled as failing that passes in an unaccelerated setting or vice versa) or produce results that are inconsistent with an unaccelerated build (e.g., the underlying reason for failure does not match with the unaccelerated build). These inconsistencies call into question the trustworthiness of CI acceleration products.

We first evaluate the time trade-off of two CI acceleration products — one based on program analysis (PA) and the other on machine learning (ML). After replaying the CI process of 100,000 builds spanning ten open-source projects, we find that the priming costs (i.e., the extra time spent preparing for acceleration) of the program analysis product are substantially less than that of the machine learning product (e.g., average project-wise median cost difference of 148.25 percentage points). Furthermore, the program analysis product generally provides more time savings than the machine learning product (e.g., average project-wise median savings improvement of 5.03 percentage points). Given their deterministic nature, and our observations about priming costs and benefits, we recommend that organizations consider the adoption of program analysis based acceleration.

Next, we study the trustworthiness of the same PA and ML CI acceleration products. We re-execute 50 failing builds from ten open-source projects in non-accelerated (baseline), program analysis accelerated, and machine learning accelerated settings. We find that when applied to known failing builds, program analysis accelerated builds more often (43.83 percentage point difference across ten projects) align with the non-accelerated build results. Accordingly, we conclude that while there is still room for improvement for both CI acceleration products, the selected program analysis product currently provides a more trustworthy signal of build outcomes than the machine learning product.

Finally, we propose a mutation testing approach to systematically evaluate the trustworthiness of CI acceleration. We apply our approach to the deterministic PA-based CI acceleration product and uncover issues that hinder its trustworthiness. Our analysis consists of three parts: we first study how often the same build in accelerated and unaccelerated

CI settings produce different mutation testing outcomes. We call mutants with different outcomes in the two settings "gap mutants". Next, we study the code locations where gap mutants appear. Finally, we inspect gap mutants to understand why acceleration causes them to survive. Our analysis of ten thriving open-source projects uncovers 2,237 gap mutants. We find that: (1) the gap in mutation outcomes between accelerated and unaccelerated settings varies from 0.11%–23.50%; (2) 88.95% of gap mutants can be mapped to specific source code functions and classes using the dependency representation of the studied CI acceleration product; (3) 69% of gap mutants survive CI acceleration due to deterministic reasons that can be classified into six fault patterns. Our results show that deterministic CI acceleration suffers from trustworthiness limitations, and highlights the ways in which trustworthiness could be improved in a pragmatic manner.

This thesis demonstrates that CI acceleration techniques, whether PA or ML-based, present time trade-offs and can reduce software build trustworthiness. Our findings lead us to encourage users of CI acceleration to carefully weigh both the time costs and trustworthiness of their chosen acceleration technique. This study also demonstrates that the following improvements for PA-based CI acceleration approaches would improve their trustworthiness: (1) depending on the size and complexity of the codebase, it may be necessary to manually refine the dependency graph, especially by concentrating on class properties, global variables, and constructor components; and (2) solutions should be added to detect and bypass flaky test during CI acceleration to minimize the impact of flakiness.

Acknowledgements

With the utmost respect, I would like to thank my supervisor, Professor Shane McIntosh, for accepting me as a master student in his research group, creating the overseas exchange opportunity, referring me to academia and industry, as well as the professional guidance and precious motivation throughout my study. He has been an exemplary researcher and adept at handling challenges, which has greatly benefited my career in immeasurable ways.

I would also express my sincere gratitude to Professor Maxime Lamothe for the meticulous feedback in shaping the papers and thesis. I really appreciate his insightful advice and invaluable encouragement during my research.

I am honored to have the chance to collaborate with brilliant researchers, Professor Hideki Hata and Master Tao Xiao, who contributed to my research and made this thesis possible. I am also grateful to Professor Raula Gaikovina Kula and Professor Matsumoto Kenichi for their hospitality at NAIST, Japan.

Special thanks to Professor Michael W. Godfrey and Professor Meiyappan Nagappan for being my thesis readers and providing constructive feedback.

Last but not least, I would like to thank my colleagues in REBELs and SWAG research groups for the warmth and happiness I received in the two years. I am proud of being a part of the UWaterloo community, the knowledge and experiences I have gained in this one of the world's top CS universities are treasures throughout my life.

This thesis is dedicated to my family, my friends and the one I love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Continuous Integration (CI) is a popular practice in modern software development [55, 74]. Accordingly, substantial effort has been invested in improving the performance of each phase of the CI process [84]. Among these phases, the test execution phase normally consumes the majority of CI execution [37, 48]. As a result, accelerating the testing process contributes to rapid feedback—a hallmark benefit of CI.

Adopting of CI acceleration is not without pitfalls. The trade-off in time and trustworthiness of a CI acceleration approach determines the practicality of the CI acceleration process. Indeed, if a CI acceleration approach takes longer to prime than to run the build, it may not be worth it [44].

However, even if the approach provides large time savings, if it mislabels change sets (e.g., a faulty build passes) [81, 86], then it may ultimately result in more work for developers [30, 75, 87]. Builds can have failing outcomes for multiple reasons, and these may not be consistent when using CI acceleration. This can allow some bugs to slip through CI when acceleration is in use. Indeed, when build behaviour is deemed *untrustworthy*, it is not uncommon for developers to rely on sub-optimal workarounds, such as repeated execution [56]. Since developers tend to prioritize correct build behaviour over efficiency [1], if CI acceleration is untrustworthy, they may hesitate to adopt it. Despite its importance, systematic approaches to evaluate the trustworthiness of CI acceleration approaches do not yet exist. Early work by Gallaba et al. [28] suggest that at a high level, accelerated outcomes of change sets tend to match (i.e., change sets with a passing/failing outcome continue to pass/fail when acceleration is applied); however, the results are based on a replay of 100 historical change sets, which may not capture a full variety or breadth of potential software modifications. To bridge this gap, we propose a mutation testing approach to study the trustworthiness of CI acceleration.

The costs and benefits of CI acceleration approaches should therefore be carefully measured. In this thesis, we use replay analysis, which can be used to evaluate the cost of time and trustworthiness. We believe that the outcomes of our study can assist CI users to better weigh the benefit and cost when selecting CI acceleration products and guide future work on improvements to the CI acceleration approaches.

1.1 Problem Statement

The cadence of software development is set by the pace at which Continuous Integration services process change sets. Rapid feedback in a CI process is a key feature. Accelerating CI processes provides several benefits, but is accompanied by limitations:

Thesis Statement: CI acceleration is not free. Replay analysis can be leveraged to evaluate the costs and benefits in terms of time and trustworthiness. Furthermore, mutation testing can complement replay analysis to provide a fine-grained assessment of the limitations that hinder CI acceleration.

The costs and benefits of a CI acceleration product determine the quality of the CI acceleration process. Indeed, it may be impractical if a CI acceleration product takes longer to prime than to run the accelerated build. Therefore, the costs and benefits of CI acceleration products should be carefully measured.

To understand and compare the costs and benefits of two families of CI acceleration products, we conduct an empirical comparison of a program analysis and a machine learning acceleration product. Furthermore, aimed at the decision-making guidelines (e.g., the dependency graph of changed code) of the program analysis product, we apply mutation testing to assess the defect patterns that exist in CI acceleration.

1.2 Thesis Overview

Below, we provide a brief overview of the scope of this thesis.

- **Chapter 2:** *Background*
  We begin by providing the necessary research background for our selected topic. We introduce the fundamental steps of continuous integration (CI) and highlight the

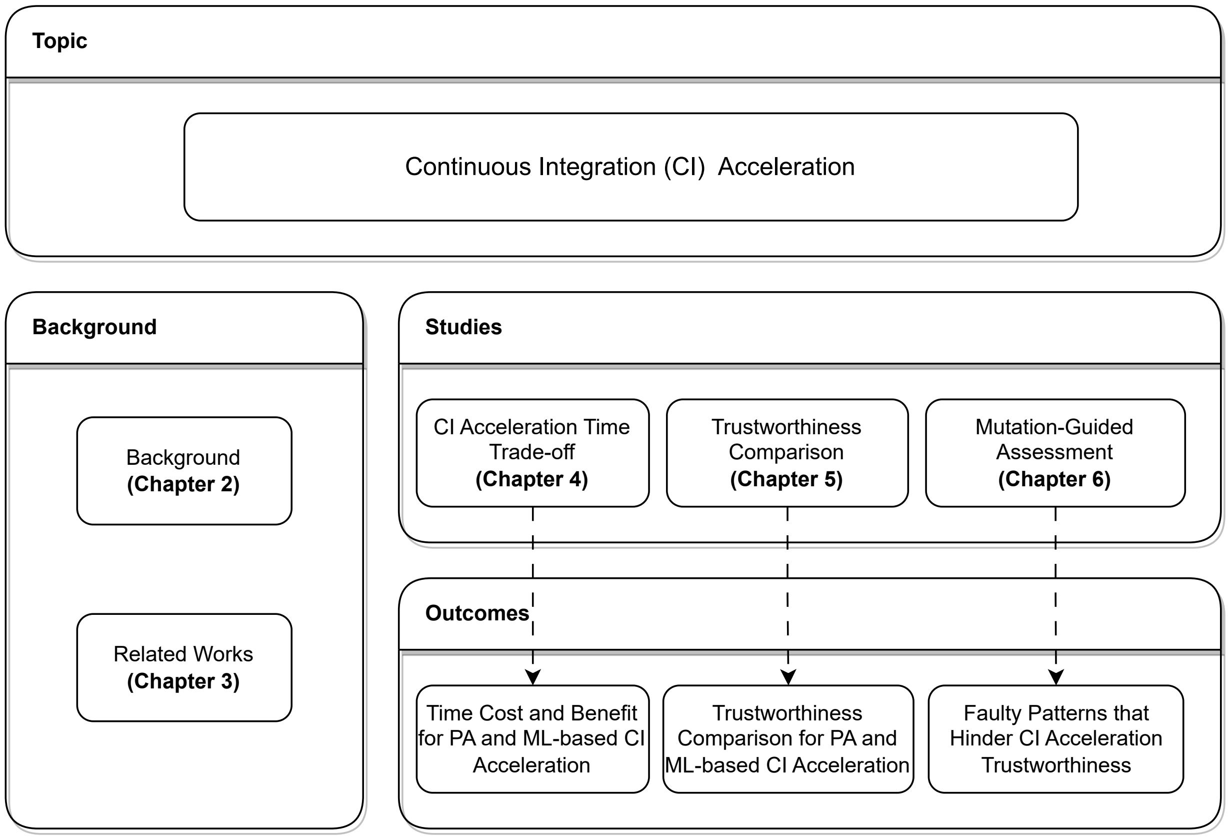
Topic
Continuous Integration (CI) Acceleration
Background
Background (Chapter 2)
Related Works (Chapter 3)
Studies
CI Acceleration Time Trade-off (Chapter 4)
Trustworthiness Comparison (Chapter 5)
Mutation-Guided Assessment (Chapter 6)
Outcomes
Time Cost and Benefit for PA and ML-based CI Acceleration
Trustworthiness Comparison for PA and ML-based CI Acceleration
Faulty Patterns that Hinder CI Acceleration Trustworthiness

Figure 1.1: An overview of the scope of this thesis.

factors that trigger the CI procedure. Subsequently, we outline and discuss various mainstream methods that can be employed to accelerate the CI process. Additionally, we describe mutation testing, the technique we apply to assess the CI acceleration.

- **Chapter 3:** *Related Work*
  In this chapter, we aim to situate this thesis with respect to the prior work in the field of continuous integration acceleration.

- **Chapter 4:** *CI Acceleration Time Trade-off*
  We study the performance of two CI acceleration products — one based on program analysis (PA) and the other on machine learning (ML). After replaying the CI process of 100,000 builds spanning ten open-source projects, we summarize the trade-off in terms of the priming costs (i.e., the extra time spent preparing for acceleration) and

time savings (i.e, the saved time save in accelerated build) of two CI acceleration products.

- **Chapter 5:** *Trustworthiness Comparison*
  We study the trustworthiness of the same program analysis and machine learning CI acceleration products. We re-execute 50 failing builds from ten open-source projects in non-accelerated (baseline), PA-accelerated, and ML-accelerated settings. We study how often the CI acceleration products align with the non-accelerated build results.

- **Chapter 6:** *Mutation-Guided Assessment*
  We propose a mutation testing approach to systematically evaluate the trustworthiness of CI acceleration. We apply our approach on a deterministic solution that uses program analysis to accelerate CI and uncover issues that hinder its trustworthiness.

1.3 Thesis Contributions

This thesis demonstrate that:

- The priming costs of the program analysis approach are substantially lower than those of the ML approach (Chapter 4).
- During the acceleration process, the program analysis approach generally provides greater time savings than the ML approach (Chapter 4).
- The results of builds accelerated by the program analysis approach more often align with non-accelerated build results (Chapter 5).
- 90% of studied projects contain mutants that survive in accelerated settings although they are killed in unaccelerated settings (Chapter 6).
- 88.95% of the mutants that only survive in the accelerated setting are traceable within the dependency graph of the studied CI acceleration product (Chapter 6).
- While 22.5% of mutants that survive in the accelerated setting but are killed in the unaccelerated setting differ due to non-deterministic build behaviour, 69% survive because of deterministic reasons, and can be broadly classified into six categories (Chapter 6).

Overall, this thesis demonstrates that CI acceleration can sacrifice trustworthiness. While there is still room for improvement for both CI acceleration approaches, the selected program analysis approach currently provides greater practical value in an explainable fashion, while also incurring less risk. We therefore encourage users of CI acceleration to carefully weigh the trustworthiness of their chosen acceleration technique before accelerating their builds to avoid compromising their build results. On the other hand, it is possible to use techniques such as mutation testing to evaluate this sacrifice and identify improvements for current approaches.

1.4 Thesis Organization

We organize the remainder of the thesis as follows. Chapter 2 outlines the background for our research. Chapter 3 introduces the related research in Continuous Integration, CI acceleration and mutation testing. Chapter 4 presents the trade-off in time of program analysis product and machine learning product. Chapter 5 discloses the gap of trustworthiness of both products. Chapter 6 describes the mutation-guided assessment of the trustworthiness of CI acceleration. Chapter 7 concludes the thesis and discusses promising directions for future work.

Chapter 2

Background

In this chapter, we introduce the concepts that underpin the three key components of our study. First, we describe fundamental steps of continuous integration (CI) and highlight the factors that trigger the CI procedure. Then, we explain acceleration of the CI process. Finally, we provide the rationale of mutation testing through our discussion of mutant categories and mutation operators.

2.1 Continuous Integration

2.1.1 Continuous Integration Procedure

The CI process begins when a developer (or automated tool) introduces a change in their version control system [49]. A CI build can then be triggered to automatically invoke a series of steps to assess whether the change set integrates safely [22]. When a build is triggered, a build job is created using project-specific criteria (e.g., compile the code then run all tests). To process a build job, a node downloads a specified (typically, the latest) version of the source code and initiates the build process, including acquiring dependencies [51], compiling the code, and running tests [77]. Then, the node sends the build results to a reporting service to broadcast them to the development team. This enables a development feedback loop that empowers its users.

2.1.2 Software Builds

Software builds refer to the process of assembling software artifacts to create or update installable or updated software packages, which consists of five steps [52]:

- **Configuration** sets the necessary parameters and user-preferred environment to prepare for the building the software.
- **Construction** invokes language tools, such as compilers and linkers, to process source code to generate raw deliverables (i.e., executable files).
- **Certification** executes tests to inspect the quality of raw deliverables generated from the construction step.
- **Packaging** assembles the validated deliverables with the necessary libraries, documentation, and data files.
- **Deployment** releases the software package to the targeted platform and makes it available for download or update.

2.2 CI Acceleration

2.2.1 CI Acceleration Techniques

There are various CI acceleration approaches. Some approaches rely on using historical code changes [50] to train classifiers [45, 54], which are then used to select tests for execution. Indeed, the machine learning approach we explored in this thesis belongs to this category. Other popular methods for CI acceleration rely on rule-based test case prioritization [13, 73] and test case selection [21]. These techniques accelerate builds by running a subset of the complete test suites or skipping steps of within test cases. The program analysis approach that we study in this thesis belongs to this category. In addition, there are techniques that rely on greedy algorithms [79], which decompose the build targets to make the builds faster. Finally, there are techniques that cache environments and infer dependencies [7, 12, 28], these techniques speed up builds by exploiting the dependency relations embedded in the dependency graphs of builds.

2.2.2 Dependency Graph

In a system, software artifacts may depend on each other. The dependency graph in a build system records dependencies from targets to other targets or files.

As the decision-making kernel of the studied program analysis CI acceleration product (PA-product), the dependency graph stores information that is used to accelerate subsequent test runs. It provides guidelines for the build with a minimal number of test runs required. Specifically, throughout the build process, the PA-product traces the execution path of each test, gains insights into the involved files and methods, and stores information about them as dependencies of that test in the dependency graph. By utilizing this graph, PA-product skips tests whose dependencies were not updated by corresponding code changes, delivering the same test results as running the entire test suite.

2.2.3 Target and Confidence Measures

Our studied Machine learning CI acceleration product (ML-product) trains a machine learning classifier to predict potential failures in each test. The source of the training set is the historical test results (i.e., names and paths of test cases and test files, test status of each test case, and the duration of each test case) and code changes (i.e., names and paths of changed files, number of changed lines, commit hashes associated with changes, and authors associated with changes).

The ML-product contains two types of optimization parameters: target and confidence. The Target measure reflects how small of a subset of tasks the acceleration aims to produce. The Confidence measure reflects how high the model-estimated likelihood needs to be before it decides to skip a task. By aiming at different optimization targets, the ML-product can provide recommended subsets of tests of varying levels of aggressiveness to accelerate builds.

2.3 Mutation Testing

Mutation testing is a program analysis approach that is generally used to evaluate the quality of test suites. It consists of artificially perturbing source code to inject (likely) faulty behaviour. The defects introduced by mutation testing are called mutants. Mutants (i.e., perturbed versions of the code under test) are produced, and test suites are re-executed to determine if the mutants are *killed* (i.e., at least one test that passes for the unperturbed version fails) or *survive* (i.e., tests continue to pass on the perturbed version). Based on this principle, we further introduce 5 types of mutants that occur in our study context.

- **Killed mutant:** mutants that have been killed by at least one test case;
- **Survived mutant:** mutants that have not been killed by any test case;
- **Timeout mutant:** mutants where test execution exceeds an upper limit setting;
- **Suspicious mutant:** mutants where the test suite took 10 times as long as the baseline;
- **Skipped mutant:** mutants that are skipped due to syntax errors in the mutated code or excessive use of system resources (e.g., memory exhaustion).

2.3.1 Mutation Operators

Mutation operators are concrete rules applied to the source code of a program in mutation testing. We provide ten types of mutation operators and their details:

- **Operator Mutation:** change the operators used in expressions (e.g., replacing '+' with '-', '*' with '/', or '==' with '!=');
- **Keyword Mutation:** modify keywords within the code (e.g., replacing 'if' with 'while' or 'return' with 'break');
- **Number Mutation:** change a number from positive to negative or modify its magnitude;
- **Name Mutation:** modify variable and function names in the code;
- **String Mutation:** change string literals in the code (e.g., modifying the content of a string or replacing it with a different string);
- **Argument Mutation:** modify function arguments (e.g., changing the order of arguments or replacing arguments with different values);
- **And/or Mutation:** replace 'and' with 'or' or vice versa in conditional statements;
- **Lambda Mutation:** modify lambda expressions in the code (e.g., changing the parameters or the body of a lambda function);

- **Expression Mutation:** change the order of operands or replace sub-expressions with ‘None’;
- **Decorator Mutation:** remove or change the decorators applied to functions or classes.

2.4 Chapter Summary

This chapter introduces the basic concepts of Continuous Integration (CI) and its acceleration as well as mutation testing. Specifically, we describe the CI procedure and the definition of software builds. We also present various of CI acceleration techniques, outline the key components of studied CI acceleration products, dependency graph and Target and Confidence Measurements. Finally, we provide the rationale and concrete rules of mutation testing, which we use to assess the trustworthiness of a CI acceleration product.

In the next chapter, we correspondingly survey prior research on Continuous Integration, CI acceleration and mutation testing to situate the empirical comparison and mutation-guided assessment with respect to the broader body of knowledge.

Chapter 3

Related Works

This chapter concentrates on existing work related to continuous integration (CI), CI acceleration techniques and their pitfalls, and mutation testing.

3.1 Continuous Integration

3.1.1 CI Procedure

Continuous Integration is best understood in terms of its process, which is composed of four steps: build triggering events, build job creation, build job processing, and build job reporting [24, 30]. Indeed, there are various prior studies that focus on each of these steps. For example, Macho et al. [49] introduce the starting point where a normal CI process begins. Dösinger et al. [22] explain that builds require a triggering event to automatically run the CI pipelines and subsequent procedures. Macho et al. [51] also study the acquisition of dependencies during the build process and propose an approach to repair dependency-related build breakages. Testing is yet another key step in the CI procedure [77]. Gallaba et al. [28] demonstrate that test execution takes up the majority of the CI process duration. As a result, accelerating test execution is considered a best practice, to keep CI processes short, so that feedback is provided quickly [76]. In this thesis, we concentrate on the CI process and its acceleration.

3.1.2 CI Applications

The presence of CI increases the efficiency and quality of software builds [2]. This has made it popular in both proprietary and open-source settings [38, 80]. Since the emergence of CI services, cloud-based CI providers (e.g., TravisCI,[1] CircleCI[2] and GitHub Actions[3]) have bridged the gap between professional CI services and individual users [10, 23, 57, 72]. The widespread adoption of CI has enabled improvements to software quality in various settings [68, 78]. Due to their widespread adoption and scale, increases in the performance of CI systems are highly sought after [59]. Gallaba et al. [29] conduct an exploratory case study of CircleCI and propose suggestions to improve the efficiency and robustness of CI applications. Esfahani et al. [26] introduce the mechanism to accelerate the CI procedure that is applied in Microsoft's internal projects. Rossi et al. [72] describe the application of continuous deployment on mobile development in Facebook. While prior works mainly focus on the benefit of CI, not much attention has been paid to the costs and risks that it incurs. In this thesis, we concentrate on both costs and benefits of CI applications.

3.2 CI Acceleration

3.2.1 CI Acceleration Techniques

While CI is an improvement over a scheduled build process, if the build process itself is slow, feedback delay can hinder development progress [32]. To reduce this delay, CI acceleration approaches have been proposed. Some approaches rely on using historical code changes [50] to train classifiers [45, 54], which are then used to select tests for execution. Other popular methods for CI acceleration rely on rule-based test case prioritization [13, 73] and test case selection [21]. These techniques accelerate builds by running a subset of the complete test suites or skipping steps of within test cases. In addition, there are techniques that rely on greedy algorithms [79], which decompose the build targets to make the builds faster. Finally, there are techniques that cache environments and infer dependencies [7, 12, 28], which speed up builds by exploiting dependency relations for each build. To achieve representative observations, we respectively select two kinds of commercial-grade CI acceleration products based on the machine learning (ML) and program analysis (PA)

[1] https://www.travis-ci.com
[2] https://circleci.com
[3] https://github.com/features/actions

domains for time trade-off and trustworthiness comparison, and conduct mutation-guided assessment to the PA product.

3.2.2 CI Acceleration Costs

Although CI acceleration provides many benefits, it suffers from challenges. First, most CI acceleration requires some preparation steps, or priming, before being ready to accelerate a build [71]. Moreover, the efficiency of CI acceleration techniques is not always obvious. Indeed, running a build acceleration approach on a system with few test cases may lack meaning [46]. Even in cases where the benefits of CI acceleration are clear, applying an acceleration approach can be a challenge in and of itself [83]. For example, some CI acceleration techniques only support a single programming language [70]. This language barrier can prevent developers from applying the acceleration approach to their projects, despite their desire for rapid feedback [8].

In cases where the development language aligns with a CI acceleration approach, the time spent on preparing the acceleration environment might have a high cost [32, 44]. However, while these costs may add up, the benefits of CI acceleration are expected to outweigh it [33, 35].

Moreover, CI acceleration approaches are not without their pitfalls. Indeed, Gallaba and McIntosh [30] showed that the misuse of CI services can cause additional failures. The occurrence of test failures in software systems is unstable, and varies based on the evaluation of the system [53, 89]. As a result, even if an acceleration approach achieves substantial savings, if it introduces instability with respect to the unaccelerated build, teams will likely hesitate to adopt it.

A systematic mapping study [48] found that prior research shows a preference for test selection and prioritization approaches that make use of failure history, execution history, and test coverage. Approaches that use failure [85, 88] and execution history [47, 58] are at the mercy of prior failures. Furthermore, approaches based on test coverage [14] attempt to maximize the number of faults detected in a given time period. Therefore, by design, such approaches might miss faults due to the side effects of test case prioritization.

A systematic literature review of test case selection and prioritization [60] based on machine learning found that these approaches concentrate only on a few, easily obtainable features. Indeed, many approaches concentrate on code complexity metrics [60]. These approaches would also likely suffer from low trustworthiness when issues occur in specific components in the codebase (e.g., class properties, global variables, and constructors) since those areas are unlikely to have low complexity yet might cause test errors. Approaches based on textual data such as the approach by Aman et al. [4] prioritize test cases by targeting those that have different textual contents and, therefore, are more likely to test different aspects of the source code. Similarly, to traditional approaches based on code coverage, such approaches might also present low trustworthiness for issues that occur in less efficient tests. Finally, machine learning test selection and prioritization approaches that concentrate on code coverage are likely to suffer from similar issues as the ones presented in this thesis. Indeed, even state-of-the-art approaches [11] make use of class-level dependency analysis, which does not prevent issues such as those covered by configuration-purpose issues in the codebase.

3.3 Mutation Testing

Mutation testing is a program analysis approach [17] that consists of artificially perturbing source code to inject (likely) faulty behaviour. Mutation testing is primarily used to evaluate the quality of test suites [62, 64, 65].

In practice, mutation testing can be found in both experimental and industrial settings. Indeed, mutation testing has been used at the experimental level to evaluate a model-based approach for security protocols research [15, 16], to guide the input for combinatorial interaction testing [61], and in order to address test flakiness [36]. Meanwhile, at the enterprise level, mutation testing has been applied to optimize Google's deployment workflow [64, 66], and has been integrated into a Facebook commit-oriented scenario within their production environment [9]. In this thesis, we use mutation testing to verify the outcomes of a CI acceleration technique.

As a white box technique, a significant feature of mutation testing is its portability [67], which makes it possible to apply in various programming languages and multiple software testing levels (e.g. unit, integration) [41]. Since Python became one of the mainstream programming languages in the software development community, the application of mutation testing aimed at Python projects has become widespread in recent years [19, 20]. In this thesis, we conduct mutation tests on Python-specific CI acceleration. Specifically, we select Mutmut,[4] a mature mutation tool for Python with 12 mutation strategies [27], to generate mutants within our subject systems for the purpose of studying the impact of CI acceleration.

The key metric of mutation testing is the percentage of killed mutants (a.k.a. mutation score) [39, 63], which is calculated by dividing the number of killed mutants (i.e., mutants

[4]https://mutmut.readthedocs.io/en/latest/

that cause test cases to fail) by the total number of generated mutants. Indeed, mutation analysis has been used to improve various aspects of software engineering [5, 18, 41, 42]. However, to the best of our knowledge, mutation testing has yet to be applied to assess CI acceleration approaches.

3.4 Chapter Summary

In this chapter, we present the prior work on Continuous Integration and its acceleration, as well as mutation testing, from which we find that (1) although CI acceleration is widely used and offers time-saving benefits, it suffers from challenges, such as its priming costs and trustworthiness; and (2) while prior evaluations of CI acceleration have been conducted, they have yet to capture the full variety or breadth of potential software modifications. Moreover, (3) mutation testing is suitable for program analysis and has yet to be used for CI acceleration assessment.

As a result, the remainder of this thesis describes our empirical studies that set out to tackle these limitations. We begin in the next chapter, by comparing the time trade-off of two families of CI acceleration products based on different principles.

Chapter 4

CI Acceleration Time Trade-off

An earlier version of the work in this chapter has been submitted to IEEE Software.

4.1 Introduction

To understand and compare the costs and benefits of two families of CI acceleration products, in this chapter, we conduct an empirical comparison of a PA and an ML acceleration product. We specifically concentrate on readily available commercial tools to determine the costs and benefits of products that are currently available to practitioners. We systematically evaluate the priming costs and time savings of these products with respect to non-accelerated builds. To achieve our goal, we first select ten projects on which to run the CI acceleration products. Using the full range of parameter settings for the PA and ML products, we replay the builds of 50 commits from ten large and active open-source projects in PA-accelerated, ML-accelerated, and non-accelerated settings. This results in a total of 100,000 builds. This corpus of data allows us to answer the following research questions:

RQ1:How do the priming costs of CI acceleration products compare?

Outcome: Unsurprisingly, the priming cost is highly correlated with the original build time for both acceleration products; however, the priming costs of the PA-product is a project-wise average of 148.25 median percentage points less than those of the ML-product.

RQ2: What are the time savings of CI acceleration products?

Outcome: The PA-product shows an improvement of 5.03 percentage points compared with the ML-product in terms of time savings. However, the machine learning product tends to close the gap as projects accrue more data for training.

4.2 Study Design

Below, we present how we filter (Section 4.2.1) and rank (Section 4.2.2) candidates to obtain our set of studied projects. We then explain how we extract CI data from each studied project (Section 4.2.3). Finally, we present the approaches that we use to answer our research questions (Section 4.2.4).

4.2.1 Data Filtering

We begin by retrieving a dataset of GitHub repositories from Google BigQuery.[1] The dataset contains the activity and property information for the repositories on GitHub.

Select python projects (DF1): To mitigate the influence of different programming languages on our experimental results, we select projects that use single programming language. We focus on Python because it is supported by our selected acceleration products. We query projects that have ‘Python’ as a field within the ‘language’ table in the BigQuery dataset. After applying our first filter, 549,098 projects survive.

Select pytest projects (DF2): After removing all non-Python projects, we use a second filter to select projects that use the pytest framework. We select the pytest framework because previous research demonstrated that it has a more stable performance profile than other testing frameworks [6]. After applying our second filter, 29,849 projects survive.

[1]https://cloud.google.com/bigquery/public-data

4.2.2 Project Ranking

Although our experimental procedures are largely automated, repeating the experiment on thousands of systems is untenable. We therefore apply a ranking procedure to systematically select a set of studied projects.

Compute projects relevance measures (PR1): We inspect the number of commits, files and test cases, which respectively correspond to the repositories' activity, scale of production code, and scale of testing code. Repositories with a high number of commits and files contain more code changes, and are more likely to offer an adequate volume of data for validation. The number of test files is likely associated with the speed at which a build completes. We expect the time consumed by the build of a studied project to be large enough to allow us to perform a cost-benefit analysis.

Transform measures to ranking (PR2): To obtain the impact of metrics during project selection, we sum the above indicators, and rank the surviving projects in descending order.

Select diversified projects (PR3): To gain a diverse set of projects from which reliable conclusions can be drawn, we exclude top projects from the same domain, selecting the next highest ranked project from another domain as a replacement. Finally we obtain 10 studied projects from different domains, which are shown in Table 4.1.

Table 4.1: The studied projects of CI Acceleration Time Trade-off

Repository	# Commits	# Files	# Test Cases	Description	Domain
psf/requests	6,107	126	554	HTTP library	Network
apache/airflow	15,629	14,062	9,890	Workflow management platform	Workflow management
ansible/ansible	52,514	5,693	3,626	IT automation platform	IT automation
asciinema/asciinema	821	91	26	Terminal session recorder	Productivity
numpy/numpy	30,262	1,963	23,644	Scientific computing Python package	Statistics
bokeh/bokeh	19,609	4,409	9,784	Interactive data visualization tool	Visualization
dask/dask	7,280	447	593	Binary analysis platform	Task scheduling
scikit-learn/scikit-learn	28,198	5,338	9,753	Machine learning library	Machine learning
cloud-custodian/cloud-custodian	3,786	705	8,270	Cloud security and governance tool	Cloud management
explosion/spaCy	15,546	1,216	4,400	Natural Language Processing tool	NLP

4.2.3 Data Extraction

After selecting the studied projects, we extract data from each using our Data Extraction (DE) procedure. Figure 4.1 provides an overview of the procedure. We describe each step below.

Extract changesets (DE1): CI acceleration operate on changesets. Thus, we extract changesets from the Git repositories of our studied projects. We separately select 50 consecutive passing-build commits and 50 failed-build commits for each studied project by walking the commit history in reverse chronological order. The exact information for each commit that we study is listed in our online appendix.[2]

Label commits with CI outcomes (DE2): We manually label each of the selected commits according to CI outcomes (e.g., pass or fail). We conduct this manual labelling because of the inconsistency in CI checks for different commits. The CI checks[3] in this chapter refer to the GitHub's check run for the public repository, which contains the passed/failed outcomes from various CI services like GitHub Actions, CircleCI and Codecov. We obtain relevant information by checking the passed or failed label listed after the commit date. Each commit has a different number of checks, which varies with commit branches and build environments. GitHub's automatic system labels commits as failed if at least one check fails. For example, we found an inconsistency in commit **f3719bf** in the project **cloud-custodian/cloud-custodian**. In that instance, GitHub's automatic system labels the commit as failed; however, the build passes when we reproduce it locally.

4.2.4 Data Analysis

For both acceleration products, we first establish a build environment for the preparation of acceleration (i.e., the priming stage), then execute the accelerated build (i.e., the acceleration stage). Figure 4.1 provides an overview of our data analysis procedure. Below, we describe each step.

Prime acceleration technology (DA1a): The priming stage for the PA-product requires constructing a dependency graph to store the dependency relations between every element in the program. The dependency graph's construction is completed during the execution of the *cold build*, i.e., the initial, complete build in which no steps are skipped. We use the duration of the cold build to estimate the priming time.

The priming stage for the ML-product requires training the machine learning classifier to predict potential failures in each test. The source of the training dataset comes from past builds and their source code changes. The priming time of the ML-product contains the time spent on creating a training set, running the full test suite, recording test results, and subsetting the test suite based on the training results and parameter settings. The

[2] https://doi.org/10.5281/zenodo.7042150

[3] https://docs.github.com/en/developers/apps/guides/creating-ci-tests-with-the-checks-api

source of the training set is the historical test results (i.e., names and locations of test cases, test status and duration of each test case) and code changes (i.e., names and locations of changed files, number of changed lines, commit hashes and authors of changes). In this chapter, the complete training set we applied comes from the 50 passing-build commits we select for acceleration. We remove failing builds that do not run any tests and train on consecutive commits to match the real-world use case of the products. Specifically, we apply build acceleration on N commits, where we use the previous commits (i.e., from the 1st to $(N-1)$th commit) for training. The acceleration of the first commit is guided by the untrained machine learning classifier. The classifier uses the previous commits for training starting from the second commit. We use the median of the data from the full range of target and parameter settings to represent the priming time in each commit.

We measure the priming time by placing timers at each step of the priming process. We compare the cost of setting up a build acceleration environment across the studied approaches.

Execute accelerated builds (DA1b): During a warm (a.k.a., accelerated) build, the PA-product uses the graph generated during the cold build to locate changed lines of code and their dependencies. By traversing this graph, the approach can skip cases that are irrelevant to a code change.

During a warm build, the ML-product creates a subset of the test suites using different optimization targets. We select the full range of possible confidence and percentage time targets in increments of 0.01 to study their influence on the acceleration performance.

We measure the time savings by calculating the difference between their cold and warm build times. We then compare the time savings of both approaches.

4.3 Study Results

In this section, we present the results of our study with respect to our research questions.

4.3.1 RQ1: How do the priming costs of CI acceleration products compare?

RQ1: Approach. To study the costs of the priming stages, we analyze a series of time measurements. For the PA-product, we record the time spent during a cold build and the subsequent construction of the dependency graph. For the ML-product, we record the time

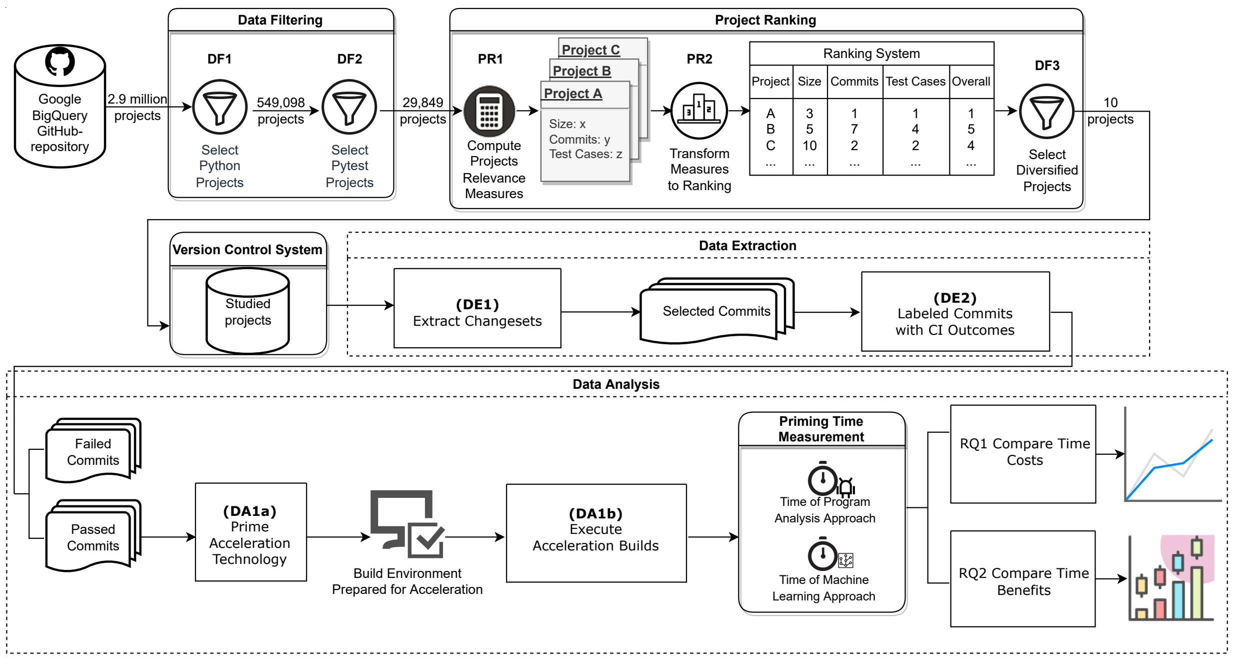
Google BigQuery GitHub-repository
2.9 million projects
Data Filtering
DF1
Select Python Projects
549,098 projects
DF2
Select Pytest Projects
29,849 projects
Project Ranking
PR1
Compute Projects Relevance Measures
Project C
Project B
Project A
Size: x
Commits: y
Test Cases: z
PR2
Transform Measures to Ranking
Ranking System
Project | Size | Commits | Test Cases | Overall
A B C ... | 3 5 10 ... | 1 7 2 ... | 1 4 2 ... | 1 5 4 ...
DF3
Select Diversified Projects
10 projects
Version Control System
Studied projects
Data Extraction
(DE1) Extract Changesets
Selected Commits
(DE2) Labeled Commits with CI Outcomes
Data Analysis
Failed Commits
Passed Commits
(DA1a) Prime Acceleration Technology
Build Environment Prepared for Acceleration
(DA1b) Execute Acceleration Builds
Priming Time Measurement
Time of Program Analysis Approach
Time of Machine Learning Approach
RQ1 Compare Time Costs
RQ2 Compare Time Benefits

Figure 4.1: The study design for each studied project of CI Acceleration Time Trade-off

spent in preparing the training set, running the full test suite, recording test results, and identifying a relevant subset of the test suite based on the training results.

For each of studied projects, we plot of the priming costs of both acceleration approaches. We define the priming cost for an acceleration product as the difference between the time spent on the priming stage for a commit and the cold build time of that commit, i.e., the extra time spent preparing the acceleration environment. We then compare the priming costs of both products and their distributions over 50 commits for each project by conducting Mann-Whitney U tests (two-tailed, paired, Bonferroni-corrected $\alpha = 0.005$ for ten comparisons). To estimate the effect size of the difference, we use Cliff's delta, which is negligible when $0 \leq \delta < 0.147$, small when $0.147 \leq \delta < 0.33$, medium when $0.33 \leq \delta < 0.474$, and large otherwise.

RQ1: Results. Below, we present two observations with respect to the priming cost trends among the studied approaches.

Observation 1 – The PA-product has a lower priming cost than the ML-product. We find that in the priming process of all studied projects, the average priming cost of the PA-product is 148.25 percentage points less than that of the ML-product. Indeed, Figure 4.2 shows that the highest priming cost of the PA-product generally outperforms the lowest priming cost of the ML-product. Only the *Ansible* project breaks this trend, and in that case, all of the commits used with the PA-product still have a lower priming cost than 96% of the commits for the ML-product. Furthermore, the trend holds across a wide range of build times. Indeed, in the shortest cold build (i.e., project *asciinema* with an average cold build time of 2 seconds), the ML-product needs more than 20 seconds on average to finish the priming stage, while the PA-product takes only 2.41 seconds on average. The largest difference occurs in the project with the longest cold build time, (i.e., project *airflow* with an average cold build time of 39 min 23 second), where the average priming cost for the ML-product is 26 minutes longer than the PA-product.

The p-value of all selected projects is much less than the assigned significance level, indicating that the null hypothesis can be rejected and that the PA and ML samples are different to a statistically significant degree. Furthermore, the Cliff's delta varies from 0.80 to 0.86, indicating that the practical differences between the PA and ML priming costs are large.

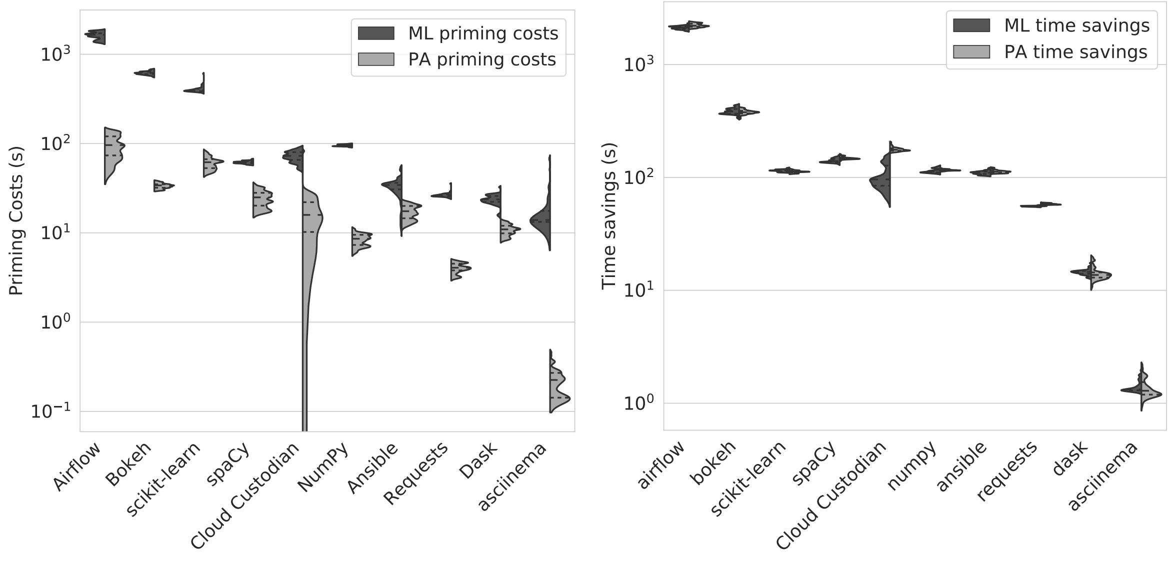

Figure 4.2: Comparison of the priming costs and time savings across 10 projects

Observation 2 – The time spent on the priming process of the two acceleration products correlates with the original build time. We also find that as build times change, both acceleration products present proportional changes in the time spent on the priming stage. This trend holds across all projects (see our online appendix)[2].

To quantify this phenomenon, we calculate the Spearman's ρ correlation coefficient between non-accelerated build time and the time spent on the priming stages of each product. We find that $\rho = 0.9028$ for the ML-product and $\rho = 0.9564$ for the PA-product.

Overall, the PA-product likely has a slightly higher correlation because the priming process for this product hinges on building a dependency graph which is unlikely to face significant changes from one commit to the next. This likely allows the product to observe a relatively stable priming stage. Meanwhile, the ML-product may present more sporadic fluctuations due to the internal workings of the various steps it uses to accelerate a build. However, when exploring each product's respective trends, we find that neither product presents a significant increasing or decreasing trend in priming time as the build times change.

Outcome 1: The priming cost of the PA-product is substantially less than that of the ML-product. The priming time is highly correlated with the original build time in all cases.

4.3.2 RQ2: What are the time savings of CI acceleration products?

RQ2: Approach. To measure the time savings, we first record the non-accelerated, PA-accelerated, and ML-accelerated build times and then compute the relative and absolute differences between the non-accelerated build time for each studied acceleration product. We define time savings as the time difference between the accelerated and non-accelerated build times of a commit.

For each of our ten studied projects, we compare the time savings of both products over 50 commits for each project by conducting Mann-Whitney U tests (two-tailed, paired, Bonferroni-corrected $\alpha = 0.005$ for ten comparisons). Similar to RQ1, we use Cliff's delta to estimate the effect size of the difference. We also explore the range and trends in time savings for both approaches using the builds of 50 passing commits for each project.

RQ2: Results. Below, we present two observations with respect to the time savings of our studied approaches.

Observation 3 – The PA-product tends to provide more time savings than the ML-product. Looking at the topmost points of the plots in Figure 4.2, we observe that the PA-product performs better in 7/10 projects in maximum time saving and 6/10 projects in median time saving.

The *Bokeh* project is one of the four projects where the ML-product provides more time savings. The trend figure shows that starting from the 34th commit, the ML-product outperforms the PA-product. The trend is especially clear in later commits, where the ML-product has accrued a larger amount of training data. A similar trend is observed in the three other projects where the ML-product outperforms the PA approach (see Section 2 of the online appendix[2] for the figures). However, this improvement for the ML-product is not always present because the content of changesets varies with each project. Furthermore, even when the ML-product provides more time savings, the difference in savings (i.e., effect size) is never large. As a result, the PA-product provides larger time savings in general, and particularly when few prior data points (i.e., prior builds) are available.

Observation 4 – The ML-product shows a greater variation in time savings over multiple commits. According to the Mann-Whitney U test results, the p-values of eight of the ten studied projects are less than the significance level. This means the null hypothesis can be rejected and the time savings of the acceleration products are different to a statistically significance degree.

When referring to the practical differences between each product, there are four projects whose statistically significant effect sizes lie in the large interval; three projects in the medium interval; and one project in the small interval. Indeed, six of the seven projects with large and medium effect sizes show better time savings for the PA-product. Only two projects, *scikit-learn* and *Dask*, show statistically significant results where the ML-product presents greater time savings. Of those, *Dask* shows a small effect size and *scikit-learn* shows a medium effect size between the products.

Moreover, the ML-product presents a broader range of time savings for six projects. The four projects that show a broader range of time savings for the PA-product generally have shorter non-accelerated build times.

Outcome 2: The PA-product tends to outperform the ML-product in terms of time savings and stability.

4.4 Chapter Summary

The rapid feedback of CI processes allows users to verify if their source code changes integrate cleanly with their existing systems. CI acceleration promises to further accelerate the CI process while maintaining its benefits. However, using CI acceleration is not without costs. To evaluate the practical implications of CI acceleration technology, we conduct an empirical study of 100,000 builds across ten projects to compare two kinds of commercial-grade CI acceleration products across two dimensions: priming cost and time savings. We make four observations from which we conclude that:

- The priming costs of the PA-product are substantially less than those of the ML approach.
- During the acceleration process, the PA-product generally provides more time savings than the ML product.

The outcomes of this empirical study indicate that the selected PA-product currently exhibits higher commercially practical value than the selected ML-product.

In the next chapter, we continue the empirical comparison of the same CI acceleration products. We turn our focus to the trustworthiness of two families of CI acceleration products. We explore which product more often align with the baseline when applied to known failing builds

Chapter 5

CI Acceleration Trustworthiness Comparison

An earlier version of the work in this chapter has been submitted to IEEE Software.

5.1 Introduction

In this chapter, we systematically evaluate the trustworthiness of two products with respect to non-accelerated builds. There are two types of errors that an acceleration approach can make: (a) builds that should have failed, but are labelled as passing; and (b) build that should have passed, but are labelled as failing. Previous research has shown that CI outcomes are consistent when using CI acceleration[28] (i.e., type (b) errors are uncommon). However, builds can have failing outcomes for multiple reasons, and these may not be consistent when using CI acceleration. This can allow some bugs to slip through CI when acceleration is in use. Therefore, we focus on the effect of CI acceleration on failing builds. Using the full range of parameter settings for the PA and ML products, we replay the builds of 50 failing commits from ten large and active open-source projects in PA-accelerated, ML-accelerated, and non-accelerated settings. This results in a total of 100,000 builds. This benchmark allows us to answer the following research question:

RQ: How does the trustworthiness of CI acceleration products compare?

> Outcome: While the performance of neither product shows a clear trend across our studied projects, the PA-product is more trustworthy, producing rates of agreement with non-accelerated counterparts that are 4.28–75.66 percentage points higher than the ML-product. Furthermore, the most trustworthy ML parameter settings vary so broadly that the best (top 10) settings are only consistent for at most 11–25 commits out of 50 within the studied projects, and 14–18 commits out of 50 across the studied projects.

We conclude that while there is still room for improvement for both CI acceleration products, the selected PA-product currently provides more trustworthy results. Future work is needed to combat the tendency of ML-based acceleration to product unstable and untrustworthy results.

5.2 Study Design

Below, we present how we filter (Section 5.2.1) and rank (Section 5.2.2) candidates to obtain our set of studied projects. We then explain how we extract CI data from each studied project (Section 5.2.3). Finally, we present the approaches that we use to answer our research questions (Section 5.2.4).

5.2.1 Data Filtering

We begin by retrieving a dataset of GitHub repositories from Google BigQuery.[1] The dataset contains the activity and property information for the repositories on GitHub.

Select Python projects (DF1): To mitigate the influence of different programming languages on our experimental results, we select projects that use a single programming language. We focus on Python because it is the only common language that is supported by both of our selected acceleration products. We query projects that have 'Python' as a field within the 'language' table in the BigQuery dataset. After applying our first filter, 549,098 projects survive.

Select pytest projects (DF2): After removing all non-Python projects, we use a second filter to select projects that use the pytest framework. We select the pytest framework

[1] https://cloud.google.com/bigquery/public-data

Table 5.1: The projects' details and their trustworthiness performance

Project name	# Commits	# Files	# Test Cases	Description	Domain	% of build results that fully align with original build results	
						PA build	ML build
Requests	6,107	126	554	HTTP library	Network	72%	0.26%
Airflow	15,629	14,062	9,890	Workflow management platform	Workflow management	50%	0.03%
Ansible	52,514	5,693	3,626	IT automation platform	IT automation	52%	0.84%
asciinema	821	91	26	Terminal session recorder	Productivity	66%	23.79%
Numpy	30,262	1,963	23,644	Scientific computing Python package	Statistics	40%	35.72%
Bokeh	19,609	4,409	9,784	Interactive data visualization tool	Visualization	46%	1.24%
Dask	7,280	447	593	Binary analysis platform	Task scheduling	70%	23.23%
scikit-learn	28,198	5,338	9,753	Machine learning library	Machine learning	44%	36.02%
Cloud Custodian	3,786	705	8,270	Cloud security and governance tool	Cloud management	76%	0.34%
spaCy	15,546	1,216	4,400	Natural Language Processing tool	NLP	46%	2.22%

because previous research demonstrated that it has a more stable performance profile than other testing frameworks [6]. After applying our second filter, 29,849 projects survive.

5.2.2 Project Ranking

Although our experimental procedures are largely automated, repeating the experiment on thousands of systems is untenable. We therefore apply a ranking procedure to systematically select a set of studied projects.

Compute projects relevance measures (PR1): We inspect the number of commits, files and test cases, which respectively correspond to the repositories' activity, scale of production code, and scale of testing code. Repositories with a large number of commits and files contain more code changes, and are more likely to offer an adequate volume of data for validation. The number of test files is likely associated with the speed at which a build completes. We expect the time consumed by the build of a studied project to be large enough to allow us to perform a meaningful analysis of trustworthiness.

Transform measures to ranking (PR2): To obtain the impact of metrics during project selection, we sum the above indicators, and rank the surviving projects in descending order.

Select diversified projects (DF3): To obtain a diverse set of projects from which reliable conclusions can be drawn, we exclude projects from previously sampled domains, selecting the next highest ranked project from another domain as a replacement. Finally we obtain 10 studied projects from different domains, which are shown in Table 5.1.

5.2.3 Data Extraction

After selecting the studied projects, we extract their data using our Data Extraction (DE) procedure. Figure 5.1 provides an overview of the procedure. We describe each step below.

Extract changesets (DE1): CI acceleration operates on changesets. Thus, we extract changesets from the Git repositories of our studied projects. We separately select 50 consecutive commits that pass the build and 50 non-consecutive commits that fail for for each studied project by walking the commit history in reverse chronological order. The exact listing of studied commits is available in our online appendix.[2]

Label commits with CI outcomes (DE2): We manually label each of the selected commits according to CI outcomes (e.g., pass or fail). We conduct this manual labelling because of the inconsistency in CI checks for different commits. The CI checks[3] in this chapter refer to the GitHub's check run for the public repository, which contains the passed/failed outcomes from various CI services like GitHub Actions, CircleCI and Codecov. We obtain relevant information by checking the passed or failed label listed after the commit date. Each commit has a different number of checks, which varies with commit branches and build environments. GitHub's automatic system labels commits as failed if at least one check fails. For example, we found an inconsistency in commit **f3719bf** in the project **cloud-custodian/cloud-custodian**. In that instance, GitHub's automatic system labels the commit as failed; however, the build passes when we reproduce it locally.

5.2.4 Data Analysis

Figure 5.1 provides an overview of our data analysis procedure. Below, we describe each step in the procedure.

Execute accelerated builds (DA1): We consider builds that undergo build acceleration to be *warm builds*. For the PA-product, the dependency graph's construction is completed during the execution of the *cold build*, i.e., the initial, complete build in which no steps are skipped. During a warm build, the product uses the graph generated during the cold build to locate changed lines of code and their dependencies. By traversing this graph, the product can skip cases that are irrelevant to a code change.

For the ML-product, the source of the training set is the historical test results (i.e., names and locations of test cases, test status and duration of each test case) and code

[2] `https://doi.org/10.5281/zenodo.7641215`

[3] https://docs.github.com/en/developers/apps/guides/creating-ci-tests-with-the-checks-api

changes (i.e., names and locations of changed files, number of changed lines, commit hashes and authors of changes). In this chapter, the complete training set we applied comes from the 50 passing-build commits we select in DE1. Specifically, we apply build acceleration on N commits, where we use the previous commits (i.e., from the 1st to $(N-1)$th commit) for training. The acceleration of the first commit is guided by the untrained machine learning classifier. The classifier uses the previous commits for training starting with the second commit. During a warm build, the product creates a subset of the test suites using different optimization targets.

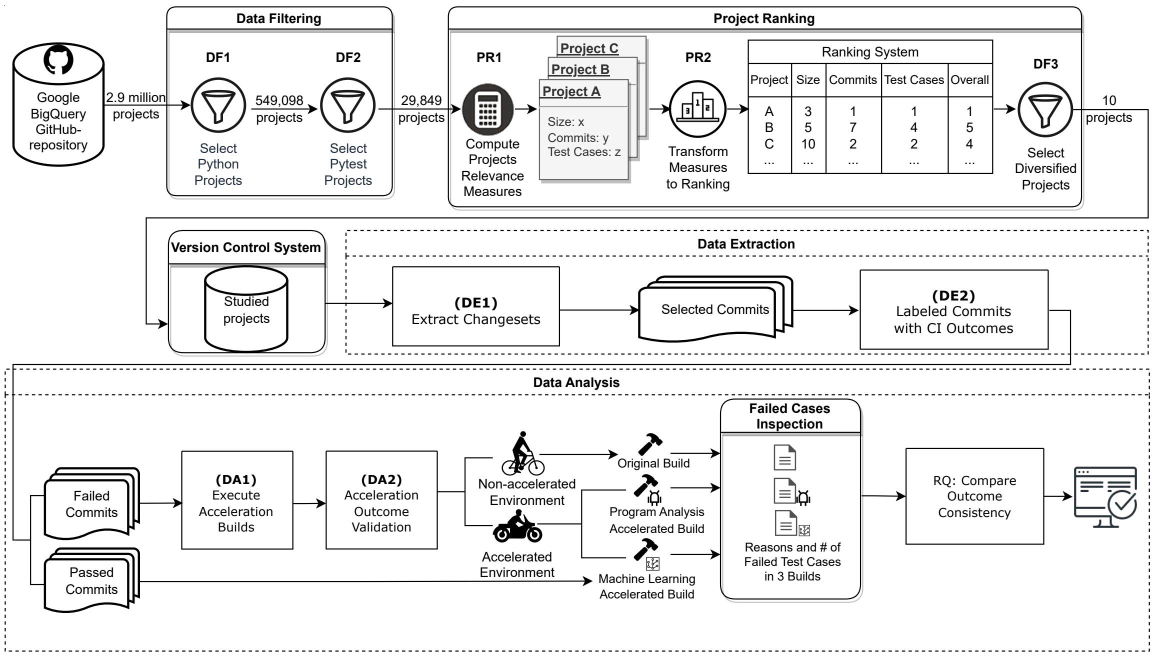
Data Filtering
Project Ranking
Google BigQuery GitHub-repository
2.9 million projects
DF1
Select Python Projects
549,098 projects
DF2
Select Pytest Projects
29,849 projects
PR1
Compute Projects Relevance Measures
Project C
Project B
Project A
Size: x
Commits: y
Test Cases: z
PR2
Transform Measures to Ranking
Ranking System
Project Size Commits Test Cases Overall
A 3 1 1 1
B 5 7 4 5
C 10 2 2 4
DF3
Select Diversified Projects
10 projects
Version Control System
Studied projects
Data Extraction
(DE1) Extract Changesets
Selected Commits
(DE2) Labeled Commits with CI Outcomes
Data Analysis
Failed Commits
Passed Commits
(DA1) Execute Acceleration Builds
(DA2) Acceleration Outcome Validation
Non-accelerated Environment
Accelerated Environment
Original Build
Program Analysis Accelerated Build
Machine Learning Accelerated Build
Failed Cases Inspection
Reasons and # of Failed Test Cases in 3 Builds
RQ: Compare Outcome Consistency

Figure 5.1: The overview of study design of CI Acceleration Trustworthiness Comparison

Acceleration outcome validation (DA2): To validate the trustworthiness of our studied build acceleration products, we conduct a comparison of failed builds between accelerated and non-accelerated environments. We therefore separately explore the pytest summary for the original build, the PA accelerated build, and the ML accelerated build.

We analyze the differences between the sources of failures and the number of failed cases for all three outcomes. Our studied CI accelerated products are based on test skipping, so differences in deliverables will not affect our study results. Therefore, passing builds cannot produce mistakes in our study setting. Nevertheless, this would make an interesting direction for future work that analyzes other CI acceleration approaches.

We judge the trustworthiness of accelerated builds by comparing the test outputs of the two acceleration products against the original build. To do so, we determine whether the acceleration products' build outcomes are consistent with the original build outcomes (i.e., they present identical outcomes), and label them as either consistent or inconsistent. Through automated scripts and manual spot-checks, we inspect the consistency of two acceleration products within the 50 studied failing commits for each studied project.

5.3 Study Results

In this section, we present the results of our study with respect to our research question. Specifically, we present the approach followed by our observations.

5.3.1 RQ: How does the trustworthiness of CI acceleration products compare?

Approach. To compare the trustworthiness of our selected CI acceleration products, we compare the failed build results for 50 builds accelerated by both the PA and ML-products for each subject project. Indeed, we select 50 failing builds for each project because we seek to determine whether the CI acceleration products will yield the same failing test results as non-accelerated builds. After identifying 50 failing builds for each project (Section 2, DE2), we extract the number of errors and failures according to the test reports generated by the pytest module. To compare the two products, we use formula (5.1), where EF_A represents the sum of errors and failures for accelerated builds and EF_U represents the sum of errors and failures for non-accelerated builds, to represent the trustworthiness for each commit.

$$trustworthiness = \frac{EF_A}{EF_U} \tag{5.1}$$

Since the studied acceleration products results in fewer tests run by the build, the resulting acceleration cannot identify more errors and failures than the non-accelerated build. As a result, this fraction cannot exceed 1.

To further analyze the optimal parameter setting for the ML-product, we explore how often specific parameter settings offer highly trustworthy results. We do this for the full range of confidence settings (i.e., how high does the model-estimated likelihood need to be before it decides to skip a task) and target settings (i.e., how small of a subset of tasks should the acceleration aim to produce). Thus, we identify for which settings and how often the ML-product's trustworthiness ranks in the top 20% range for the 50 commits studied for each project. The aim of this exploration is to uncover the most trustworthy settings for ML acceleration.

Results. Below, we present three observations with respect to the trustworthiness of studied acceleration products.

Observation 1 – The PA-product more often aligns with non-accelerated build results. Figure 5.2 shows that the overall trustworthiness of a PA accelerated build is higher than the trustworthiness of a ML accelerated build. As shown in Table 5.1, a PA accelerated build can fully align with non-accelerated build results for up to 76% of builds (i.e., the number of errors and failures for the accelerated product are equal to the number of errors and failures in the non-accelerated build). We also observe that this alignment fluctuates between 40% and 76% for the PA-product. We believe that this fluctuation is caused by greedy test skipping.

While the ML test acceleration product can sometimes present a trustworthiness that matches or even surpasses the PA-product in specific commits, this is generally not the case. Indeed, the median value for the ML-product in Figure 5.2 is less than 80% trustworthy. While we only present results for the Requests project, the situation is similar for other projects. Indeed, the PA-product outperforms the ML-product in terms of overall trustworthiness for all of our studied projects. A complete comparison of the trustworthiness for the two products for all studied projects is available in our online appendix.[2]

Observation 2 – The trustworthiness of both acceleration products is project dependent. Neither product presents a unified trustworthiness trend in our studied projects. Indeed, we could not identify a trend for either product when comparing their fluctuations in trustworthiness across our studied projects.

For PA-accelerated builds, we observed that the fluctuating ranges for each projects are different. For example, the PA trustworthiness for *Requests* shows a slight fluctuation around 100%. However, the trustworthiness of the product alternates irregularly between 0 and 100% in the *NumPy* and *asciinema* projects. This could be explained by the small number of original errors and failures in these projects. Indeed, the non-accelerated build contains at most two failures in *NumPy* and three failures in *asciinema*. When the PA-product skipped test cases, the accelerated build failed to run any erroneous or failing test cases, leading to a trustworthiness value of 0%.

A small number of original errors and failures also affects the trustworthiness of ML accelerated builds. The distribution of trustworthiness values hovers around 0% trust-

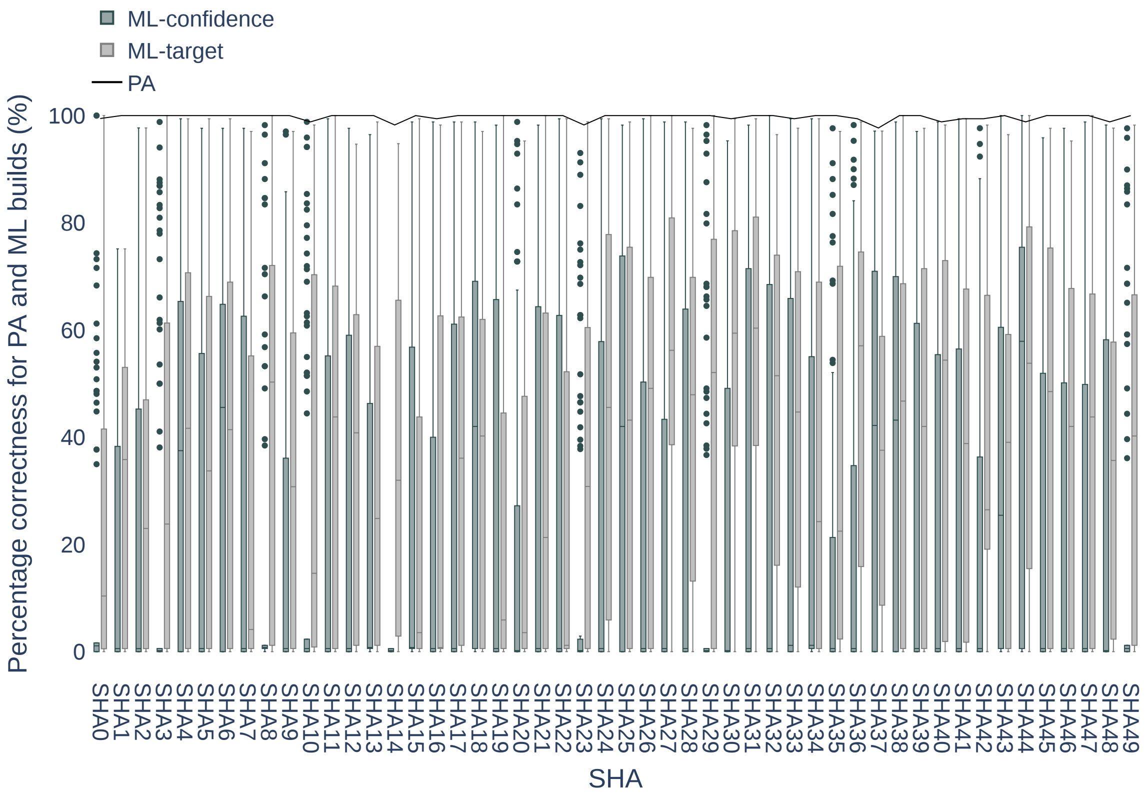

Figure 5.2: Trustworthiness of both CI acceleration products using all parameter settings for 50 failing Requests builds

worthiness in projects with few original errors and failures (e.g. *NumPy* and *asciinema*). Meanwhile, in the projects with more original errors and failures, the distribution of ML trustworthiness is located in different trustworthiness intervals (e.g., 20% - 60% in *Requests*).

Observation 3 – The ML-product does not present any optimal parameter setting for trustworthiness. As shown in Figure 5.3, the parameter setting with the highest trustworthiness (i.e., 81% confidence) is only the best setting 25% of the time in the Requests project. Indeed, this trend holds for all of our projects. The parameter setting with the highest incidence of trustworthiness is the 67% target setting in the scikit-learn project, and even in that case, the setting was only the best 50% of the time. We provide the figures for our other studied projects in our online appendix.[2]

In Figure 5.3, we also observe that the top five trustworthiness incidences belong to the confidence setting. Thus, the confidence setting generally yields build results that more closely align with non-accelerated builds in the *Requests* project. However, this trend does not hold for the remaining projects. Indeed, the confidence setting presents the highest trustworthiness in only three of nine studied projects, while the target setting presents highest trustworthiness in five of nine projects. Both settings present the same maximum occurrence of trustworthiness in the *airflow* project. Overall, the highest trustworthiness occurs in the range of 14 to 18 commits out of 50. The situation indicates that there are no ideal target and confidence settings in ML-accelerated builds. The top ten most frequent trustworthiness settings, both within projects and across projects, are shown in our online appendix.[2]

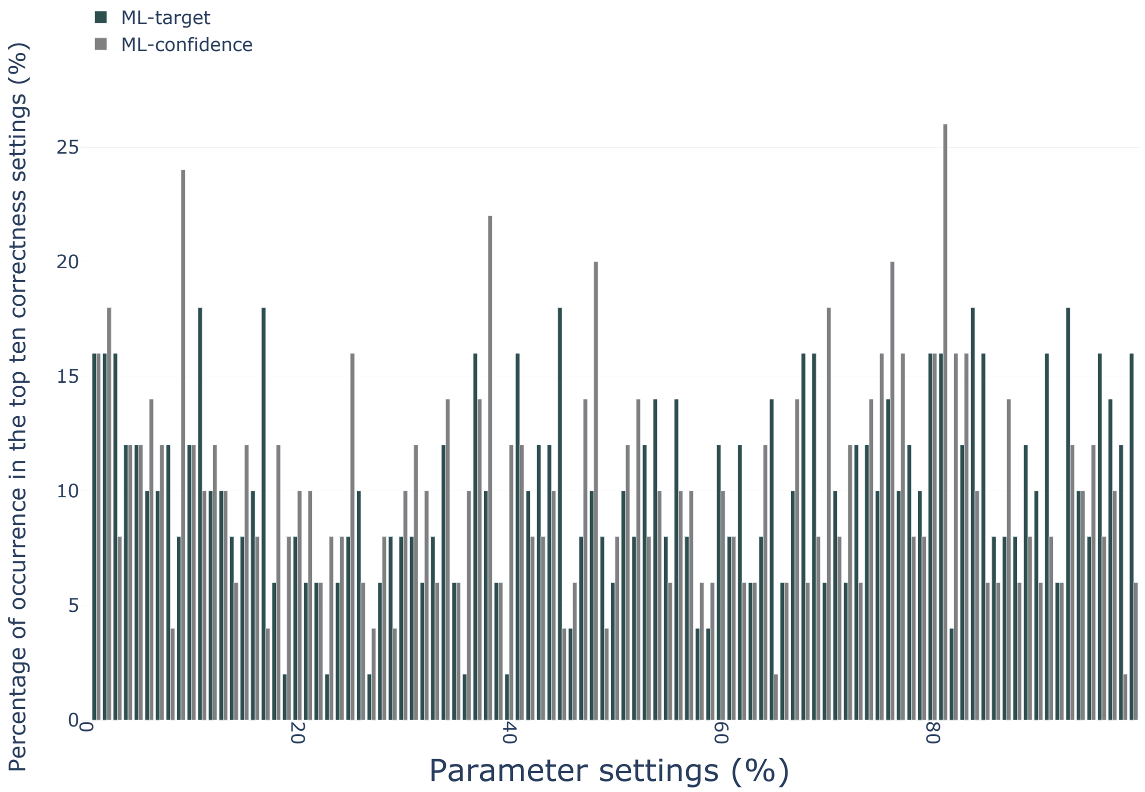

Figure 5.3: Top 10 trustworthiness occurrence frequency of the ML-product using different parameter settings for 50 failing Requests builds

Outcome: While the performance of neither product shows a clear trend across our studied projects, the PA-product provides more trustworthy results(4.28–75.66 percentage points). Furthermore, the optimal ML parameter settings vary so broadly that the best (top 10) settings are consistent for at most 50% of commits within projects, and at most 36% across projects.

5.4 Chapter Summary

Users of CI expect to obtain rapid software development feedback, allowing them to verify if their source code changes integrate cleanly with their existing systems. CI acceleration promises to further accelerate the CI process while maintaining its benefits. However, using CI acceleration is not without costs. To evaluate the practical implications of CI acceleration technology, we conduct an empirical study of 100,000 builds that span ten projects to compare two kinds of commercial-grade CI acceleration techniques in terms of trustworthiness. We make three observations from which we conclude the results of builds accelerated by the PA-product more often aligns with non-accelerated build results.

The outcomes of this empirical study indicate that the selected PA-product currently exhibits a higher degree of trustworthiness than the selected ML-product. The gap with non-accelerated build results indicates that both acceleration products could benefit from improvements to achieve more robust acceleration.

Indeed, this study has shown that acceleration performance may sacrifice trustworthiness. Adams[1] showed that developers (at least those within the Linux kernel development context) prefer a correct build to a fast one, so CI acceleration that sacrifices correctness may not be appealing to stakeholders. We therefore encourage future work to explore how the trustworthiness gap can be bridged.

Knowing that the trustworthiness of neither PA or ML-product fully align with the baseline (non-accelerated setting), in the next chapter, we conduct an in-depth analysis of the PA-product, to determine the reasons that hinder its trustworthiness, mutation testing is applied to uncover the fault patterns that causes the low trustworthiness.

Chapter 6

Mutation-Guided Assessment

An earlier version of the work in this chapter has been submitted to an international conference.

6.1 Introduction

In this chapter, we assess the trustworthiness of CI acceleration by comparing the outcomes of mutation testing of accelerated and unaccelerated builds of studied change sets. We first measure the percentage of mutants that only survive in the accelerated setting (i.e., gap mutants) to quantify the discrepancy between the accelerated and unaccelerated settings. Then we associate gap mutants with the dependency representation of the studied CI acceleration product to understand the limitations of its decision-making. Finally, we summarize patterns in these limitations by inspecting the gap mutants to identify root causes of discrepancies that undermine the trustworthiness of the studied CI acceleration product. Our analysis of ten open-source projects reveals 2,237 gap mutants, which can be grouped into six patterns. This benchmark allows us to answer the following research questions:

RQ1: How often do mutants survive due to CI acceleration?

Outcome 1: While 60% of the studied mutants survive both accelerated and unaccelerated settings, a gap in mutation outcomes between both settings exists and varies from 0.11%–23.50% across the studied projects.

RQ2: How are mutants that survive due to CI acceleration mapped to the source code?

Outcome 2: 88.95% of gap mutants can be mapped to specific source code functions and classes using the dependency representation of the studied CI acceleration product; however, 6.66% of gap mutants appear outside of the scope of source code classes, and 4.38% of gap mutants are completely absent from the dependency representation.

RQ3: What causes mutants to survive in CI acceleration?

Outcome 3: A majority (69%) of gap mutants survive in the accelerated setting due to deterministic reasons that can be classified into six fault patterns. However, a considerable proportion (22.5%) of gap mutants survive in CI acceleration because of non-deterministic build behaviour (e.g., mutants that timeout in the unaccelerated setting and survive in the accelerated setting). An analysis of the literature suggests that at least the six deterministic fault patterns can apply to, and help improve, other CI acceleration approaches.

6.2 Research Questions

In this study, we verify the percentage of survived mutants in accelerated and unaccelerated builds to quantify the gap between accelerated and unaccelerated CI outcomes. To structure this analysis, we formulate the following research question (RQ):

RQ1: How often do mutants survive due to CI acceleration?

Mutation testing is a code-based analysis method [62] that can complement code coverage analysis [82]. Indeed, mutation testing generates mutants in the source code and provides the precise location of each survived mutant [19]. This feature enables us to associate survived mutants with features of the programming language. Moreover, it may help to identify limitations in the decision-making process of the CI acceleration approach. Accordingly, we formulate this inquiry through our second research question:

RQ2: How are mutants that survive due to CI acceleration mapped to the source code?

Mutation testing can be used to evaluate the quality of test suites [40]. Mutation testing can therefore allow us to compare the quality of an original unaccelerated test suite to the quality of the same test suite when it is the target of CI acceleration. By comparing the results of these two scenarios, we seek to gain insight into the root causes of discrepancies that harm the trustworthiness of our studied CI acceleration product. More precisely, we study why some mutants only survive in accelerated builds through our third research question:

RQ3: What causes mutants to survive in CI acceleration?

6.3 Study Design

The goal of our study is to verify the outcomes of an emerging CI acceleration product and uncover reasons for outcomes that can erode trust. To realize our goal, we apply mutation testing and inspect the mutants that survive in the unaccelerated and accelerated builds. To that end, we study the locations in which these mutants appear and reason about why they survive in ten well-established OSS.

Below, we present how we filter (Section 6.3.1) and rank (Section 6.3.2) candidates to obtain our studied projects. We then explain how we verify the mutation results for each studied project (Section 6.3.3). Finally, we present the approaches that we use to answer our research questions (Section 6.3.4).

6.3.1 Data Filtering

We begin by retrieving a dataset of GitHub repositories from Google BigQuery.[1] This open-source dataset of software repositories was exported from GitHub and allows legacy and standard SQL queries. The dataset contains the activity (e.g., commits) and property information (e.g., programming language) for the repositories that are hosted on GitHub.

[1]https://cloud.google.com/bigquery/public-data

Select Python Projects (DF1): To mitigate the influence of different programming languages on our experimental results, and reduce confounding factors, we focus on the builds of projects that are written in a single programming language.

To that end, we group candidate projects by programming language and select those that are primarily implemented in Python. We focus on the Python programming language because it is the programming language for which the selected CI acceleration product can guarantee the most stable and trustworthy acceleration. We apply the filter by querying for projects that have 'Python' as a field within the 'language' table in the BigQuery dataset. The query returns the projects that predominantly use the Python programming language (i.e., Python makes up the majority of the source code). After applying our first filter, 549,098 projects survive.

Select Pytest Projects (DF2): After removing all non-Python projects, we select the Python projects that use the pytest framework.[2] We select the pytest framework because previous research demonstrated that it has a more stable performance profile than other testing frameworks [6]. After applying our second filter, 29,849 projects survive.

6.3.2 Project Ranking

Although our experimental procedures are largely automated, the execution cost for each studied project is large. Thus, repeating the experiment on thousands of systems is untenable. We therefore apply a ranking procedure to systematically select a set of subject systems.

Compute Project Relevance Measures (PR1): We inspect the surviving projects using three metrics, the number of commits, the number of files, and the number of test cases. We consider the number of commits and files because repositories with a high number of commits and files are more likely to offer an adequate volume of data for validation. Meanwhile, the number of test files is likely associated with the speed at which a build completes. In this chapter, we expect the time consumed by the build of a studied project to be large enough to perform meaningful CI acceleration. Since we cannot build every project to estimate its time cost, we therefore use the number of test files as an estimate of this measure.

Transform Measures to Rankings (PR2): To comprehensively quantify the impact of our three chosen measures on our project selection, we rank each subject system. The

[2]https://docs.pytest.org/en/7.1.x/contents.html

details of our ranking system are shown in Figure 6.1. Given this procedure and the above three measures, we rank the surviving projects in descending order.

Select Studied Projects (DF3): Mutation testing is computationally expensive [43, 69]. To maintain a manageable experimental process, we set an upper bound for mutation time when selecting projects. Specifically, we set the maximum allowable mutation duration to one week per project.

To obtain a diverse set of projects from which reliable conclusions can be drawn, we exclude projects from any previously sampled domains, selecting the next highest-ranked project from another domain as a replacement. For example, if the *Supervisor* and the *Nagstamon* projects both belong to the process management domain, we select *Supervisor* because it is the highest-ranked project from the domain in our ranking. We then omit *Nagstamon* and any other projects that belong to the same domain. Finally, we obtain 10 studied projects from different domains, which are shown in Table 6.2.

6.3.3 Data Extraction

After selecting the studied projects, we extract their data using our Data Extraction procedure. Figure 6.1 provides an overview of the procedure. We describe each step below.

Extract Changesets (DE1): CI acceleration approaches operate on changesets. In the case of our studied projects, we extract changesets from the Git version control system. We select commits with release tags for each studied project. Indeed, release code usually contains fewer failed test cases for a given project, because the release tag can indicate that the code has undergone a rigorous testing process and has been deemed stable enough for release. The exact information for each commit that we studied is listed in our online appendix.[3]

6.3.4 Data Analysis

To determine the trustworthiness of our selected CI acceleration product, we study the quantity, locations and reasons for the mutants that survive in the accelerated build, but do not survive in the unaccelerated build. First, we perform an initial unaccelerated build on the parent commit of a studied release for each studied project. This is done to enable the CI acceleration product to generate its internal graph, which is used to reason about build and test invocation steps that may be skipped in the future. Next,

[3]https://doi.org/10.5281/zenodo.7872663

we checkout two copies of the studied release—one for executing unaccelerated (baseline) builds and another for executing accelerated builds. We then apply mutation testing on the independent copies of the codebase, ensuring that the same set of mutants is generated in both settings. Finally, we compare the mutants that we extract from the unaccelerated and accelerated mutation reports. Figure 6.1 provides an overview of our data analysis procedure. Below, we describe each step in the procedure.

Perform Initial Builds (DA1): We perform initial builds using two settings: (a) unaccelerated build – the complete build of a given project release in which no steps are skipped; (b) accelerated build – the complete build of a given project release in which our chosen CI acceleration product skips tests based on its inferred dependency graph. During the accelerated build, we use the inferred dependency graph generated using an instrumented build of the parent commit of the release. This graph from the parent commit replicates the state that the acceleration product would be in prior to the release commit. By traversing the graph, the CI acceleration product skips tests that are deemed irrelevant to a particular code change.

Generate Mutants (DA2): When the initial builds are finished, we conduct mutation testing using Mutmut.[4] Mutmut generates mutants according to established rules and provides a mutation report. Most relevant to this study context, Mutmut allows us to produce the same set of mutants that we evaluate in both accelerated and unaccelerated settings.

Inspect Mutants (DA3): The generated mutation reports contain detailed information about the mutation process and its outcome, such as the number of killed, timeout, suspicious, survived and skipped mutants, as well as the mutation operator used for each mutant, and the mutated lines of code. We inspect the mutants that survive in the accelerated and unaccelerated builds of each studied project. The difference between the rates of mutant survival in the two settings quantify the gap between accelerated and unaccelerated CI outcomes.

Compare Mutants (DA4): When comparing the mutants that belong to the unaccelerated and accelerated builds of a single release, there are three kinds of mutants: (1) the mutants that are killed in both settings, (2) the mutants that survive in both settings, and (3) the mutants that only survived in the accelerated build. Type 1 and 2 mutants indicate agreement between the two settings, indicating that the test suite is either capable of detecting the mutant or not, respectively; however, type 3 mutants suggest that the acceleration procedure has erroneously omitted a test that would kill the mutant. Thus, in this

[4]https://mutmut.readthedocs.io/en/latest/

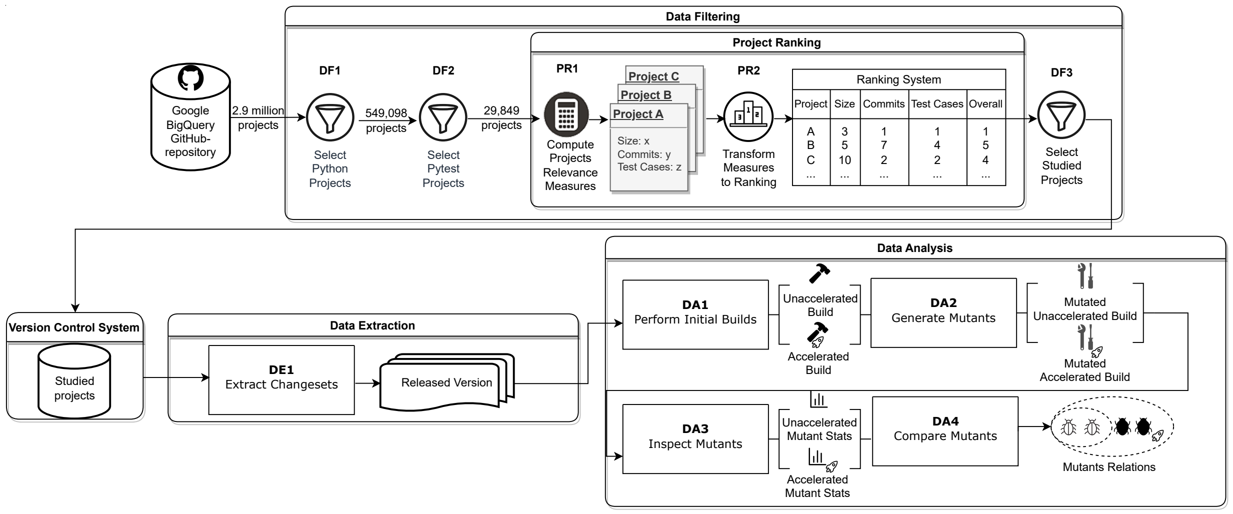

Figure 6.1: The study design for each selected project of mutation-guided assessment

chapter, we concentrate on type 3 mutants, i.e., those that only survive in the accelerated setting.

The principle for this verification is based on the assumption that in the ideal case, a CI acceleration product should only skip test cases that can safely be skipped (i.e., would not affect the result of the build). In this ideal case, the mutation testing outcomes for unaccelerated and accelerated builds should be consistent (i.e., exactly the same). However, if mutation testing were to show that an accelerated build presents different survived mutants than an unaccelerated build, then the CI acceleration product's trustworthiness may be in jeopardy.

6.4 Study Results

In this section, we present the results of our study with respect to our RQs. For each RQ, we present our approach, followed by our observations. We also extend the discussion of our patterns from the CI acceleration product studied in RQ3 to CI acceleration approaches presented in prior works.

Table 6.1: Confusion Matrix for Gap Rate Calculation

	Accelerated	Non-accelerated
Survived	a	b
Killed	c	d
Timeout	e	f
Suspicious	g	h

6.4.1 RQ1: How often do mutants survive due to CI acceleration?

RQ1: Approach. To measure the gap between accelerated and unaccelerated CI outcomes, we calculate the percentage of survived mutants in each studied project. Specifically, we find the number of mutants that survive in accelerated and unaccelerated CI builds in their respective mutation reports, and use Equation 6.1 to measure the *Gap Rate*, which indicates the percentage of mutants that only survive during acceleration.

Mutmut reports one of five outcomes for each mutant:

- **Killed mutant:** mutants that have been killed by at least one test case;
- **Survived mutant:** mutants that have not been killed by any test case;
- **Timeout mutant:** mutants where tests took a long time, but not long enough to be fatal;
- **Suspicious mutant:** mutants where the test suite took 10 times as long as the baseline;
- **Skipped mutant:** mutants that are skipped due to certain reasons, such as syntax errors in the mutated code or insufficient memory.

Since there are no skipped mutants in the mutation process of all of our studied projects, we consider the remaining four mutant outcomes in our *Gap Rate* formula.

$$Gap\,Rate = \frac{|a \cap d|}{|a \cup c \cup e \cup g|} \tag{6.1}$$

RQ1: Results. Here, we describe our observations pertaining to the gap between accelerated and unaccelerated CI outcomes.

Observation 1 – While the percentage of survived mutants varies for each project, many mutants tend to survive, even in unaccelerated builds. Table 6.2 shows the number of mutants that survive in unaccelerated and accelerated builds. We observe that the mutant survival rate in more than half (6 out of 10) of our studied projects is over 50 percent in both accelerated and unaccelerated builds. Indeed, for project *scholia* and *dvc*, over 90 percent of mutants survived in both accelerated and unaccelerated builds.

The average mutant survival rate in our ten studied projects is 58.46% for the unaccelerated builds and 62.45% for the accelerated builds. This high mutant survival rate is important to note because it suggests that test suites of our studied projects are in general not particularly effective at detecting mutants. Indeed, capacity of a test suite to detect mutants may affect the trustworthiness of the CI acceleration product. If a test suite is of poor quality, e.g., if it contains redundant, irrelevant, or flaky tests, the CI acceleration product may erroneously skip tests that should not be skipped.

Observation 2 – CI acceleration indeed allows mutants that do not survive unaccelerated builds to survive in an accelerated setting. The Gap Rate for our studied projects is presented in Table 6.2. With an average Gap Rate of 7.24% and a standard deviation of 8.17%, our ten studied projects show that the trustworthiness of CI acceleration greatly depends on the project. Indeed, Table 6.2 illustrates that while the gap between mutants that survive in accelerated and unaccelerated builds can be zero (see the *mackup* project), it can also be as high as 23.5% (see the *retext* project).

In the ideal CI acceleration scenario, the Gap Rate is zero. Indeed, this is the case for the *mackup* project, where there is no difference between mutants that survive accelerated and unaccelerated builds. In this project, mutation testing provides consistent outcomes, indicating that CI acceleration safely skips test cases, and does not affect the results of the build. However, *mackup* is the only project which presented this scenario. The remaining nine studied projects showed varying degrees of inconsistency in the mutation verification outcomes.

A non-zero Gap Rate indicates that CI acceleration performs untrustworthy test skipping. Indeed, in the case of the project *retext*, a Gap Rate of 23.5% was observed. This implies that nearly a quarter of the mutants generated for this project survived in the accelerated build, despite the fact that they could be killed by the original test suite. By using the Gap Rate, we can identify mutants that should not survive, which we then inspect more closely to address RQ2 and RQ3.

Table 6.2: Survived Mutants and Gap Rate of each studied project

Projects	Domain	Survived Mutants (%)		Total Mutants	Gap Mutants	Gap Rate
		Unaccelerated Settings	Accelerated Settings			
mackup	Configuration Tool	408 (86.26%)	408 (86.26%)	473	0	0
dvc	Data Version Control	18165 (99.69%)	18185 (99.80%)	18221	20	0.11%
scholia	Organization	4295 (90.14%)	4320 (90.66%)	4765	25	0.58%
httpie	Http Client	3070 (77.19%)	3150 (79.21%)	3977	80	2.54%
bottle	Python Web Framework	62 (2.23%)	64 (2.30%)	2783	2	3.13%
supervisor	Process Management	38 (0.42%)	40 (0.44%)	9153	2	5.00%
asciinema	Productivity	905 (86.44%)	974 (93.03%)	1047	69	7.08%
ranger	File Management	7036 (62.36%)	8037 (71.23%)	11283	1001	12.45%
asciidoc	Documentation	2579 (32.91%)	3147 (40.61%)	7836	568	18.05%
retext	Text Editor	1530 (46.99%)	2000 (61.43%)	3256	470	23.50%

Outcome 1: The overall percentages of mutants that survive in both accelerated and unaccelerated settings of the 10 studied projects are high. Although the quantity varies from project to project, there exist mutants that survive in accelerated settings and were killed in unaccelerated settings in most (9 of the 10) studied projects.

6.4.2 RQ2: How are mutants that survive due to CI acceleration mapped to the source code?

RQ2: Approach. To study the distribution of mutants that survive due to CI acceleration (i.e., gap mutants) in the source code, we propose four mapping categories. These categories describe the location of each mutant with respect to the source code of the project, and the dependency graph generated by our studied CI acceleration product.

- **No Mapping:** The mutated statement appears within the scope of a function, but this mapping is absent in the dependency graph of the CI acceleration product;
- **Not in Class:** The mutated statement appears outside of the scope of source code classes (e.g., in a configuration file), and is untracked in the dependency graph;
- **Mapping in Class:** The mutated statement appears within the scope of a class definition (but outside of the scope of a method) and is correctly mapped in the dependency graph;

- **Mapping in Function:** The mutated statement appears within the scope of a function and is correctly mapped in the functions of the dependency graph.

The dependency graph generated by the CI acceleration product can accurately represent the function-level granularity of a project, which consists of three components: the project's abstract syntax tree, known tests, and their dependencies. To classify the above mapping relations, we first locate each gap mutant in the source code. Then we check if the mapping exists in the dependency graph.

Specifically, if the mutated statement is within the scope of a function, we query for it in the function level of the dependency graph and label the mutant as 'Mapping in Function'. Similarly, if the mutated statement is within the scope of a class but outside of the scope of any function when we query for it in the class level of the dependency graph, and label the mutant as 'Mapping in Class'. Additionally, there are mutants that do not appear within functions or classes. In these cases, the CI acceleration product is unable to include them in the dependency graph, and they are labeled as 'Not in Class'. The mutant labeled as 'No Mapping' appears within the scope of a function, but is untracked in the dependency graph. The 'No Mapping' category is a direct indication of incomplete or missing information in the dependency graph.

This question aims to associate survived mutants with the CI acceleration dependency graph by finding the mapping relation. Through this categorization we aim to identify the limitations of the dependency graph generated by studied CI acceleration product.

RQ2: Results. Below, we describe our observations pertaining to the mapping of gap mutants in the dependency graph.

Observation 3 – Most of the gap mutants are mapped in the function category of the dependency graph. We extract the 2,237 gap mutants from ten studied projects. As shown in Table 6.3, nearly 90% of gap mutants could be tracked in the dependency graph. Among them, the majority (95.5%) of gap mutants are in the 'Mapping in Function' category, whereas only 88 gap mutants (4.4%) are in the 'Mapping in Class' category. For the 'Mapping in Function' category, we randomly select a gap mutant to verify its validity. We first locate the mutated statement in the source code, and confirm the location is outside the function but inside the class scope. Then we query for the class name in the dependency graph and successfully track the class name under the test dependencies.

This distribution is consistent with the encapsulation feature of object-oriented programming, i.e., keeping variables and corresponding methods (functions) together in a single unit (class). When we inspect the source code distribution of our ten studied projects,

Table 6.3: Category and Frequency of Gap Mutant Mapping

Mapping Category	Frequency	
Mapping in Function	1902	85.02%
Mapping in Class	88	3.93%
Not in Class	149	6.66%
No Mapping	98	4.38%
• Statement in function	90	4.02%
• Statement in class	8	0.36%

we find that, on average, 11.50% of the code (lines of code) is written outside the scope of classes and functions. This effectively indicates that around 90% of gap mutants should be tracked in the dependency graph. However, this also implies that the CI acceleration product is making mistakes despite having dependency graph data.

Observation 4 – Incomplete information exists in the dependency graph. According to the Table 6.3, the 'No Mapping' category is rare. Specifically, 4.02% of gap mutants located in function scope and 0.36% of gap mutants located in class scope could not be mapped within the dependency graph generated by the CI acceleration product.

We also study the distribution of the 'No Mapping' mutants across the studied projects. As shown in the Table 6.4, the highest project-specific 'No Mapping' rate is 100%. All mutants from both the *bottle* and *supervisor* projects could not be mapped within the dependency graph. Conversely, all of the gap mutants in the *dvc* and *scholia* projects could be mapped within the dependency graph. For the remaining six projects, the 'No Mapping' rate ranges between 1.45% and 4.70%. We believe that alternative ways to identify test cases in the 'No Mapping' category are needed. Indeed if we could force the CI acceleration product not to skip these tests, trustworthiness would be improved. The detailed mapping information for each studied project is shown in our online appendix.[3]

Observation 5 – A small number of gap mutants are not mapped within the dependency graph due to their source code locations. There are 6.66% of gap mutants whose mutated statements are neither within the scope of functions nor the scope of classes (e.g., global variable initialization). Since the dependency graph of the studied acceleration product only records function and class-level information, the dependencies for these mutants cannot be traced within the dependency graph. These may therefore result in unsafe test skipping, and therefore lower trustworthiness. We consider this to be a limitation of the CI acceleration product that should be considered by future products.

Table 6.4: No Mapping Distribution for Studied Projects

Projects	Gap Mutant	No Mapping	
mackup	0	0	0
bottle	2	2	100.00%
supervisor	2	2	100.00%
dvc	20	0	0
scholia	25	0	0
asciinema	69	1	1.45%
httpie	80	3	3.75%
retext	470	18	3.83%
asciidoc	568	25	4.40%
ranger	1001	47	4.70%

Another test acceleration approach [31] bypasses this type of limitation by using data-flow tracing instead of class or method tracing. Specifically, this approach records dependencies using a data-flow model without any location constrains for the statements when deciding the starting point of the data-flow. As a result, this approach is able to track statements outside the scope of classes and functions.

Outcome 2: Most of the gap mutants are traceable within the dependency graph. Incomplete dependency data and the locations where mutations are applied mutants explain only a small proportion of the gap mutants that lack mappings.

6.4.3 RQ3: What causes mutants to survive in CI acceleration?

RQ3: Approach. To uncover the reasons that cause mutants to survive in CI acceleration, we conduct a qualitative analysis of 200 mutants. Through our analysis, we uncover categories of fault patterns that exist in our studied CI acceleration product. Before conducting this analysis, we first design a priming procedure to associate the gap mutants with the dependency graph constructed by the CI acceleration product. The priming procedure consists of three steps:

(1) Reproduce failure in the unaccelerated build: To verify the integrity of gap mutants (and reduce the impact of flaky tests), for each gap mutant, we first re-apply the suspect mutation on the codebase and re-execute the build without acceleration;

Table 6.5: Reasons and Frequency of Gap Mutant Survival

Reasons	Frequency	
Dependencies untracked by the CI acceleration product	138	69.00%
• Class properties	40	20.00%
• Global variable	31	15.50%
• Constructor	23	11.50%
• Static method decorator	17	8.50%
• Conditional statement	14	7.00%
• Configuration-purpose string	13	6.50%
Non-deterministic build behaviours	45	22.50%
• Inconsistent Labeling	24	12.00%
• Flaky Tests	21	10.50%
Other (Failed to classify)	17	8.50%

(2) Confirm the test is skipped by CI acceleration: By rerunning the mutated code, we can locate failed tests and verify if the test is skipped in the accelerated build;

(3) Associate the mutant-exposing test case with the gap mutant: We inspect the test and the source code being tested to determine why the test was skipped.

This priming procedure, and more specifically the inspection in step (3), is onerous. Indeed, it is impractical to inspect all 2,237 identified gap mutants using our approach. Therefore, to achieve analytical generalization [25], we use the principle of saturation. Specifically, we continue to check new gap mutants until no new patterns of survival have been discovered for 50 consecutive gap mutants. In the end, using this criterion, we inspect 200 gap mutants, covering all studied projects except *mackup*, since there is no gap between accelerated and unaccelerated settings in this project.

RQ3: Results. Below, we describe the observations that we make with respect to the reasons why gap mutants survive in the accelerated build. Table 6.5 provides an overview of the patterns that we observe from the 200 inspected cases. For each pattern, we provide its definition and a mutant example.

Observation 6 – 69% of the inspected gap mutants that survive in the accelerated build had dependencies that were untracked by the CI acceleration product.

The inspected gap mutants in this category fall into six patterns, which we define and characterize below.

Class properties: The most common (20% of the time) pattern is the mishandling of class properties. Indeed, a missing class-related dependency (e.g., a mishandled class property) in the dependency graph would cause the CI acceleration product to incorrectly label test cases that depend on this code as independent. Therefore, the CI acceleration product would erroneously label these test cases as safe to skip. We uncovered this issue during our analysis when checking the dependencies of class properties, where we discovered class fields with null values in the CI acceleration product's dependency graph. In those cases, if a skipped test case depends on the variable initialization in the class property, it can lead to a mutant surviving mutation testing when it should not, implying a lack of trustworthiness for the CI acceleration product.

```
'''
./supervisor/options.py
@@ -69,7 +69,7 @@
'''
    pass

class Options:
-    stderr = sys.stderr
+    stderr = None
    stdout = sys.stdout
    exit = sys.exit
    warnings = warnings
```

Pattern 6.1: The gap mutant survived due to class properties

Static method decorator: 17% of inspected gap mutants had code statements mutated in a function annotated with *@staticmethod*. *@staticmethod* is a decorator in Python that defines a static method within a class. Static methods belong to a class rather than its objects, and can be called without creating an instance of the class. This can cause the static method dependencies to be inaccurately captured by CI acceleration.

```
'''
./asciidoc/asciidoc.py
@@ -1693,7 +1693,6 @@
'''
        if not skipsubs:
            Title.attributes['title'] = Title.dosubs(Title.attributes['title'])

-    @staticmethod
    def dosubs(title):
```

Pattern 6.2: The gap mutant survived due to static method decorator

Global variable: 15.5% of the inspected gap mutants survive because of a lack of, or incomplete dependencies due to, global variables. Indeed, in these cases, the CI acceleration product fails to correctly record the dependency relations of the global variables in the source code. Changes made to global variables can affect the behaviour of the code under test, and skipping tests that depend on those variables can lead to erroneous results. In our experiments, this can occur when global variables are mutated and tests are erroneously skipped. Indeed, if the CI acceleration product cannot accurately record global variable dependencies, it may skip tests that are necessary to ensure the trustworthiness of the accelerated build.

```
'''
./supervisor/options.py
@@ -60,7 +60,7 @@
'''
    version_txt = os.path.join(mydir, 'version.txt')
    with open(version_txt, 'r') as f:
        return f.read().strip()
-VERSION = _read_version_txt()
+VERSION = None

 def normalize_path(v):
    return os.path.normpath(os.path.abspath(os.path.expanduser(v)))
```

Pattern 6.3: The gap mutant survived due to incorrect handling of global variables

Constructor: 11.5% of the inspected gap mutants survived because the CI acceleration product did not detect the dependency between test cases and a constructor. Specifically, these failures occur when the mutated statements are located within the scope of a constructor. In those cases, the dependency graph fails to record the dependency information for the constructor, and thus, the test cases that test these methods are erroneously skipped.

```
'''
asciinema/pty_.py
@@ -153,7 +153,7 @@
'''
 class SignalFD:
    def __init__(self, signals: List[signal.Signals]) -> None:
-        self.signals = signals
+        self.signals = None
        self.orig_handlers: List[Tuple[signal.Signals, Any]] = []
        self.orig_wakeup_fd: Optional[int] = None
```

Pattern 6.4: The gap mutant survived due to constructor

Conditional statement: 7% of inspected gap mutants survive due to complex conditional statements in the source code. Specifically, these mutated statements typically involve multiple conditions that are combined using logical operators such as ‘and’, ‘or’, and ‘not’. Mutmut generates mutants in these logical keywords and we use these mutations to discover incomplete information for the functions that contain these complex conditional statements in the dependency graph. We find that the dependencies for the complex conditional statements are not captured accurately by the CI acceleration product, as a result, the mutants survive during acceleration.

```
'''
./ReText/tab.py
@@ -185,7 +185,7 @@
'''
        errMsg = errMsg.replace('<a href="%s">', '').replace('</a>', '')
      return '<p style="color: red">%s</p>' % errMsg
    headers = ''
-   if includeStyleSheet and self.p.ss is not None:
+   if includeStyleSheet and self.p.ss is  None:
      headers += '<style type="text/css">\n' + self.p.ss + '</style>\n'
    elif includeStyleSheet:
```

Pattern 6.5: The gap mutant survived due to conditional statement

Configuration-purpose string: 6.5% of inspected gap mutants have mutated statements (e.g., version and date strings) belonging to the configuration files. These strings are often separate from the body of functional code; however, configurations can also affect the behaviour of the program and the tests that need to be executed. The CI acceleration product does not track changes in the strings of configuration files, so it skips tests that are relevant to the changes made in the configuration file, further leading to the survival of gap mutants.

```
'''
httpie/__init__.py
@@ -4,7 +4,7 @@
'''
 __version__ = '3.2.1'
-__date__ = '2022-05-06'
+__date__ = 'XX2022-05-06XX'
 __author__ = 'Jakub Roztocil'
 __licence__ = 'BSD'
```

Pattern 6.6: The gap mutant survived due to configuration-purpose string

Observation 7 – 22.5% of the inspected gap mutants survive due to non-deterministic build behaviour originating from test flakiness and inconsistent

mutation outcomes.

Flaky Tests: 10.5% of the inspected gap mutants survive due to flaky tests. These tests produce non-deterministic outcomes when executing on the same program version [3]. Flakiness is often caused by test code that has external dependencies or relies on non-deterministic algorithms [34]. These flaky tests can cause mutants to survive (i.e., the build passes) for a mutant that should have been killed (i.e., the build should have failed). Flakiness can therefore causes mutants to be incorrectly labeled as gap mutants.

Inconsistent Labeling: Inconsistent labeling from our selected mutation tool also results in non-deterministic build behaviour between accelerated and unaccelerated builds. When running mutation testing on both unaccelerated and accelerated builds, some mutants are classified as 'suspicious' or 'timeout' in the unaccelerated build, and are meanwhile classified as 'survived' in the accelerated build. This causes such mutants to be labeled as gap mutants by our approach. These gap mutants, caused by non-deterministic build behaviour, cannot be used to reason about the trustworthiness of CI acceleration. Indeed, it is possible to reduce the impact of such issues by repeating the experiment, but there is no guarantee that such experimental noise can be fully eradicated.

In addition, 17 (8.5%) mutants in our qualitative analysis were not classifiable based on our taxonomy. These mutants survive because of project-specific situations. For example, in the *httpie* project, an http request failed because of a mutated URL. In other cases, we lack the domain knowledge to diagnose the issue. We attempt to keep this "Other" category as small as possible by using a cyclic procedure when analyzing the mutants. Specifically, whenever a new category appears, we repeatedly re-check whether any of the "Other" mutants belong to this new category.

Outcome 3: Most (69%) mutants survive CI acceleration due to deterministic reasons that can be generally classified into six fault patterns. However, a smaller, but not insignificant proportion (22.5%) of mutants survive CI acceleration because of non-deterministic build behaviour (e.g., flaky tests).

Discussion – Generalizability of the observations. According to the systematic literature review for CI acceleration costs (Section 3.2.2), prior research shows a preference for test selection and prioritization approaches that make use of failure history [85, 88], execution history [47, 58], and test coverage [14]. We believe that rare failures are therefore potential pitfalls for these approaches. All the causes for gap mutant survival could also occur in rarely modified code. Using mutation-based testing to analyze the trustworthiness of these approaches might therefore allow the simulation of rarer failure cases and improve the trustworthiness of these approaches. Overall, by design, such approaches might miss some faults due to the side effects of test case prioritization. Indeed, while other issues might exist as well, all the reasons for gap mutant survival found in our research might also exist in these approaches if the tests to detect these reasons are judged to be inefficient. Therefore, while our chosen approach specifically makes use of program analysis for test selection, we firmly believe that our methodology, as well as the issues that we detected could help improve other traditional test selection approaches.

The takeaways of this chapter could be extended to additional types of CI acceleration techniques other than program analysis techniques. A systematic literature review aimed at machine learning acceleration [60] found that these test selection or prioritization approaches concentrate only on a few, easily obtainable features. Indeed, these ML-based approaches that concentrate on code coverage are likely to suffer from similar issues as the ones presented in this chapter. Moreover, even state-of-the-art approaches [11] make use of class-level dependency analysis, which does not prevent issues such as those covered by configuration-purpose issues. Therefore, while we make use of a traditional program analysis CI acceleration approach, our methodology, and the issues that we detected could also help improve machine learning test selection and prioritization approaches.

6.5 Threats to Validity

Below, we describe the threats to the validity of our study.

6.5.1 Construct Validity

We apply the default setting for the selected mutation tool without any customization. The default mutation strategy was not originally designed for the purpose of assessing the trustworthiness of CI acceleration. Within the pre-established strategy, we analyze a limited number of mutants. An analysis of the impact of different configuration settings may prove fruitful; however, we believe that our results have already demonstrated the promise of our mutation-based approach to assessing the trustworthiness of CI acceleration.

6.5.2 Internal Validity

Our capacity to discover fault patterns is based on the inspection of gap mutants. If the mutation tool fails to generate mutants that relate to the changed code, we cannot detect trustworthiness problems associated with this part of the code. This, in turn, prevents us from detecting the complete set of fault patterns in the CI acceleration decision-making. However, in principle, the selected mutation tool (i.e., Mutmut) applies mutations to a broad range of statements.

6.5.3 External Validity

We set an upper bound of mutation time to select projects for our experiment that can be analyzed within practical time limits. This choice constrains the scale of the studied projects. Moreover, we study GitHub-hosted open-source projects that use Python and pytest. Because GitHub is one of the most popular hosts for open source projects and Python is one of the most popular programming languages, our findings apply to a wide range of projects. Similarly, pytest is one of the most popular unit testing frameworks for Python and mutation testing is a mature program analysis approach. Nonetheless, we encourage extensions of our study using other approaches, projects, languages, and test frameworks.

6.6 Chapter Summary

Users of CI expect to obtain rapid and reliable feedback, allowing them to verify if their source code changes integrate cleanly with their existing systems. CI acceleration promises to further accelerate the CI process while maintaining its trustworthiness. However, the trustworthiness of CI acceleration is not guaranteed. To evaluate the trustworthiness of a commercial-grade CI acceleration product, we apply mutation testing within accelerated and unaccelerated settings across ten projects to measure the gap between accelerated and unaccelerated builds. We make seven observations (see Section 6.4) from which we conclude that:

- 90% of studied projects contain mutants that survive in accelerated settings although they are killed in unaccelerated settings.
- 88.95% of the mutants that only survive in the accelerated setting are traceable within the dependency graph of the chosen CI acceleration product.
- While 22.5% of mutants only survive in the accelerated setting because of non-deterministic build behaviour, 69% survive because of deterministic reasons, and can be broadly classified into six categories.

In conclusion, this study shows that while CI acceleration may sacrifice trustworthiness, it is possible to use techniques such as mutation testing to evaluate this sacrifice and identify improvements for current approaches. For future work, our study demonstrates that the following improvements for PA-based CI acceleration approaches would improve their trustworthiness: (1) depending on the size and complexity of the codebase, it may be necessary to manually refine the dependency graph, especially by concentrating on class properties, global variables, and constructor components; and (2) solutions should be added to detect and bypass flaky test during CI acceleration to minimize the impact of flakiness.

6.7 Data Availability

To enable replication and foster further enhancements of our results, we make a detailed online appendix available to the research community.[3]

Chapter 7

Conclusion

Continuous Integration (CI) is a practice that allows developers to quickly integrate and verify projects modifications. CI acceleration products are a boon to developers seeking rapid feedback. However, using CI acceleration is not without costs. Indeed, the priming stage of CI acceleration incurs a time cost, which hinders the efficiency of CI acceleration products. Furthermore, even if a product provides an enticing trade-off between time costs and benefits, if the outcomes vary between accelerated and non-accelerated settings, the trustworthiness of the acceleration is called into question.

In this thesis, to understand the trade-offs between costs and benefits of two commercial-grade CI acceleration products, we conduct an empirical comparison of a program analysis and a machine learning acceleration product. We also conduct a fine-grained assessment of the decision-making guidelines (e.g., the dependency graph of changed code) of the program analysis product. The outcomes of this thesis show that achieving acceleration may be accompanied by a loss of trustworthiness, suggesting that users of CI acceleration techniques should carefully weigh their trade-offs.

7.1 Contribution and Findings

Prior works that focus on the performance evaluation of CI acceleration are mostly high-level evaluations (e.g., Gallaba et al. [28] directly compare the CI acceleration outcomes with the traditional CI service CircleCI, but do not inspect the internal decision-making). Prior to the work conducted for this thesis, a fine-grained analysis of the fault patterns that exist in CI acceleration decision-making was lacking. To uncover the in-depth factors that affect the practicality of CI acceleration products, we claim that:

Thesis Statement: CI acceleration is not free. Replay analysis can be leveraged to evaluate the costs and benefits in terms of time and trustworthiness. Furthermore, mutation testing can complement replay analysis to provide a fine-grained assessment of the limitations that hinder CI acceleration.

The empirical comparisons of two families of CI acceleration products and the mutation-guided assessment reveal that:

- *CI Acceleration Time Trade-off*
  The empirical study of 100,000 builds across ten projects to compare PA and ML CI acceleration products indicates that the selected PA-product currently exhibits lower priming cost and greater time savings than the selected ML-product (Chapter 4).
- *CI Acceleration Trustworthiness Comparison*
  The result from re-executing 50 failing builds from ten open-source projects in non-accelerated (baseline), PA-accelerated, and ML-accelerated settings indicates both acceleration approaches still have potential to achieve more robust acceleration, while the selected PA-product currently exhibits a higher degree of trustworthiness than the selected ML-product (Chapter 5).
- *Mutation-Guided Assessment*
  The fine-grained analysis of the PA-product shows that while CI acceleration may sacrifice trustworthiness, it is possible to use techniques such as mutation testing to evaluate this sacrifice and identify improvements for current approaches (Chapter 6).

7.2 Opportunities for Future Research

Our research shows that there is still room for improvement for both PA and ML-based CI acceleration approaches. Aiming at closing the trustworthiness gap and reducing the priming cost, we describe the path for future work to improve the performance of each CI acceleration product.

7.2.1 Program analysis based CI acceleration

We summarize the fault patterns that hinder the trustworthiness of a PA product. By concentrating on class properties, global variables, and constructors, the components with the greatest frequency of missing dependencies in our analysis, it may be necessary to manually refine the dependency graph if the the size and complexity of the codebase is simple. Future work may complement our fault pattern set by conducting a larger scale of mutation-guided assessment. Additionally, some solutions could be added to detect and bypass flaky test during CI acceleration to minimize the impact of flakiness.

7.2.2 Machine learning based CI acceleration

We select 50 past build commits as the sample for our time-related comparisons. It is possible that the acceleration approaches do not have enough data to reach steady-state, and that we are actually measuring the models while they are still being trained rather than fully accelerating. The ML product does show an improvement in time savings as projects accrue build data beyond our initial training. Future work may investigate whether the accelerated builds more frequently align with the non-accelerated builds by feeding the machine learning model with additional training data.

References

[1] Bram Adams. Co-evolution of source code and the build system. In *Proc. of the International Conference on Software Maintenance*, pages 461–464, 2009.

[2] Bram Adams and Shane McIntosh. Modern Release Engineering in a Nutshell: Why Researchers should Care. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering*, pages 78–90, 2016.

[3] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. Predicting flaky tests categories using few-shot learning. *arXiv preprint arXiv:2208.14799*, 2022.

[4] Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. A comparative study of vectorization-based static test case prioritization methods. In *Proc. of the International Conference on Software Engineering and Advanced Applications*, pages 80–88, 2020.

[5] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2012.

[6] Lívia Barbosa and Andre Hora. How and why developers migrate python tests. In *Proc. of the International Conference on Software Analysis, Evolution and Reengineering*, pages 538–548, 2022.

[7] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proc. of the Joint Meeting on Foundations of Software Engineering*, pages 770–781, 2015.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Proc. of the International Conference on Mining Software Repositories*, pages 356–367, 2017.

[9] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *Proc. of the International Conference on Software Engineering: Software Engineering in Practice*, pages 268–277, 2021.

[10] Jean-Marcel Belmont. *Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI.* Packt Publishing, 2018.

[11] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. Learning-to-rank vs ranking-to-learn: Strategies for regression testing in continuous integration. In *Proc. of the International Conference on Software Engineering*, page 1–12, 2020.

[12] Qi Cao, Ruiyin Wen, and Shane McIntosh. Forecasting the Duration of Incremental Build Jobs. In *Proc. of the International Conference on Software Maintenance and Evolution*, page 524–528, 2017.

[13] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21(3):445–478, 2013.

[14] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. Optimizing test prioritization via test distribution analysis. In *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 656–667, 2018.

[15] Frédéric Dadeau, Pierre-Cyrille Héam, and Rafik Kheddam. Mutation-based test generation from security protocols in hlpsl. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 240–248, 2011.

[16] Frédéric Dadeau, Pierre-Cyrille Héam, Rafik Kheddam, Ghazi Maatoug, and Michael Rusinowitch. Model-based mutation testing from security protocols in hlpsl. *Software Testing, Verification and Reliability*, 25(5-7):684–711, 2015.

[17] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[18] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. Towards mutation analysis of android apps. In *Proc. of the International Conference on Software Testing, Verification and Validation Workshops*, pages 1–10, 2015.

[19] Anna Derezińska and Konrad Hałas. Analysis of mutation operators for the python language. In *Proc. of the International Conference on Dependability and Complex Systems DepCoS-RELCOMEX*, pages 155–164, 2014.

[20] Anna Derezinska and Konrad Halas. Experimental evaluation of mutation testing approaches to python programs. In *Proc. of the International Conference on Software Testing, Verification and Validation Workshops*, pages 156–164, 2014.

[21] 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, 2015.

[22] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *Proc. of the International Conference on Automated Software Engineering*, pages 374–377, 2012.

[23] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 291–295, 2019.

[24] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[25] Kathleen M Eisenhardt. Building theories from case study research. *Academy of management review*, 14:532–550, 1989.

[26] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. Cloudbuild: Microsoft's distributed and caching build service. In *Proc. of the International Conference on Software Engineering Companion*, pages 11–20, 2016.

[27] Matheus Ferreira, Lincoln Costa, and Francisco Carlos Souza. Search-based test data generation for mutation testing: a tool for python programs. In *Anais da IV Escola Regional de Engenharia de Software*, pages 116–125, 2020.

[28] Keheliya Gallaba, John Ewart, Yves Junqueira, and Shane McIntosh. Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *IEEE Transactions on Software Engineering*, 48(8):2040–2052, 2022.

[29] Keheliya Gallaba, Maxime Lamothe, and Shane McIntosh. Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. In *Proc. of the International Conference on Software Engineering*, page 1330–1342, 2022.

[30] Keheliya Gallaba and Shane McIntosh. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects that (mis)use Travis CI. *IEEE Transactions on Software Engineering*, 46(1):33–50, 2020.

[31] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 1–11, 2018.

[32] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24:2102–2139, 2019.

[33] Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[34] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of flaky tests in python. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 148–158, 2021.

[35] Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane McIntosh. The impact of the adoption of continuous integration on developer attraction and retention. In *Proc. of the International Conference on Mining Software Repositories*, pages 491–494, 2017.

[36] Sarra Habchi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. On the use of mutation in injecting test order-dependency. *arXiv preprint arXiv:2104.07441*, 2021.

[37] Mary Jean Harrold. Testing: a roadmap. In *Proc. of the International Conference on the Future of Software Engineering*, pages 61–72, 2000.

[38] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proc. of the International Conference on Automated Software Engineering*, pages 426–437, 2016.

[39] Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, and Alexander Pretschner. Nequivack: Assessing mutation score confidence. In *Proc. of the International Conference on Software Testing, Verification and Validation Workshops*, pages 152–161, 2016.

[40] Mostafa Jangali, Yiming Tang, Niclas Alexandersson, Philipp Leitner, Jinqiu Yang, and Weiyi Shang. Automated generation and evaluation of jmh microbenchmark suites from unit tests. *IEEE Transactions on Software Engineering*, 2022.

[41] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010.

[42] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proc. of the International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[43] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. Prioritizing mutants to guide mutation testing. In *Proc. of the International Conference on Software Engineering*, pages 1743–1754, 2022.

[44] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In *Proc. of the International Conference on Foundations of Software Engineering*, pages 821–830, 2017.

[45] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. System-level test case prioritization using machine learning. In *Proc. of the International Conference on Machine Learning and Applications (ICMLA)*, pages 361–368, 2016.

[46] Eero Laukkanen, Maria Paasivaara, and Teemu Arvonen. Stakeholder perceptions of the adoption of continuous integration–a case study. In *Proc. of the Agile Development Conference*, pages 11–20, 2015.

[47] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *Proc. of the International Conference on Software Engineering*, page 688–698, 2018.

[48] Jackson A Prado Lima and Silvia R Vergilio. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, 121:106268, 2020.

[49] Christian Macho, Shane McIntosh, and Martin Pinzger. Predicting Build Co-Changes with Source Code Change and Commit Categories. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering*, pages 541–551, 2016.

[50] Christian Macho, Shane McIntosh, and Martin Pinzger. Extracting build changes with builddiff. In *Proc. of the International Conference on Mining Software Repositories*, pages 368–378, 2017.

[51] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *Proc. of the International Conference on Software Analysis, Evolution and Reengineering*, pages 106–117, 2018.

[52] Shane McIntosh. *Studying the Software Development Overhead of Build Systems*. PhD thesis, Queen's University, 557 Goodwin Hall, Kingston, ON, Canada, September 2015.

[53] Shane McIntosh, Bram Adams, and Ahmed E Hassan. The evolution of java build systems. *Empirical Software Engineering*, 17(4):578–608, 2012.

[54] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E Hassan. Mining co-change information to understand when build changes are necessary. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 241–250, 2014.

[55] Ade Miller. A hundred days of continuous integration. In *Proc. of the Agile Development Conference*, pages 289–293, 2008.

[56] Peter Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc*, 19(1):14–25, 1998.

[57] Ioannis K Moutsatsos, Imtiaz Hossain, Claudia Agarinis, Fred Harbinski, Yann Abraham, Luc Dobler, Xian Zhang, Christopher J Wilson, Jeremy L Jenkins, Nicholas Holway, et al. Jenkins-ci, an open-source continuous integration system, as a scientific data and image-processing platform. *SLAS DISCOVERY: Advancing Life Sciences R&D*, 22(3):238–249, 2017.

[58] Armin Najafi, Weiyi Shang, and Peter C. Rigby. Improving test effectiveness using test executions history: An industrial experience report. In *Proc. of the International Conference on Software Engineering: Software Engineering in Practice*, pages 213–222, 2019.

[59] Agneta Nilsson, Jan Bosch, and Christian Berger. Visualizing testing activities to support continuous integration: A multiple case study. In *Proc. of the International Conference on Agile Software Development*, pages 171–186, 2014.

[60] Rongqi Pan, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. Test case selection and prioritization using machine learning: a systematic literature review. *Empirical Software Engineering*, 27(2):29, 2021.

[61] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Proc. of the International Conference on Software Testing, Verification and Validation*, pages 1–10, 2014.

[62] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[63] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proc. of the International Conference on Software Engineering*, pages 537–548, 2018.

[64] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proc. of the International Conference on software engineering: Software engineering in practice*, pages 163–171, 2018.

[65] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Does mutation testing improve testing practices? In *Proc. of the International Conference on Software Engineering*, pages 910–921, 2021.

[66] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, 48(10):3900–3912, 2021.

[67] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proc. of the International Conference on Software Testing, Verification and Validation Workshops*, pages 47–53, 2018.

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

[69] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157:110388, 2019.

[70] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

[71] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *Proc. of the International Conference on Mining Software Repositories*, pages 345–355, 2017.

[72] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. Continuous deployment of mobile software at facebook (showcase). In *Proc. of the International Conference on Foundations of Software Engineering*, pages 12–23, 2016.

[73] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proc. of the International Conference on Software Maintenance-1999.'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 179–188, 1999.

[74] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proc. of the International Conference on Software Engineering*, pages 724–734, 2014.

[75] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proc. of the International Conference on Mining Software Repositories*, pages 189–200, 2016.

[76] Daniel Ståhl and Jan Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *Proc. of the IASTED International Conference on Software Engineering*, pages 736–743, 2013.

[77] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[78] Daniel Ståhl and Jan Bosch. Industry application of continuous integration modeling: a multiple-case study. In *Proc. of the International Conference on Software Engineering Companion (ICSE-C)*, pages 270–279, 2016.

[79] Mohsen Vakilian, Raluca Sauciuc, J David Morgenthaler, and Vahab Mirrokni. Automated decomposition of build targets. In *Proc. of the International Conference on Software Engineering*, volume 1, pages 123–133, 2015.

[80] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proc. of the International Conference on foundations of software engineering*, pages 805–816, 2015.

[81] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *Proc. of the International Conference on Software Engineering*, pages 105–115, 2019.

[82] Moshi Wei, Yuchao Huang, Jinqiu Yang, Junjie Wang, and Song Wang. Cocofuzzing: Testing neural code models with coverage-guided fuzzing. *IEEE Transactions on Reliability*, 2022.

[83] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of travis ci. In *Proc. Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 647–658, 2019.

[84] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *Proc. of the International Conference on Software Engineering*, pages 419–429, 2003.

[85] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. Terminator: Better automated ui test case prioritization. In *Proc. of Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[86] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 2020.

[87] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. A large-scale empirical study of compiler errors in continuous integration. In *Proce. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 176–187, 2019.

[88] Yuecai Zhu, Emad Shihab, and Peter C. Rigby. Test re-prioritization in continuous testing environments. In *Proc. of the International Conference on Software Maintenance and Evolution*, pages 69–79, 2018.

[89] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhénuc. Do not trust build results at face value-an empirical study of 30 million cpan builds. In *Proc. of the International Conference on Mining Software Repositories*, pages 312–322, 2017.