Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-Graduação em Sistemas e Computação
Doutorado Acadêmico em Ciência da Computação

Understanding the Relationship between Continuous Integration and Code Coverage

José Diego Saraiva da Silva

Natal-RN

Maio de 2023

José Diego Saraiva da Silva

Understanding the Relationship between Continuous Integration and Code Coverage

Doctoral Thesis presented to the Graduate Program in Systems and Computing of the Department of Informatics and Applied Mathematics at the Federal University of Rio Grande do Norte as a partial requirement for obtaining the degree of Doctor of Computer Science.

Research Area:
Software Engineering

Advisor:

Uirá Kulesza

Co-Advisor:

Daniel Alencar da Costa

PPgSC – Programa de Pós-Graduação em Sistemas e Computação

DIMAp – Departamento de Informática e Matemática Aplicada

CCET – Centro de Ciências Exatas e da Terra

UFRN – Universidade Federal do Rio Grande do Norte

Natal-RN
Maio de 2023

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Setorial Prof. Ronaldo Xavier de Arruda - CCET

Silva, José Diego Saraiva da.
Understanding the relationship between continuous integration and code coverage / José Diego Saraiva da Silva. - 2023.
xviii, 147 f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Departamento de Informática e Matemática Aplicada, Programa de Pós-Graduação em Sistemas e Computação. Natal, RN, 2023.
Orientação: Uirá Kulesza.
Coorientação: Daniel Alencar da Costa.

1. Computação - Tese. 2. Teste de software - Tese. 3. Integração contínua - Tese. 4. Cobertura de código - Tese. 5. Estudo empírico - Tese. I. Kulesza, Uirá. II. Costa, Daniel Alencar da. III. Título.

RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Ata nº. 110

ATA DA SESSÃO DE AVALIAÇÃO DE TESE DE DOUTORADO DO PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO.

Aos trinta e um dias do mês de maio de dois mil e vinte três (31/05/2023), às 15h, por videoconferência, foi instalada a Comissão Examinadora responsável pela avaliação da tese de doutorado intitulada "***Understanding the Relationship between Continuous Integration and Code Coverage***", como trabalho final apresentado pelo(a) candidato(a) **JOSÉ DIEGO SARAIVA DA SILVA** ao Programa de Pós-Graduação em Sistemas e Computação, da Universidade Federal do Rio Grande do Norte, e parte dos requisitos para obtenção do título de **DOUTOR(A) EM CIÊNCIA DA COMPUTAÇÃO**. A Comissão Examinadora foi presidida pelo(a) professor(a) **Dr. UIRÁ KULESZA** (Orientador - UFRN) e contou com a participação de **Dr. DANIEL ALENCAR DA COSTA** (UO – NZL) **Dr. EDUARDO HENRIQUE DA SILVA ARANHA** (UFRN), **Dr. GUSTAVO HENRIQUE LIMA PINTO** (UFPA), **Dr.ª ROBERTA DE SOUZA COELHO** (UFRN) e **Dr. RODRIGO BONIFACIO DE ALMEIDA** (UnB) na qualidade de examinadores. A sessão teve a duração de 4 horas e a Comissão emitiu o seguinte parecer: o trabalho e desempenho do candidato atenderam aos requisitos necessários a uma Tese de Doutorado, tendo a Comissão Examinadora, portanto **APROVADO** o trabalho.

[illegible]
Examinador(a) Externo(a): **Dr. DANIEL ALENCAR DA COSTA**

Eduardo Aranha
Examinador(a) Interno(a): **Dr. EDUARDO HENRIQUE DA SILVA ARANHA**

[illegible]
Examinador(a) Externo(a): **Dr. GUSTAVO HENRIQUE LIMA PINTO**

Roberta S Coelho
Examinador(a) Interno(a): **Dr.ª ROBERTA DE SOUZA COELHO**

[illegible]
Examinador(a) Externo(a): **Dr. RODRIGO BONIFACIO DE ALMEIDA**

[illegible]
Presidente**: Dr. UIRÁ KULESZA**

José Diego S. da silva
Discente: **JOSÉ DIEGO SARAIVA DA SILVA**

JOSÉ DIEGO SARAIVA DA SILVA

"*Understanding the Relationship between Continuous Integration and Code Coverage*"

Esta Tese foi julgada adequada para a obtenção do título de Doutor(a) em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Natal, 31 de maio de 2023

Acknowledgements

Hello there! As I approach the culmination of my Ph.D. studies, I reflect on the journey that has led me to this momentous milestone. With heartfelt appreciation and profound gratitude, I acknowledge the significant contributions and support of those who have played instrumental roles in shaping my academic and personal growth.

First and foremost, I humbly offer my heartfelt praise and gratitude to God, the Almighty, for His abundant blessings, wisdom, and the invaluable opportunities bestowed upon me. Through His grace, I have successfully completed this thesis, and I am profoundly grateful for His guidance throughout this journey.

I extend my profound appreciation to my mentor, Uirá Kulesza, whose exceptional guidance has been instrumental in shaping the trajectory of my work. Their expertise and dedication have proven invaluable throughout this dissertation. Additionally, I am immensely grateful to my co-advisor, Daniel Alencar, for their mentorship, which enabled me to conduct the pivotal studies that form the cornerstone of this research. Their insights and assistance have played a crucial role in the successful completion of this work.

I am genuinely grateful for the unwavering support and love of my family: José Martiniano da Silva Neto, Maria de Fátima Saraiva da Silva, and José Diogenes Saraiva da Silva. Their constant presence and dedication have enabled me to embark on this Ph.D. journey and face its challenges head-on. The foundation of a loving family has provided me with the strength and motivation to pursue and complete this thesis.

Furthermore, I am indebted to my girlfriend Nathália, whose sacrifices, patience, and understanding have been the cornerstone of my success. Her constant encouragement and belief in me have given me the strength and inspiration to overcome challenges and persevere.

Lastly, but certainly not least, I express my deepest gratitude to my friends, whose unwavering encouragement has been invaluable throughout this thesis journey. Their presence, camaraderie, and insightful discussions have enriched my research and kept me motivated.

I am convinced that it is God who reveals Himself in the orderly harmony of what exists.

Gregor Mendel

Understanding the Relationship between Continuous Integration and Code Coverage

Author: José Diego Saraiva da Silva

Advisor: Uirá Kulesza

Co-advisor: Daniel Alencar da Costa

Abstract

Continuous Integration (CI) is a widely adopted software engineering practice emphasizing frequent software integration through an automated build process. Although CI has been shown to detect errors earlier in the software life cycle, the relationship between CI and code coverage still needs to be clarified. Our work aims to fill this gap by investigating the quantitative and qualitative aspects of this relationship.

In the quantitative study, we compare 30 CI projects and 30 projects that have never adopted CI (NOCI projects) to investigate whether CI is associated with higher code coverage rates. We analyzed 1,440 versions from different projects to identify trends in code coverage. Our findings reveal a positive association between CI and higher code coverage rates.

Our qualitative study consisted of a survey and a document analysis. The survey revealed several significant findings, including a positive association between continuous integration (CI) and higher code coverage rates, indicating the value of CI in promoting testing practices. Additionally, our survey highlighted the importance of using code coverage during the authoring and review process, which can help identify potential issues early in the development cycle.

The document analysis focused on coverage-related themes in pull request conversations in CI-enabled projects. Through this analysis, we discovered the main topics related to coverage usage during pull requests, which can provide valuable insights into how developers use coverage to improve code quality. This information can help guide the development of best practices for using coverage in CI-enabled projects, ultimately improving the quality and reliability of software products.

Our work offers insights into the evolution of code coverage in CI, which can assist researchers and practitioners in adopting tools and practices for better monitoring, maintenance, and even enhancement of code coverage.

Keywords: Software Testing, Continuous Integration, Code Coverage, Empirical Study.

Entendendo a relação entre integração contínua e cobertura de código

Autor: José Diego Saraiva da Silva

Orientador: Uirá Kulesza

Coorientador: Daniel Alencar da Costa

Resumo

Integração Contínua, em inglês Continuous Integration (CI), é uma prática amplamente adotada na engenharia de software que enfatiza a integração frequente do software por meio de um processo de *builds* automatizado. Embora tenha sido demonstrado que a CI detecta erros mais cedo no ciclo de vida do software, a relação entre CI e cobertura de código ainda precisa ser esclarecida. Nosso trabalho tem como objetivo preencher essa lacuna investigando os aspectos quantitativos e qualitativos dessa relação.

No estudo quantitativo, comparamos 30 projetos com CI e 30 projetos que nunca adotaram CI (projetos NOCI) para investigar se a CI está associada a maiores taxas de cobertura de código. Analisamos 1.440 versões de diferentes projetos para identificar tendências na cobertura de código. Nossas descobertas revelam uma associação positiva entre a CI e maiores taxas de cobertura de código.

Nosso estudo qualitativo consistiu em um *survey* e uma análise de documentos. A pesquisa revelou várias descobertas significativas, incluindo uma associação positiva entre a integração contínua (CI) e maiores taxas de cobertura de código, indicando o valor da CI na promoção de práticas de teste. Ademais, nossa pesquisa enfatizou a relevância do uso de cobertura de código durante o processo de autoria e revisão, pois isso pode auxiliar na detecção precoce de possíveis problemas ao longo do ciclo de desenvolvimento.

A análise de documentos se concentrou em temas relacionados à cobertura nas discussões dos *Pull Requests* de projetos que adotam CI. A partir dessa análise, identificamos os principais tópicos associados ao uso da cobertura durante os *Pull Requests*, o que pode fornecer informações valiosas sobre como os desenvolvedores utilizam a cobertura para aprimorar a qualidade do código. Essas informações são capazes de orientar o desenvolvimento de melhores práticas para o uso da cobertura em projetos que adotam CI, contribuindo para aprimorar a qualidade e a confiabilidade dos produtos de software.

O nosso trabalho permitiu encontrar percepções sobre a evolução da cobertura de código em projetos que adotam CI, as quais podem auxiliar pesquisadores e profissionais a adotarem ferramentas e práticas para monitorar, manter e, inclusive, aprimorar a cobertura de código.

Palavras-chave: Teste de Software, Integração Contínua, Cobertura de Código, Estudo Empírico.

List of Figures

List of Tables

Contents

1 Introduction

The software industry constantly struggles with frequently changing requirements and the pressure to deliver products to market faster, which significantly strains the software development process. In this context, software testing is of utmost importance as it ensures that the software meets the required specifications and features, is reliable, and performs as intended. Although, in the short term, software testing may appear to increase time-to-market, as it adds extra practice in the development process. However, effective testing can ultimately reduce time-to-market by identifying and resolving issues early in the development process, reducing the risk of costly errors and later rework. Testing can also help ensure that the software meets customer requirements and expectations, increasing the chances of success in the market. In other words, investing in software testing upfront can save time and resources in the long run by preventing delays, improving quality, and reducing the need for rework and maintenance. Additionally, thorough testing can increase customer satisfaction and loyalty by providing a high-quality product that meets their expectations. Therefore, software testing is an activity intrinsically related to software development methodologies.

In the past, it was a common practice to integrate different software parts during development only after each part was fully implemented, often leading to isolated integrations. However, as projects become more complex and agile methodologies gain popularity promoting iterative and incremental development cycles with constant modifications, the need to increase the frequency of code integration between software parts has arisen, giving rise to the Continuous Integration practice.

Continuous Integration (CI) is an incremental software development technique that helps maintain the software in a working state by integrating changes more frequently. Its goal is to provide a systematic approach to software development, reducing risk and increasing software product quality (Felidré et al., 2019).

Despite being introduced as one of the *eXtreme Programming (XP)* practices by Beck K. (Beck and Gamma, 2000), Continuous Integration (CI) only gained widespread recognition and adoption after several years. Today, due to its promised benefits, CI has become a popular software development practice.

A key idea of CI is that the more often a project is integrated, the less likely the project will suffer from problems when in production (Hilton, Tunnell, et al., 2016). Also, as automated builds are performed at each push, errors can be detected and located earlier in the project's lifetime. As such, CI is intended to bring many benefits, such as risk reduction, improvement of project visibility and predictability (Duvall, Matyas, and Glover, 2007), greater confidence in the software product (Laukkanen, Paasivaara, and Arvonen, 2015), reducing code integration problems in a collaborative environment (Vasilescu, Schuylenburg, et al., 2014), and helping to maintain code quality. Therefore the stability of the code in the main branch repository is expected.

While Continuous Integration is a practice that can be implemented without any specific tools, some authors attribute its widespread adoption to modern tools and services that have significantly automated the steps necessary for compiling, building, and testing software (Felidré et al., 2019). The availability of such tools removes many of the barriers developers may face when adopting CI, making integrating it into their workflow easier.

In recent years, researchers have conducted empirical studies to confirm the benefits of Continuous Integration (CI) and determine whether it lives up to its reputation

(Hilton, Tunnell, et al., 2016). The research scope includes various aspects, such as practices, environments, tools, potential benefits and problems, fundamental concepts, motivation, and others (Hilton, Tunnell, et al., 2016; Pinto et al., 2018; Ståhl and Bosch, 2013; Soares et al., 2022).

Vasilescu et al. (Vasilescu, Yu, et al., 2015) studied the correlation between CI, software quality, and productivity. The authors found that CI correlates with positive quality outcomes and improves the productivity of project teams. Hilton et al. (Hilton, Nelson, et al., 2017) studied the usage of CI in open-source projects and concluded that projects that use CI tend to release more frequently than projects that do not use CI. In a complementary study, Hilton et al. (Hilton, Tunnell, et al., 2016) report that CI projects are more likely to catch software defects earlier. Vassallo et al. (Vassallo et al., 2016a) found evidence that CI could be associated with time-to-market reduction. However, Bernardo et al. (Bernardo, Costa, and Kulesza, 2018) observed that CI may not always reduce the time-to-market for delivering new functionalities to end users. Guo & Leitner (Guo and Leitner, 2019) conducted a conceptual replication of the work by Bernardo et al. (Bernardo, Costa, and Kulesza, 2018), confirming their results through the use of the Discontinuous Regression Design (RDD) statistical model. On the other hand, the authors observed that CI is associated with the delivery of more functionalities per software release.

Automated tests play an essential role in CI (Fowler and Foemmel, 2006) as they represent a fundamental part of the build automation process to detect errors as quickly as possible. Some researchers consider that, without automated tests, a project should not be deemed as adopting CI (Duvall, Matyas, and Glover, 2007). Writing tests assists developers in revealing faults embedded in the software. As such, the effectiveness of a test suite directly impacts the quality of a software project (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021). The effectiveness of a test suite can be partially measured by code coverage (i.e., code coverage is a measure of the degree to which a test suite exercises a software system (Ivanković et al., 2019). Measuring code coverage can benefit the development workflow by offering an objective, industry-standard metric for test quality (Hora, 2021).

1.1 Problem Statement

Continuous Integration (CI) practices have made valuable contributions to software development culture, but many assumptions in the community remain unexplored empirically. Although some practices are adopted based on the expectation of positive outcomes, there is still a vague idea of the trade-offs and actual impact on software process quality.

Soares et al. (Soares et al., 2022) have presented some studies that provide evidence to support the association between CI and good test practices, specifically concerning the number of tests and code coverage level. However, their findings were based primarily on surveys (73.91%), and none of the studies they reviewed explicitly addressed how code coverage changes over time after CI adoption. This suggests a need for further investigation into the relationship between CI and code coverage over an extended period.

The literature often links CI with higher code coverage (Humble and Farley, 2010; Felidré et al., 2019), likely because of the emphasis on automated testing. While this assumption is intuitive, it remains unclear whether adopting CI results in a sustained improvement in code coverage over time. So far, the research works that have explored the relationship between CI and good test practices, particularly in terms of the number of tests and test coverage, have predominantly relied on surveys, accounting for 73.91% of the studies, as noted by Soares et al. (Soares et al., 2022). However, these studies have not directly addressed the question at hand. This highlights the importance of conducting additional empirical investigations to examine the long-term impact of CI adoption on code coverage and other aspects of test practices.

The present thesis is an effort to reduce the lack of empirical investigation into the relationship between code coverage and the adoption of CI.

1.2 Current Research Limitations

The work by Nery et al. (Nery, Costa, and Kulesza, 2019) investigated the relationship between CI and test ratio. They compared projects that eventually adopted CI (CI projects) and projects that never adopted CI (NOCI projects). The study results suggest that adopting CI may positively affect the test ratio. However, test ratio is a global measure of the proportion of test code in a project and does not necessarily capture the amount of code exercised by the tests. In their work, they performed a preliminary study on the association between CI and code coverage. The authors analyzed a subset of projects (10 CI projects and 10 NOCI projects), where their preliminary results reveal a potential positive association between CI and code coverage. However, their study about the relationship between code coverage and CI was designed with a relatively small number of projects and builds, making it unclear whether their results can be generalized.

Hilton et al. (Hilton, Bell, and Marinov, 2018) conducted a comprehensive analysis of code coverage evolution over a large number of builds for multiple projects. However, their primary focus was on evaluating the impact of patches on the quality of a project's test suite.

Therefore, the authors investigated the coverage of individual patches, i.e., the changed statements within the system under test (SUT), instead of assessing the overall coverage of the SUT. It's worth noting that the study did not specifically examine the integration of code coverage analysis with Continuous Integration (CI) practices, as the authors' main objective was to evaluate the effect of patches on the test suite's quality.

Indeed, the investigation of the impact of CI on code coverage remains an open question.

1.3 Thesis Proposal

The central research question of this thesis is: *What is the relationship between the adoption of continuous integration and the evolution of code coverage?*

The thesis proposes to analyze the the relationship between code coverage and continuous integration from both quantitative and qualitative perspectives. The goal of the quantitative study (**Study 1**), detailed in Section 1.3, is to compare code coverage rates between continuous integration projects and non-continuous integration projects (NOCI) and identify trends in code coverage over time. The goal of the two-part qualitative studies is to investigate how developers use code coverage in continuous integration projects. The first part, in the **Study 2**, described in Section 1.3, uses a survey approach to gather developers' data. Next, in the second part, the **Study 3**, outlined in Section 1.3, uses document analysis to examine the use of code coverage in continuous integration projects.

The subsections below provide further details on the motivation and methodology of each study.

Study 1 — The Impact of the Adoption of Continuous Integration on Code Coverage

Our quantitative study investigates whether CI is associated with an improvement in code coverage. The study, which is detailed in Chapter 3, compares the code coverage rates of 30 projects with continuous integration (CI) and 30 projects without CI (NOCI). A total of 1,440 versions were analyzed to observe trends in code coverage related to CI. The results indicate that CI is associated with an improvement in code coverage.

Specifically, more projects with rising trends in code coverage were observed in CI projects compared to NOCI projects, and CI projects tend to stabilize at a higher code coverage rate than NOCI projects. The statistical evidence also supports the positive association between CI and a higher code coverage rate. Considering the CI dataset, the coverage improvement after CI adoption is estimated to be approximately 10%, while it is negligible in the NOCI dataset.

Study 2 — Uncovering the Relationship Between Continuous Integration and Code Coverage: An Exploratory Investigation

This study is the first part of a two-part qualitative investigation aimed at addressing the research gap in understanding the underlying mechanisms that drive the relationship between continuous integration (CI) and code coverage. The main objective of this study is to gain a comprehensive understanding of the impact of CI on code coverage through a survey of developers, providing additional qualitative insights and examining how developers utilize code coverage in CI projects. Furthermore, this study seeks to enforce previous quantitative findings by comparing them with developers' perspectives.

Study 3 — Investigating Discussions on Code Coverage in CI-Enabled Projects: An Exploratory Document Analysis

This study is the second part of our qualitative investigation into the relationship between continuous integration (CI) and code coverage. Our primary objective is to investigate the connection between code coverage and continuous integration (CI) using document analysis technique. After establishing a positive relationship between code coverage and CI, we aim to explore the specific aspects of code coverage that developers discuss during pull requests. To accomplish this, we conducted a comprehensive document analysis of 30 CI projects, employing a two-step approach involving inductive and deductive analysis techniques.

1.4 Thesis Contributions

We outline the contributions of this thesis below. The contributions are grouped by their respective study.

Study 1 — The Impact of the Adoption of Continuous Integration on Code Coverage

- Our study found compelling evidence for the positive relationship between continuous integration (CI) and code coverage. Specifically, we observed that 50% of CI projects exhibited increasing trends in code coverage, while only 10% of non-CI (NOCI) projects showed similar trends.

- The data analysis conducted in our study indicates a statistically significant growth in code coverage following the adoption of CI. These findings support the notion that implementing CI can effectively improve code quality and test coverage. Moreover, our research has revealed a positive correlation between code coverage and the implementation of CI, further reinforcing this relationship.

- Interestingly, our study also found that projects that eventually adopt CI tend to have a significantly higher number of code changes that lead to increased coverage, a consequence of the positive correlation between code coverage and CI implementation. This suggests that the decision to implement CI may be motivated by a desire to improve testing practices and increase code coverage.

Study 2 — Uncovering the Relationship Between Continuous Integration and Code Coverage: An Exploratory Investigation

- Our study found that developers use code coverage information during both the code authoring stage and code reviews. They find it useful for identifying trivial bugs, detecting uncovered code paths, and improving software quality by potentially preventing bugs and controlling side effects when refactoring or changing code behavior.
- Participants attributed the increase in code coverage to the adoption of CI, but we wanted to ensure that this was not due to other testing practices. After asking participants if they adopted any new practices after adopting CI, we found that all practices mentioned fell under the umbrella of CI practices. This supports the association between CI and coverage. Additionally, participants associated coverage fluctuations with the proper use of CI practices.

Study 3 — Investigating Discussions on Code Coverage in CI-Enabled Projects: An Exploratory Document Analysis

- Our findings suggest that there may be projects that monitor their code coverage, as evidenced by the existence of automated coverage reports. This finding is in line with the monitoring practices that developers reported in our qualitative study.
- Developers face challenges when integrating coverage tools with CI, with compatibility issues between local and CI server environments being a critical factor in coverage within CI environments. Study 1 - RQ3 (Section 3.3.3) findings support this, as integration problems were identified as a major reason for decreased coverage, and multiple mentions of compatibility issues were observed.

- In Study 1 - RQ3 (Section 3.3.3), we did not find any clear evidence of prioritizing coverage for newly added code over preexisting uncovered code. While some participants mentioned the significance of covering new lines, the frequency of such comments was not substantial enough to suggest a systematic trend towards prioritizing new code coverage. Based on our findings, the coverage of new code does not appear to be a primary concern in the projects surveyed.

- The coverage maintenance theme is the second most common topic, making up 22.87% of all comments. This aligns with the results from both the quantitative study (Chapter 3) and survey (Chapter 4), as it reflects the efforts made by developers to increase coverage. The reduction in discussions related to coverage debt is also expected, as our RQ1 findings (Section 3.3.1) suggest that CI projects tend to stabilize at higher coverage levels.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide the background material to the reader. In Chapter 3 presents our quantitative study to investigate the evolution of code coverage and its relationship with the adoption of continuous integration. Chapter 4 and Chapter 5 present the two-part quantitative study on how developers use code coverage in CI projects. In Chapter 6, we present related research with respect to this thesis. Finally, Chapter 7 draws conclusions.

2 Background

In this chapter, we describe the key concepts that are necessary to understand the studies that are performed in this thesis.

2.1 Software Testing

Software testing is a fundamental discipline within software engineering that seeks to enhance the quality and reliability of software systems by detecting software faults and ensuring that the system functions as intended without errors or non-conformities, thereby fulfilling the customer's requirements (Mahdieh et al., 2020). This role is critical in the software development lifecycle, and the effectiveness of testing is paramount (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021). According to the academic literature, fixing a bug becomes increasingly costly throughout the software development life cycle (Boehm and Papaccio, 1988), and discovering all faults in a program is essentially unviable. As a result, testing considers only a limited number of possible program states, meaning that there is a possibility of existing faults that the test suite cannot discover. Therefore, the focus of software testing is on eliminating as many defects in the software system as soon as possible, which adds value to the software development process by reducing repair costs.

Consequently, testing software always involves a trade-off between the cost of (further) testing and the potential cost of undiscovered faults in a program (Gopinath, Jensen, and Groce, 2014). In this context, the quality of the test suite is crucial in measuring the effectiveness of testing. The effectiveness of the test suite concerns its ability to uncover faults, making it a mandatory requirement for the software testing discipline to achieve its goals. As a result, software developers aim to write high-quality tests to reveal faults embedded in the software and estimate the overall quality of the system under test (SUT).

The fault detection measure is a perfect method for measuring the effectiveness of the test suite. However, collecting faults and analyzing failures requires substantial effort, making it an unrealistic method for measuring test suite effectiveness in a practical software development cycle. Instead, software developers use methods that predict fault detection capability based only on the test suite itself and the current version of the SUT (Gopinath, Jensen, and Groce, 2014).

One well-established metric to measure the effectiveness of a test suite is code coverage, which is the most popular metric used to compare test suites' adequacy (Ahmed et al., 2016). Numerous organizations set testing requirements in terms of coverage levels (Gligoric et al., 2013).

Code coverage is a metric that estimates the proportion of the execution paths within the code under test that the test suite exercises. Due to its simplicity and computational efficiency, code coverage has become a popular metric in both industry and research for evaluating the effectiveness of software testing. By measuring the code coverage achieved by the test suite, developers can identify untested or under-tested areas of the code, which can help to reveal potential faults and improve the overall quality of the software. Nevertheless, its effectiveness in improving testing in practice remains a matter of debates (Inozemtseva and Holmes, 2014; Ivanković et al., 2019).

The proliferation of practices such as continuous integration and test-driven development has promoted testing to new prominence, proclaiming to help developers achieve high code coverage. Thus, having a comprehensive test suite is essential to these practices.

2.2 Code Coverage

Code coverage is a metric that expresses which proportion of application code of a software project is executed when running all test cases. Code coverage is a widely-used metric in software testing that measures which proportion of application code of a software project is executed when running all test cases (Zhu, Hall, and May, 1997). By using code coverage-based metrics, software developers can evaluate the effectiveness of their test suites and identify untested or poorly-tested areas of the code. This information can then be used to improve the testing process and increase the reliability and quality of the software.

However, this loose definition allows for the implementation of coverage at different levels, objectives and measure criterion. Coverage measure criteria provide a set of requirements for the code under test and measure how many of these requirements are satisfied by a given test suite. As a result, we found a variety of code coverage-based metrics in the literature.

Structural coverage metrics, which explore code structure to determine how much of the code is exercised by the test suite, are among the most popular coverage-based metrics in software engineering. These metrics provide a way to measure the extent to which code has been executed, and several of them have been proposed in the literature. In the following paragraphs, we briefly discuss some of the most common structural coverage metrics and their applications.

Statement coverage is a coverage metric used to determine the proportion of executable statements that are executed at least once during testing (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021). It is calculated as the number of executed statements divided by the total number of statements. Statement coverage is the most widely used coverage metric.

Line coverage is a metric that quantifies the proportion of executable lines of code in a program that have been executed at least once during testing (Someoliayi et al., 2019). To compute line coverage, each executable line is assigned a coverage point value of either 0 or 1, depending on whether it has been exercised or not. The total coverage is then calculated as the ratio of coverage points of all exercised lines to the total number of executable lines.

Branch coverage measures the proportion of conditional branches that are exercised by a test suite (Homès, 2013). Branch coverage is equivalent to edge coverage for all conditional edges in the control flow graph (Zhu, Hall, and May, 1997), and it subsumes statement coverage once all branches are examined.

Decision coverage measures the proportion of decisions in the program that are executed by its test suite (Inozemtseva and Holmes, 2014). A decision is a compound of one or more conditions, and a 100% decision coverage requires that each decision is evaluated to both **true** and **false**. Decision coverage is slightly more difficult to satisfy and measure than statement coverage.

Condition coverage measures whether each condition in a decision is evaluated to both possible outcomes at least once (Kelly J. et al., 2001). It does not require that the decision take on all possible outcomes at least once, and therefore a 100% condition coverage does not guarantee a 100% decision coverage.

Modified condition/decision coverage (MC/DC) measures whether every condition within a decision in the program has taken on all possible outcomes at least once, each condition has been shown to independently affect the outcome of the decision, and that each entry and exit point have been traversed at least once (Rajan, Whalen, and Heimdahl, 2008). Achieving MC/DC requires, in general, a minimum of n+1 test cases for a decision with n inputs.

Recently, novel coverage-based criteria have been proposed to evaluate and compare the effectiveness of test suites. In the following paragraphs, we describe some of them.

Vanoverberghe et al. (Vanoverberghe et al., 2011) introduced state coverage as a new metric to measure the ratio of state updates that are read by assertions to the total number of state updates. Additionally, the authors presented efficient algorithms for measuring state coverage and reported that it helped to identify multiple unchecked properties and detect several bugs in their preliminary evaluation.

Someoliayi et al. (Someoliayi et al., 2019) proposed a novel test coverage metric called *Program State Coverage (PSC)*, which considers the number of distinct program states in which each line is executed. The only difference between PSC and line coverage is the method of assigning a coverage point to each line. PSC assigns a real number between 0 and 1 to each line based on the number of distinct program states in which that line is executed. The authors reported that their approach outperformed the traditional line coverage metric.

Aghamohammadi et al. (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021) presented a new coverage metric called *Statement Frequency Coverage* that subsumes *Statement Coverage*. Statement frequency coverage aims to address the limitations of branch and statement coverage as they do not consider the executed statement frequency. Thus, Statement Frequency Coverage assigns a continuous value to a statement that is proportional to the number of times it is executed during test execution. The authors reported that their coverage metric outperformed the existing code coverage metrics, namely statement coverage and branch coverage.

2.3 Continuous integration

Software systems are highly intricate, consisting of numerous interdependent components that require careful management. Even a minor change in one file can result in unintended consequences that compromise the entire system's accuracy and stability. As a result, integrating software, which involves combining all developers' working copies into a shared mainline, poses significant challenges that increase in complexity with the size of the system and team.

Consequently, a significant portion of software development time is often spent with the system in an unusable state (Humble and Farley, 2010). Developers typically only attempt to build the entire application after completing their work, and the process of merging multiple active branches into the mainline can be painful, particularly as the number of branches and their duration of existence increase. This integration process is critical to the system's overall success, and any errors or complications during the process can significantly impact the final product.

CI advocates for the use of an automated build process with each push, enabling early error detection and location in the project's life cycle. By doing so, CI is expected to bring numerous benefits, including risk reduction, improved project visibility and predictability (Duvall, Matyas, and Glover, 2007), increased confidence in the software product (Laukkanen, Paasivaara, and Arvonen, 2015), reduced code integration problems in collaborative environments (Vasilescu, Schuylenburg, et al., 2014), and maintenance of code quality. Therefore, stability in the code of the main branch repository is expected. Given its promised benefits, CI is widely used in software development practice today.

The practice of continuously integrating code changes dates back to the 1970s (Brooks Jr, 1995), but it has only gained widespread adoption in recent decades thanks to the popularization of various CI services, such as TravisCI[1] CircleCI[2], and Jenkins[3]. Although Continuous Integration can be implemented without this specific tooling, these tools provide the underlying infrastructure to automate software compilation, building, and testing, significantly lowering the barriers to CI adoption. CI services have become a cornerstone of modern software development practices, employed in both commercial and open-source projects alike (Felidré et al., 2019).

The adoption of continuous integration has been increasing among software development projects (Hilton, Tunnell, et al., 2016). Today continuous integration is a critical aspect of software development, serving as the centerpiece for ensuring software health and quality (Duvall, Matyas, and Glover, 2007).

2.4 Regression Discontinuity Design

The Regression Discontinuity Design (RDD) is a quasi-experimental research design that is well-suited for examining the impact of an intervention based on a cutoff score or threshold on a outcome continuous variable (Hyman, 1982). RDD involves the assignment of treatment based on whether the participant's observed covariate value is above or below a known threshold or cutoff point. The discontinuous jump in the probability of treatment assignment at the threshold creates a natural "quasi-experiment" that can be used to estimate the average treatment effect. By comparing outcomes closely on either side of the threshold, RDD provides an estimate of the causal effect of the intervention that is robust to potential confounding factors. The fundamental premise of RDD is that if an intervention has no effect on the outcome, there will be no discontinuity, and the outcome will remain continuous over time.

In other words, RDD divides the sample into two groups based on the value of

[1]https://www.travis-ci.com
[2]https://circleci.com
[3]https://www.jenkins.io

the threshold variable, with those above the threshold assigned to the treatment group and those below the threshold assigned to the control group. The basic idea behind RDD is that units on either side of the threshold are similar in all relevant respects except for their treatment assignment. Therefore, any outcome difference between the treatment and control groups can be attributed to the treatment effect rather than to differences in baseline characteristics or other confounding factors.

The relationship between an assignment variable and an outcome in RDD is depicted in Figure 2.1. When an actual effect of the intervention is present, a discontinuous change or jump in the outcome value is observed when the assignment variable exceeds the threshold. This abrupt change in the outcome corresponds to the effect of the treatment or policy being analyzed.

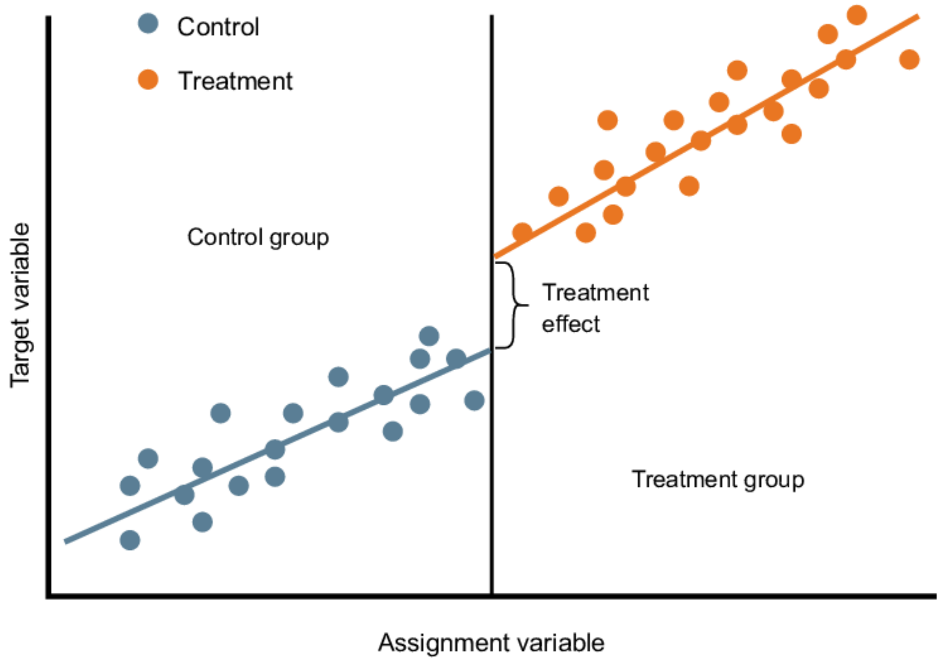

Figure 2.1: Relationship between assignment variable and outcome in a regression discontinuity design.

RDD makes several key assumptions that are essential for valid causal inference. Firstly, there must be a clear and deterministic rule for assignment to treatment based on a score variable, which is fully known in advance. This score variable should not be related to any other confounding variables that may affect the outcome. If there is no misassignment, the design is called sharp. On the other hand, if there are some misassignments or treatment crossovers, the design is called fuzzy.

Therefore, in the *sharp RDD*, the threshold variable is used to divide the sample precisely into the treatment and control groups. Specifically, all units with assignment variable values above a certain threshold value are assigned to the treatment group, while all units with values below the threshold are assigned to the control group. This sharp separation creates a discontinuity in the probability of treatment at the threshold, which enables the estimation of causal effects by comparing the outcomes of units on either side of the threshold. In contrast, the *fuzzy RDD* involves a threshold variable that influences the probability of being assigned to the treatment group. Consequently, the separation between the treatment and control groups is less precise than in the sharp RDD. However, the fuzzy RDD also allows for the estimation of causal effects by exploiting the discontinuity in the probability of treatment at the threshold.

The second assumption is there should be no manipulation of the score variable by individuals, as this can lead to bias in the results. The third assumption is that the assignment of units to the treatment or control group should be as good as random at the threshold point. Finally, the treatment and control groups should be comparable in terms of observable characteristics, both before and after the threshold. If these assumptions are met, then RDD can provide unbiased estimates of causal effects at the threshold point.

There are two widely used methods for estimating treatment effects in RDD: non-parametric and parametric approaches. The difference between non-parametric and parametric RDD estimation approaches lies in the assumptions made about the functional form of the relationship between the running variable and the outcome variable.

In parametric RDD estimation assumes a specific functional form, typically a polynomial regression, to model the relationship between the running variable and the outcome variable. This method is often used when the relationship between the running variable and the outcome variable is well-known or when the data conform to a specific functional form. In this case, a parametric model may have greater statistical power to detect the treatment effect. The statistical model for parametric RDD using polynomial regression can be represented as:

$$Y_i = \alpha + \beta_1 S_i + \beta_2 S_i^2 + ... + \beta_i S_i^k + \gamma T_i + \epsilon_i \tag{2.1}$$

where Y_i is the outcome variable for unit i, S_i is the score variable used for assignment, T_i is a binary variable indicating treatment status (1 for treatment, 0 for control), α is the intercept, γ is the treatment effect, β_1 through β_k are the coefficients for the polynomial terms of S_i, and ϵ_i is the error term. The polynomial degree k is chosen based on the data and the degree of nonlinearity in the relationship between S_i and Y_i.

On the other hand, non-parametric RDD refers to an estimation approach that does not assume any specific functional form for the relationship between the score variable and the outcome of interest. Instead, it estimates the treatment effect by fitting a flexible function to the data using techniques such as local linear regression or other non-parametric regression methods (Lee and Lemieux, 2010). The most common non-parametric method used in the RDD context is a local linear regression. This is of the form:

$$Y_i = \alpha + \beta W_i + \gamma (W_i - c) + \epsilon_i \tag{2.2}$$

where Y_i is the outcome variable for the i-th observation, W_i is the assignment variable (e.g., test score), c is the threshold value, α is the intercept, β is the effect of W_i on Y_i away from the threshold, γ is the effect of the distance between W_i and c on Y_i, and ϵ_i is the error term.

Non-parametric methods in RDD offer a significant advantage by providing estimates based on data closer to the cutoff, which is intuitive and reduces bias that can arise from using data farther away from the cutoff to estimate the discontinuity. Local linear regressions are commonly preferred because they have better bias properties and convergence. Nonetheless, using both parametric and non-parametric methods, if feasible, can be a useful way to validate the estimated results and reduce reliance on a specific approach (Lee and Lemieux, 2010).

2.5 Document Analysis

Document analysis is a systematic procedure for reviewing or evaluating any type of document, including written texts (printed and electronic), images, audio recordings, or video footage (Bowen, 2009).

It involves a range of methods for analyzing documents, including close reading, textual analysis, and discourse analysis, with the goal of extracting insights, identifying patterns, or gaining a deeper understanding of the document(s) being analyzed. Document analysis can be used in various fields, including social sciences, humanities, law, and business.

Document analysis is often used in combination with other qualitative research methods such thematic analysis (see Section 2.6), means of triangulation and discourse analysis.

Mixed-methods studies, which combine both quantitative and qualitative research techniques, can use document analysis to enhance the comprehensiveness of data analysis and interpretation.

Document analysis is a common qualitative research method that is sometimes used in mixed-methods studies to provide a more comprehensive analysis and interpretation of the data. Mixed-methods studies are research designs that combine both quantitative and qualitative research techniques. By analyzing qualitative data (surveys, surveys, open-ended, semi-structured interviews with reviews of documents), researchers can gain a deeper understanding of the topic being studied

Document analysis is a valuable qualitative research method that has advantages over other research methods. One of its key advantages is that it is an efficient method that requires data selection instead of data collection, making it less time-consuming than other methods. Additionally, it is a cost-effective method that is often the method of choice when collecting new data is not feasible. Another advantage is that documents are 'unobtrusive' and 'non-reactive', meaning that they are unaffected by the research process and do not require reflexivity, which is inherent in other qualitative research methods. Furthermore, documents are stable, making them suitable for repeated reviews, and they provide exact names, references, and details of events, enhancing the precision of the research process. Finally, documents provide broad coverage, spanning a long time period and many events and settings, making them a valuable tool for researchers seeking a comprehensive understanding of complex social phenomena (Merriam, 1988; Yin, 1994; Bowen, 2009).

Despite the advantages of document analysis, there are also some inherent limitations that researchers should consider. One limitation is that documents may lack sufficient detail since they are produced for a purpose other than research and are independent of a research agenda. Another limitation is that document retrievability can be low or difficult, and access to documents may be deliberately blocked. Furthermore, an incomplete collection of documents may suggest biased selectivity, which can occur in an organizational context where available documents may align with corporate policies and procedures or reflect the emphasis of a particular unit that handles record-keeping, such as Human Resources. As a result, researchers should be cautious in using document analysis as a standalone research method and should consider complementing it with other qualitative or quantitative research methods to address these limitations (Merriam, 1988; Yin, 1994; Bowen, 2009).

Document analysis involves skimming (superficial examination), reading (thorough examination), and interpretation. This approach combines elements of both content analysis (Neuendorf, 2016) and thematic analysis(See Section 2.6). Content analysis involves organizing information into categories related to the central questions of the research, while thematic analysis involves identifying patterns, themes, and meanings within the data.

2.6 Thematic Analysis

Thematic Analysis (TA) is a qualitative research method used to identify, analyze, and report patterns (themes) in qualitative data (i.e., audio, text, or video). Therefore, Thematic Analysis focuses on identifying and describing implicit and explicit ideas within the data, namely, themes. This technique moves beyond counting explicit words or phrases; interpreting and analyzing the themes to gain insights into the research question (Braun and Clarke, 2006).

Thematic Analysis is a family of methods (Fugard and Potts, 2020). Although the thematic analysis methods have some characteristics in common, they also have significant divergences in underlying research values, the conceptualization of core constructs, and analytical procedures. The literature categorizes the thematic analysis methods into three broad approaches (Braun and Clarke, 2019): Coding reliability approaches, Reflexive approaches, and Codebook approaches.

2.6.1 Coding Reliability Thematic Analysis

Coding reliability Thematic Analysis (coding reliability TA) approaches involve early theme development and conceptualize coding as a process of identifying evidence for themes (Guest, MacQueen, and Namey, 2011). According to Boyatzis "Coding Reliability Thematic Analysis is a translator of those speaking the language of qualitative analysis and those speaking the language of quantitative analysis" (Boyatzis, 1998). Qualitative data are gathered and examined using qualitative techniques of coding and theme development; the data are reported qualitatively as themes.

The coding process involves breaking down the data into smaller units of meaning, such as phrases, sentences, or paragraphs, and then labeling or categorizing these units based on commonalities or patterns. In coding reliability TA, themes are typically conceptualized as domain/topic summaries (often derived from data collection questions) (Braun, Clarke, et al., 2018), analytic inputs, and outputs – they drive the coding process and are the output of the coding process.

The coding is an identification process driven by a codebook/coding frame, which usually contains a list of codes/themes – each has a label/name, a definition, information on how to identify the code/theme, a description of any exclusions or qualifications to identifying the code/theme, and even examples (Boyatzis, 1998). The coding process typically demands multiple coders working independently to apply the codebook to the data, reaching reliability and replicability.

The final coding is determined through consensus between coders. So any bias related to coders is managed by measuring the level of agreement between coders. The coding reliability TA assumes that a high level of agreement equals reliable coding.

There are several coding reliability approaches that researchers can use, depending on the research question, data type, and analytical approach (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021; Braun and Clarke, 2021).

Some common approaches include:

1. **Inter-coder reliability**: This approach involves having multiple coders independently code the same data and comparing their codes to assess the consistency of coding. Inter-coder reliability can be assessed using various measures, such as Cohen's kappa, Holsti's method, Scott's pi, Krippendorff's alpha, percent agreement, or intra-class correlation coefficient.

2. **Intra-coder reliability**: This approach involves having the same coder code the same data multiple times and comparing their codes to assess the consistency of coding. In contrast to inter coder reliability, intra-coder reliability measures the consistency of coding within a single researcher's coding. Intra-coder reliability can be assessed using various measures, such as test-retest reliability or split-half reliability.

3. **Consistency coding**: This approach involves having the same coder code a subset of the data at two different points in time and comparing their codes to assess the consistency of coding. This approach is builds toward a singular, shared, and "correct" analysis of the data. This approach involves having the same coder code a subset of the data at two different points in time and comparing their codes to assess the coding consistency. This approach is built toward a singular, shared, and "correct" data analysis.

2.6.2 Reflexive Thematic Analysis

Reflexive Thematic Analysis (Reflexive TA) (Braun and Clarke, 2019) is a variation of the thematic analysis method that emphasizes the researcher's subjective interpretation of the data. This approach acknowledges that the researcher's beliefs, values, and experiences shape how they interpret and analyze the data, encouraging researchers to reflect on their positionality throughout the research process. Thus, in reflexive thematic analysis, the researcher engages in an iterative data analysis and reflection process that incorporates their experiences and perspectives into the analysis.

Although the more traditional thematic analysis approaches prioritizes the objectivity of the research process, the reflexive thematic analysis approach emphasizes the researcher's reflexivity and subjectivity.

Themes do not arise spontaneously from the data or coding; they are not inherent in the data, waiting to be identified and retrieved by the researcher. Instead, themes are creative and interpretive stories about the data, resulting from the interaction of the researcher's theoretical assumptions, analytical abilities, and the data itself (Terry et al., 2017).

In reflexive TA, the theme process is about identifying meaning-based patterns rather than data features.

According Braun and Clarke (2019), the theoretical approach to analysis in RTA can be agnostic, meaning that it is not tied to a specific theoretical framework or approach. However, they suggest that RTA can be used with a range of approaches, including inductive/deductive, semantic/latent, critical realist/constructionist, or a mix of those.

1. **Deductive/Inductive**: The deductive/inductive approach involves the researchers using their theoretical framework and hypotheses to inform the development of codes. The inductive approach involves building generalizations from specific observations, where the researcher starts with the data and generates themes and concepts based on emerging patterns and trends. On the other hand, the deductive approach starts with a theoretical framework or pre-existing concepts, and the researcher looks for evidence in the data to support or refute these preconceptions.the choice between inductive and deductive approaches may depend on the research question and the

available data.

2. **Semantic/latent**: In semantic/latent approach to reflexive thematic analysis, the researcher analyzes the surface-level meaning of the data (semantic) as well as the underlying or implicit meanings (latent). The semantic approach focuses solely on the explicit or obvious meanings conveyed by the data (i.e. what participants say). The latent approach involves exploring the ideas, assumptions, and conceptualizations that underpin the data, and how they may influence participants' perspectives and experiences (Clarke and Braun, 2021). This approach allows for a more nuanced and comprehensive understanding of the data, and can reveal deeper insights that may not be immediately apparent at the semantic level.

3. **Critical realist/constructionist**: The realist/constructionist approach to RTA acknowledges the co-construction of reality between the researcher and the participants. This approach recognizes that knowledge is constructed through social interaction and language use and that multiple perspectives and experiences contribute to creating meaning. In the realist approach, researchers make claims about the objectivity of people's experiences as they report them (realist approach). Alternatively, they may adopt a constructionist approach, focusing on how people perceive a situation (constructionist approach).

2.6.3 Codebook Thematic Analysis

The Codebook approach to Thematic Analysis (*Codebook TA*) is positioned between the coding reliability and reflexive approaches, as it incorporates the structured coding process of coding reliability TA (although without relying on coding reliability measures), while also aligning with the qualitative philosophy of reflexive TA (Braun, Clarke, et al., 2018). This method involves pre-determining some, if not all, of the themes before conducting the full analysis and conceptualizing the themes as domain summaries. The initial them could be derived from data research questions.

The aforementioned approach is particularly appropriate for describing and summarizing qualitative data, along with participants' perspectives regarding a certain topic or technology. These codebooks can be presented as a template (Brooks et al., 2015), thereby providing a structure for an article, or as a framework (J. Smith and Firth, 2011) that emphasizes displaying each stage of the analysis, or as a matrix. It does not measure (inter)reliability. There are two main types of codebook approaches:

1. **Deductive codebook approach**: In this approach, the researcher creates a set of codes based on existing theories or previous research in the area. The codes are pre-determined before the analysis of the data begins.

2. **Inductive codebook approach**: In this approach, the researcher does not pre-determine the codes, but instead develops them through a close reading and analysis of the data. The codes are generated from the data itself, rather than being imposed on the data from a pre-existing theoretical framework.

The codebook approach has clear pragmatic advantages. The codebook provides a structured and systematic way to analyze qualitative data, allowing for efficient and consistent analysis across multiple researchers or studies. It also allows for easy comparison and synthesis of findings across studies that use the same codebook. Additionally, the use of a codebook can provide a clear and transparent method for reporting results and can help ensure that all relevant aspects of the data are captured and analyzed.

The codebook approach offers evident pragmatic benefits as it furnishes a structured and systematic technique to examine qualitative data, ensuring reliable and consistent analysis across various researchers or studies (Braun, Clarke, et al., 2018). Moreover, it facilitates easy comparison and synthesis of results across studies utilizing the same codebook. Furthermore, the utilization of a codebook can provide a lucid and transparent procedure for reporting findings. It can aid in guaranteeing that all significant aspects of the data are captured and analyzed.

3 The Impact of the Adoption of Continuous Integration on Code Coverage

Earlier versions of the work in this chapter appears in the proceedings of the International Conference on Mining Software Repositories (MSR 2023) (Silva et al., 2023)

In this chapter, we present a quantitative study that aims to investigate the correlation between CI and code coverage. We analyze a total of 1,440 versions to identify trends in code coverage associated with CI by comparing the code coverage rates of 30 projects with continuous integration (CI) and 30 projects without CI (NOCI). The results demonstrate a positive relationship between CI and the improvement of code coverage. Notably, CI projects exhibit a higher frequency of increasing code coverage trends compared to NOCI projects, and they tend to maintain a consistently higher code coverage rate. Moreover, the statistical evidence provides further support for the association between CI and the attainment of a higher code coverage rate.

Chapter organization. The subsequent sections of this chapter are structured as follows: Section 3.1 introduces the research question that serves as the central focus and primary objective of our investigation. In Section 3.2, we present the study design, followed by Section 3.3, which presents the findings of our empirical study. The principal implications of our investigation are deliberated upon in Section 3.4, while Section 3.5 highlights the potential threats to its validity. Finally, Section 3.6 provides a comprehensive analysis and draws insightful conclusions based on our study.

3.1 Research Questions

In particular, we address the following research questions:

- **RQ1 - What are the evolution trends of code coverage within CI and NOCI Projects?**
  Rationale: This research question aims to gain insights into the patterns and trends of code coverage over time in projects with continuous integration (CI) and projects without CI (NOCI). By examining the evolution of code coverage within these two groups, we can identify any significant differences in how code coverage changes over time between CI and NOCI projects. Understanding these trends can provide valuable information about the impact of CI on code coverage and shed light on the effectiveness of CI in improving code coverage over the course of a project.

- **RQ2 - Is there a significant correlation between CI and code coverage values?**
  Rationale: In RQ1, we examine the distinct patterns of code coverage trends between CI projects and NOCI projects, which demonstrate a noticeable difference. Building upon these findings, RQ2 delves deeper into a quantitative analysis of the code coverage within our studied projects. The objective is to investigate whether the adoption of CI is significantly correlated with improved code coverage. Establishing a positive association between CI and code coverage values would provide evidence

of the effectiveness of CI in positively influencing code coverage outcomes.

- **RQ3 - What types of code changes affect the code coverage of CI and NOCI projects?**
  Rationale: The rationale behind this research question is delve deeper into understanding the dynamics of code coverage evolution, RQ3 explores how different types of code changes impact code coverage in projects with continuous integration (CI) and projects without CI (NOCI).

3.2 Study Setup

In this section, we explain the design of our study. We describe how we select the projects and collect the data for our analyses.

3.2.1 Subject Projects

To perform our investigations, we leverage the dataset made available by Nery et al. (Nery, Costa, and Kulesza, 2019).

As mentioned before, our objective is to investigate the possible association between the adoption of CI and the evolution of code coverage. This target demands available of two different sets of study objects: (i) projects that have adopted CI at some point in their lifetime (CI projects) and (ii) projects that, to the best of our knowledge, never adopted CI have never adopted CI during their lifetime (NOCI projects).

We use their dataset because it contains projects that fulfill our requirements (detailed in the following sections) and comprises two groups: CI projects and NOCI projects.

The process of data collection for constructing both datasets, as elucidated by Nery et al. (Nery, Costa, and Kulesza, 2019), is depicted in Figure 3.1 and Figure 3.2.

To ensure that this thesis is self-contained, we explicate the methodology employed in creating mentioned datasets, including supplementary procedures for collecting pertinent data for our research.

The CI Dataset

The CI dataset consists of projects that eventually adopted CI during their lifetime. As this dataset is used to analyze the evolution of code coverage before and after adopting CI, it must have considerable historical data before and after the adoption of CI. The CI dataset contains two years of data for each project, one year before and one year after CI adoption.

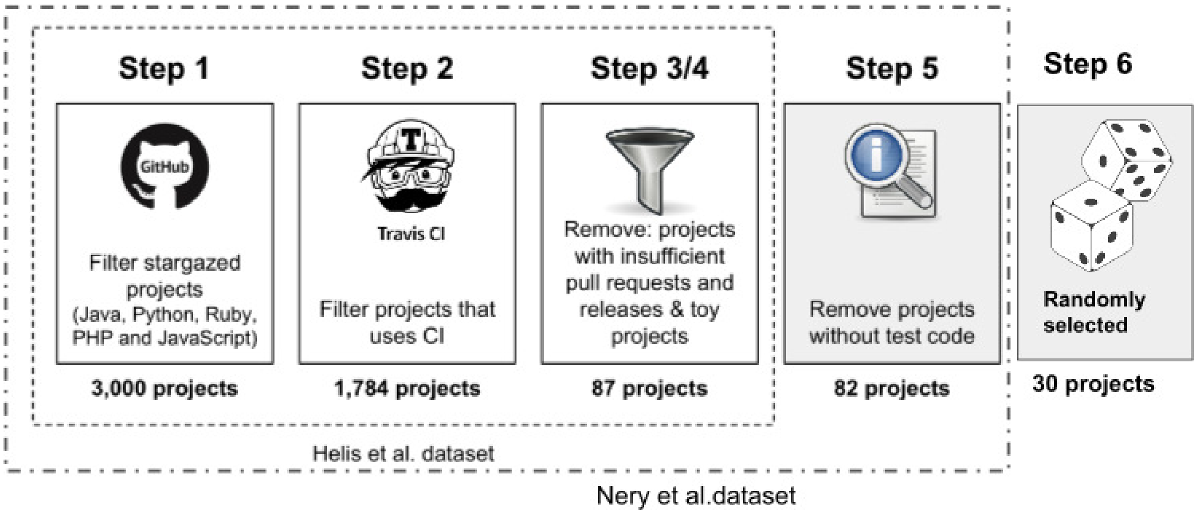
Step 1
GitHub
Filter stargazed projects (Java, Python, Ruby, PHP and JavaScript)
3,000 projects
Step 2
Travis CI
Filter projects that uses CI
1,784 projects
Step 3/4
Remove: projects with insufficient pull requests and releases & toy projects
87 projects
Step 5
Remove projects without test code
82 projects
Step 6
Randomly selected
30 projects
Helis et al. dataset
Nery et al.dataset

Figure 3.1: CI Projects.

In **step 1** (Figure 3.1), the 3,000 most popular GitHub projects (as measured by the number of stars) were selected. These projects were written in Java, Python, Ruby, PHP, and JavaScript. Next, projects that adopted a CI service were selected (TravisCI[1] in this case). Since August 2019, GitHub has officially supported CI through GitHub Actions (GHA). Although GHA has become the dominant CI service in the market, as mentioned before, our study reuses a consolidated dataset used in other CI-related studies

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

(Bernardo, Costa, and Kulesza, 2018; Nery, Costa, and Kulesza, 2019), which is based on projects that use TravisCI. These projects were carefully filtered and fulfilled our requirements, as detailed below. We also need the entire CI history from such projects, making other services like Jenkins[2] inappropriate.

In **Step 2**, builds associated with projects on TravisCI were checked to select projects that adopt CI. Projects without any build on TravisCI were removed. Additionally, the first TravisCI build date of a project was used as the moment at which a project started using CI. Of the 3,000 projects, 1,784 remained after this stage (59.5%).

Afterward, toy projects[3] (**step 3**) and projects with less than 100 linked pull requests[4] before and after CI adoption (**step 4**) were filtered out to improve the dataset's quality. 87 projects remained after steps 3 and 4. And for linked pull requests, we understand any pull requests bind to a release.

87 projects remained after steps 3 and 4. To investigate the relationship between CI and test code evolution (which includes test coverage), five projects, which did not have any test code, were excluded (**Step 5**). A total of 82 projects remained.

Lastly, in **Step 6**, because our goal is to study the relationship between CI and code coverage, we further process the dataset made available by Nert et al. (Nery, Costa, and Kulesza, 2019). To collect the code coverage information, we *set up and run* each version of the 82 projects in order to execute their test suites and compute the coverage information. This task proved challenging for several reasons.

First, for each project version, we need to download all the involved libraries (some of them old, unavailable, or dependent on the OS), configure the build tools (which changed considerably from version to version, especially for JavaScript projects), and

[2]https://jenkins.io/

[3]Toy projects are small and simple projects that are created for learning or experimentation purposes rather than for actual use. Toy projects are typically characterized by a small number of files, a limited quantity of commits, and infrequent updates, often spanning a significant period of time.

[4]A linked pull request was defined as any pull request that is associated with a specific software release (Bernardo, Costa, and Kulesza, 2018).

run the tests.

Second, interpreting the result of tests is not trivial, as a failing test could be due to a configuration error from our side instead of a real failure in a particular version of the project, requiring us to debug and make assumptions about the domain knowledge of the project. These challenges incur compiling and replicating the required environment, using virtual machines, to execute the automated tests, and collecting the code coverage data for each version of our projects, which made this task exceptionally time-consuming. If a specific version (commit) was not compilable, we skipped it and proceeded to the nearest compilable version (commit) within the same month.

Due to these challenges, we further sampled 30 randomly selected projects out of the previous 82 projects (**step 6** in Figure 3.1). We collected the code coverage information for all versions of the remaining 30 projects (8 Python, 5 Java, 9 JavaScript, 4 Ruby, and 3 PHP projects).

The NOCI Dataset

The NOCI dataset consists of projects that never adopted CI. This dataset is essential for checking the presence of bias in our observations (control data). For example, if we observe an increase in code coverage after a project adopts CI, we can check whether a similar trend occurs in the NOCI projects. If that is the case, it would be unreasonable to conclude that CI could be a precursor for the improvement in code coverage (as NOCI projects would also have presented such an improvement). Figure 3.2 illustrates the process for building the NOCI dataset.

In **Step 1**, again, 3,000 projects with the most stars were selected. Next, projects with at least one build in TravisCI were discarded (**Step 2**), since the NOCI dataset must contain projects that have never adopted CI. Next, in **step 3**, to avoid studying inactive and immature projects, we select projects that have at least four years of activity.

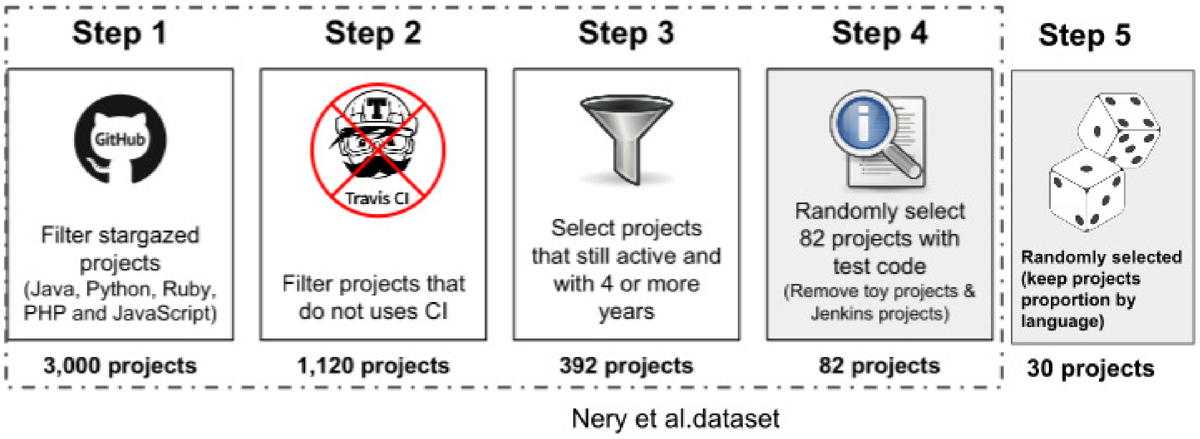
Step 1
Filter stargazed projects (Java, Python, Ruby, PHP and JavaScript)
3,000 projects
Step 2
Travis CI
Filter projects that do not uses CI
1,120 projects
Step 3
Select projects that still active and with 4 or more years
392 projects
Step 4
Randomly select 82 projects with test code (Remove toy projects & Jenkins projects)
82 projects
Step 5
Randomly selected (keep projects proportion by language)
30 projects
Nery et al.dataset

Figure 3.2: NOCI Projects.

After this step, 392 NOCI projects remained.

In **step 4**, 82 projects were randomly selected from the previous step. The reason for selecting 82 projects is to balance the samples between the CI and NOCI datasets. The random selection process occurred in several iterations. First, a NOCI project was randomly selected from the dataset. Then, two criteria were checked: (i) whether the project is not a toy project and (ii) whether the project has a test suite available. If a given NOCI project fails to fulfill both criteria, the project is discarded, and another project is randomly selected to replace it.

Sizílio et al. (Nery, Costa, and Kulesza, 2019) also manually verified whether projects contain configuration files from other popular CI services like Jenkins or CircleCI[5]. All projects using another CI service provider were excluded. To strengthen the certainty that the selected projects never adopted CI, the members of each project were contacted via development channels and emails. This process was repeated until 82 NOCI projects remained to match the CI projects number.

Nery et al. (Nery, Costa, and Kulesza, 2019) received confirmation replies for about 48% of the NOCI projects. For the remaining 52% of NOCI projects, they confirmed the NOCI attribute by manually inspecting their code to find CI services configuration

[5]https://jenkins.io/

files.

Lastly, due to the challenges involved in computing code coverage information (as explained before), we further sampled 30 randomly selected projects out of the 82 NOCI projects (**Step 5** in Figure 3.2). We collected the code coverage information for all versions of the remaining 30 projects. We sought to maintain the same proportion of projects across languages in the final dataset, ensuring that there was no more than a 5% difference in the number of projects for each language.

3.2.2 Collecting Versions

Once our projects are selected, we collect the versions of each project to perform our analyses, since code coverage varies across versions. A version is the nearest commit (snapshot) on the project's default branch that occurred at the beginning of a month, which can also be the end of a previous month (Zhao et al., 2017; Nery, Costa, and Kulesza, 2019).

As our goal is to analyze the impact of adopting CI on code coverage, we collect historical data before and after the adoption of CI. To determine when CI was adopted in a project, we use the approach of previous studies (Zhao et al., 2017; Nery, Costa, and Kulesza, 2019; Bernardo, Costa, and Kulesza, 2018), which considers the first build on TravisCI as the reference point for when CI was adopted in a project. Commits made after the reference point are categorized as belonging to the After-CI group, while those made before are categorized as belonging to the Before-CI group.

Similar to Zhao et al. (Zhao et al., 2017), we disregarded 30 days around the adoption of CI (15 days before and 15 days after), since it may represent an unstable and noisy period for data collection (Zhao et al., 2017).

Figure 3.3 illustrates this process, where each red arrow indicates a checkout of the nearest version. We collect 12 months of historical data before CI and 12 months after CI for each project.

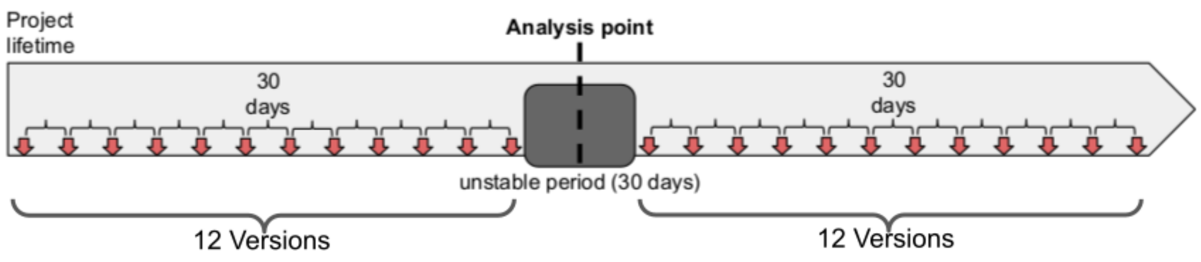

Figure 3.3: Overview of versions selection process (Nery, Costa, and Kulesza, 2019).

Concerning NOCI projects, it is also necessary to establish a moment to divide their history into two periods, which is important to double-check the observations found when studying CI projects. One could argue that NOCI projects could be split based on their coverage variation, where a peak in coverage could become the splitting-event time point. However, if we consider the CI projects, we primarily consider the adoption of CI as the splitting-event time point regardless of variations in code coverage. Thus, to ensure a fair comparison between CI and NOCI projects, we follow the methodology applied by Nery et al. (Nery, Costa, and Kulesza, 2019). Specifically, we select a splitting-event time point for the NOCI projects that is proportional to their own lifetime.

To find this point in time for NOCI projects, we first analyze the CI projects to compute the number of months before the adoption of CI. Next, we identify the time that CI projects took to adopt CI in relation to their own lifetime. We found that the median proportional time to adopt CI was 27.36% of the projects' lifetime. As such, we use 27.36% of a project's lifetime as the splitting-event time point for NOCI projects.

We fetched 24 versions from each project, i.e., 12 versions before the splitting-event and 12 versions after the splitting-event, for both CI and NOCI projects (see Figure 3.3). For CI projects, the splitting-event is the adoption of CI, whereas, for NOCI projects, the splitting-event is the 27,36% timepoint within the projects' lifetime. The Table 3.1 resumes all datasets.

Table 3.1: Study datasets.

Dataset	Definition
Before-CI	The dataset includes the data of our CI projects before adopting CI.
After-CI	The dataset includes the data of our CI projects after adopting CI.
Early-NOCI	The dataset contains the first 27,36% of the lifetime of our NOCI projects.
Late-NOCI	The dataset contains the remaining data of the NOCI projects

3.2.3 Collecting Coverage

Code coverage is a widely used quality metric with a long history in software engineering (Vanoverberghe et al., 2011). Code coverage indicates the parts of the software that are exercised by the test suite.

There is a wide variety of coverage metrics, including statement coverage, branch coverage, edge-pair coverage, path, predicate, clause, and data flow coverage (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021). Statement coverage, i.e., the number of instructions executed by tests over the total number of instructions, is the simplest but most commonly used coverage metric applied in studies on the evolution of code coverage (Hilton, Bell, and Marinov, 2018; Gopinath, Jensen, and Groce, 2014; Ahmed et al., 2016). Due to this reason, we choose statement coverage as our starting point.

We computed the coverage percentage (PC) using the following formula: $PC = EL/VL$ where EL stands for *executed lines* and VL stands for *valid lines* (i.e., lines that are not comments and blank lines).

Nevertheless, configuring, building, and running each project to compute the test coverage requires considerable effort. Therefore, we sample 30 projects from the database to analyze, according to the process presented in Section 3.2.1. We compiled, tested, and collected the code coverage in each version of each project studied in this thesis. We ran the builds on a set of Ubuntu 14.04 docker containers targeting a compatible and stable environment for the datasets, which include projects with versions dating back to as early as 2012/2013.

We collected coverage information using the following coverage tools: `Coverage.py`[6] to instrument Python projects, `Open Clover`[7] for Java projects, `Istanbul`[8] for JavaScript projects, `SimpleCov`[9] for Ruby projects, and `PHP Unit Coverage`[10] for PHP projects. To unify the output from all these libraries, we instruct them to produce two kinds of format: Clover-formatted output and the Cobertura formatted output.

3.3 Results

In this section we disclose the motivation, approach, and results for our research questions (RQ).

3.3.1 RQ1 - What are the evolution trends of code coverage within CI and NOCI Projects?

Motivation

Although recent research has studied associations between CI and several software development factors (Nery, Costa, and Kulesza, 2019; Beller, Gousios, and Zaidman, 2016), they have not investigated whether CI is associated with better code coverage in software projects, which is an important investigation, since CI is known for its focus on automated tests.

In this RQ, we study the evolution trends of code coverage in both CI and NOCI projects. The trends will allow us to understand whether code coverage evolves differently

[6] https://coverage.readthedocs.io
[7] ttps://openclover.org
[8] https://istanbul.js.org
[9] https://github.com/simplecov-ruby/simplecov
[10] https://phpunit.de

in CI projects compared to NOCI projects. For example, CI projects may be associated with a rising trend of code coverage, whereas NOCI projects may be associated with maintaining or falling trends of code coverage (or vice-versa). Our goal is to investigate whether CI is associated with enhancements in code coverage.

Approach

To address this research question, we use both datasets containing 30 CI projects and 30 NOCI projects, as explained in Section 3.2. We compare the CI projects against the NOCI projects to study the potential differences in code coverage trends between CI and NOCI projects.

The evolution of code coverage over time can be interpreted as time series data. Therefore, we use the Dynamic Time Warping (DTW) algorithm to identify the dominant trends of coverage evolution.

The *Dynamic Time Warping (DTW)* algorithm (Kate, 2016) is a robust method used to measure the similarity between time series by aligning their offsets (Berndt and Clifford, 1994; Salvador and Chan, 2007), meaning that two similar time series can be grouped together even if they span over different time periods. As such, DTW allows us to group coverage trends into clusters. Although DTW supports cluster discovery, we must first set the optimal number of clusters (Milligan and Cooper, 1985), so they can better represent the coverage trends. We use the Gap Statistic (Tibshirani, Walther, and Hastie, 2001) approach to determine the appropriate number of clusters. We varied the number of clusters from 2 to n-1, where n is the number of projects.

Results

Each plot in Figures 3.5, 3.4, 3.6 and 3.7 contains the distribution of code coverage (Y-axis) across the versions of the CI or NOCI projects (X-axis). The vertical dotted line indicates the moment at which CI was adopted (in CI projects), or the *splitting-event* in the lifetime of NOCI projects. We use the numbers on the top gray bars of the plots to refer to a specific cluster in our observations.

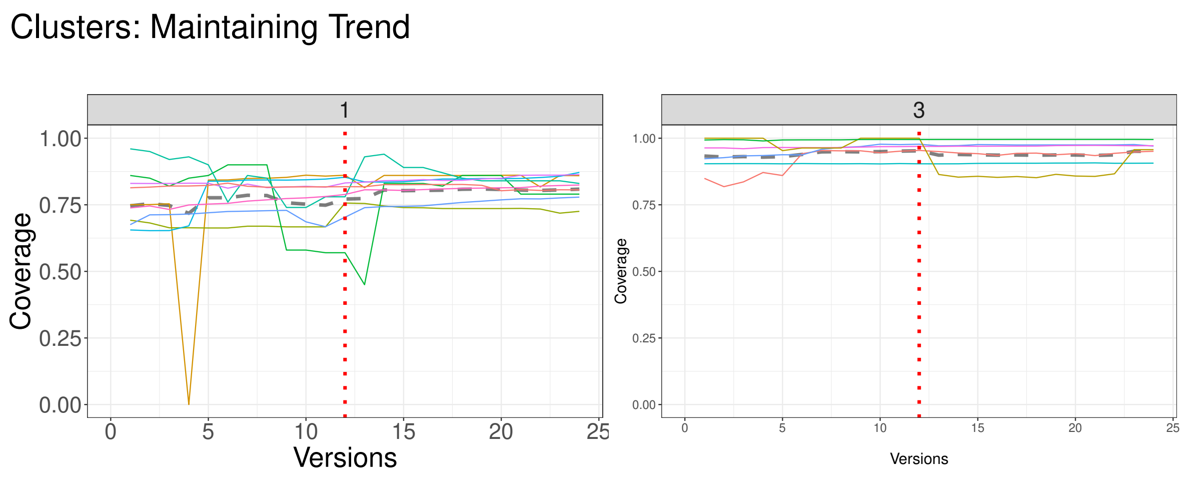

Figure 3.4: CI Projects - Maintaining Trend

Figure 3.4 shows that the centroids (i.e., the gray horizontal dashed lines) of clusters 1 ($\frac{9}{30}$ projects) and 3 ($\frac{6}{30}$) of CI projects indicate a maintaining trend of code coverage. However, although these are maintaining trends, they are stabilized at high values (i.e., around or above 75%).

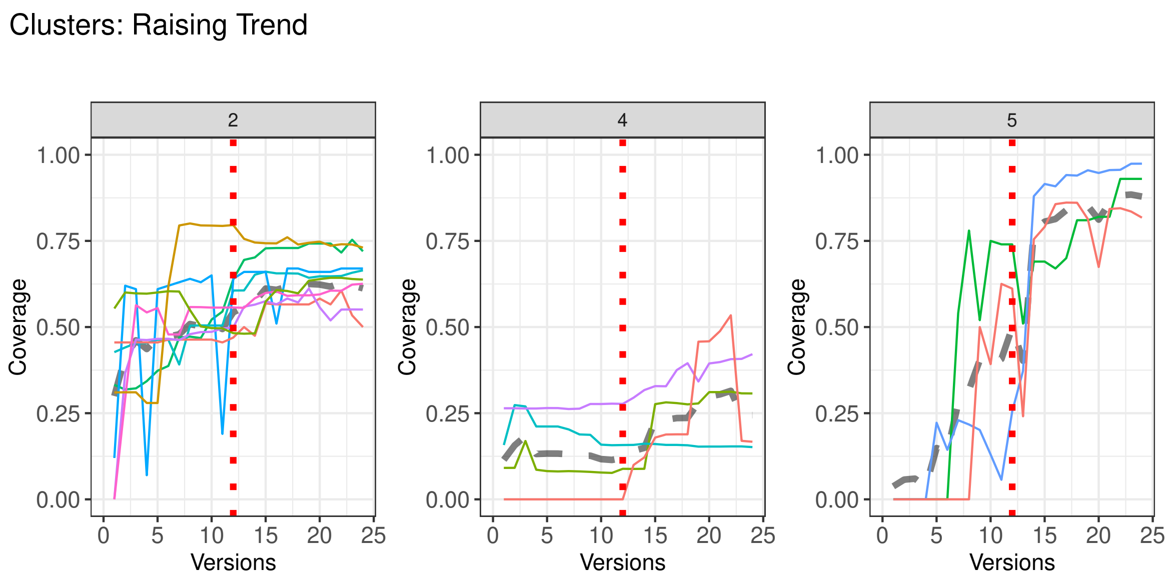

Figure 3.5: CI Projects - Raising Trend

Figure 3.5 shows clusters 2, 4, and 5, which reveal rising code coverage trends of CI projects. Clusters 2 ($\frac{8}{30}$ projects) and 5 ($\frac{3}{30}$ projects) reveal a significant increase in the code coverage, i.e., varying from 25% to 75%. On the other hand, cluster 4 ($\frac{4}{30}$ projects) reveals only a slight increase of code coverage (i.e., from around 13% to 25%).

Figure 3.6 shows the rising code coverage trends for NOCI projects. Only cluster 1 reveals a rising trend of code coverage ($\frac{3}{30}$ projects). We observe that the code coverage for those projects stabilizes at values around below 65%, thus, lower than the values observed in most of the rising trends of CI projects.

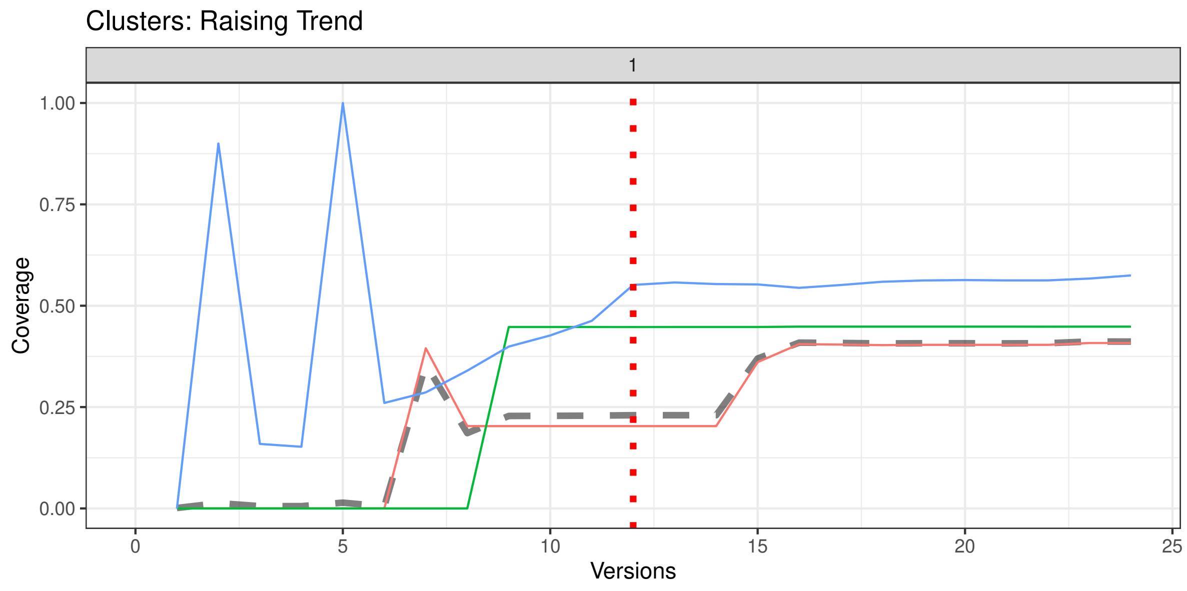

Figure 3.6: NOCI Projects - Raising Trend

The remaining clusters shown in Figure 3.7 reveal maintaining code coverage trends. Clusters 2, 3, and 10 containing $\frac{3}{30}$ projects; clusters 4, 7, 8, 9, and 16 containing ($\frac{2}{30}$ projects; and clusters 5, 6, 11, 12, 13, 14, 15, and 17 containing $\frac{1}{30}$ projects. Overall, 90% of NOCI projects reveal a maintaining trend of code coverage ($\frac{27}{30}$ projects).

When comparing the code coverage distributions between the *before-CI* and *after-CI* periods, we observe a notable increase in the median code coverage following the adoption of CI. In the *before-CI* period, the median code coverage is 0.663 with a mean of

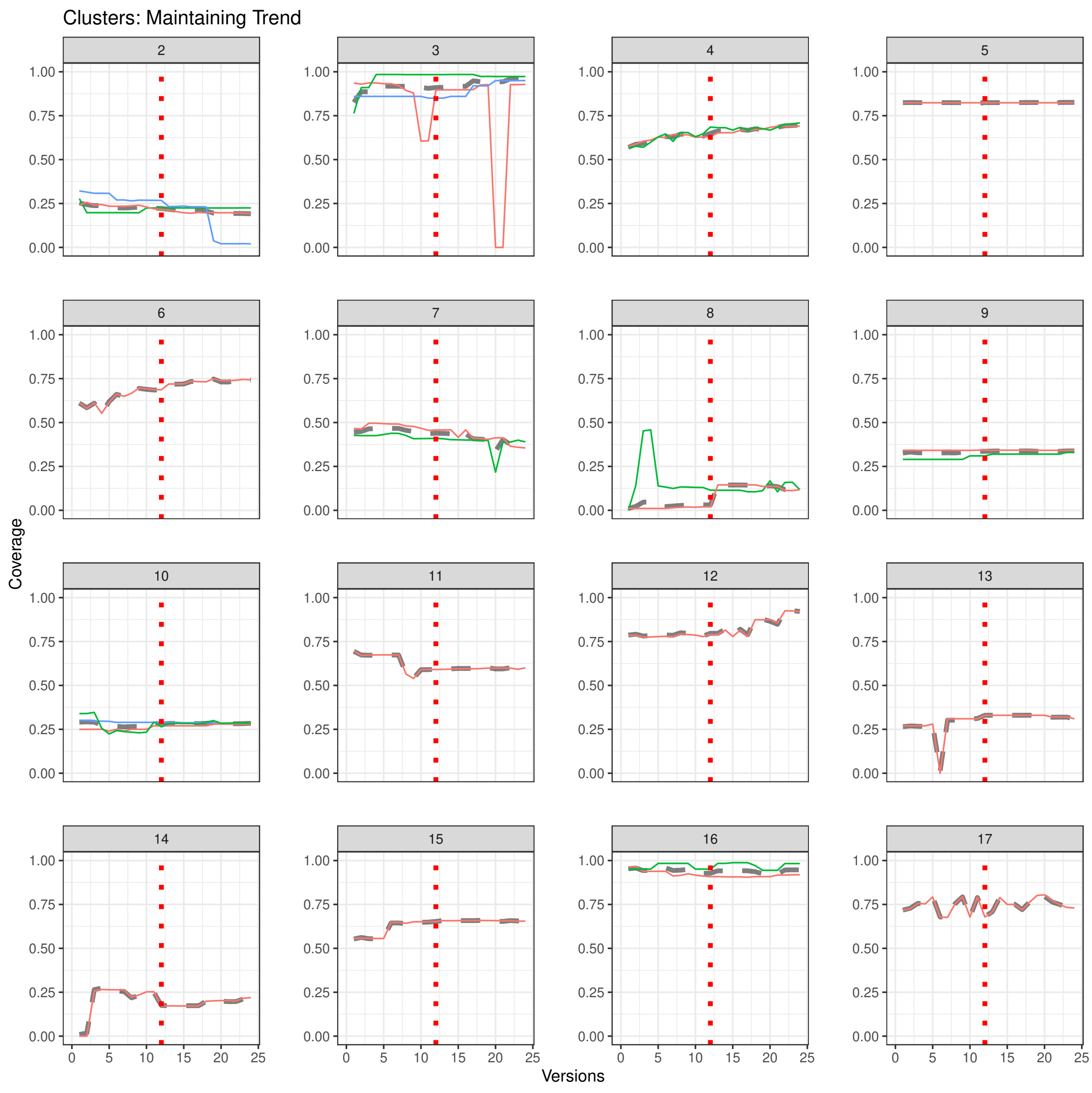

Figure 3.7: NOCI Projects - Maintaining Trend

0.72 and a standard deviation of 0.315. In the *after-CI* period, the median code coverage is 0.79 with a mean of 0.629 and a standard deviation of 0.225. A Wilcoxon rank-sum test reveals a statistically significant difference between the distributions ($p-value = 2e-07$). Additionally, we calculate a Cliff's delta of -0.2173, indicating a *small* but meaningful difference in code coverage values. Thus, the improvement in coverage across sets is on the order of magnitude of 10%.

In a similar vein, when comparing the code coverage distributions between the *early-NOCI* and *late-NOCI* periods (as shown in Table 3.1), we observe the following results. In the *early-NOCI* period, the median code coverage is 0.435 with a mean of 0.482 and a standard deviation of 0.293. In the *late-NOCI* period, the median code coverage is 0.448 with a mean of 0.510 and a standard deviation of 0.287. Despite a slight increase in code coverage during the *late-NOCI* period, our Wilcoxon test (with alternative='less') indicates no significant difference between the *early-NOCI* and *late-NOCI* periods ($p - value = 0.1$). Thus, the difference in coverage levels between the *early-NOCI* and *late-NOCI* periods is negligible or statistically insignificant.

Findings 3.3.1

We observe more projects with rising trends of code coverage in CI projects (50%, $\frac{15}{30}$) than NOCI projects (10%, $\frac{3}{30}$). We also observe that maintaining trends differ, as CI projects tend to stabilize their code coverage at a higher value than NOCI projects.

3.3.2 RQ2 - Is there a significant correlation between CI and code coverage values?

Motivation

In RQ1, we observe a notable difference in code coverage trends when comparing CI projects against NOCI projects. As such, in RQ2, we dive deeper and quantitatively analyze the code coverage of our studied projects. The goal is to investigate whether the adoption of CI has a significant association with better code coverage. This investigation allows us to better understand whether CI has the potential to benefit code coverage.

Approach

To address this research question, we applied the regression discontinuity design (RDD) (Imbens and Lemieux, 2008) to assess the potential association between adopting CI and better code coverage. As this research question considers non-randomized trials, as is often the case with software engineering data, techniques such as quasi-experiments are suitable. The RDD is a robust quasi-experimental design to evaluate the presence and extent of the impact introduced by an intervention (Cook and Campbell, 1979). It works by assigning a cut-off above or below a given intervention. RDD assumes that the trend would continue without changes if the intervention had not existed. In our study, the intervention refers to the adoption of CI; a cut-off relates to the moment at which CI was introduced. RDD has already been applied by existing research investigating the potential benefits of CI (Cassee, Vasilescu, and Serebrenik, 2020; Zhao et al., 2017), e.g., to study whether CI may influence software quality or productivity in software development. Given that RDD fits the purpose of our research question, we apply RDD in RQ2.

In this RQ, the RDD model was formulated in the following manner. Let Y be the outcome of the dependent variable (coverage) of the project version with and without the effect of the intervention (i.e., using CI or not), respectively. $D_i \in 0, 1$ represents the intervention, which, in our case, refers to the moment at which a CI service was adopted. Finally, X_i, as the forcing variable, represents the time in months from the start of the observation period. It determines the value of D_i with the cutpoint c. $D_i = 1X_i > c$. We can specify the following linear regression model to estimate the level and trend in Y before and after the adoption of CI:

$$E[Y_i(d)|X_i, D_i] = \alpha + \beta * c + \tau * D_i + \beta * X_i$$

Where $E[Y_i(d)|X_i, D_i]$ represents the estimation of the coverage given X_i and D_i; τ as the treatment effect, α represents the intercept of a fitted regression line, and β represents the slope. Solving the regression gives us the coefficients, which, if significant, can help us reason about the effects of adopting CI, if any. We report on the models having significant coefficients in the regressions ($p < 0.05$) as well as their effect size.

Results

We fitted an RDD model, as described above, to model the trend in coverage levels over time as a function of the adoption of CI. Our data is centered around the moment at which CI was adopted, having an equal number of points, 12, on each side. The model is fitted by `lm` function in `R`. This function models a regression with the Ordinary Least Squares (OLS) technique.

Figure 3.8 shows the outcome from the fitted model. A discontinuity is observed after introducing CI. The dots represent individual coverage levels. In the center of the plot, we note the cut-off point, before which are the samples after the adoption of CI, and after which are the versions that precede the adoption of CI. The effect of the intervention is estimated as a discontinuity between the intervention groups before and after the cut-off point (the adoption of CI in our case).

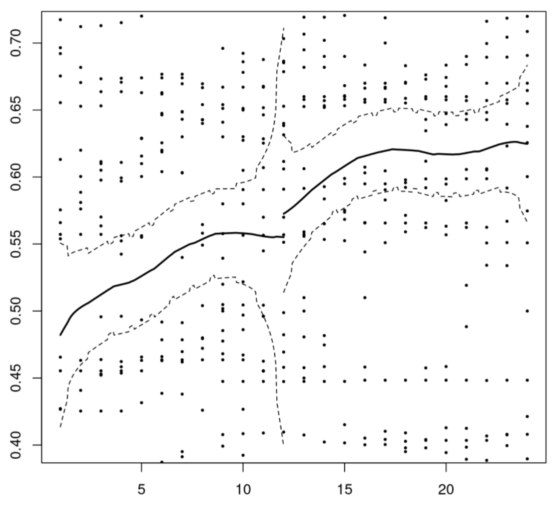

Figure 3.8: RDD model for CI Projects

Table 3.2 summarizes the results. We observe a statistically significant D of 0.12710 in coverage, meaning that adopting CI increases code coverage. The R-squared (R^2) is a measure that provides information about how well the data fits the regression model (the goodness of fit).

Table 3.2: Coverage model Results.

Coefficients	Estimate Std	Error	τ	$Pr(> \tau)$
Intercept	0.59177	0.01444	40.977	<2e-16 ***
D	0.12710	0.02042	6.223	8.27e-10 ***

Statistical significance is indicated by ***p<0.001, **p<0.01,*p<0.05

Figure 3.9 shows a boxplot of the R^2 values for the analyzed projects. The horizontal line represents the median of R^2 values (i.e., 0.46652) for all projects. Indeed, the linear model achieves an R^2 value higher than 0.35 for 60% of the projects, and 50% of them achieve an R^2 value higher than 0.5, meaning that adopting CI can explain a reasonable portion of the coverage variation.

On the other hand, we did not find any discontinuities when applying the RDD model to the NOCI dataset (Figure 3.10).

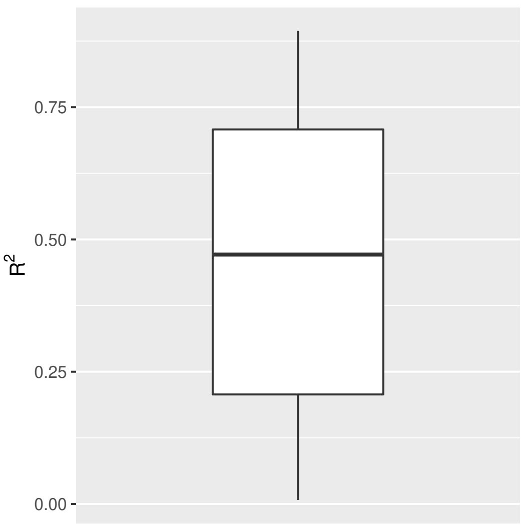

Figure 3.9: Box plot of R^2 from each project.

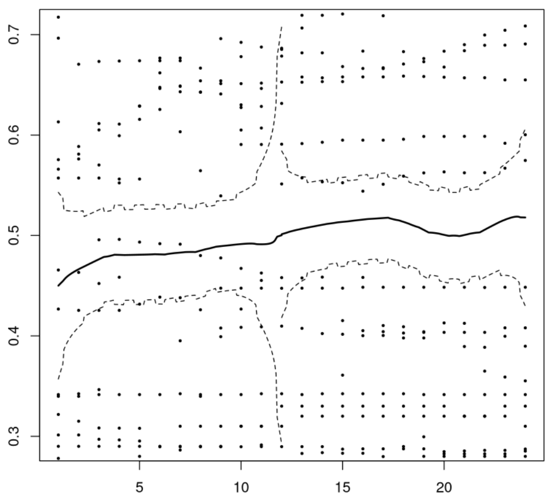

Figure 3.10: RDD model for NOCI Projects

Findings 3.3.2

The statistical evidence indicates that adopting Continuous Integration is associated with increased code coverage.

3.3.3 RQ3 - What types of code changes affect the code coverage of CI and NOCI projects?

Motivation

In RQ1 and RQ2, we study whether CI is associated with the evolution of code coverage in general. As our results reveal that CI has a significant association with code coverage, we investigate how code coverage evolves in a finer granularity, i.e., we investigate different types of code changes that contribute to the evolution of code coverage. For example, is CI associated with more legacy code being covered? Or is CI associated with the removal of legacy and uncovered code, which ultimately increases the total coverage? These are the type of questions we investigate in RQ3.

Current tools offer little support for understanding how changes to the source code can affect code coverage (Hilton, Bell, and Marinov, 2018). From a naïve perspective, code coverage increases when existing lines of code become covered and decreases when existing lines are no longer covered. However, when considering code evolution, i.e., comparing two consecutive project versions, coverage can vary for diverse reasons. For example, adding new and uncovered source code will cause code coverage to decrease. Likewise, removing existing and uncovered source code will increase code coverage without new tests. Finally, as one would expect, code coverage can be impacted due to the addition or deletion of test code. In conclusion, there are several ways that code coverage can increase or decrease. Our purpose with RQ3 is to study whether CI has a strong association with the changes that impact the evolution of code coverage.

Approach

To address this research question, we again use our dataset containing 30 CI projects and 30 NOCI projects. We combine code coverage data with code change data to identify the types of code changes that affect code coverage. For each project version, we find the lines of code (LoC) added, modified, or deleted compared to the previous version (i.e., the commit patch representing the changes between versions). To track lines of code that have moved or shifted within the codebase, we employ the Git Diff command. This allows us to compare each version of the code with its previous version and obtain a list of added and removed lines. We also create a mapping that identifies the unchanged lines by pairing the line numbers of each version. To generate these mappings, we use the Git Diff command for each changed file and determine the line numbers for each line as compared to the previous version of that file. We used a global identifier to track the position of each line across the versions of each project. Lastly, we aggregate each studied project version's coverage and code change information. With this combined information, it is possible to determine whether the increase or decrease in code coverage occurs due to the addition of new LoC or deletion of existing LoC (covered or uncovered). We group the change types of covered or uncovered LoCs into the categories shown in Figure 3.11, taking the median number of LoC that contributes to each category and the overall number of changes throughout the versions.

Results

Figure 3.11 provides an overview of how each type of code change contributes to the variation in code coverage. Comparing the CI dataset with the NOCI dataset, we observe that the coverage of new lines is responsible for nearly 16% of changes in code coverage, whereas, in the NOCI dataset, only nearly 5% of the changes in coverage are due to covering newly added lines. Indeed, a Wilcoxon test shows that CI

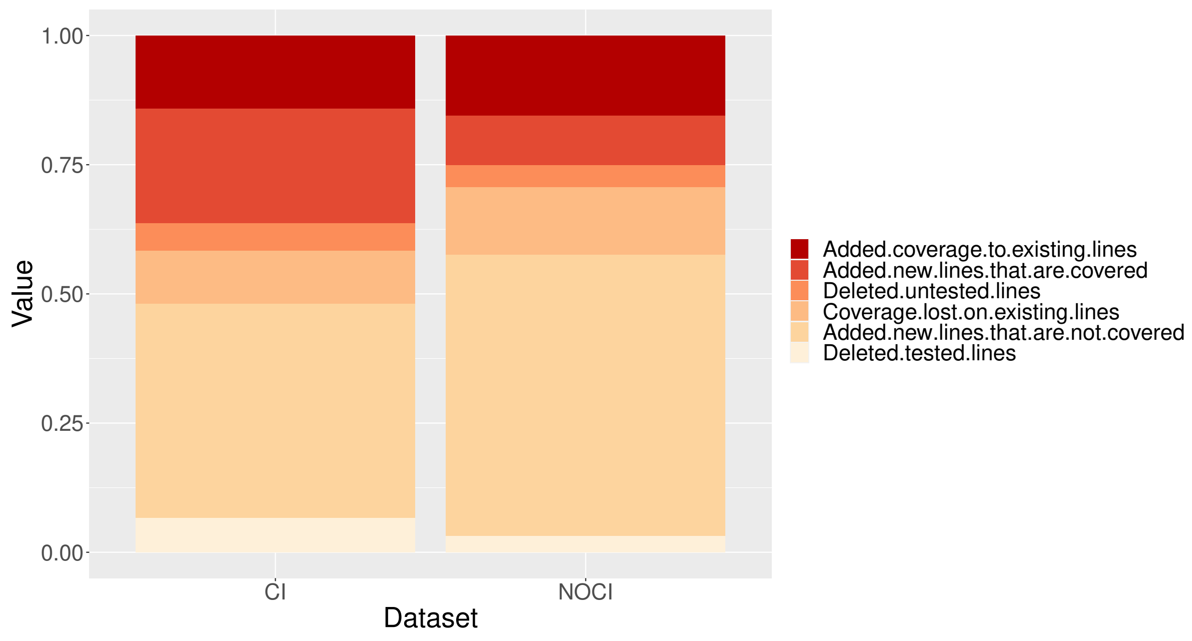

Figure 3.11: How much each code change contributed to the net change in coverage.

is positively associated with the coverage of newly added lines ($p = 0.0003$ with a *large* effect $d = -0.543$).

Regarding the changes that increase coverage, we find that 40% of code changes in CI projects increase the coverage, whereas only about 26% of code changes in NOCI projects increase the coverage. A Wilcoxon test reveals that this difference is statistically significant ($p = 3.255e - 05$ with a *large* Cliff's delta of $d = -0.602$). Furthermore, we notice a statistically significant difference between CI and NOCI projects ($p = 0.0016$ with a *large* Cliff's delta of -0.475) regarding deleted test lines. CI projects lose more covered lines because they have a higher coverage. Nevertheless, we do not have enough evidence to reject the null hypothesis for the remaining types of changes, e.g., *deleted untested lines* etc. Table 3.3 summarizes the obtained results for the CI and NOCI comparisons.

We now shift the focus of our analysis to investigate which types of changes impact the code coverage during the before-CI and after-CI periods (for CI projects), and during the early-NOCI and late-NOCI periods (for NOCI projects).

Figure 3.12 shows an overview of the prevalence of each type of code change

Table 3.3: Results of statistical tests applied in NOCI and CI dataset

Metric	Wilcoxon (p)	Cliff's Delta
Added coverage to existing lines	0.398	-
Added new lines that are covered	0.0003***	-0.543 (Large)
Deleted untested lines	0.0821	-
Coverage lost on existing lines	0.9474	-
Added new lines that are not covered	0.948	-
Deleted tested lines	0.0016**	-0.475 (Large)
Changes that increase coverage	3.255e-05***	-0.602 (Large)
Changes that decrease coverage	0.6125	-

Statistical significance is indicated by ***p<0.001, **p<0.01, *p<0.05

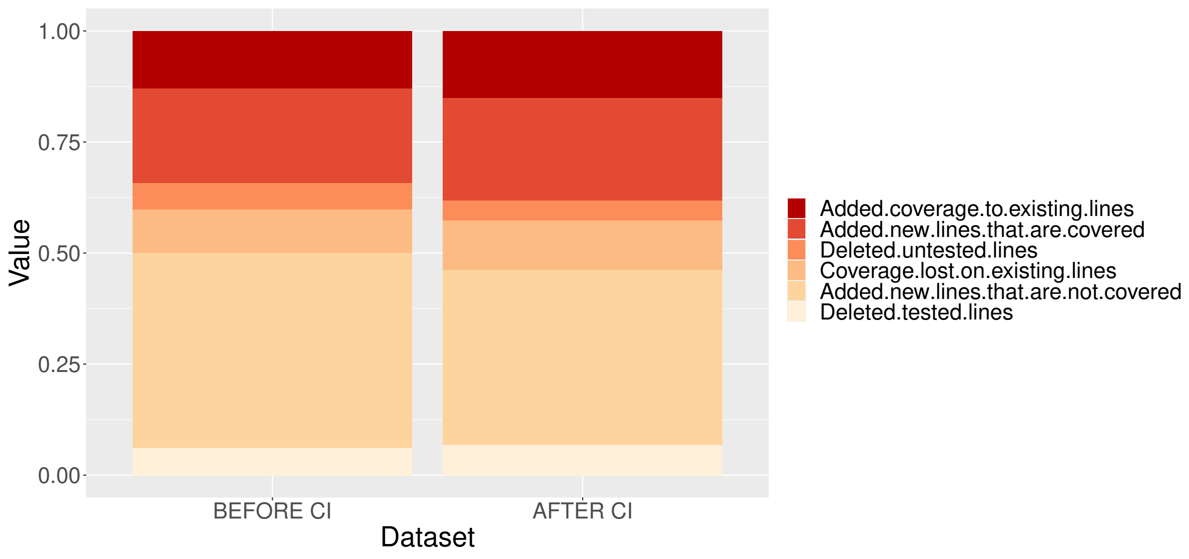

Figure 3.12: How much each code change contributed to the net change in coverage into CI dataset.

during the before- and after- CI periods. We observe that CI has a positive association with the coverage of existing lines. The median and mean of existing and covered lines during the before-CI period are 0.663 and 0.592, respectively. The median and mean of existing and covered lines during the after-CI period are 0.7900 and 0.7189, respectively. Our Wilcoxon test reveals a significant difference between the before-CI and after-CI periods ($p = 0.01966$) with a *small* effect size $d = -0.243$. Overall, the after-CI period has 14% more changes that increase the code coverage than the before-CI period ($p = 0.04049$ with a *small* effect size $d = -0.233$). Nevertheless, we do not have enough evidence to reject the null hypothesis for the remaining types of changes, e.g., *deleted untested lines* etc. Table 3.4 summarizes the test results.

Table 3.4: Results of statistical tests applied in before-CI and after-CI dataset

Metric	Wilcoxon (p)	Cliff's Delta
Added coverage to existing lines	0.01966*	-0.243 (Small)
Added new lines that are covered	0.1358	-
Deleted untested lines	0.3892	-
Coverage lost on existing lines	0.2621	-
Added new lines that are not covered	0.8553	-
Deleted tested lines	0.1332	-
Changes that increase coverage	0.04049*	-0.233 (Small)
Changes that decrease coverage	0.04049	-

Statistical significance is indicated by ***p<0.001, **p<0.01, *p<0.05

Regarding the NOCI projects, in Figure 3.13, we observe a statistically significant difference ($p = 0.02479*$ with a *small* delta of $d = -0.215$) in the coverage lost on existing lines. We also obtain a statistically significant difference ($p = 0.03213$) in deleted tested lines with a *small* effect $d = -0.215$. Similarly, with respect to deleted and uncovered lines, a Wilcoxon test reveals a statistically significant ($p = 0.01776$) difference with a *small*, but non-negligible effect ($d = 0.234$).

The remaining statistical tests show that we do not have enough evidence to reject the null hypothesis for the other types of changes, corroborating the strong association

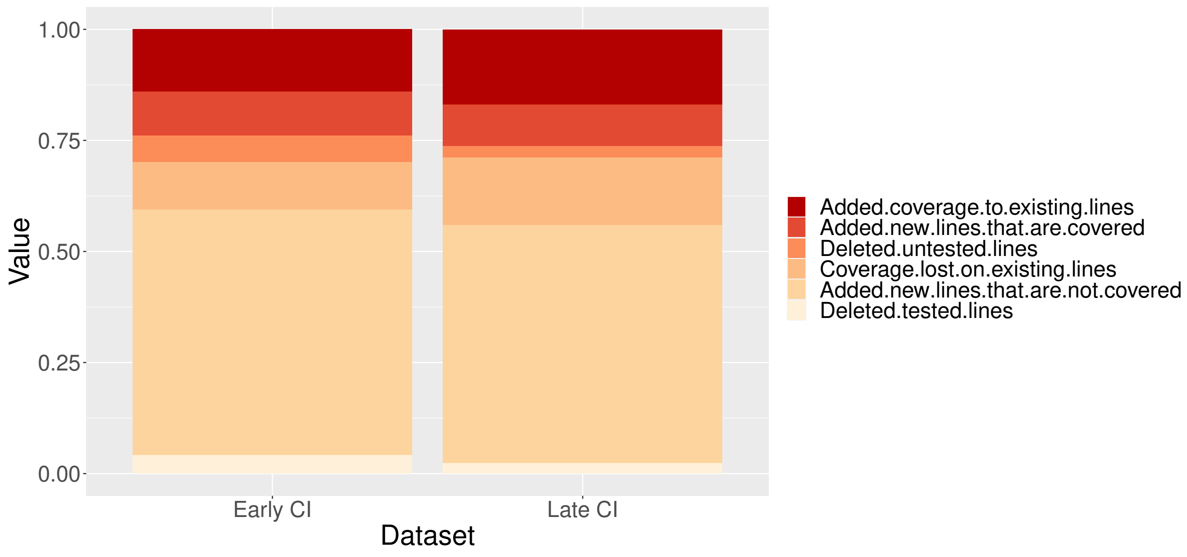

Figure 3.13: How much each code change contributed to the net change in coverage into NOCI dataset.

between the adoption of CI and an improved code coverage. Table 3.5 summarizes the test results.

Table 3.5: Results of statistical tests applied in early-NOCI and late-NOCI dataset.

Metric	Wilcoxon(p)	Cliff's Delta
Added coverage to existing lines	0.06356	-
Added new lines that are covered	0.06018	-
Deleted untested lines	0.01776*	0.234 (Small)
Coverage lost on existing lines	0.02479*	-0.215 (Small)
Added new lines that are not covered	0.812	-
Deleted tested lines	0.03213*	0.217 (Small)
Changes that increase coverage	0.4045	-
Changes that decrease coverage	0.5158	-

Statistical significance is indicated by ***p<0.001, **p<0.01, *p<0.05

Reasons for the decrease in coverage. Analyzing the evolution of code coverage during the after-CI period, we observe that some projects underwent substantial reductions in code coverage levels across successive versions. We manually inspect the issues, code commits, and pull requests to better understand these reductions in code coverage. These reductions were mostly observed in the SENSU[11], AFRAME[12], FOG[13], and GOLLUM[14], projects. Figure 3.14 shows the coverage evolution in these projects.

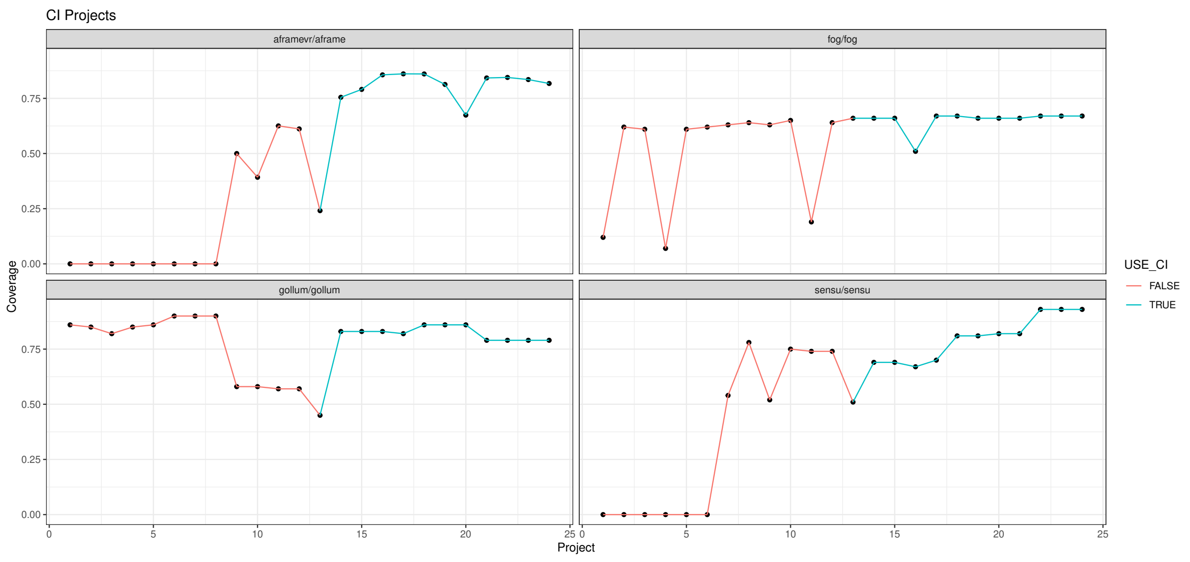

Figure 3.14: Projects with a sharp drop in code coverage at some point after CI adoption.

For example, pull request #305[15], from the SENSU project, addressed a premature termination of tests due to a timeout issue. To fix the premature termination, developers increased the test waiting time, allowing for the proper completion of CI tests. If tests are terminated prematurely, the code they exercise is not executed, which impacts the computation of code coverage, i.e., if the test suite exercises fewer statements, there exists an unexpected reduction in code coverage.

Another example is pull request #303[16], also from the SENSU project, which addressed caching issues. In that case, test code relies on variables in an external cache server to test each code branch properly. Due to caching issues, tests failed to exercise some code execution flows, consequently reducing code coverage (i.e., these execution flows were not considered when computing code coverage). We speculate that this kind

[11]https://github.com/sensu/sensu
[12]https://github.com/aframevr/aframe
[13]https://github.com/fog/fog
[14]https://github.com/gollum/gollum
[15]https://github.com/sensu/sensu/pull/305
[16]https://github.com/sensu/sensu/pull/303

of problem may indicate the use of naïve tests, as the tests relied on specific values that would be present at Redis at a specific time. Inadequate testing is one of the main issues in a CI environment (Pinto et al., 2018), potentially due to the lack of a testing culture suitable to CI practices.

Concerning the GOLLUM project, we find that the most probable reason for the drop in coverage is related to problems such as the one described in issue #649[17]. The issue exposes that the test suite relies on a specific version of *libxml2* in the CI server (TRAVISCI). Thus, mismatch between library versions caused the breakage of tests, which impacted the computation of code coverage. Indeed, existing research hints that dependency management and version changes are two critical issues when building and testing software systems (Pinto et al., 2018).

Concerning the AFRAME project, we found issues where developers were troubleshooting CI build failures. For instance, issue #515[18] shows that setting up a CI server can be challenging. For example, test failures occurred due to problems integrating WebGL into the CI server, leading to a drop in coverage levels. The lack of integration between WebGL and the CI server forced developers to inactivate related tests, leading to a decrease in code coverage.

Reasons for the increase in coverage. Next, we investigate the substantial increase in code coverage levels during the after-CI period for some of our CI projects. In particular, we observe sharp increases in code coverage in the following projects: SERVERLESS[19], and KIVY[20]. Figure 3.15 shows the coverage evolution for these projects.

The main reason for the increase in coverage levels is the improvements of test suites. Projects like SERVERLESS represent this trend; note the plot on the right side of Figure 3.15. For example, in the SERVERLESS project, it is possible to find issues such

[17]https://github.com/gollum/gollum/issues/649
[18]https://github.com/aframevr/aframe/issues/515
[19]https://github.com/serverless/serverless
[20]https://github.com/kivy/kivy

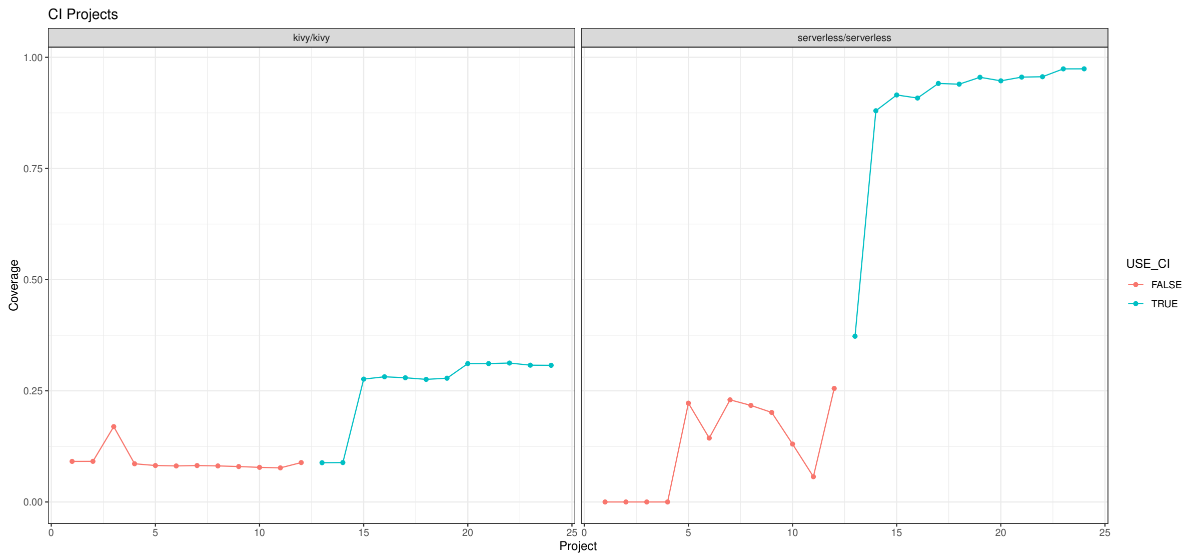

Figure 3.15: Projects with positive change in coverage after CI.

as the following: Issue #1089[21] ("Write unit tests for Plugin methods"), issue #1203[22] ("Deep test coverage for awsCompileFunctions plugin"), and issue #3275[23] ("Return code coverage to proud 100%"), i.e., issues addressing the lack of tests or the need to improve existing tests.

Lastly, in the Kivy project (the plot on the left side of Figure 3.15), we observe an 89% increase in issue reports and pull requests targeting test suites after adopting CI. Pull requests such as #721[24] ("add a test suite for App(), only two tests for now"), and commits such as #3f76ea8[25] ("Added tests toward full coverage") became a common-place.

Findings 3.3.3

CI projects tend to significantly increase the types of code changes that improve code coverage (e.g., newly added and covered lines). NOCI projects tend to lose coverage on existing lines.

[21]https://github.com/serverless/serverless/issues/1089
[22]https://github.com/serverless/serverless/issues/1203
[23]https://github.com/serverless/serverless/issues/3275
[24]https://github.com/kivy/kivy/pull/721
[25]https://github.com/kivy/kivy/commit/3f76ea8

3.4 Discussions and Implications

3.4.1 Implications for Practitioners

The result of RQ3 (Section 3.3.3) indicates that projects that use CI generally have better coverage than NOCI projects even before the adoption of CI. We also obtained evidence that projects that do not adopt CI tend to lose coverage, especially on existing lines. We conjecture that projects that happen to take care of their code coverage may be better positioned to eventually adopt CI, which may boost the code coverage even more (see RQ2 (Section 3.3.2) and RQ3 (Section 3.3.3). Therefore, we suggest that, even if a project does not plan to adopt CI, minding its code coverage through a quality and well maintained test suite can only help the project.

In RQ3 (Section 3.3.3), we study the different reasons (i.e., the code change types) as to why coverage can increase or decrease in CI and NOCI projects. Given that code coverage may fluctuate for a variety of reasons, we believe current tools should show how the change in coverage has occurred over time. We conjecture that proper visualizations of such changes over time (e.g., a project has increased its code coverage due to an exceptional coverage of new lines) may encourage developers to maintain good testing practices and increase their confidence, similarly to how CI is already known for increasing the confidence of developers (Soares et al., 2022). Furthermore, a finer-grained analysis of how coverage evolves may equip developers to better understand the current state of their automated tests.

3.4.2 Implications for Researchers

Our research opens avenues for future empirical studies that address other coverage metrics such as branch, MC/DC, method, file, and path coverage. We acknowledge that the relationship between CI and other coverage metrics can differ from the results obtained for statement coverage and CI, which was investigated in our research. In addition, our study uses data from public open-source projects, so further studies can consider commercial projects. This is important as non-commercial projects may have a lower release frequency than commercial projects, and their CI pipelines may be completely different. Moreover, similarly to Felidré et al. (Felidré et al., 2019), a possible direction for future work lies in investigating whether code coverage evolution is generally correlated with the adherence to CI practices. Lastly, future studies can investigate whether better visualizations for monitoring code coverage may influence the evolution of code coverage (e.g., as developers would me more aware of how their code coverage evolves). Lastly, future work may consider the expansion of our analysis to include project using GitHub Actions.

3.5 Threats to the Validity

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

3.5.1 Construct Validity

Construct validity concerns the research design, i.e., whether it is inaccurate in a way that it does not measure what it claims or purports. The statistical methods, including two non-parametric tests Wilcoxon test (Wilks, 2011) and Cliff's delta (Macbeth, Razumiejczyk, and Ledesma, 2011), and Regression Discontinuity Design (RDD) (Imbens and Lemieux, 2008) fulfill the requirements of this work once we are researching for an association between CI and coverage and not necessarily causality. The potential threats related to constructing validity mainly include the processes of collecting data from projects. We mitigate this threat by relying on a consolidated dataset constructed and validated by other studies (Bernardo, Costa, and Kulesza, 2018; Nery, Costa, and Kulesza, 2019; Guo and Leitner, 2019).

3.5.2 Internal Validity

Internal threats are concerned with the ability to draw conclusions from the relationship between the dependent variable (code coverage) and independent variables (e.g., use of CI, number of projects, and versions). Over the years, researchers have proposed several coverage-based criteria, including statement coverage, branch coverage, modified condition coverage, and others. We focused on statement coverage and recognized that the relationship between other coverage metrics and CI could differ. However, this is a starting study, and we plan to extend it to other coverage metrics in the future. Furthermore, we only investigate open-source projects whose testing activities may differ from commercial software. Given that our work is an exploratory study, we did not seek causal relationships necessarily. Hence this category of risks is less important. Regardless, we plan to include non-OSS in the study in the future.

3.5.3 External Validity

External threats are concerned with the extent to which we can generalize our results (Perry, Porter, and Votta, 2000). In this work, we analyzed 1,440 versions of 60 popular open-source projects from GitHub. All CI projects adopt a popular CI server, i.e., `TravisCI`. We excluded projects that do not use the most used CI services for obtaining the NOCI dataset. However, we acknowledge that our results are restricted to our projects' setup (and potentially similar setups in other projects). Further analysis (e.g., additional projects) should be performed in future work, which we are planning to do.

3.6 Conclusions

We investigated the potential association between CI and code coverage evolution. We compared 30 CI and 30 NOCI projects to study whether the adoption of CI is associated with better code coverage. In total, we studied 1,440 versions.

Our analysis showed that continuous integration can be empirically associated with better code coverage. The coverage values after the adoption of CI were significantly higher than before the adoption of CI. Moreover, after CI is adopted, the types of code changes that increase code coverage also increased across our studied projects. We also identified that CI projects revealed a lower tendency of losing covered lines when compared to NOCI projects. Lastly, our study demonstrated that, after the adoption of CI, there was no difference in the prioritization to cover new or existing code. Indeed, we did not observe a statistically significant difference between new or existing code when it came to becoming covered. Conversely, when we compared CI projects (before and after CI was adopted) to NOCI projects, we noticed that the coverage of newly added lines was significantly higher in CI projects.

Overall, our work does not only revealed that the adoption of CI is associated with a stronger code coverage, but also revealed that projects that eventually adopt CI tend to maintain a significantly better code coverage than NOCI projects throughout their lifetimes. Our recommendation is that, if CI cannot be adopted from the beginning, maintaining good levels of code coverage may place the project in a better position to heap the benefits of CI at a later stage.

4 Uncovering the Relationship Between Continuous Integration and Code Coverage: An Exploratory Investigation

Despite the widespread adoption of continuous integration (CI) and code coverage, there are still many unanswered questions regarding their relationship. In Chapter 3, we quantitatively investigated this relationship and presented a positive association between CI and code coverage. However, we did not explore the underlying mechanisms that drive this relationship. To address this gap in knowledge and gain a comprehensive understanding of developers' use of CI and coverage, we conducted a two-part qualitative study consisting of a preliminary exploratory phase based on a survey, followed by a document analysis (see Chapter 5).

Our main aim is to gain a comprehensive understanding of the impact of continuous integration (CI) on code coverage through a survey of developers working on the studied systems. This approach will allow us (i) to gather additional qualitative insights from developers that cannot be obtained through quantitative analysis alone; (ii) to examine how developers utilize code coverage in the context of CI projects; and (iii) to validate our quantitative findings by comparing them with the developers' perspectives.

The remainder of this chapter is organized as follows. Section 4.1 introduces the research question that serves as the central focus and objective of our investigation. In Section 4.2, we outline the study design, detailing the methodology and approach employed to address the research questions. Section 4.3 presents the results, highlighting the key findings and outcomes obtained from our study. We acknowledge the limitations and constraints of our research in Section 4.4. Section 4.5 delves into the broader implications for different stakeholders of our study. Finally, in Section 4.6, we draw conclusions.

4.1 Research Questions

The following research questions guide our research:

- **RQ4 - How the code coverage information is used in the CI projects?**
  Rationale: Our research question aims to explore how developers utilize code coverage information in practice, i.e., their day-to-day coding activities. Specifically, we are interested in understanding how valuable developers perceive code coverage when writing and reviewing code.

- **RQ 5 - Does the adoption of Continuous Integration increase the code coverage?**
  Rationale: Qualitative investigation into the relationship between adopting Continuous Integration and increasing code coverage can provide valuable insights that quantitative metrics may not capture. Exploring the subjective experiences and perspectives of developers can reveal nuances and complexities that quantitative analysis alone may not capture. Qualitative research can uncover whether developers perceive an increase in code coverage after CI adoption and, if so, what factors

they attribute this improvement to.

4.2 Study Design

In this work, we developed a survey based on the guidelines proposed by Kitchenham et al. (Kitchenham et al., 2002), which includes six phases: planning, survey creation, target audience definition, evaluation, survey administration, and result analysis. To increase survey participation, we employed several principles recommended in the literature (E. Smith et al., 2013), including personalized invitations, and the use of closed, direct questions. Prior to distributing the survey, we developed an initial draft and sought feedback from colleagues and researchers. Based on their suggestions, we refined some of the questions and explanations to improve their clarity and precision. Participation was voluntary and the estimated time to complete the survey was 10-15 minutes.

Our data collection method involves surveying the developers of the CI projects included in our study. To select the most active contributors, we gathered the names and emails of all contributors using the GitHub Rest API and ranked them by the number of contributions (commits) during the period analyzed in our previous study (see Chapter 3).

We aimed to select the top 20% of contributors (90 out of the total) from each project and expected a response rate of one-third. By focusing on the most active contributors, we hope to gather the most relevant and insightful data about the impact of CI on code coverage in our selected projects. However, due to the insufficient number of responses for our research, we had to modify our initial procedure. As a result, we sent the survey to the top 40% (200) remaining active developers to try to increase the sample size. In the end, we obtained 17 responses, resulting in a 5.86% of response rate.

Table 4.1: Survey questions.

ID	Section	Question
Q1.0	Audience	Have we correctly identified you as someone who has contributed to the Ansible project ?
Q2.0	Demographic	For how many years have you been developing software?
Q2.1	Demographic	How would you describe your main roles in the project you work the most (e.g., core developer, bug-fixer, reviewer) ? Choose all that apply.
Q2.2	Demographic	How would you classify yourself in terms of experience using continuous integration?
Q2.3	Demographic	Have you experienced the process of introducing CI to a project that was not previously following the practice?
Q3.0	Perceptions about CI Adoption and Code Coverage	To what extent do you agree with the following statement: The Ansible project systematically monitors code coverage?
Q3.1	Perceptions about CI Adoption and Code Coverage	How is coverage monitored in Ansible project?
Q3.2	Perceptions about CI Adoption and Code Coverage	Would you support the systematic monitoring of code coverage in your CI pipeline? Why or why not?

Continued on next page

Table 4.1: Survey questions *(continued)*

ID	**Section**	**Question**
Q3.3	Perceptions about CI Adoption and Code Coverage	What are the main tools/frameworks that you have used to monitor code coverage?
Q3.4	Perceptions about CI Adoption and Code Coverage	Do you find code coverage useful when authoring code and/or reviewing code? Why or why not?
Q3.5	Perceptions about CI Adoption and Code Coverage	Do you believe that test suites with high code coverage will typically find more real faults or reduce the search space to find a bug? Explain your reasoning.
Q3.6	Perceptions about CI Adoption and Code Coverage	What code coverage metric do you use in your project (if any)?
Q4.0	Study Results	When analyzing the code coverage of project Ansible, we observed an increasing trend in code coverage after CI was adopted. In your opinion, is this increase in coverage related to the use of CI? If yes, please explain. If not, which are the reasons that you could associate with it?

Continued on next page

Table 4.1: Survey questions *(continued)*

ID	**Section**	**Question**
Q4.1	Study Results	There are many ways in which code coverage can fluctuate. For instance, newly added and uncovered code can reduce the overall coverage, whereas the deletion of existing and uncovered code can increase the overall coverage. In your opinion, do you think CI influences how code coverage fluctuates? If so, in what ways? If not, why not?
Q4.2	Study Results	Are there any development or testing practices/techniques/tools that were introduced after the adoption of CI, which you believe must have influenced the quality of your project? Could you elaborate on the nature of such contribution?
Q4.3	Study Results	Analyzing the evolution of code coverage, we observe notable variations (increase/reduction) in the levels of code coverage across successive versions, as illustrated in graphic above (before and after CI adoption). What are the main reasons that explain such variations (e.g., broken tests, inadequate test culture, code refactoring, test suite improvement) ?

Continued on next page

Table 4.1: Survey questions *(continued)*

ID	Section	Question
Q5.0	Ending Our Questionnaire	Would you like to be informed about our findings and receive additional reports about the code coverage of your project?
Q5.1	Ending Our Questionnaire	Would you be willing to be contacted for an online follow-up interview at a time that is convenient for you?
Q5.2	Ending Our Questionnaire	If you checked yes on one of the two questions above and have not yet left your e-mail, please leave it below.
Q5.3	Ending Our Questionnaire	Do you have further comments for us?

Our survey consisted of 21 questions, 9 of which were open-ended (free-response). Table 4.1 shows the survey questions. The survey was divided into five sections: (1) Audience, (2) Demographic Questions, (3) Perceptions about CI Adoption and Code Coverage, (4) Study Results, and (5) Ending Our Questionnaire.

1. *Audience:* As an initial step, we asked participants to confirm whether they had contributed to the project under investigation. This allowed us to ensure that our sample consisted of individuals with first-hand experience of the project, enabling us to gather valid information about their perceptions and experiences.

2. *Demographic:* This group of questions aimed to gather information about the technical backgrounds, expertise, and experience of the participants. It included four demographic questions, namely, the number of years of experience in software

development (Q2.0), the role they played in the project they worked on the most (Q2.1), their familiarity with CI techniques (Q2.2), and whether they have ever configured a project to use CI techniques (Q2.3).

3. *Perceptions about CI Adoption and Code Coverage:* In this section, we included seven questions to gather information on participants' perceptions about code coverage monitoring and its impact on their projects. First, we asked for their opinion on an affirmative statement about coverage monitoring in their project (Q3.0), followed by a question on how coverage is monitored in their project (Q3.1). Then, we inquired if they consider supporting systematic monitoring of code coverage in their CI pipeline and their reasons for that decision (Q3.2). We also asked about the main tools/frameworks they have used to monitor code coverage (Q3.3) and whether they found code coverage helpful when authoring code and/or reviewing code. If so, we asked them to elaborate on why they believe that (Q3.4). Furthermore, we asked for their opinions on the usage of code coverage as a metric to assess test suite effectiveness (Q3.5). Finally, we asked if they use any code coverage metric in their project and, if yes, which one they use (Q3.6).

4. *Study Results:* We have questions about the results of our previous study (see Chapter 3). In the first question (Q4.0), we presented charts related to each project that depict the code coverage trends before and after the adoption of CI. These charts were used to help participants reason about the behavior of code coverage in their project. We began by asking for their opinions on the observed coverage trends in their project. Next, we asked for their opinions on how the adoption of CI may have influenced the fluctuations in code coverage (Q4.1), and whether they believe that any development or testing practices, techniques, or tools introduced after the adoption of CI can impact software quality. If so, we asked them to elaborate on why they believe this is the case (Q4.2). Finally, we asked participants to explain

any notable variations in the levels of code coverage on the charts and to discuss their motivations for using CI techniques (Q4.3).

5. *Ending Our Questionnaire:* In this section, we asked participants if they would like to be informed about our findings and receive additional reports about the code coverage of their project (Q5.0) and their availability for an interview (Q5.1). Lastly, we reminded them to leave an email for contact (Q5.2).

To give respondents the most flexibility, no question was mandatory. To avoid unreliable responses, developers were free to skip questions that they did not feel confident to answer.

4.3 Study Results

4.3.1 Demographics Analysis

In this section, we present basic demographic information about the participants in our study. Figure 4.1 displays the distribution of years of development experience among the participants in our survey (Q2.0). We observed that all participants have at least five years of experience in software development, and a significant number of them reported having twenty or more years of experience.

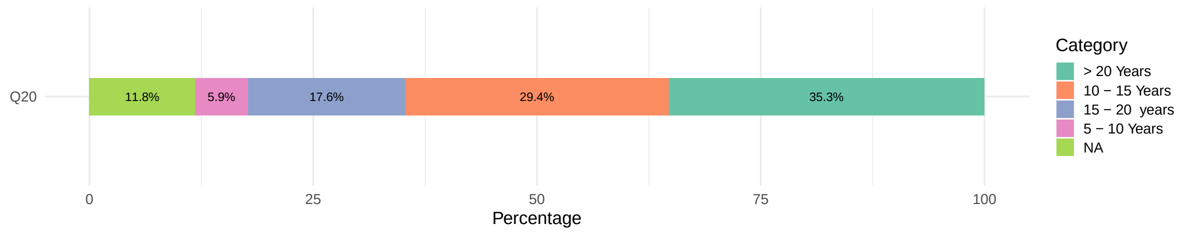

Figure 4.1: Experience across participants.

Among the respondents, 35.3% reported working in software development for more than twenty years, while 17.6% had between 10-15 years of experience, and 5.9%

had between 5-10 years of experience. A small proportion (11.8%) did not respond to this question. These results indicate that the survey was answered by a highly experienced group of software development professionals.

We also asked developers to indicate the roles they perform in the projects they work on the most (Q2.1). Two participants did not provide any answer, and one participant declared that they worked exclusively as a tester. The remaining participants declared that they work in more than one role. The most common role mentioned was *Back-end/Core Developer*, with 30% of the respondents, followed by *DevOps Developer*, with 15% of the respondents reporting this role. Interestingly, one participant declared that they work as a CEO, which was unexpected.

This result confirms the diversity of the audience for this survey, as all participants who provided an answer stated that they work or have worked in a role related to development. The roles mentioned included developers, bug fixers, testers, reviewers, and others, highlighting the varied backgrounds and experiences of those who responded to the survey.

The Q2.2 question inquired about the participants' experience in using continuous integration. The majority of respondents reported having experience with CI, which is promising for our study as we can leverage their experience for the next set of questions. Figure 4.2 summarizes the results of this demographic question.

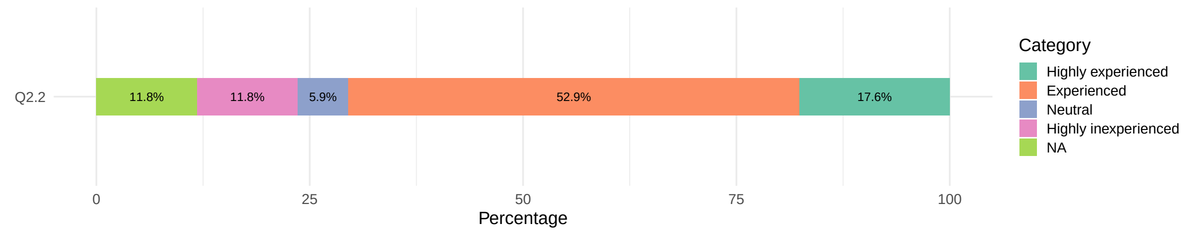

Figure 4.2: Experience of CI usage across participants.

As Figure 4.2 suggests, 17.6% of the respondents said that they are *"Highly Experienced"* with CI techniques, 52.9% said that they are *"Experienced"*, 11.8% are *"Highly* *inexperienced"*, 5.5% are *"Neutral"*, and no respondent mentioned being *"Inexperienced"*. It is worth noting that 11.8% of the participants did not respond to this question. These results indicate that the survey was answered by an experienced group in CI techniques.

Lastly, in our Q2.3 question, we asked participants whether they had experience introducing CI to a project that was not previously following CI. Figure 4.3 presents an overview of our results.

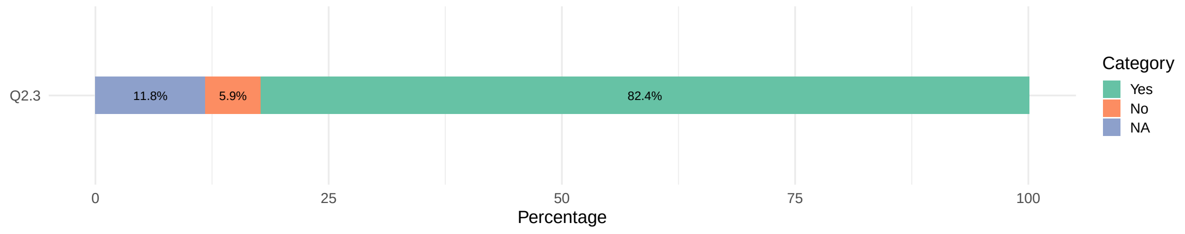

Figure 4.3: Experience level with adopting CI among participants.

Among the respondents, 82.4% of the participants mentioned that they have some experience in adopting the CI process, whereas 5.9% of the participants declared that they do not have any experience in the CI adoption process. It is noteworthy that 11.8% of the participants did not respond to this question. This experience is crucial for our research purposes, as it can provide valuable insights and confirm some of the findings from our previous study.

Although the number of participants in our study is relatively small, it is important to note that the group consists of highly experienced developers with a strong background in the adoption of CI practices. This gives us confidence that the insights and perspectives gained from our survey are valuable.

4.3.2 RQ4 - How the code coverage information is used in the CI projects?

We analyzed the answers from the *perceptions about CI adoption and code coverage* group of questions from the previous section to answer this research question. Combining the answers for these questions, we can provide a comprehensive overview of how developers use code coverage in the context of CI.

Initially, we asked participants to express their level of agreement with the statement *"The project XXX systematically monitors code coverage"* (Q3.0). As we can see in Figure 4.4, the majority of participants agreed with this statements.

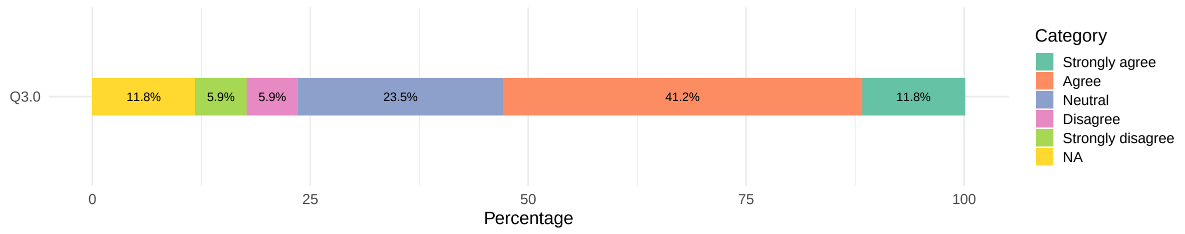

Figure 4.4: Participants' agreement with the statement on systematic code coverage monitoring.

Among the participants, 53% indicated agreement with the statement that their project systematically monitors code coverage (41.2% *"Agree"* and 11.8% *"Strongly agree"*), while 11.8% indicated disagreement (5.9% *"Disagree"* and 5.9% *"Strongly disagree"*). A small proportion (11.8%) did not respond to the question. This finding aligns with our observations from the quantitative study presented in Chapter 3, where we speculated that the majority of projects in the CI dataset have a culture of monitoring code coverage.

In the following three questions (Q3.1, Q3.2, and Q3.3), we asked respondents how code coverage monitoring occurs in their projects. In question Q3.1, we asked participants about how they monitor code coverage. We found that there were three main approaches:

- **Using coverage services** (cloud-based solution): 23.5% of participants reported using a cloud-based tool that provides code coverage reporting and analysis. The most commonly used tool was *Codecov*.

- **Using coverage tools**: Among the participants, 29.4% monitored coverage using reports generated by coverage tools.
- **Using mixed approach**: 5.9% reported using a mixed approach, where coverage monitoring was done using both traditional coverage reports and a cloud-based service.

The remaining participants either did not know how to answer this question or did not provide any feedback. However, it is noteworthy that none of the participants explicitly declared that they do not monitor code coverage.

It is worth noting that out of the top 50 most popular Python software projects hosted on GitHub, a significant majority (86%) use *Coverage.py* (Hora, 2021). However, there is currently a trend towards presenting online coverage reports, integrating them into CI/CD workflows, and facilitating code review. Tools like Codecov[1] and Coveralls[2] are capable of generating detailed coverage analysis for most programming languages, making it easier for developers to monitor and analyze their code coverage. Our findings are in line with this trend.

In question Q3.2, we further explored if the participants support the idea of systematically monitoring code coverage in their CI pipelines. As shown in Figure 4.5, 35.6% of the participants responded positively, stating that they do support systematic monitoring. On the other hand, 29.45% responded negatively, indicating that they do not support it. 5.9% of participants said that it depends on the project context, and 29.4% did not provide any answer.

Upon analyzing the responses of participants who did not support the systematic

[1]https://coverage.readthedocs.io
[2]https://coveralls.io

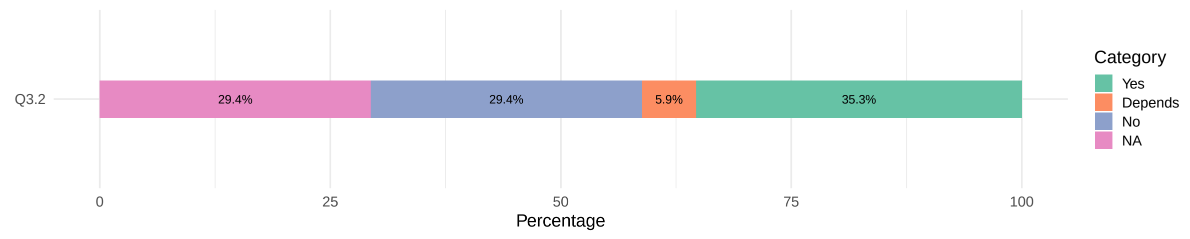

Figure 4.5: Participants' opinions on systematic code coverage monitoring in their CI pipeline (Q3.2).

monitoring of code coverage in their CI pipeline, we found that the most commonly cited reason was the high resource consumption and effort required to implement such a process. *P8* mentioned that *"Not systematic. CI resources spend energy and we need to save it"*. This observation is intriguing as it suggests that developers are not solely focused on productivity in software development, but also on resource constraints concerns. Other participants shared similar sentiments, with *P7* stating that it requires *"too much effort compared to the gain"*, and *P14* describing that it takes *"a lot of time"*.

Analyzing the responses of participants who supported systematically monitoring code coverage in their CI pipeline, the most frequently cited benefit was the ability to detect unmaintained parts of the codebase. *P4* highlighted the importance of systematically monitoring code coverage in the CI pipeline to avoid retaining dead branches of code or untested parts, stating: *"To ensure [we are] not keeping dead branches of code, or untested parts"*. However, the potential for improving software quality was another benefit mentioned, although no additional details were provided by respondents who mentioned this benefit.

Lastly, we inquired about the tools and frameworks that participants have used to monitor code coverage on a daily basis (Q3.3). The majority of respondents, representing 45.8% of all mentions, reported using *Coverage.py*. Tied for second place were PhpUnit, JaCoCo, and OpenClover with 8.3%.

In the Q3.4 question of our survey, we aimed to determine whether code coverage is considered useful when authoring or reviewing code. Our analysis revealed that 80% of respondents considered code coverage during the authoring process, while 13.3% did not. 6.7% of participants did not provide any answer. We found a similar result when analyzing the use of coverage during the code review process. The majority of respondents (86.7%) made use of code coverage when reviewing code, while only 6.7% did not consider coverage information, and another 6.7% did not provide any response.

In question Q3.5, we examined whether participants believed that test suites with high code coverage are more effective in finding faults or at least reducing the search space for a bug. Our analysis of the responses revealed that 33.3% of participants stated that test suites with high code coverage are more effective, while 26.7% declared that they do not believe this to be the case. 6.7% mentioned that the effectiveness of high code coverage test suites depends on test quality, and another 13.3% were unsure or did not provide an answer. Figure 4.6 presents an overview of our results.

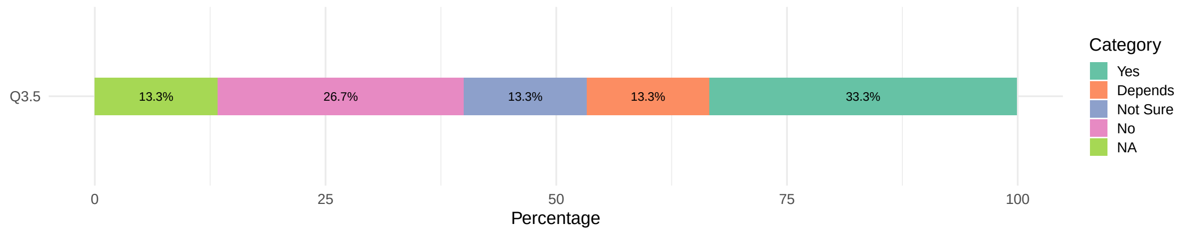

Figure 4.6: Participants' views on the effectiveness of high code coverage test suites in fault-finding (Q3.5).

Our analysis revealed technical reasons that could explain why some participants did not consider code coverage as an effective test indicator or as a way to reduce the search space for faults. Among these reasons, inadequate testing was cited as a common issue. Participants mentioned that code coverage done by "dummy tests" is not helpful and that tests with high coverage might miss edge cases or unexpected states that cause faults. *P10* argued that coverage only finds faults in simple, well-written standalone code, which is not representative of most code. They believe that bugs tend to be in higher-level logic and are caused by unexpected states from side effects, which code coverage does not measure.

As we can see in Figure 4.6, we observed that some respondents had different views regarding the effectiveness of code coverage in fault-finding. For instance, *P15* mentioned that in their experience, achieving 100% code coverage requires re-reading the code and creating test cases, P15 stated: *"Yes. In my experience, that's true. Getting to 100% coverage requires re-reading the code and creating test-cases. That's when most obvious bugs are eliminated"*. Similarly, *P12* noted that code coverage can flag bugs by identifying untested code paths, *"yes, I have observed that already coverage itself was able to identify a bug by showing that a code path isn't followed"*.

It is worth noting that most of participants suggested not to use high code coverage as a stipulated quality target. As argued by *P11*, *"No. High code coverage often means that code is structured to be covered, and protect against dead code"*. This suggests that while coverage may be useful in detecting dead code, it may not be adequate for finding faults in live production code. Furthermore, *P13* emphasized that the effectiveness of code coverage in fault-finding also depends on the quality of tests written. As *P13* stated, *"It depends on the tests. You can write tests with 100% coverage but not test for any edge cases where multiple paths are part of the problem"*. This highlights the importance of considering the comprehensiveness of test cases rather than just the code coverage percentage.

In question Q3.6, we asked participants about the code coverage metric they use. The most common metric used by participants was Statement/Line Coverage with 39.1%, with participants citing its simplicity and speed as reasons for its popularity. Branch/Decision Coverage and Functional Coverage tied for second place at 17.4%, followed by Statement Frequency Coverage at 8.7% and Path Coverage at 4.3%. Additionally, 8.7% of participants reported not using any coverage metric, while 4.3% did not provide an answer.

These findings are consistent with those reported by Vassallo et al. (Vassallo et al., 2016b), as we similarly found that statement coverage was the most commonly mentioned metric among our survey respondents. This suggests that statement coverage remains a widely recognized and valued measure of testing adequacy in software development.

Findings 4.3.1

Our study found that developers make use of coverage information both during the code authoring stage and code reviews. According to the participants, code coverage is useful for identifying trivial bugs, detecting uncovered code paths, and potentially preventing bugs and controlling side effects when refactoring or changing the code's behavior, thus improving overall software quality. Additionally, they systematically monitor code coverage, with the primary coverage metric being statement/line coverage.

4.3.3 RQ5 - Does the adoption of Continuous Integration increase the code coverage of software releases?

To address RQ5, we analyzed the claims made by participants in the results section of our survey. While question Q4.0 directly asked if adopting Continuous Integration increased code coverage, we also included the other questions from the *"Study Results"* section to isolate certain variables and provide a more accurate answer.

In question Q4.0, we specifically inquired participants about the positive association between code coverage and CI, as identified in our study in Chapter 3. Figure 4.7 shows that 46.7% of respondents confirmed the increase in coverage is related to CI adoption, while 6.7% disagreed with our result. 26.7% reported that they were unsure, and 20% did not provide an answer.

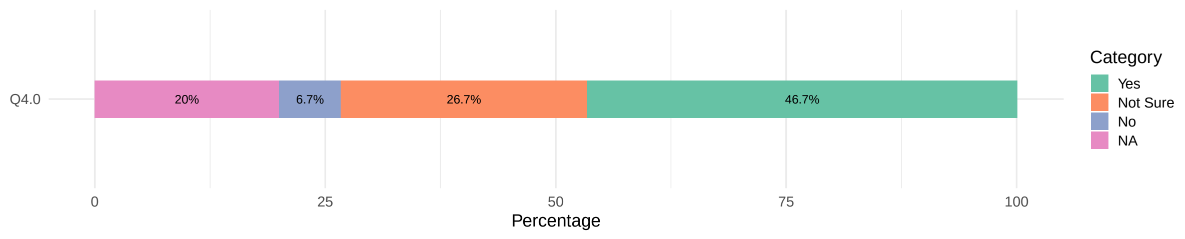

Figure 4.7: Participants' views on positive association between Coverage and CI (Q4.0).

Our findings from research questions **RQ1** (Section 3.3.1) and **RQ2** (Section 3.3.2) are in line with these outcomes, as we had hypothesized a positive association between the increase of code coverage and CI, and the majority of participants confirmed the existence of such a relationship.

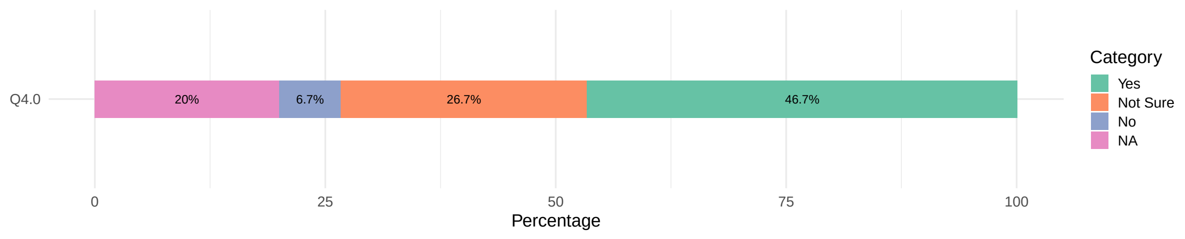

Figure 4.8: Participants' views association between coverage fluctuation and CI (Q4.1).

Our results, as presented in Figure 4.8, indicate that 46.7% of the participants stated that CI has an impact on coverage fluctuation, while 6.7% did not agree with this statement. 26.7% were unsure about the relationship, and 20% did not provide a response. These findings are aligned with the results reported in Section 3.3.2.

One scenario that may arise in this landscape is the need to control code coverage fluctuations due to project requirements. Some participants shared that their companies set boundaries for code coverage percentages. For example, *P9* mentioned that *"I've worked for companies that set boundaries for code coverage percentage. Under a certain percentage didn't allow any drop in coverage; within a certain range only permitted fluctuation of +/- 1% coverage; above a certain percentage there was no concern about slight changes in fluctuation. Drops in coverage were not permitted to be merged without* *adding tests to improve code coverage"*. *P15* shared a similar perspective, stating that *"If the project is enforcing a certain amount of coverage, there's way less fluctuation"*. Therefore, some projects may adopt policies to avoid significant code coverage fluctuations after CI adoption.

The most frequently mentioned impact of CI on code coverage is related to improved monitoring. Many participants highlighted that, in addition to project policies that may require a certain level of code coverage for code contributions, CI makes it easy to track changes in code coverage over time, enabling better maintenance. On the other hand, participants who did not believe that CI had any impact on coverage fluctuations generally did not provide detailed explanations.

To identify the factors that can impact software quality, we asked the participants about any development, testing practices, techniques, or tools introduced after adopting CI that they believed influenced the quality of their projects (Q4.2). We identified four development and testing practices/techniques/tools that participants believe have positively influenced the quality of their projects after adopting CI. These include automatic tests, static analysis, functional tests, and continuous delivery.

Participants most frequently mentioned automated tests as a development or testing practice that positively impacted their software quality. One participant mentioned that automated testing, although not specifically coverage, gave developers more confidence to advance at a faster pace without fearing breaking anything in production. Another respondent stated that being able to automatically test on many platforms and specific problem areas, such as successful compilation and installation, helped improve their software quality.

One participant provided an interesting insight into the use of functional tests, stating that they go beyond just unit tests and include tests that represent real scenarios. These tests are run in a simplified environment to reduce the cost of full integration tests. The participant's comment highlights the importance of testing not only individual units but also the interactions between components in a real-world scenario. By using functional tests, teams can gain a more comprehensive understanding of how their code performs in practical situations, without incurring the high costs associated with full integration testing. In addition, *P5* mentioned the importance of continuous delivery, stating that *"CD as that has really put the spotlight on producing code that actually really works on deployment"*. Finally, a participant mentioned mutation testing, a technique that can test the quality of test suites by introducing faults into the code and observing if the tests can detect them.

Regarding tools, the most commonly mentioned one was *linter*. Participants highlighted its role in detecting and preventing coding errors and enforcing code quality standards. There were also mentions of automated AST-based refactoring tools and automated code-style fixers. These tools were seen as helpful in improving code maintainability and reducing technical debt.

As part of the study, we asked participants to examine the changes in code coverage across successive versions of their projects, before and after the adoption of continuous integration (CI) (Q4.3). We presented the variations in code coverage levels in a chart and asked participants to identify the main reasons for such changes, including broken tests, inadequate test culture, code refactoring, and test suite improvement.

An inadequate test culture was cited as the primary reason for negative variations in code coverage levels across successive versions, as reported by the majority of respondents. This finding is consistent with the results of our manual investigation, which was presented in Section 3.3.3. However, during our analysis, we encountered two noteworthy quotes from participants *P4* and *P5*.

According to a statement by *P4*, legacy code was the reason for variations in coverage due to the effort required to set up tests for older code, as described by *P4*:

"Legacy. Nobody wants to set up tests for older code, as it leads to much yack shaving... ". There could be several reasons why people might avoid setting up tests for older code. One reason could be that the code was written a long time ago, and the original developers may no longer be available to guide how to set up the tests. Furthermore, inadequate documentation can make it challenging to comprehend how the code operates and create effective tests for it. Another reason is that testing older code can be time-consuming and expensive. In some cases, it may require setting up an entirely new development environment or toolchain, which can be a significant investment of time and resources. Additionally, the code may not have been designed with testing in mind, making it difficult to isolate individual components and write effective tests. Legacy code can also lead to a phenomenon known as "yak shaving," which refers to the process of performing a series of tasks that are not directly related to the desired outcome but are necessary to achieve it. This can happen when working with older code because the code may depend on outdated libraries or technologies or rely on specific hardware or infrastructure that is no longer available. Consequently, developers may find themselves spending a lot of time and effort "shaving yaks," i.e., dealing with these ancillary tasks, instead of actually making progress on the code itself. It can be frustrating and time-consuming and may discourage developers from working with older code in the first place.

P5 highlighted that fatigue can negatively impact code coverage levels as writing tests can become monotonous and tedious over time. As *P5* describes it, *"As the novelty of gamification fades, writing tests become a chore"*. This refers to the decreasing interest or excitement that developers experience over the use of gamification techniques, such as incorporating game-like elements in the process of writing tests. Several factors, such as becoming accustomed to the technique or feeling that it is no longer effective, may contribute to this decline in interest. Consequently, *P5* is suggesting that the initial motivation to write tests that was boosted by the gamification approach may decline, leading to lower code coverage levels. Furthermore, *P5* highlighted that writing comprehensive tests can also be repetitive and unexciting, which can lead to a lack of motivation and lower code coverage levels. These findings suggest the need for further investigation into the impact of legacy code on code coverage and the required effort to maintain high code coverage levels throughout the software lifecycle.

Regarding positive variations, the primary reason was found to be the maturing process of the project. We also found some mentions to enforce coverage policy after CI adoption.

Findings 4.3.2

Most of the participants (46.7%) attributed the increase in code coverage to the adoption of CI. However, this impression could have arisen from testing practices, techniques, or tools not necessarily related to CI. Therefore, we asked participants if they adopted any practices after adopting CI. All mentioned practices fell under the umbrella of CI practices, which strengthens their claim about the association between CI and coverage. In addition, upon analyzing our answers, participants also associated coverage fluctuation with the proper use of CI practices.

4.4 Limitations

As an empirical study, our work has several limitations that need to be considered. Firstly, our conclusions are based on a limited number of respondents. Although our participants have diverse backgrounds and expertise, including CI adoption experience, they also share certain characteristics, such as having many years of software development experience.

Additionally, a significant limitation of our study is the low number of responses, which limits the generalizability of our findings. Further investigations with larger sample sizes are needed to confirm and extend our results. Besides, our participants are open-source contributors, so our findings may not be generalizable to developers working in other contexts, such as companies or organizations that produce non-open-source software systems.

Regarding the ethical implications of our email invitations, we are confident that the risks and discomforts associated with the invitations are minimal. We took precautions to protect the developers' privacy by not sharing their email addresses and ensuring that each developer was only invited once. Additionally, each developer was free to choose whether or not to participate in the survey. Finally, the developers were unaware of the total number of invitations sent and thus were not subject to any potential issues related to mass email campaigns.

4.5 Implications

This work has implications for different kinds of stakeholders. Here we discuss some of them.

Relevance for Researchers

In our study, we investigated the utilization of code coverage information in CI projects and identified its significant role in the authoring and reviewing process. These findings suggest a promising direction for future research to explore the potential benefits of code coverage information during these stages in more detail.

Participants in our study indicated that the increase in code coverage values, as observed in response to **RQ1** (see Section 3.3.1), is linked to the CI adoption. This finding strengthens our overall conclusions. Furthermore, we suggest that future research should explore whether the evolution of code coverage is generally correlated with adherence to CI practices. According to our participants, the observed increase in code coverage values is attributed to test automation. However, further research is required to substantiate this conclusion. Another potential avenue for future research could investigate the effectiveness of coverage policies to avoid significant coverage fluctuations.

To expand on our findings, our study revealed that developers encounter challenges in maintaining code coverage for legacy code. This highlights the need for future research to investigate strategies and techniques that can help improve coverage for legacy codebases.

Relevance for Practitioners

In our study, we explored how code coverage information is utilized in CI projects and found that it plays a significant role during the authoring and reviewing process. Moreover, our findings suggest that participants believe that leveraging code coverage can enhance software quality, identify uncovered code paths, and potentially prevent bugs. This investigation is particularly crucial for decision-makers such as team leaders, as it can provide insights on how to explore code coverage information to achieve better software quality.

In our study, we perceived a lack of testing culture as the main reason for notable code coverage negative variation, which might motivate practitioners to place additional care in providing testing courses and tools.

Relevance for Tool builders

Our findings suggest that developers take code coverage information into consideration during the authoring and review process. In addition, our work also reveals that there are multiple factors that can contribute to changes in code coverage levels (see Section 3.3.3 for additional context). To address this challenge, tool builders should strive to provide developers with more comprehensive coverage information that not only shows what changed but also explains why coverage levels have increased or decreased.

Our study also has implications for tool builders who create code coverage tools and services. We identified that developers struggle to determine the reasons why code coverage may increase or decrease. While current tools show the change in coverage, they do not provide an explanation for the change. If developers are aware of how their coverage is changing and are not expecting it to change based on their latest pull request, they can then evaluate whether their tests are unreliable or if their latest change impacted the system coverage in unexpected ways. Therefore, tool builders can explore ways to provide more informative and transparent code coverage data to help developers better understand the reasons behind the changes in coverage.

4.6 Conclusions

In our survey study, we aimed to explore the relationship between code coverage and CI by identifying the most active developers from our CI dataset and conducting a qualitative research analysis. Our findings unveiled several intriguing insights into the relationship between code coverage and CI, including some less obvious ones such as a lack of testing culture and the use of coverage during the authoring and review process.

For future research, we plan to broaden the scope of our study by gathering more responses from a wider range of projects, including those that do not use CI. We also aim to seek out additional CI projects to increase the number of responses in both sets. Additionally, we plan to conduct semi-structured interviews with a selected group of developers from our CI dataset to gain further insights and validate our findings. Furthermore, we intend to triangulate our results with additional data sources to increase the robustness of our conclusions.

Another avenue for future research could be to investigate the impact of code coverage on software maintenance and evolution, and whether it can improve the understandability and maintainability of code. Finally, another possible direction for future work is to explore other code coverage metrics and techniques to gain a more comprehensive understanding of their usefulness in the context of CI.

5 Investigating Discussions on Code Coverage in CI-Enabled Projects: An Exploratory Document Analysis

Previous research has demonstrated the benefits of continuous integration (CI) in enhancing the overall quality of software projects (Soares et al., 2022). However, the actual interplay between executing automated builds on a CI server and coverage remains unexplored. We aim to address this gap by empirically investigating the interplay between pull request discussions and the use of code coverage in CI projects.

This chapter represents the second part of our two-part qualitative study on the relationship between CI and code coverage. In Chapter 4, we presented a initial exploratory analysis.

Building upon the positive association between Continuous Integration (CI) and code coverage demonstrated in our quantitative study (Chapter 3), we seek to delve deeper into the specific aspects of code coverage that developers discuss during pull requests. Since pull requests trigger CI builds, by gaining insights into the discussions regarding code coverage during pull request conversations, we can better understand developers' approaches to code coverage in the context of CI, identify challenges they encounter, and explore effective strategies to address these challenges.

In the next section, Section 5.1, we explain the design of our empirical study. The study procedure is described in Section 5.2. Section 5.3 presents the results and discussions. The threats to the validity are discussed in Section 5.5. Finally, we draw conclusions and avenues for future work in Section 5.6.

5.1 Study Setup

In this section, we explain the design of our study. We describe the data collection process and the adopted methodology for each investigated research question. Figure 5.1 provides an overview of all steps involved in our project selection approach, which are described over the next subsections.

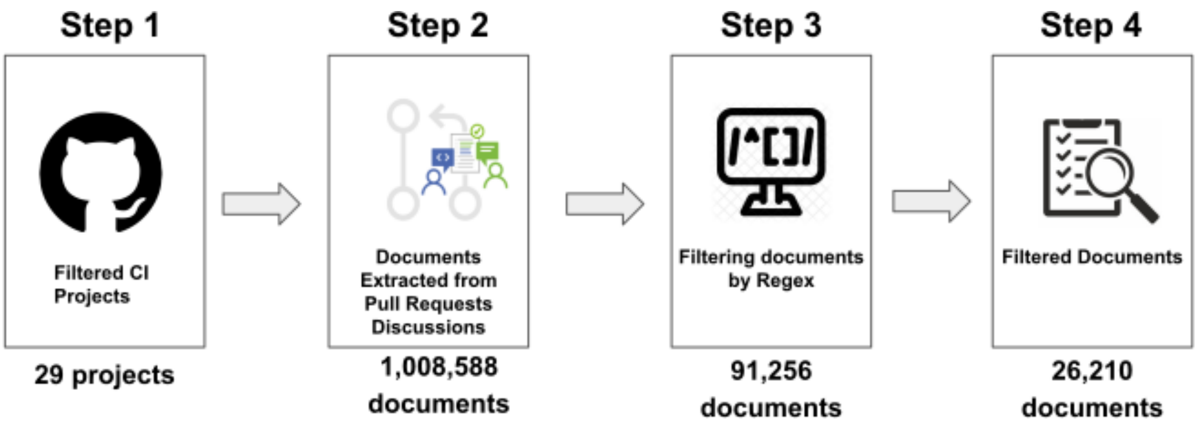

Figure 5.1: An overview of our methodology.

5.1.1 Projects Selection

To carry out our empirical investigation, we need to generate a dataset containing many pull request discussions from different consolidated projects that adopted CI at some point in their history. The dataset should exclude non-relevant projects (e.g., projects with sporadic traces of activity and contributions). The projects in the CI dataset presented in Section 3.2.1 fulfilled all these requirements. However, at the time of our study, the authors of the *Faker.js* project initiated a new project from scratch and removed the old project from GitHub. Because of that, we removed this project from our final dataset (**Step 1** of the Figure 5.1).

5.1.2 Data Collection

In **Step 2** (Figure 5.1), our goal was to collect the pull request discussions for each project. Since all projects are hosted on GitHub, we used the GitHub Restful API[1] to collect all data associated with its pull requests.

Each pull request contains a series of conversations (i.e., comments) discussing a few topics concerning the committed code. We consider the whole sequence of conversations associated with a pull request and the pull request itself as a document. We collect 1,008,588 conversation documents from 104.785 users for our study.

All collected data were saved in a relational database to facilitate future querying.[2]

5.1.3 Data Filtering

Since our aim is to investigate the discussions about code coverage among developers, we need to filter the documents that contain such discussions. To identify these code coverage-related conversations from the documents, we have crafted the following regular expressions:

```
"(\y(test(s)?) | (\ycover)”                                    (5.1)
```

The above regular expression searches for words related to tests and coverage.

[1]https://developer.github.com/v3/
[2]The relational database used was PostgreSQL 15.2

Specifically, it matches the word "*test*" or "*tests*" surrounded by word boundaries, or the word "*cover*" also surrounded by word boundaries. The `\y` in the regular expression is used to match word boundaries, ensuring that the regular expression matches only the exact word forms specified. For example, the regular expression will match "*coverage*" or "*covering*", but not "*discovery*", because the word boundary modifier at the beginner of (`\ycover`) part. The | operator allows the regular expression to match either the first or the second expression.

Consequently, any document that satisfies the aforementioned regular expression is considered eligible for analysis. Following the implementation of these filters, **Step 3** in Figure 5.1 reveals that 91,256 documents remained out of the original 1,008,588 selected documents. However, not all matched documents are necessarily related to coverage discussions. In other words, the regular expression may produce false-positive errors.

We manually inspected each candidate document to eliminate any erroneous matches in **Step 4**, as shown in Figure 5.1. At last, a dataset comprising 26,210 documents was assembled after applying all steps.

5.2 Analysis Procedure

Our objective is to gain insight into the topics and issues discussed by developers regarding code coverage. To achieve this, we transitioned from the quantitative domain and adopted a qualitative approach called Document Analysis (Bowen, 2009). One of the primary advantages of this approach is that documents are considered "unobtrusive" and "non-reactive", indicating that they remain stable and can be read and reviewed numerous times without being impacted by the research process (Bowen, 2009).

Our document analysis methodology, depicted in Figures 5.2 and 5.3, comprises two main phases: (i) an inductive phase; and (ii) a deductive phase.

5.2.1 Inductive Phase

The first phase employs an inductive approach to identify the main discussion topics among developers regarding code coverage. To this end, we considered a representative sample with a confidence level of 95% and a confidence interval of 5%, resulting in samples with 380 documents selected from a population of 26,210 documents.

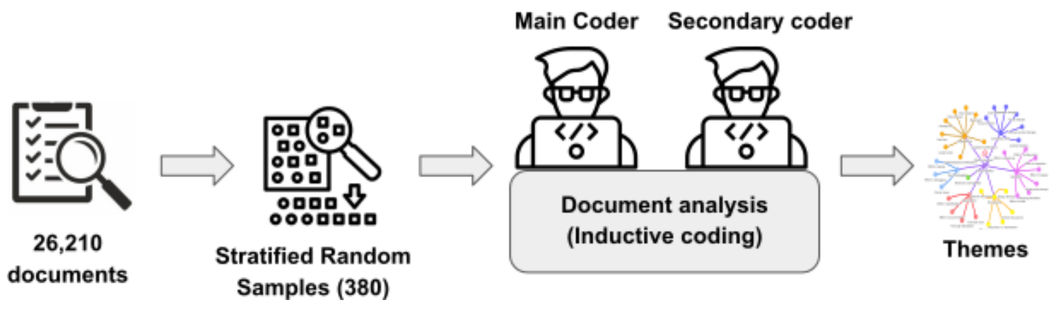
Main Coder
Secondary coder
26,210 documents
Stratified Random Samples (380)
Document analysis (Inductive coding)
Themes

Figure 5.2: An overview of the Inductive phase.

Subsequently, the main and the secondary coder independently scrutinized all the selected pull request discussions from the same sample and created themes based on the topics discussed in these pull requests. At last, the main coder discussed the themes and their meanings with the secondary coder. After several iterations and reflections, a complete set of themes was generated (see Section 5.3.1 for the complete set of themes).

5.2.2 Deductive Phase

Once the set of themes is created (see Section 5.3.1), the second phase uses the generated themes to guide our deductive analysis. In this deductive analysis, the main coder and a third coder independently assessed all documents from the another representative sample (i.e., a different sample than the one used in the inductive phase). Unlike the previous phase, this phase focuses on measuring the accuracy of our perception of the debate within a pull request comment.

Given that our dataset involves a multi-rater multi-label scenario, we employed both Fleiss's Kappa (Hripcsak and Heitjan, 2002) and Krippendorff's Alpha (α) (Krip-

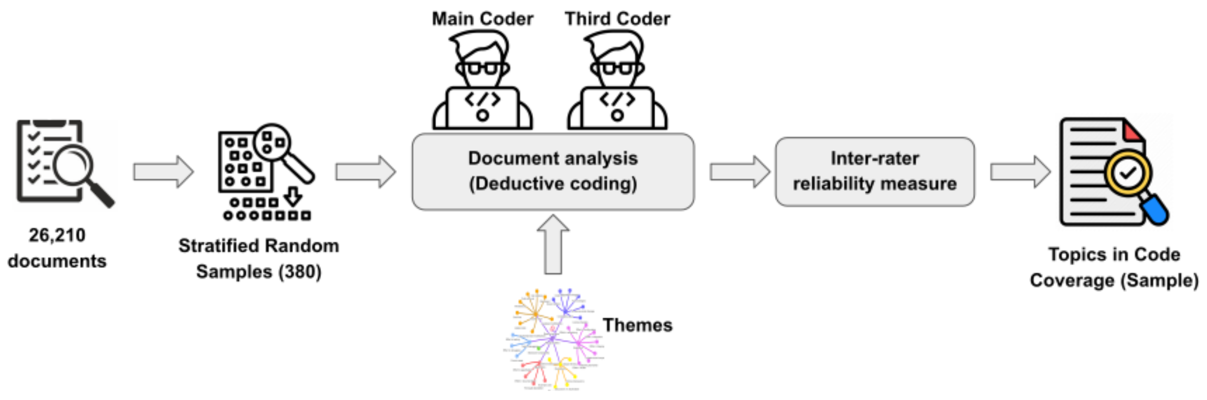
Main Coder
Third Coder
26,210 documents
Stratified Random Samples (380)
Document analysis (Deductive coding)
Inter-rater reliability measure
Topics in Code Coverage (Sample)
Themes

Figure 5.3: An overview of the deductive phase.

pendorff, 1970) as measures of inter-rater reliability. Fleiss's Kappa assesses the level of agreement among raters when multiple categories are being evaluated and considers the possibility of chance agreement. It can accommodate nominal, ordinal, or interval data and is specifically designed for situations in which there are multiple raters and categories that may only partially agree with one another. Similarly, Krippendorff's Alpha is a versatile agreement measure that can handle any level of measurement (nominal, ordinal, interval, or ratio) and any number of raters. Like Fleiss Kappa, it is capable of handling partially agreeing raters. Krippendorff's Alpha offers the advantage of managing missing data, which is especially helpful in cases where certain items have not been rated by all raters.

In conclusion, both Fleiss Kappa and Krippendorff's Alpha are suitable inter-rater reliability measures for cases where multiple raters and multiple categories are being assessed and can accommodate partial agreement among raters.

In the context of a multi-label problem where N documents need to be annotated using themes from a set of themes, we adopted the subsequent methodology for calculating agreement for multi-label annotation. First, we selected a weighted agreement coefficient, such as Krippendorff's Alpha and/or Fleiss Kappa, to measure the extent of agreement among annotators. Next, we determined the method for computing the distance between two sets of themes. Among the potential alternatives, we considered the MASI distance (Bannard and Callison-Burch, 2005) and Jaccard Jaccard (Salton and McGill, 1986).

Jaccard distance measures the dissimilarity between two sets by calculating the ratio of the size of the intersection of the sets to the size of their union. It ranges from 0 (complete similarity) to 1 (complete dissimilarity) (Weng et al., 2008). The formula for Jaccard distance is:

$$J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Where A and B are two sets, $|A \cap B|$ is the size of their intersection, and $|A \cup B|$ is the size of their union.

MASI distance is a variation of Jaccard distance that calculates the similarity between two sets by taking into account the size and overlap of their segments (Frantzi and Ananiadou, 2000). It ranges from 0 (complete similarity) to 1 (complete dissimilarity). The formula for MASI distance is:

$$MASI(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

where A and B are two sets, $|A \cap B|$ is the size of their intersection, $|A|$ is the size of set A, and $|B|$ is the size of set B. MASI distance penalizes for false positives, i.e., items that are present in one set but not in the other, and false negatives, i.e., items that are present in the other set but not in the current set.

The following Table 5.1 summarizes the results obtained for Fleiss Kappa and Krippendorff's Alpha, using the Jaccard and MASI distances.

Table 5.1: Inter-rater reliability measures.

Measure	Jaccard	MASI
Fleiss Kappa	0.6437	0.6159
Krippendorff's Alpha	0.6288	0.6032

Krippendorff's alpha and Fleiss Kappa are both metrics that measure inter-rater agreement, with their values ranging from -1 to 1. A value of 1 represents perfect agreement between the raters, while a value of 0 suggests that the raters are guessing randomly. Negative values indicate that the raters are systematically disagreeing, which means that they are not able to come to a consensus on the coding of the data. While there is no consensus on the cutoff values to use for subjective labels like 'strong' agreement when interpreting Krippendorff's alpha and Fleiss' Kappa, some standard guidelines have been proposed in the literature. In this study, we adopt the interpretation presented in Table 5.2 (Koo and Li, 2016).

Table 5.2: Interpretation of Krippendorff's alpha and Fleiss's Kappa coefficients.

Coefficient	Interpretation
0-0.20	Slight agreement
0.21-0.40	Fair agreement
0.41-0.60	Moderate agreement
0.61-0.80	Substantial agreement
0.81-1.00	Almost perfect agreement

The results demonstrate substantial agreement across all metrics used (Table 5.1). As a result, we have been able to consolidate the theme set and achieve a high level of alignment in the meaning of each theme. Finally, the main coder used the final set of themes to assess all documents in the dataset.

5.3 Results

In this section, we report on the themes that emerged from our document analysis (Section 5.3.1) and provide an in-depth analysis of each theme (Section 5.3.2).

5.3.1 Summary of Themes

In this section, we provide a summary of the themes that emerged from our document analysis of pull request discussions related to code coverage. These themes were identified through an inductive analysis of a representative sample of 380 pull request discussions, gathered from a larger dataset of 26,210 pull request discussions. The themes represent the key topics and issues that developers discussed in relation to code coverage. By summarizing these themes, we aim to provide a high-level overview of the main findings of our study and highlight the most salient points for further discussion and analysis.

Coverage Measurement. This theme emerges whenever there is an explicit indication of the need to measure the coverage or coverage report found at issues or pull requests. This theme represents 16.93% of all mentions.

Coverage Maintenance. Whether there was an effort related to maintaining or improving coverage. Out of all mentions, 6.06% pertain to this theme.

Encourage Coverage. Whenever there are clear discussions related to encouraging coverage. For example, discussions about identifying or requesting to solve a lack of coverage, requests for coverage tools, etc. We highlighted that 0.53% of all mentions are classified under this theme.

Reflecting on Coverage. This theme denotes discussions that revolve around the overall system coverage, such as discussions about coverage level status. This theme provides insights into how developers evaluate the coverage of a system as a whole and can reveal patterns or trends in how they approach and think about code coverage. We pointed out that this theme makes up 1.44% of all mentions.

Coverage Debt. Coverage debt is a term commonly used in software development to describe the gap between the actual code coverage achieved by a software project and the expected or desired level of coverage. This theme emerges whenever there is an explicit indication that the current coverage level is insufficient, but developers fail to address it due to constraints such as a lack of resources or time to develop and maintain tests. Our analysis revealed that this theme constitutes 0.22% of all mentions.

Lack of coverage. Whenever there was a report of a lack of coverage. This theme means that developers recognize that certain features of the software are not tested, and as a result, potential bugs or errors may remain undetected. It is essential to distinguish **lack of coverage** from **coverage debt**, as the former denotes an issue where a particular aspect of the software has not been tested, while the latter indicates the accumulation of gaps in test coverage over time due to various factors, such as resource limitations or inadequate test design. We drew attention to the fact that this theme accounts for 0.38% of all mentions.

Coverage Constraints: This theme arises when there are limitations in the coverage environment that impose restrictions on activities related to code coverage. It is worth noting that 0.06% of all mentions are attributed to this theme.

Coverage Configuration. This theme refers specifically to the process of setting up and configuring the coverage analysis tool that will be used to measure code coverage during testing or existing practices. This can include selecting the appropriate tool, configuring its settings, and integrating it with the development environment. Our analysis revealed that this theme constitutes 0.97% of all mentions.

Coverage Support. This theme refers to discussions or actions related to improving the effectiveness and efficiency of the tools used to measure code coverage. It can include identifying new tools or features to support coverage analysis, improving the integration of existing tools with the development environment, or providing training and support to developers to help them better use the tools. We drew attention to the fact that this theme accounts for 0.06% of all mentions.

Coverage Computation. Coverage computation is the process of measuring the level of code coverage achieved by software testing. This theme emerges whenever there are clear discussions on how coverage should be computed. Our analysis revealed that this theme constitutes 0.03% of all mentions.

Coverage Scope. This code denotes the scope in which coverage will be calculated. It refers to the extent or range of code that is subject to testing or measurement in a software project. For example, we will only perform coverage on this module, not the others. This theme represents 0.03% of all mentions.

Coverage Strategy. This code emerges when there is a plan or guideline for achieving a desired level of code coverage. It may specify the types of tests to be used, such as unit tests, integration tests, or functional tests, as well as the coverage metric, such as branch coverage, path coverage, etc. Additionally, it may also define a schedule for achieving coverage goals. Out of all mentions, 0.12% pertain to this theme.

Coverage Inquiry. Whenever there were questions or uncertainties related to code coverage. These discussions aim to clarify and address any issues or gaps related to code coverage. By addressing these questions, developers can ensure that the coverage process is comprehensive. This theme represents 0.39% of all citations.

Coverage Challenge. This theme refers to difficulties or obstacles that developers encounter when attempting to achieve a certain level of code coverage in a software project, which are unrelated to external constraints in the coverage environment. Our analysis revealed that this theme constitutes 0.47% of all mentions.

Coverage Threshold. This theme refers to conversations or debates among developers regarding the minimum acceptable level of code coverage for a given software project or component. These discussions typically involve establishing a coverage threshold, the minimum percentage of code that automated tests should cover. Coverage threshold discussions aim to establish a common understanding among team members about the level of testing required to ensure quality and reduce the risk of defects in the software.

Out of all mentions, 0.01% pertain to this theme.

Unknown. Whenever there is not enough information to classify a document. This theme represents 1.03% of all mentions.

Unrelated. This theme is used to indicate the presence of a document that is unrelated to the research and was not filtered out by the screening process. 72.46% of all mentions are classified under this theme.

5.3.2 In-Depth Analysis of Themes

After identifying the themes that emerged from our document analysis, we now present an in-depth analysis of each theme. Our aim in this section is to provide a detailed understanding of the discussion topics (themes) related to code coverage that developers discussed in pull request discussions. For each theme, we discuss the frequency of occurrence, the context in which it was mentioned, and the main points that emerged from our analysis. By delving into each theme, we hope to shed light on the different perspectives and experiences related to code coverage in software development projects

To ensure a more targeted discussion in this subsection, we will exclude the **Unrelated** and **Unknown** themes from our analysis, as they represent false positives and unclassified documents, respectively. By excluding these themes, we aim to focus our analysis on the documents directly related to code coverage, reducing the dataset noise. Thus, in this section, we only considered the 6,949 documents from 26,210 documents.

Figure 5.4 presents a visualization of the discussion topics (themes) that emerged from our inductive analysis (detailed in Section 5.2). The main object of analysis, which is the *code coverage*, is situated at the center (or root) of the network. The second-level (or axial) themes are related to discussions on *Coverage Strategy*, *Coverage Issues*, *Coverage Report*, *Coverage Discussions*, *Coverage Tools*, and *Coverage Maintenance*. The third-level themes are more specific and are grouped based on their relationship with the second-level themes. Table 5.3 shows the high-level themes and their definitions, except for *Coverage Strategy* and *Coverage Maintenance*, which are defined in Section 5.3.1.

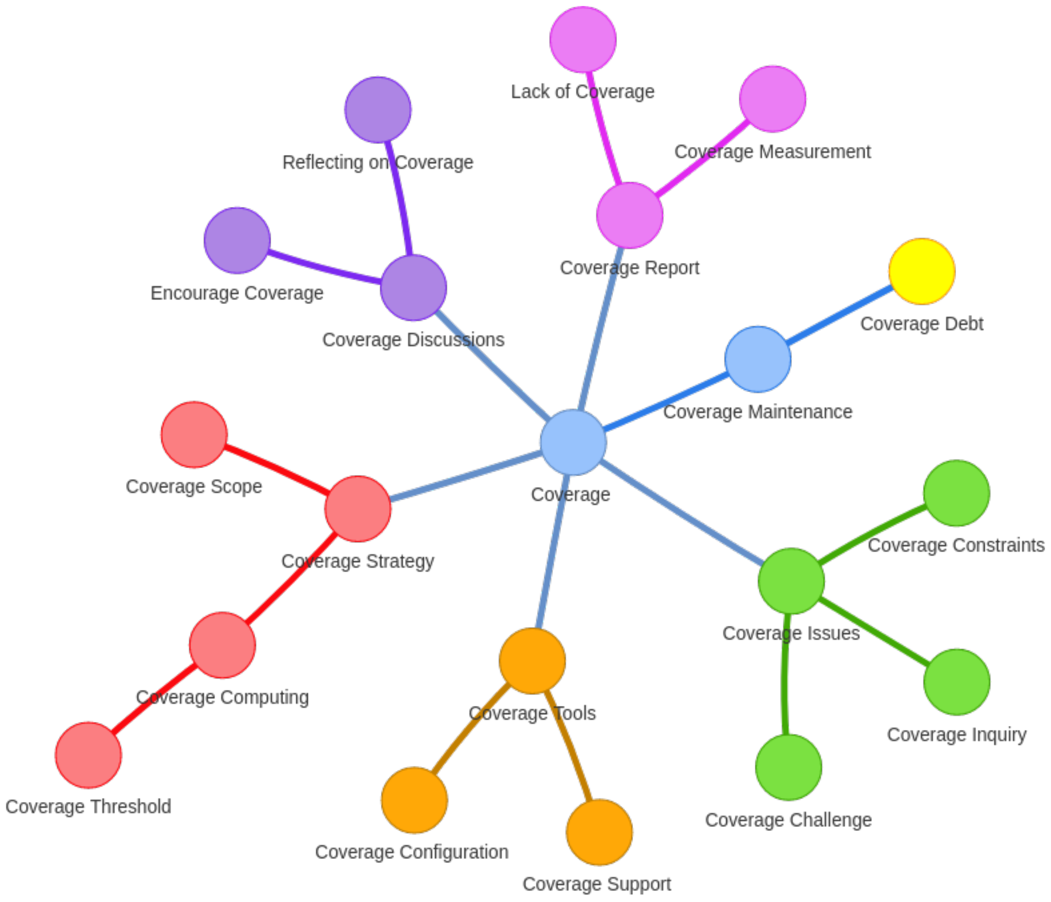

Figure 5.4: An overview of themes that emerged from our inductive analysis.

5.3.3 Coverage Report

The *Coverage Report* axial theme encompasses reports generated manually or automatically that indicate the percentage of code covered by tests. In our dataset, we identified 4537 (17.31%) citations related to this theme, with 4438 (63.87%) citations relating to *Coverage Measurement* and 99 (1.42%) to *Lack of Coverage*.

For instance, we encountered comments expressing concerns about no coverage at all:"*This module (and basically all of mpl_toolkits) currently has no test coverage*",

Table 5.3: High-level Themes emerged from the Document Analysis.

High-level themes	Description
Coverage Report	This theme refers to reports that can be generated manually or automatically, indicating the percentage of code covered by tests.
Coverage Issues	Denotes problems or challenges related to achieving or maintaining adequate test coverage in software development.
Coverage Discussion	Refers to debates and reflections about the current level of code coverage and the possible implications of it. These discussions can involve topics such as identifying areas of the codebase that require additional testing and analyzing the impact of test coverage on software quality
Coverage Tools	This theme refers to conversations or debates among developers about the different coverage tools or coverage support tools.

low coverage levels, such as: "*Numbers are scary low: 50% line coverage, 55% function coverage, and 23% branch coverage*", and positive feedback about good coverage levels, such as: "*Code coverage for this is solid. The only areas to consider improving are: - Core/HermitePolynomialApproximation - 79% - lots of important looking math is not called. - DynamicScene/CzmlDefaults.js - 80% - no call to createVisualizers*". We also observed several comments celebrating 100% coverage, such as: *"Coverage is 100%, and all the comments have been addressed. Let me know if there is anything else"*.

These comments indicate that developers place importance on both overall coverage and patch coverage. It is worth noting that patch coverage exhibits significant variability across different projects. Notably, the correlation between patch coverage and overall coverage, as pointed out by Hilton, Bell, and Marinov (2018), is not evident.

Remarkably, we discovered that certain projects employ distinct coverage metrics, as mentioned in the initial comment. These metrics can concentrate on specific aspects of code coverage, such as statement coverage, branch coverage, or path coverage. By utilizing multiple metrics, projects aim to achieve a more comprehensive understanding of their codebase and ensure that different coverage criteria are effectively addressed.

Our analysis indicated that the overwhelming majority of coverage reports, approximately 92%, were automatically generated by tools designed to monitor code coverage, with Codecov[3] and Coveralls[4] being the most commonly used ones. These tools perform similar tasks by commenting on pull requests about code coverage.

For projects that adopt these tools, we observed a slight increase in the median number of comments on merged pull requests. This increase may be attributed to contributions reducing code coverage, leading to discussions between maintainers and contributors. We can identify these through the presence of comments related to "lack of coverage" and "encourage coverage" during the discussions.

Notably, the increase in comments was most pronounced in the initial phase after the bot adoption, as contributors may have been unfamiliar with the tools' feedback. Interestingly, we observed that the median number of comments on merged pull requests tends to decrease each month after this initial phase. Our observations are inline with the results report by Wessel et al. (Wessel et al., 2020). The authors reported that after the adoption of the code review bot, they observed a slight increase in the median number of comments on merged pull requests. This increase could be due to contributions that significantly decreased the code coverage, leading to discussions between maintainers and contributors. This effect was most pronounced in the initial phase after the bot adoption, as contributors might be unfamiliar with the bot's feedback. Therefore, we believe that we detected a similar phenomenon.

In summary, the analysis of the *Coverage Report* axial theme revealed that coverage reports, which indicate the percentage of code covered by tests, played a significant role in the discussions during pull requests. A substantial number of citations were related to coverage measurement and the lack of coverage. The comments expressed concerns about no coverage, low coverage levels, positive feedback about good coverage levels, and celebrations of 100% coverage. These comments indicate that developers place

[3]https://github.com/marketplace/codecov
[4]https://github.com/coveralls

importance on monitoring overall and patch coverage. Additionally, the analysis indicated that most coverage reports (approximately 92%) were automatically generated by tools specifically designed to monitor code coverage. Popular tools like Codecov and Coveralls were commonly used for this purpose. These tools comment on pull requests about code coverage, facilitating discussions between maintainers and contributors. Therefore, based on the findings, it can be concluded that projects do actively monitor code coverage and engage in discussions related to coverage issues during the pull request process.

Findings 5.3.1

We found evidence that some CI projects are monitoring code coverage, as indicated by the presence of automatic coverage reports and a substantial number of citations related to coverage measurement and the lack of coverage. This observation is consistent with the self-reported monitoring practices of developers mentioned in our qualitative study (see Chapter 4).

5.3.4 Coverage Issues

The high-level *Coverage Issues* theme identifies, through PR discussions, debates, and reflections among developers, the challenges (*Coverage Challenge*), uncertainties (*Coverage Inquiry*) or constraints (*Coverage Constraint*) related to code coverage in software development. *Coverage Issues* theme comprises only 0.92% of the topics, indicating that while code coverage is a well-known topic, it is not a significant issue in most of the projects analyzed.

The *Coverage Challenge* is the most prevalent issue identified in the *Coverage Issue* high-level theme, accounting for 1.76% of all mentions. One of the most common challenges reported by developers is related to the integration between Continuous Integration (CI) and coverage tools. For instance, one developer encountered issues generating coverage data using the *lcov* tool, as shown in the quote, *"I'm not entirely sure what is going on. In travis.yml I'm trying with tape which generates lcov data, but it seems to fail somewhere"*. Another developer split tests to avoid timeouts caused by enabling coverage reports, as indicated in *"Split vyos tests to avoid timeouts with coverage"*.

These examples highlight developers' significant challenges in seamlessly integrating their CI systems and coverage tools. The lack of tool integration emerges as a prevalent challenge developers face, reinforcing the findings of our previous survey study (Chapter 4). During interviews, participants expressed a strong desire for enhanced integration between their CI systems and coverage tools. These sentiments further underscore the importance of addressing the issue of tool integration to streamline the development and testing processes and improve overall code coverage practices.

In addition to the aforementioned challenges, developers encounter difficulties when dealing with enabled coverage reports for browsers, determining when tests are completed, and notifying workers of completion, as mentioned in *"enabled coverage reports for browsers Solves #8632: Yes. You still need to figure out when the tests have completed and notify the worker of completion"*. The quote, *"With coverage enabled, the 2016 hosts are getting close to the group limits and sometimes even exceed the timeouts"*, highlights the challenge of enabling coverage while encountering timeouts and exceeding group limits.

Moreover, while CI tools offer flexibility in their usage, they often require extensive configuration, even for simple workflows. This complexity is further amplified when integrating with coverage tools, making the process more challenging for developers.

In light of these challenges, developers must carefully navigate the complexities of enabling coverage reports, managing test completion, and integrating coverage tools within their CI workflows. Finding effective solutions and streamlining the configuration process can help alleviate these difficulties and facilitate seamless integration between CI and coverage tools.

Finally, the quote, *"To my knowledge, we are working through some testing decisions about how we will properly test both process models without overloading the CI infrastructure, but still get enough coverage"*, highlights the challenge of balancing testing needs and CI infrastructure. The developer is trying to test both process models without overloading the CI infrastructure while still ensuring enough coverage.

Interestingly, our findings reveal that developers often face trade-offs between test execution speed and coverage accuracy. Nevertheless, the incorporation of extra tests within the CI pipeline may decrease the overall efficiency of the process, which is also viewed as a concern by developers. Rothermel et al. identified a trade-off between accurately testing code and managing build times as a challenge (Rothermel et al., 1999). This trade-off also affects coverage reports. Tool developers can assist developers by creating tools to conveniently execute subsets of their testing suites (Yoo and Harman, 2012).

Another frequently encountered problem is related to coverage services, as evidenced by comments such as (I) *"Coveralls seems to have issues again"*, (II) *"Coveralls seems to be drunk"*, (III) *"Code coverage with Coveralls won't work correctly anymore"*, and (IV) *"The coverage results are not uploaded at all now"*. Surprisingly, issues with traditional coverage tools like *coverage.py* and *Istanbul* were less common. Our analysis suggests that both CI services and code coverage tools have significant room for improvement.

This finding is consistent with the results reported by Zampetti et al. (Zampetti et al., 2019), who also found that CI pipeline misconfiguration was a common topic in Pull Request discussions.

In the *Coverage Inquiry* theme, which accounts for 1.48% of all mentions, we did not identify any predominant topic. We found debates about coverage services and tools similar to the previous theme. However, we noticed some discussions about how to interpret the coverage reports generated by the tools. For example, developers questioned the reason for coverage reports dropping even though it had increased (*"Not sure why coveralls is complaining of coverage dropped, as it actually increases...? !"*). Another issue was that Istanbul sometimes showed code as not covered by tests (*"Besides that, for some reason Istanbul shows as if my code is not covered by the tests I wrote. Any ideas?"*), leading to confusion. These comments often stemmed from misunderstandings about coverage concept and how local changes made in a pull request can affect the overall coverage.

The literature shows no correlation between patch coverage and overall coverage (Hilton, Bell, and Marinov, 2018). In other words, having high patch coverage does not necessarily increase non-patch coverage.

Moving on to the *Coverage Constraint* theme, it denotes constraints in the coverage environment that limits the developers' activities regarding coverage. The most common constraints were related to compatibility issues or other limitations in the development, as shown in the following quotes: *"Add ansible-test constraint to avoid coverage 5.0+"* and *"Update constraints for coverage on Python 3.8"*. It is worth noting that 0.24% of all mentions are attributed to this theme.

We found some quotes emphasizes the challenges of testing GUIs, particularly in terms of ensuring good test coverage due to the complexity of managing multiple GUI frameworks in the test process. For instance: *"We do not have good test coverage on the UI interaction (which is why the bug slipped in!) because GUI testing is just hard to begin with and due to difficulty in managing multiple GUI frameworks in the test process..*

Testing GUI (Graphical User Interface) can be challenging due to the complex interaction between the user and the system. Unlike testing individual functions or methods, GUI testing involves testing how users interact with the system through various inputs, events, and navigation. This complexity requires consideration of various factors, such as data validation, screen layout, and error handling. GUI testing is time-consuming and requires specialized tools and techniques to ensure complete coverage of all possible user scenarios and system states. Testers must ensure that the GUI is tested under different conditions, such as different screen resolutions and input devices. Additionally, GUI testing is more susceptible to changes in the user interface design, making it necessary to update and maintain test cases as the system evolves.

One noteworthy quote that stands out is: *"Coverage could show more than just a line number. This. I like all the new fancy null coalescing functionality that PHP now has, but I very much dislike how it hides coverage PCOV can't generate path coverage, but even if it could, I don't think that Codecov supports it"*. This quote not only highlights a limitation of Codecov on supporting PHP's coalescing functionality, but also raises a broader issue with how we perceive and interpret code coverage. It underscores the importance of not relying solely on code coverage ratios as a measure of testing quality and instead considering other factors that could impact the reliability of our tests, such as non-deterministic behavior and changes to covered statements.

Findings 5.3.2

Developers commonly report challenges related to integrating coverage tools with Continuous Integration (CI), with compatibility issues between local and CI server environments being a critical factor in coverage within CI environments. Our findings related to Study 1 / RQ3 (see Section 3.3.3) support it, as we identified integration problems as a main reason for decreased coverage and observed several mentions of compatibility issues.

5.3.5 Coverage Discussion

We have consolidated the *Reflections on Coverage* and *Encourage Coverage* themes into an axial theme called *Coverage Discussion* because both themes represent debates around the ideal coverage level. By examining the conversations and debates surrounding code coverage, we can gain insights into how developers use the coverage information and implement code coverage practices.

In our dataset, we identified 517 (1.97%) citations related to this theme, with 377 (5.43%) citations relating to *Reflections on Coverage* and 140 (2.01%) to *Encourage Coverage*.

The *Reflections on Coverage* themes group comments or feedback from developers on the current state or effectiveness of code coverage. These reflections may include observations, insights, or critiques on the current coverage, suggestions for improving it, or discussions about how to interpret or use coverage metrics. For example, the comment *"I guess that a lesson learned from all this is that test coverage is high but not totally effective."* indicates that code coverage alone is not always a reliable effectiveness indicator.

Research studies typically prioritize evaluating the effectiveness of fault prediction criteria and identifying approaches that can detect the most faults. However, from a developer's perspective, the more common question is whether it is worthwhile to allocate additional effort to improve a specific test suite (Gopinath, Jensen, and Groce, 2014).

We also found comments indicating that code coverage provides developers with a sense of confidence, such as in the comments *"I guess the best way to be convinced that it's correct would be to have enough test coverage, especially since it's easy to verify if an indefinite integral is correct."* and *"And will be worth the effort spent in coverage - especially since our notification system has been a bit dodgey."*

Most of the comments within the *Reflections on Coverage* theme are related to discussions about the level of code coverage achieved in the project. These discussions may include observations or critiques about the current coverage level, suggestions for how to improve coverage, or debates about the appropriate level of coverage to aim for. For example, comments such as *" Code coverage for this is solid. The only areas to consider improving are ... "*, *"Coverage is great"*, and *"I don't believe we should strive for 100% test coverage. It's really not about the percentage, but more about covering what needs to be covered and maintaining that."*.

We can deduce that developers engage in discussions regarding the desired level of testing for each project and consider the allocation of resources necessary to achieve a specific coverage level. Some projects may prioritize extensive and thorough testing, even if it results in slower execution times. On the other hand, some developers express the view that achieving 100% test coverage may not be necessary, emphasizing the importance of covering what needs to be covered and maintaining that level. According to the literature (Chen et al., 2020), a higher level of code coverage, which typically necessitates larger test sets, is associated with an increased probability of detecting faults. Consequently, developers may take this into consideration when determining the appropriate level of testing and allocating resources for their projects.

Regarding the *Encourage Coverage* theme, the majority of comments are related to encouraging developers to provide more coverage. However, we also found comments indicating that developers consider patch coverage during the code review process. For example, one comment stated, *"Yes. There are a few significant lines that are not covered. (I left comments above.) Since this is new code, the best time to get it covered is while all the code is fresh in mind."* This suggests that covering newly added code as soon as possible is a good practice, which shows that *coverage debt*, a theme also found in our analysis, should be avoided. When new code is added to a project but not covered by tests, it creates a gap in the test coverage that represents coverage debt. This debt will need to be paid off eventually by writing tests to cover the code, but the longer it remains uncovered, the higher the risk of undetected errors or bugs. Therefore, addressing coverage debt as soon as possible by writing tests for newly added code can help to reduce the risk of bugs and improve overall code quality.

During our analysis, we came across comments that pertained to incidental code coverage. This term refers to the unintentional coverage of code that occurs when tests designed for other purposes inadvertently cover additional code. It is important to notice that incidental coverage is considered a false positive, as it can create a deceptive perception of adequate testing quality. An example of this situation is when our test suite executes all of our code, but there are certain routines or segments that lack dedicated tests. This undermines the accuracy of the coverage metric since it may give a false impression of comprehensive testing. We came across a comment illustrating the challenge of managing incidental coverage in practice: *"I would also like to remove the dependency on these tests. However, they currently provide incidental code coverage for parts of Ansible that are not covered by other tests. Once the ansible-base migration has been completed, we can start working on replacing these (and other tests), providing intentional coverage."*.

The mentioned comment implies that incidental coverage can be advantageous in certain scenarios. For instance, incidental coverage can be helpful if the goal is to achieve 100% coverage. However, relying solely on incidental coverage falls short of harnessing the full potential of the test suite. It is important to recognize that changes or improvements made to the application can occur without the reassurance of a comprehensive regression suite. Therefore, while incidental coverage may offer some benefits, it should not be considered as the primary method of testing code, as it may leave gaps in test coverage and pose risks in maintaining code quality.

Findings 5.3.3

- Considering the discussions about coverage, the most common comments are related to coverage levels, which reinforce our impression about coverage monitoring as mentioned in the previous section. In addition, these concerns were also expressed by developers during the survey (Section 4.3.3).
- In addressing Study 1 - RQ3 (section 3.3.3), we did not identify any discernible coverage prioritization for newly added code over preexisting uncovered code. Although some participants mentioned the importance of covering new lines, the frequency of such comments (approximately 0.14%) was not significant enough to suggest a systematic movement towards prioritizing new code coverage. Our findings suggest that the coverage of new code is not a dominant concern in the surveyed projects.

5.3.6 Coverage Tools

The axial theme *Coverage Tools* refers to conversations or debates among developers about the different tools available for measuring and supporting code coverage in software development. These discussions may include topics such as the benefits and limitations of different coverage tools, how to configure and integrate coverage tools into the development process, and best practices for using coverage tools effectively.

In our dataset, we identified 268 (1.02%) citations related to this theme, with 253 (3.64%) citations relating to *Coverage Configuration* and 15(0.22%) to *Coverage Support*.

The theme of *Coverage Configuration*, which deals with configuring code coverage, was the most frequently discussed, comprising 94.40% of the comments in regarding *Coverage Tools* axial theme. Many of these comments focused on issues related to the

Coverall and *Codecov* tools, which are the most commonly used coverage servers in the market. Despite their popularity, developers still encounter challenges in configuring these tools effectively. The quotes *"removing coveralls support for Ruby 1.9.2 in hopes of fixing #1921"* and *"Disable coveralls on travis with jruby"* describe two actions taken to address issues related to *Coveralls*. In the first case, support for Ruby 1.9.2 was removed in the hopes of resolving a specific problem. In the second case, *Coveralls* was disabled on Travis, a CI platform, when running with JRuby.

The remaining of comments in the *Coverage Configuration theme* were related to coverage tools specific to each project and programming language, such as *Istanbul* and *Coverage.py*. For example, in the quote *"Backport PR #22326: CI: ban coverage 6.3 that may be causing random hangs in fork test"*, the proposed change involves banning the use of version 6.3 of the Coverage.py tool in the CI environment due to its tendency to cause "random hangs" during "fork tests".

The existing literature has already recognized the trade-off between flexibility and simplicity that developers encounter (Hilton, Nelson, et al., 2017). This trade-off requires developers to strike a balance between harnessing the flexibility and power offered by highly configurable coverage and embracing the ease of use associated with simplicity. In general, systems that are overly configurable tend to have a negative impact on usability (Xu et al., 2015).

We identified 15 (0.22% of all mentions) comments related to the *Coverage Support* theme, as described in Section 5.3.1, which pertains to the underlying tools and infrastructure that support the coverage environment. Some examples of these comments include: *"It turns out that the tool can be used to convert multiple JSON files at once, so it is not necessary to try and merge all coverage files!"* and *"Try using phpdbg to speed up coverage build."*. These comments suggest that developers are actively seeking ways to improve the efficiency their coverage tools and processes.

Findings 5.3.4

- Our investigation revealed that configuring the coverage environment poses a challenge for both the use of coverage services and traditional coverage methods. These findings align with those presented for Study 2 - RQ4 (Section 4.3.2), where developers reported that monitoring coverage in the CI pipeline may not be worth the effort.

- Developers seek coverage tools that balance high flexibility and ease of setup and maintenance. However, there is a trade-off between these two objectives, as increasing configurability introduces complexity while simplification aims to reduce complexity.

5.3.7 Coverage Strategy

The axial theme of *Coverage Strategy* encompasses all discussions related to the approach, plan, or guidelines that a development team employs to attain a desired level of code coverage. It is composed of two sub-themes: *Coverage Scope* and *Coverage Computing*.

In the *Coverage Strategy* theme, we identified discussions primarily focused on determining the appropriate coverage metric, as exemplified by the following comment: *"We should definitely test method coverage somewhere. That would be a good place to test if cdef class methods are properly covered, and, thus, we'd test cdef classes in combination with coverage."*; deciding on the best approach to achieve coverage, as illustrated by: *"Split each C# module into its own test target, making it easier to determine what is covered and to separate them from the repository in the future."*. The developers also suggest how to effectively address coverage, as evident in the following quote: *"There should be 3 separate status indicators: (1) Overall library coverage, (2) Percent of coverage for the* *changes, (3) Number of lines covered in the tests"* which indicate that the developers are actively seeking ways to improve their coverage monitoring practices. This theme stands out as it represents 0.73% of all mentions.

We discovered 8 (0.12%) occurrences related to the *Coverage Scope* debates. For instance, comments such as *"We should only measure coverage for one Python version. The slow tests probably shouldn't be counting toward coverage anyway"*, *"Maybe we should only do coverage testing for the non-slow tests on one Python version"*, and *"Code-coverage scope is limited to codes under whitelist config"* provide insight into the type of code that is intentionally excluded from coverage reports. Moreover, we identified some comments highlighting flawed coverage reports, such as *"One disturbing thing I notice is that core tests cover a small percentage of the code; the actual coverage is much higher when _other_ non-core code uses the core. It seems like the tests for a given module should address coverage or else when non-core (for example) tests are changed you could be changing the coverage of the codebase."* These comments expose potential issues with coverage analysis that could be harmful to the project.

The *Coverage Computing* theme accounted for 12 (0.17%) of the total citations and encompasses discussions related to the computation and interpretation of code coverage metrics. This theme also involves the *Coverage Threshold* theme, which pertains to establishing a minimum acceptable level of code coverage.

The discussions related to the *Coverage Computing* theme highlights the challenges that developers face when trying to compute the coverage metric. An example of a discussion related to the *Coverage Computing* theme is: *"Despite adjusting limits and boundaries, these tests sometimes fail. Continuously increasing the coverage threshold undermines the purpose of these tests, and instead, we can skip them when generating coverage."* The comment suggests that despite adjusting the limits and boundaries, the tests sometimes fail. The person who made the comment also suggests that continuously increasing the coverage threshold undermines the purpose of these tests, which is to ensure adequate test coverage. Instead, the comment suggests that developers should skip the tests that fail to achieve the desired coverage level when generating coverage metrics. This comment reflects a common concern among developers about the trade-offs between test coverage and the costs of maintaining and updating test suites.

We also identified 3 (0.04%) citations that fell under the *Coverage Threshold Discussions* theme. These comments focused on debates surrounding the appropriate level of coverage threshold. For instance, one comment suggested setting the threshold at -0.01%, which would force developers to pay attention and merge coverage-decreasing pull requests sparingly.

Findings 5.3.5

Within the *Coverage Strategy* theme, we observed that discussions primarily centered around determining the most appropriate coverage metric to use. This finding aligns with our expectations since, in Study 2 - RQ4 for questions Q3.5 and Q3.6 (see Section 4.3.2), developers reported varying coverage metrics and expectations regarding their usage. These differences in opinion can lead to debates among developers as they consider which metric is most appropriate for a given project, considering the project's goals and constraints. As such, our results highlight the importance of carefully considering which coverage metric to use and engaging in collaborative discussions with other developers to ensure that the chosen metric aligns with the project's needs.

5.3.8 Coverage Maintenance

The *Coverage Maintenance* axial theme captures the actions taken by developers to attain the desired level of code coverage, as seen in examples such as *"Improved Code*

Coverage from 54.55% to 77.27% for ...", *"Added a regression test to ensure we don't do this in the future."*, and *"Improved Coverage for Series module"*. We identified 1647 occurrences (22.68% of the total) related to this theme.

The *Coverage Maintenance* theme consists of the *Coverage Debt* theme, which refers to comments about coverage debts, or gaps between the actual coverage level and the expected or desired level of coverage. Out of the total occurrences within the *Coverage Maintenance* axial theme, 58 (which represents 3.52%) were related to coverage debts. We identified two main reasons for these coverage debts: (1) the difficulty of writing tests, as evidenced by comments like *"I haven't found any test that currently evaluates this function, so adding a test case wasn't as easy as expected. Because of this, I provided this fix without any test."*, *"Please let me know if you have any solution to this so I can increase the test coverage."*, and *"The test coverage for finding the roots of quartic polynomials and finding the grazing altitude location are around 70%-80% because it's difficult to come up with equations/rays for all cases."*; and (2) the effort required to maintain the tests, as demonstrated by comments such as *"Covering tests for CLI outputs is not that worthwhile and too hard to maintain. Otherwise, we'd have to add tests for every error message we display too."*, and *"Writing a test for it might be more trouble than it's worth though [...]"*.

Findings 5.3.6

The coverage maintenance theme represents the second most recurrent topic, accounting for 22.87% of all comments. This result is in line with both our quantitative study (see Chapter 3) and survey (see Chapter 4), as it reflects the actions taken by developers to improve coverage. These studies predicted a significant volume of such actions, given that we detected an increase in coverage. Therefore, this finding corroborates the results of both studies. Additionally, we observed a reduced number of discussions related to coverage debt. This was an expected result, given that our findings from **RQ1** (Section 3.3.1) indicate that CI projects tend to stabilize at higher coverage levels.

5.4 Implications

This work has implications to different kinds of stakeholders. Here we discuss some of them.

Relevance for Research

Our research opens avenues for future empirical studies that address projects that have not adopted Continuous Integration (CI) practices. By examining the discussions on code coverage in pull requests from both types of projects and comparing the results with those of this study, we can identify potential differences in development practices and priorities. Specifically, we can investigate whether there is a significant difference in the emphasis placed on coverage between the two groups and gain insights into how NOCI projects approach and use coverage. This analysis can shed light on which CI practices effectively impact coverage and help explain why NOCI projects may struggle to maintain high coverage levels.

Relevance for Practitioners

We have identified several themes related to code coverage in the comments of pull requests. By pinpointing common themes related to coverage, our research can assist practitioners in understanding the intricacies involved in achieving and sustaining an adequate code coverage level, and in devising strategies for enhancing their coverage practices. Furthermore, our research can help practitioners discover tools and practices that can be employed to boost their code coverage efforts.

In additional, tool developers can leverage our research findings to better understand the practical needs and challenges of developers related to code coverage. By considering these insights, tool builders can improve their tools to provide more effective solutions for monitoring, maintaining, and enhancing code coverage.

5.5 Threats to the Validity

5.5.1 Construct Validity Threats

The construct validity of a research design refers to its accuracy in measuring what it claims to measure. Potential threats to construct validity are primarily related to the data collection process. To mitigate this threat, we relied on a validated and consolidated dataset from a previous study (Nery, Costa, and Kulesza, 2019).

5.5.2 Internal Validity Threats

Internal threats are concerned with the ability to draw conclusions from the relationship between the dependent variable (code coverage) and independent variables.

In terms of our qualitative analyses, it is important to acknowledge that there is always the potential for bias resulting from the authors' subjective experiences. It is possible that different experiences could lead to different themes emerging from the data.

However, we employed rigorous qualitative methods to mitigate these potential threats, such as recruiting multiple coders and calculating inter-rater agreement measures. To establish a common vocabulary and standardize the understanding of the themes, we conducted both deductive and inductive phases on a sample with a 95% confidence level and a 5% interval. Once the themes were confirmed by two independent coders, the main coder applied them to the entire dataset.

Moreover, it is important to recognize that inductive analysis is an ongoing process and the emergent themes may not be exhaustive. Therefore, the themes identified in our study should be taken as a foundational framework that can be further refined and improved by future researchers.

In conclusion, while our approach may not be flawless due to our lack of technical or domain knowledge comparable to that of a developer from our chosen projects, we are still able to assess the topics debated when analyzing pull request comments.

5.5.3 External Validity Threats

External threats represent the extent to which we can generalize our results to other environments (e.g., other software projects).

The open-source projects analyzed in this study are hosted on the GitHub platform and use Travis-CI as their continuous integration (CI) service. It is important to note that our findings are limited to the context of these specific projects. Further research is required to replicate our study and explore whether the results can be generalized to other projects with varying characteristics.

5.6 Conclusions

Our research focuses on investigating the relationship between code coverage and continuous integration (CI) by analyzing qualitative data. To achieve this goal, we conducted a document analysis of 30 CI projects using a two-step approach: an inductive analysis and a deductive analysis. We found several themes related to code coverage in the comments of pull requests, representing the key issues and topics that developers discuss regarding code coverage. These themes include *Coverage Measurement*, *Coverage Maintenance*, *Encourage Coverage*, *Reflecting on Coverage*, *Coverage Debt*, *Lack of Coverage*, *Coverage Constraints*, *Coverage Configuration*, *Coverage Support*, C*overage Computation*, *Coverage Scope*, *Coverage Strategy*, *Coverage Inquiry*, *Coverage Challenge*, and *Coverage Threshold*. By analyzing these themes, we gained valuable insights into how developers approach and think about code coverage and identified potential gaps and issues related to code coverage in software development projects.

6 Related Work

In this chapter, we review previous studies related to continuous integration and code coverage, providing a comprehensive understanding of the current state of the related work and identifying gaps that this thesis aims to address.

This chapter discusses previous work on CI and code coverage, highlighting the strengths and limitations of different approaches. We begin by reviewing studies that investigate the impact of CI on software development, including its effects on quality, productivity, and team collaboration (Section 6.1). We then discuss research that focuses specifically on code coverage (Section 6.2).

6.1 Previous CI Studies

Several authors have listed a set of practices and principles related to CI (Duvall, Matyas, and Glover, 2007; Vasilescu, Schuylenburg, et al., 2014; Soares et al., 2022), and existing research have used these practices/principles to evaluate the usage of CI in software projects.

Felidré et al. (Felidré et al., 2019) observed, through the analysis of 1,270 open source projects using TravisCI, that about 60% of the studied projects did not follow adequate CI practices. They studied four bad practices: (1) performing infrequent commits to the mainline repository; (2) building a project with poor test coverage; (3) allowing the build to remain in a broken state for long periods; and (4) using CI with long build durations. They observed that 748 (60%) projects face infrequent commits, and the average code coverage for 51 projects was 78%.

Elaszhary et al. (Elazhary et al., 2021) also discuss the benefits and challenges of CI practices. The authors found that CI practices are broadly implemented but how they are implemented varies depending on the project context and their perceived benefits.

Santos et al. (Santos, Alencar da Costa, and Kulesza, 2022) studied how CI sub-practices may influence the productivity and quality of open-source projects. They analyzed the build duration, build activity, build health, the length of time to fix a broken build, and the commit activity sub-practices, in 90 open-source projects, over a period of two years. The authors found a correlation between build activity and commit activity with the number of merged pull requests. They also found a correlation between build activity, build health, and time to fix broken builds with the number of bug-related issues. The authors also report that projects with the best values for CI sub-practices face fewer CI-related problems.

Bernardo et al. (Bernardo, Costa, and Kulesza, 2018) empirically investigated the impact of adopting CI on the time-to-market of new features. The authors analyzed 162,653 PRs from 87 GitHub projects to explore the factors that affect the delivery time of these PRs. They reported that only 51.3% of the projects merged pull requests faster before adopting CI. The authors also observed that the number of PR submissions increased considerably after CI adoption. They reported that CI might not necessarily reduce the time to deliver features.

Vasilescu et al. (Vasilescu, Yu, et al., 2015) studied the quality and productivity outcomes regarding CI on GitHub. The authors analyzed the code quality (e.g., the number of bug reports raised in a project each month) and team productivity (e.g., efficiency on pull request integrations). Their findings reveal that CI improves productivity without reducing code quality. Zhao et al. (Zhao et al., 2017) empirically investigated the impact of CI in development practices. They studied the CI transition of projects using an RDD model, exploring variables such as commit frequency, code churn, pull request closing, and issue closing. They concluded that PRs are more frequently closed after adopting Travis CI.

Compared to previous studies, our research provides a comprehensive and in-depth analysis of the relationship between Continuous Integration (CI) and code coverage, both qualitatively and quantitatively. Unlike prior works, our primary focus is to examine the coverage evolution in CI projects and determine whether CI have any impact. Thus, our study complements existing research by providing a novel perspective on the empirical body of knowledge related to CI.

Zampetti et al. (Zampetti et al., 2019) investigated the relationship between Pull Request discussions and Continuous Integration build failures. Although the quantitative analysis showed that the build status had limited impact on PR merging, the qualitative analysis highlighted that PR discussions primarily focused on testing and static analysis issues, with a significant number of discussions revolving around CI pipeline configuration problems. The study emphasized the challenges in configuring and maintaining a CI pipeline, which may lead to maintenance difficulties or unnecessary build failures despite the advantages of automated continuous builds. Similar to previous studies, we utilized pull request discussions as a primary data source for our qualitative analysis. However, our approach differed from prior research which focused on how developers use CI outcomes and build logs during code review. Instead, we examined discussions related to code coverage within pull request discussions. Additionally, we conducted a quantitative study to complement our previous work and provide an overview of how coverage evolves when adopting CI practices.

Pinto et al. (Pinto et al., 2018) conducted a survey of 158 users of continuous integration systems to investigate the benefits and problems associated with their use.

The findings revealed that developers are uncertain about how to define a successful build, often due to factors such as flaky tests or misconfigured CI jobs.

A survey conducted among 152 developers at a large financial organization by Vassallo et al. (Vassallo et al., 2016b) investigated their adoption of Continuous Integration and delivery pipeline in their development activities, focusing on managing technical debt and test automation practices. The survey results provide insights into the adoption of agile methods in practice and challenge common assumptions and findings from other studies in some cases. In addition, the authors found that statement coverage was the most frequently mentioned coverage metric. Specifically, 84% of the respondents reported striving for a coverage level of at least 80% when considering statement coverage. Similar to Vassallo et al. (Vassallo et al., 2016b), we conducted a survey with developers. However, our study aimed to gather insights into how developers utilize coverage information during pull requests, contrasting with their investigation of the impact of continuous integration on code quality and testing practices. Moreover, we analyzed the growth of code coverage to gain insights into whether it has been increasing or decreasing since before CI was adopted. This allowed us to understand further the impact of CI on code coverage in the studied projects.

To the best of our knowledge, our work is the first to focus exclusively on investigating the evolution of code coverage by (i) comparing the trends in coverage evolution in both NOCI and CI projects, (ii) applying multiple regression models to analyze potential effects of an intervention (in our case, the adoption of CI), and (iii) conducting a qualitative study to complement the quantitative analyses.

6.2 Previous Coverage Studies

Code coverage is a topic largely explored in existing literature (Hilton, Bell, and Marinov, 2018; Ivanković et al., 2019; Nery, Costa, and Kulesza, 2019; Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021; Kochhar et al., 2017). Many coverage criteria have been proposed, such as statement, branch, and path coverage (Aghamohammadi, Mirian-Hosseinabadi, and Jalali, 2021).

Grano et al. (Grano et al., 2019) presented a study about the possibility of using source-code metrics to predict the coverage achieved by test data generation tools using a Random Forest Regressor. The authors studied 79 factors belonging to four categories that might correlate with branch coverage. Our work also aims to evaluate the code coverage of CI projects, but instead of predicting a possible branch coverage for a specific software version, we study the association between the adoption of CI and code coverage.

Zaidman et al. (Zaidman, Van Rompaey, Deursen, et al., 2011; Zaidman, Van Rompaey, Demeyer, et al., 2008) studied whether production code and developer test code co-evolve. They investigated the version system, code coverage reports, and code size metrics of software projects. The authors examined class coverage, method coverage, and statement coverage for two Java projects. The findings reveal that code coverage positively correlates with the percentage of test code in the system. Our research complements their work (Zaidman, Van Rompaey, Deursen, et al., 2011; Zaidman, Van Rompaey, Demeyer, et al., 2008) by investigating the relationship between code coverage and CI adoption.

Hilton et al. (Hilton, Bell, and Marinov, 2018) performed a large-scale study about code coverage evolution over 7,816 builds of 47 projects. The authors observed that simply measuring the change to statement coverage does not capture the nuances of code evolution. Thus, fine-grained analysis (i.e., changed statements in commits) is needed to better capture coverage changes over time. Our work differs from their work by investigating the relationship between CI and code coverage evolution for a period of 2 years. Differently from our work, Hilton et al. (Hilton, Bell, and Marinov, 2018) only considered the last 250 commits of the projects, which may not capture the evolution of code coverage over time.

Hora (Hora, 2023) investigated code that is excluded from coverage reports. The author analyzed popular Python projects to investigate commit messages and coverage documentation. The author also computed test coverage by running test suites. The main findings reveal that (1) over 1/3 of the projects perform coverage exclusion; (2) 75% of the code is already created using the exclusion feature; (3) developers exclude non-runnable, debug-only, defensive code, platform-specific and conditional importing; (4) most code is excluded because it is already untested, low-level, or complex; and (5) code exclusion may impact test coverage. Unlike our work, the author's focus was on analyzing practices related to the exclusion of code coverage.

The closest work to ours is the study by Nery et al. (Nery, Costa, and Kulesza, 2019). The authors studied the relationship between CI and test ratio, and coverage. They compared 82 projects that eventually adopted CI (CI projects) and 82 projects never adopted CI (NOCI projects). The authors observed that the adoption of CI is associated with a consistent increase of test ratio. They also reported that test ratio is largely explained by the project inherent context rather than by the source code or development process. However, regarding code coverage, their study analyzed a small number of projects and versions (10 CI projects and 10 NOCI projects), which, unfortunately, yields little empirical evidence to support their results. Also, the work by Nery at al. (Nery, Costa, and Kulesza, 2019) did not employ the RDD analysis, which is more appropriate for checking the potential effects of an intervention (in our case, the adoption of CI). Lastly, our work investigated the type of changes that impact the code coverage in different directions (e.g., addition of new and uncovered lines vs. deletion of existing and uncovered lines) instead of considering code coverage as a single-dimensional measure. Finally, our work distinguishes itself from theirs by incorporating a qualitative study. Including qualitative research methods provides a unique perspective and adds depth to our findings. By delving into pull request discussions about coverage, we uncovered valuable insights about how code coverage is used and its main issues.

While previous studies have primarily focused on understanding how code coverage impacts code quality, our present study investigates the impact of adopting continuous integration on code coverage using a mixed-method approach of qualitative and quantitative analyses. Our quantitative approach aims to identify patterns and trends in the coverage evolution; while the qualitative study captured the experiences, opinions, and attitudes of developers regarding coverage.

7 Conclusions

The potential advantages of implementing Continuous Integration (CI) have drawn the attention of researchers, who have studied its benefits in various aspects of software development. Despite the significant progress made, there are still several assumptions in the community that have not been empirically investigated. In this thesis, we conducted empirical studies to explore the impact of adopting CI on code coverage. In the following sections, we highlight the contributions of this thesis and suggest promising directions for future research.

7.1 Contributions and Findings

Thesis Statement

While the association between CI and higher code coverage seems intuitive, given the emphasis on automated testing, the long-term relationship between CI and code coverage remains unexplored empirically.

The overarching goal of this thesis investigates the relationship between Continuous Integration (CI) and code coverage, exploring the extent to which developers' perceptions of the benefits of CI align with empirical evidence. Through an empirical study, this research seeks to provide insights into the impact of adopting CI on code coverage and to identify factors that influence this relationship. The findings of this study can inform practitioners in adopting effective tools and practices for improving code coverage in software development projects.

To conduct our research, we analyzed data from 60 GitHub projects, surveyed team members involved in these projects, and conducted a document analysis of pull request discussions. Our research seeks to address the following questions:

Study 1 – The Impact of the Adoption of Continuous Integration on Code Coverage

- **RQ1 - What are the evolution trends of code coverage within CI and NOCI Projects?** Our results show that the adoption of CI is associated with positive code coverage trends. We found that 50% of CI projects reveal rising trends in code coverage. In contrast, only 10% of NOCI projects reveal rising code coverage trends.

- **RQ2 - Is there a significant correlation between CI and code coverage values?** Our analysis demonstrates a statistically significant increase in code coverage following the adoption of CI. We observed a D value of 0.12710, indicating a significant enhancement in coverage as a result of CI implementation.

- **RQ3 - What types of code changes affect the code coverage of CI and NOCI projects?** Our results show that projects that eventually adopt CI have a significantly higher number of code changes that increase coverage.

Study 2 – Uncovering the Relationship Between Continuous Integration and Code Coverage: An Exploratory Investigation

- **RQ4 - How the code coverage information is used in the CI projects?** According to our study, developers make use of code coverage information during both the code authoring stage and code reviews, finding it valuable for identifying trivial bugs, detecting uncovered code paths, and enhancing software quality by potentially preventing bugs and controlling side effects when refactoring or modifying code behavior.

- **RQ 5 - Does the adoption of Continuous Integration increase the code coverage?** After investigating the increase in code coverage following the adoption of CI, we sought to rule out the possibility that other testing practices could be responsible for the observed effects. To this end, we asked participants whether they had adopted any new testing practices in addition to CI, and found that all of the reported practices (e.g. automated tests and build, all tests and inspections must pass, and code review.) were indeed part of CI. This finding reinforces the link between CI and code coverage. Furthermore, participants identified fluctuations in coverage as being linked to the correct use of CI practices.

Study 3 – Investigating Discussions on Code Coverage in CI-Enabled Projects: An Exploratory Document Analysis

After thoroughly analyzing the comments in pull requests, we identified several key themes that encompass the primary discussions and issues related to code coverage. By exploring the extracted themes, we gained valuable insights into how developers approach and address code coverage and the potential obstacles and deficiencies in software development projects related to code coverage.

We observe that our some of our quantitative findings are corroborated by our qualitative findings after performing our studies. According to our statistical models that are used in Chapter 3, reveal a positive association between CI and a higher code coverage rate. Our survey of participants in Section 4.3.3 also supports this conclusion, despite some differing opinions. Although there was some controversy among our participants, it is worth noting that the majority reported conducting coverage monitoring (see Section 4.3.2). This perception was further supported by our qualitative document analysis (see Chapter 5). This was evident from the presence of automatic coverage reports and active discussions between maintainers and contributors regarding coverage levels. In addition, our qualitative analysis provides us with a deeper understanding of our quantitative findings. For instance, we discovered that although the adoption of CI is generally associated with an increase in code coverage, some projects experienced significant declines in coverage across multiple versions during the after-CI period (see Section 3.3.2). Our qualitative findings shed light on the reasons behind this coverage fluctuation, revealing that some developers attributed it to a lack of adequate testing culture within the team (see Section 3.3.3).

Our study provides empirical evidence that the use of continuous integration can lead to an increase in code coverage over the long term.

7.2 Future Work

The studies that are performed in this thesis pave a way for several future work possibilities. We outline some venues for future work below.

- **Replication**. Replication studies are essential to further validate the findings of this thesis and generalize them to other contexts. Future work could replicate the studies performed in this thesis using other datasets, enabling us to test the robustness of our results. Additionally, future studies could focus on identifying the most critical CI practices that impact code coverage evolution, which can guide software development teams in improving their testing strategies.

- **Tooling**. Future work could focus on the development of tools to improve the practice of CI and code coverage.Our study identified several challenges related to coverage tools and services that could be addressed by tool builders, such as better tool integration between CI and coverage, better reports/visualization support, debugging assistance, and better notification. Additionally, developing better tools that automate the tracking and reporting of coverage metrics or provide real-time feedback on coverage during code changes could improve developers' adherence to coverage standards and facilitate the identification of code areas in need of additional testing.

- **Qualitative Study.** While our quantitative analysis sheds light on the relationship between adopting continuous integration and increasing code coverage, a more comprehensive understanding of the underlying reasons for this relationship could be gained through a new qualitative study. Such a study could involve comparing the experiences of non-CI projects with those of the projects in our study to identify potential differences in practices and factors that contribute to changes in code coverage levels. This could provide valuable insights into how CI practices can be further improved and optimized to promote sustained improvements in code coverage.

7.3 Publications

In this section, we provide a comprehensive list of all the publications that have resulted during my PhD journey, including also some publications that were not directly related to this PhD thesis.

1. J. D. S. Silva, D. A. Da Costa, U. Kulesza, G. Sizílio, J. G. Neto, R. Coelho, M. Nagappan. "**Unveiling the Relationship Between Continuous Integration and Code Coverage**". In: Proceedings of the 20th International Conference on Mining Software Repositories (MSR). ICSME '23. Melbourne, Australia: IEEE,

2023 (page 28).

2. J. D. S. Silva, J. G. Neto, U. Kulesza, G. Freitas, R. Rebouças, R. Coelho, 2021. **Technical Debt Tools: A Systematic Mapping Study**. ICEIS (2), pp.88-98.

3. J. D. S. Silva, J. G. Neto, U. Kulesza, G. Freitas, R. Rebouças, R. Coelho. **Exploring Technical Debt Tools: A Systematic Mapping Study.** In Enterprise Information Systems: 23rd International Conference, ICEIS 2021, Virtual Event, April 26–28, 2021, Revised Selected Papers (pp. 280-303). Cham: Springer International Publishing.

References

Aghamohammadi, Alireza, Seyed-Hassan Mirian-Hosseinabadi, and Sajad Jalali. "Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness". In: *Information and Software Technology* 129 (**2021**), p. 106426. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106426. URL: https://www.sciencedirect.com/science/article/pii/S0950584920301841 (pages 3, 11, 13, 15, 24, 39, 125, 126).

Ahmed, Iftekhar, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. "Can Testedness Be Effectively Measured?" In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, **2016**, pp. 547–558. ISBN: 9781450342186. DOI: 10.1145/2950290.2950324. URL: https://doi.org/10.1145/2950290.2950324 (pages 12, 39).

Bannard, Colin and Chris Callison-Burch. "Token and type: Combining distributional and taxonomic cues in similarity judgments". In: *Proceedings of the 20th national conference on Artificial intelligence-Volume 1* (**2005**), pp. 1149–1154 (page 95).

Beck, Kent and Erich Gamma. *Extreme programming explained: embrace change*. addison-wesley professional, **2000** (page 2).

Beller, Moritz, Georgios Gousios, and Andy Zaidman. *Oops, my tests broke the build: An analysis of travis ci builds with github*. Tech. rep. PeerJ Preprints, **2016** (page 40).

Bernardo, João Helis, Daniel Alencar da Costa, and Uirá Kulesza. “Studying the impact of adopting continuous integration on the delivery time of pull requests”. In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE. **2018**, pp. 131–141 (pages 3, 34, 37, 60, 123).

Berndt, Donald J and James Clifford. “Using dynamic time warping to find patterns in time series.” In: *KDD workshop*. Vol. 10. 16. Seattle, WA. **1994**, pp. 359–370 (page 41).

Boehm, B.W. and P.N. Papaccio. “Understanding and controlling software costs”. In: *IEEE Transactions on Software Engineering* 14.10 (**1988**), pp. 1462–1477. DOI: 10.1109/32.6191 (page 11).

Bowen, Glenn A. “Document analysis as a qualitative research method”. In: *Qualitative research journal* 9.2 (**2009**), pp. 27–40 (pages 21–23, 92).

Boyatzis, Richard E. *Transforming qualitative information: Thematic analysis and code development*. sage, **1998** (page 24).

Braun, Virginia and Victoria Clarke. “Using thematic analysis in psychology”. In: *Qualitative research in psychology* 3.2 (**2006**), pp. 77–101 (page 23).

Braun, Virginia and Victoria Clarke. “Reflecting on reflexive thematic analysis”. In: *Qualitative research in sport, exercise and health* 11.4 (**2019**), pp. 589–597 (pages 23, 25, 26).

Braun, Virginia and Victoria Clarke. “Can I use TA? Should I use TA? Should I not use TA? Comparing reflexive thematic analysis and other pattern-based qualitative analytic approaches”. In: *Counselling and Psychotherapy Research* 21.1 (**2021**), pp. 37–47 (page 24).

Braun, Virginia, Victoria Clarke, Nikki Hayfield, and Gareth Terry. *Handbook of Research Methods in Health Social Sciences*. Ed. by Pranee Liamputtong. Singapore: Springer Singapore, **2018**, pp. 1–18. ISBN: 978-981-10-2779-6. DOI: 10.1007/978-981-10-2779-

6_103-1. URL: https://doi.org/10.1007/978-981-10-2779-6_103-1 (pages 24, 27, 28).

Brooks, Joanna, Serena McCluskey, Emma Turley, and Nigel King. "The utility of template analysis in qualitative psychology research". In: *Qualitative research in psychology* 12.2 (**2015**), pp. 202–222 (page 28).

Brooks Jr, Frederick P. *The mythical man-month: essays on software engineering.* Pearson Education, **1995** (page 16).

Cassee, N., B. Vasilescu, and A. Serebrenik. "The Silent Helper: The Impact of Continuous Integration on Code Reviews". In: *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER).* Los Alamitos, CA, USA: IEEE Computer Society, **2020**, pp. 423–434. DOI: 10.1109/SANER48275.2020.9054818. URL: https://doi.ieeecomputersociety.org/10.1109/SANER48275.2020.9054818 (page 46).

Chen, Yiqun T., Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. "Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size". In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* **2020**, pp. 237–249 (page 110).

Clarke, Victoria and Virginia Braun. "Thematic analysis: a practical guide". In: *Thematic Analysis* (**2021**), pp. 1–100 (page 27).

Cook, T.D. and D.T. Campbell. *Quasi-experimentation: Design & Analysis Issues for Field Settings.* Houghton Mifflin, **1979**. ISBN: 9780395307908. URL: https://books.google.com.br/books?id=BFNqAAAAMAAJ (page 46).

Duvall, Paul M, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, **2007** (pages 2, 3, 16, 17, 122).

Elazhary, Omar, Colin Werner, Ze Shi Li, Derek Lowlind, Neil A Ernst, and Margaret-Anne Storey. “Uncovering the benefits and challenges of continuous integration practices”. In: *IEEE Transactions on Software Engineering* (**2021**) (page 123).

Felidré, Wagner, Leonardo B. Furtado, Daniel Alencar da Costa, Bruno Cartaxo, and Gustavo Pinto. “Continuous Integration Theater”. In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2019, Porto de Galinhas, Recife, Brazil, September 19-20, 2019.* IEEE, **2019**, pp. 1–10 (pages 2, 4, 17, 60, 122).

Fowler, Martin and Matthew Foemmel. “Continuous integration”. In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (**2006**), p. 14 (page 3).

Frantzi, Katerina T and Sophia Ananiadou. “Automatic recognition of multi-word terms: the C-value/NC-value method”. In: *Proceedings of the 1st international conference on Natural Language Processing and Industrial Applications.* Springer. **2000**, pp. 58–68 (page 95).

Fugard, Andi and H Potts. *Thematic analysis.* Sage, **2020** (page 23).

Gligoric, Milos, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. “Comparing Non-Adequate Test Suites Using Coverage Criteria”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis.* ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, **2013**, pp. 302–313. ISBN: 9781450321594. DOI: `10.1145/2483760.2483769`. URL: `https://doi.org/10.1145/2483760.2483769` (page 12).

Gopinath, Rahul, Carlos Jensen, and Alex Groce. “Code Coverage for Suite Evaluation by Developers”. In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: Association for Computing Machinery, **2014**, pp. 72–82. ISBN: 9781450327565. DOI: `10.1145/2568225.2568278`. URL: `https://doi.org/10.1145/2568225.2568278` (pages 11, 12, 39, 109).

Grano, Giovanni, Timofey V Titov, Sebastiano Panichella, and Harald C Gall. “Branch coverage prediction in automated testing”. In: *Journal of Software: Evolution and Process* 31.9 (**2019**), e2158 (page 126).

Guest, Greg, Kathleen M MacQueen, and Emily E Namey. *Applied thematic analysis*. sage publications, **2011** (page 24).

Guo, Yunfang and Philipp Leitner. “Studying the impact of CI on pull request delivery time in open source projects—a conceptual replication”. In: *PeerJ Computer Science* 5 (**2019**), e245 (pages 3, 60).

Hilton, Michael, Jonathan Bell, and Darko Marinov. “A large-scale study of test coverage evolution”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM. **2018**, pp. 53–63 (pages 5, 39, 50, 102, 107, 125, 126).

Hilton, Michael, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. “Trade-offs in continuous integration: assurance, security, and flexibility”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. **2017**, pp. 197–207 (pages 3, 113).

Hilton, Michael, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. “Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, **2016**, pp. 426–437. ISBN: 9781450338455. DOI: 10.1145/2970276.2970358. URL: https://doi.org/10.1145/2970276.2970358 (pages 2, 3, 17).

Homès, Bernard. *Fundamentals of software testing*. John Wiley & Sons, **2013** (page 14).

Hora, Andre. “What Code Is Deliberately Excluded from Test Coverage and Why?” In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. **2021**, pp. 392–402 (pages 4, 75).

Hora, Andre. “Excluding code from test coverage: practices, motivations, and impact”. In: *Empirical Software Engineering* 28.1 (**2023**), pp. 1–33 (page 127).

Hripcsak, George and Daniel F Heitjan. “Measuring agreement in medical informatics reliability studies”. In: *Journal of biomedical informatics* 35.2 (**2002**), pp. 99–110 (page 93).

Humble, Jez and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* 1st. Addison-Wesley Professional, **2010**. ISBN: 0321601912 (pages 4, 16).

Hyman, Ray. “Quasi-experimentation: Design and analysis issues for field settings (book)”. In: *Journal of Personality Assessment* 46.1 (**1982**), pp. 96–97 (page 17).

Imbens, Guido W. and Thomas Lemieux. “Regression discontinuity designs: A guide to practice”. In: *Journal of Econometrics* 142.2 (**2008**). The regression discontinuity design: Theory and applications, pp. 615–635. ISSN: 0304-4076. DOI: `https://doi.org/10.1016/j.jeconom.2007.05.001`. URL: `https://www.sciencedirect.com/science/article/pii/S0304407607001091` (pages 46, 60).

Inozemtseva, Laura and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: Association for Computing Machinery, **2014**, pp. 435–445. ISBN: 9781450327565. DOI: `10.1145/2568225.2568271`. URL: `https://doi.org/10.1145/2568225.2568271` (pages 12, 14).

Ivanković, Marko, Goran Petrović, René Just, and Gordon Fraser. “Code coverage at Google”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* **2019**, pp. 955–963 (pages 4, 12, 125).

Kate, Rohit J. "Using dynamic time warping distances as features for improved time series classification". In: *Data Mining and Knowledge Discovery* 30.2 (**2016**), pp. 283–312 (page 41).

Kelly J., Hayhurst, Veerhusen Dan S., Chilenski John J., and Rierson Leanna K. *A Practical Tutorial on Modified Condition/Decision Coverage.* Tech. rep. **2001** (page 14).

Kitchenham, Barbara A, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. "Preliminary guidelines for empirical research in software engineering". In: *IEEE Transactions on software engineering* 28.8 (**2002**), pp. 721–734 (page 65).

Kochhar, Pavneet Singh, David Lo, Julia Lawall, and Nachiappan Nagappan. "Code coverage and postrelease defects: A large-scale study on open source projects". In: *IEEE Transactions on Reliability* 66.4 (**2017**), pp. 1213–1228 (page 126).

Koo, Terry K. and M. Y. Li. "Guideline of selecting and reporting intraclass correlation coefficients for reliability research". In: *Journal of chiropractic medicine* 15.2 (**2016**), pp. 155–163 (page 96).

Krippendorff, Klaus. "Estimating the reliability, systematic error and random error of interval data". In: *Educational and psychological measurement* 30.1 (**1970**), pp. 61–70 (page 93).

Laukkanen, Eero, Maria Paasivaara, and Teemu Arvonen. "Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study". In: *2015 Agile Conference.* **2015**, pp. 11–20. DOI: `10.1109/Agile.2015.15` (pages 2, 16).

Lee, David S and Thomas Lemieux. "Regression discontinuity designs in economics". In: *Journal of Economic Literature* 48.2 (**2010**), pp. 281–355 (pages 20, 21).

Macbeth, Guillermo, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. "Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations". In: *Universitas Psychologica* 10.2 (**2011**), pp. 545–555 (page 60).

Mahdieh, Mostafa, Seyed-Hassan Mirian-Hosseinabadi, Khashayar Etemadi, Ali Nosrati, and Sajad Jalali. “Incorporating fault-proneness estimations into coverage-based test case prioritization methods”. In: *Information and Software Technology* 121 (**2020**), p. 106269. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2020.106269`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584920300197` (page 11).

Merriam, Sharan B. *Case study research in education: A qualitative approach.* Jossey-Bass, **1988** (pages 22, 23).

Milligan, Glenn W and Martha C Cooper. “An examination of procedures for determining the number of clusters in a data set”. In: *Psychometrika* 50.2 (**1985**), pp. 159–179 (page 41).

Nery, Gustavo Sizilio, Daniel Alencar da Costa, and Uirá Kulesza. “An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. **2019**, pp. 426–436 (pages 5, 32, 34, 36–38, 40, 60, 119, 125, 127).

Neuendorf, Kimberly A. “The content analysis guidebook”. In: *SAGE Publications* (**2016**) (page 23).

Perry, Dewayne E, Adam A Porter, and Lawrence G Votta. “Empirical studies of software engineering: a roadmap”. In: *Proceedings of the conference on The future of Software engineering.* ACM. **2000**, pp. 345–355 (page 61).

Pinto, Gustavo, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. “Work practices and challenges in continuous integration: A survey with Travis CI users”. In: *Software: Practice and Experience* 48.12 (**2018**), pp. 2223–2236 (pages 3, 57, 124).

Rajan, Ajitha, Michael W Whalen, and Mats PE Heimdahl. “The effect of program and model structure on MC/DC test adequacy coverage”. In: *Proceedings of the 30th International Conference on Software engineering.* **2008**, pp. 161–170 (page 14).

Rothermel, G., R.H. Untch, Chengyun Chu, and M.J. Harrold. “Test case prioritization: an empirical study”. In: *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ’Software Maintenance for Business Change’ (Cat. No.99CB36360).* **1999**, pp. 179–188. DOI: 10.1109/ICSM.1999.792604 (page 106).

Salton, Gerard and Michael J McGill. “Introduction to modern information retrieval”. In: (**1986**) (page 95).

Salvador, Stan and Philip Chan. “Toward accurate dynamic time warping in linear time and space”. In: *Intelligent Data Analysis* 11.5 (**2007**), pp. 561–580 (page 41).

Santos, Jadson, Daniel Alencar da Costa, and Uirá Kulesza. “Investigating the Impact of Continuous Integration Practices on the Productivity and Quality of Open-Source Projects”. In: *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement.* ESEM ’22. Helsinki, Finland: Association for Computing Machinery, **2022**, pp. 137–147. ISBN: 9781450394277. DOI: 10.1145/3544902.3546244. URL: https://doi.org/10.1145/3544902.3546244 (page 123).

Silva, José Diego Saraiva da, Daniel Alencar Da Costa, Uirá Kulesza, Gustavo Sizílio, José Gameleira Neto, Roberta Coelho, and Mei Nagappan. “Unveiling the Relationship Between Continuous Integration and Code Coverage”. In: *Proceedings of the 20th International Conference on Mining Software Repositories (MSR).* ICSME ’23. Melbourne, Australia: IEEE, **2023** (page 30).

Smith, Edward, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. “Improving developer participation rates in surveys”. In: *2013 6th International workshop on cooperative and human aspects of software engineering (CHASE).* IEEE. **2013**, pp. 89–92 (page 65).

Smith, Joanna and Jill Firth. “Qualitative data analysis: the framework approach”. In: *Nurse researcher* 18.2 (**2011**) (page 28).

Soares, Eliezio, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. “The effects of continuous integration on software development: a systematic literature review”. In: *Empirical Software Engineering* 27.3 (**2022**), pp. 1–61 (pages 3, 4, 59, 89, 122).

Someoliayi, Khashayar Etemadi, Sajad Jalali, Mostafa Mahdieh, and Seyed-Hassan Mirian-Hosseinabadi. “Program state coverage: a test coverage metric based on executed program states”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. **2019**, pp. 584–588 (pages 14, 15).

Ståhl, Daniel and Jan Bosch. “Experienced benefits of continuous integration in industry software product development: A case study”. In: *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*. **2013**, pp. 736–743 (page 3).

Terry, Gareth, Nikki Hayfield, Victoria Clarke, and Virginia Braun. “Thematic analysis”. In: *The SAGE handbook of qualitative research in psychology* 2 (**2017**), pp. 17–37 (page 26).

Tibshirani, Robert, Guenther Walther, and Trevor Hastie. “Estimating the number of clusters in a data set via the gap statistic”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63.2 (**2001**), pp. 411–423 (page 41).

Vanoverberghe, Dries, Jonathan de Halleux, Nikolai Tillmann, and Frank Piessens. “State Coverage: Software Validation Metrics beyond code coverage-extended version”. In: *CW Reports, volume CW610* 15 (**2011**) (pages 15, 39).

Vasilescu, Bogdan, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G.J. van den Brand. “Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub”. In: *2014 IEEE International Conference on Software Maintenance and Evolution*. **2014**, pp. 401–405. DOI: `10.1109/ICSME.2014.62` (pages 2, 16, 122).

Vasilescu, Bogdan, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. "Quality and productivity outcomes relating to continuous integration in GitHub". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. **2015**, pp. 805–816 (pages 3, 123).

Vassallo, Carmine, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. "Continuous Delivery Practices in a Large Financial Organization". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. **2016a**, pp. 519–528. DOI: `10.1109/ICSME.2016.72` (page 3).

Vassallo, Carmine, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. "Continuous delivery practices in a large financial organization". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. **2016b**, pp. 519–528 (pages 79, 125).

Weng, Jian, Xiaohua Li, Yalou Liu, and Jiliang Tang. "Topic detection using non-negative matrix factorization and semantic similarity". In: *Proceedings of the 2008 international conference on Web search and data mining*. ACM. **2008**, pp. 237–248 (page 95).

Wessel, Mairieli, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. "Effects of adopting code review bots on pull requests to oss projects". In: *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE. **2020**, pp. 1–11 (page 103).

Wilks, D.S. *Statistical Methods in the Atmospheric Sciences*. Academic Press. Elsevier Science, **2011**. ISBN: 9780123850225. URL: `https://books.google.com.br/books?id=IJuCVtQ0ySIC` (page 60).

Xu, Tianyin, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software". In: *Proceedings of the 2015 10th*

Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, **2015**, pp. 307–319. ISBN: 9781450336758. DOI: 10.1145/2786805.2786852. URL: https://doi.org/10.1145/2786805.2786852 (page 113).

Yin, Robert K. “Discovering the future of the case study. Method in evaluation research”. In: *Evaluation practice* 15.3 (**1994**), pp. 283–290 (pages 22, 23).

Yoo, S. and M. Harman. “Regression Testing Minimization, Selection and Prioritization: A Survey”. In: *Softw. Test. Verif. Reliab.* 22.2 (**2012**), pp. 67–120. ISSN: 0960-0833. DOI: 10.1002/stv.430. URL: https://doi.org/10.1002/stv.430 (page 106).

Zaidman, Andy, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. “Mining software repositories to study co-evolution of production & test code”. In: *2008 1st international conference on software testing, verification, and validation*. IEEE. **2008**, pp. 220–229 (page 126).

Zaidman, Andy, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining”. In: *Empirical Software Engineering* 16.3 (**2011**), pp. 325–364 (page 126).

Zampetti, Fiorella, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. “A study on the interplay between pull request review and continuous integration builds”. In: *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. **2019**, pp. 38–48 (pages 106, 124).

Zhao, Yangyang, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. “The impact of continuous integration on other software development practices: a large-scale empirical study”. In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. **2017**, pp. 60–71 (pages 37, 46, 123).

Zhu, Hong, Patrick A. V. Hall, and John H. R. May. “Software Unit Test Coverage and Adequacy”. In: *ACM Comput. Surv.* 29.4 (**1997**), pp. 366–427. ISSN: 0360-0300. DOI: `10.1145/267580.267590`. URL: `https://doi.org/10.1145/267580.267590` (pages 13, 14).