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

Understanding the Relationship between Continuous Integration and its Effects on Software Quality Outcomes

Gustavo Sizílio Nery

Natal-RN

February, 2020

Gustavo Sizílio Nery

Understanding the relationship between Continuous Integration and its effects on software quality outcomes

A thesis submitted to the Graduate Program in Systems and Computing (PPGSC) in conformity with the requirements for the Degree of Doctor of Philosophy

Supervisor

Prof. PhD. Uirá Kulesza

Co-supervisor

Prof. PhD. 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

February, 2020

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

Nery, Gustavo Sizílio.
Understanding the relationship between continuous integration and its effects on software quality outcomes / Gustavo Sizílio Nery. - 2020.
147f.: il.

Tese (Doutorado) - Universidade Federal do Rio Grande do Norte, Centro de Ciências Exatas e da Terra, Programa de Pós-Graduação em Sistemas e Computação. Natal, 2020.
Orientador: Uirá Kulesza.
Coorientador: Daniel Alencar da Costa.

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

RN/UF/CCET CDU 004

Elaborado por Joseneide Ferreira Dantas - CRB-15/324

GUSTAVO SIZÍLIO NERY

"*Understanding the Relationship between Continuous Integration and its Effects on Software Quality Outcomes*"

Esta Tese foi julgada adequada para a obtenção do título de doutor 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.

Prof. Dr. UIRÁ KULESZA (UFRN)
Presidente – Orientador

Profa. Dra. ANNE MAGALY DE PAULA CANUTO
Coordenadora do PPgSC

Banca Examinadora

Prof. Dr. DANIEL ALENCAR DA COSTA (University of Otago – NZ)
Examinador Externo

Prof. Dr. EDUARDO HENRIQUE DA SILVA ARANHA (UFRN)
Examinador Interno

Prof. Dr. GUSTAVO HENRIQUE LIMA PINTO (UFPA)
Examinador Externo

Prof. Dr. RODRIGO BONIFÁCIO DE ALMEIDA (UnB)
Examinador Externo

Discente: **GUSTAVO SIZÍLIO NERY**

Fevereiro, 2020

Dedico este trabalho à minha amada família, especialmente minha mãe, esposa e filhos, por permanecerem ao meu lado nos momentos mais difíceis e desanimadores. Nada faria sentido sem eles ao meu lado.

Agradecimentos

Durante uma jornada de aproximadamente 54 meses, pouco mais de quatro anos de muitos desafios e sacrífícios, nada teria evoluído sozinho. Não foram poucos os momentos de desânimo e nem poucas as atribulações que me fizeram pensar em desistir. Entretanto, Deus provê os mecanismos necessários para te fazer lembrar que tudo está sob controle e que, com Ele, nada é em vão. Somente a Ele agradeço de joelhos e me coloco sob comando.

Ainda em reconhecimento à Deus, é preciso estender minha gratidão àqueles colocados de forma estratégica para me manter de pé. Pois Ele saberia onde eu cairía, mas também saberia o necessário para me fazer levantar. É dessa maneira, repleto de reconhecimento do papel gracioso dessas pessoas em minha vida, que preciso registrar minha eterna gratidão.

Agradeço, portanto, ao meu mestre, Prof. Uirá Kulesza, por depositar em mim confiança na capacidade de realizar um bom trabalho. Foi nesta confiança que me firmava para acreditar que tudo teria, em seu desfecho, êxito e alegria. Todo ensinamento e cada conselho, durante toda esta jornada, fazem parte de mim, do ser humano e pesquisador que sou. Traços de uma pessoa melhor e mais capacitada que vai tentar levar o seu nome por toda minha jornada.

É preciso, ainda, agradecimentos especiais ao Prof. Daniel Alencar da Costa. Aqui confesso que não sei como apresentá-lo, se como mestre coorientador, amigo ou família. É notório em minha confusão que o seu convívio foi mais do que importante para mim. É preciso reconhecer o esforço incondicional empenhado em não me deixar desviar do meu caminho, sempre com conselhos e suporte que nunca terei como retribuir. Ao amigo, deixo os meus mais profundos agradecimentos,

certo de que não pouparei esforços para honrar tamanha competência que fora imprimida na sua tarefa de coorientador.

Manifesto também os meus respeitosos agradecimentos aos demais membros examinadores do meu trabalho. Meu muito obrigado, pelas orientações e pelo tempo dedicado na banca examinadora da qualificação e da defesa. Em especial, ao Prof. Eduardo Aranha, que, além de membro da banca, participou deste processo desde o início. Por me receber no programa de doutorado e me acompanhar até o desfecho, sempre manifestando interesse e palavras animadoras, o meu muito obrigado.

É preciso lembrar, ainda, das pessoas que não fizeram diretamente parte do processo acadêmico, mas que foram, cada um de uma maneira diferente, fundamentais para que todo o restante de mim entrasse em harmonia. Agradeço à minha amada família, especialmente minha mãe, esposa e filhos, por se mostrarem sempre dispostos a me apoiar em todas minhas decisões. Sei que não foi fácil também para vocês, mas agradeço pela paciência e pelo carinho. Por fim, agradeço aos meus amigos, por não se deixarem distanciar mesmo tendo sido eu uma pessoa tão ausente. Vocês fizeram parte de tudo isto, muito obrigado!

Don't accept that what's happening is just a case of others' suffering or you'll find that you're joining in "The turning away"

Gilmour & Moore, 1987

Compreendendo o relacionamento entre a Integração Contínua e seus efeitos nos resultados da qualidade do software

Autor: Gustavo Sizílio Nery
Orientador: Prof. PhD. Uirá Kulesza
Coorientador: Prof. PhD. Daniel Alencar da Costa

Resumo

Integração Contínua, em inglês Continuous Integration (CI), consiste na prática de automatizar e melhorar a frequência das integrações de código (por exemplo, através de builds diários). CI é frequentemente considerado um dos principais elementos que dão suporte às metodologias ágeis. CI ajuda a reduzir os riscos de integração no desenvolvimento de software, através de builds e testes automatizados, permitindo que equipes corrijam problemas de integração imediatamente. A adoção de CI pode ajudar as equipes a avaliar e melhorar a qualidade dos sistemas de software. Os potenciais benefícios de CI chamaram a atenção de pesquisadores da engenharia de software que buscam entender, de forma empírica, as possíveis vantagens de adoção da prática. Estudos anteriores evidenciam o impacto da adoção de CI em diversos aspectos do desenvolvimento de software. Apesar dos valiosos avanços, muitas suposições permanecem empiricamente inexploradas na comunidade.

Nosso trabalho investiga, de forma empírica, os fatores da qualidade do software e suas relações com a adoção de CI. Como contribuição, esta tese fornece um mapeamento sistemático da literatura, que apresenta um amplo cenário de como

profissionais e pesquisadores observam o efeito de CI nos aspectos relacionados ao produto de software. Além disso, melhoramos algumas premissas, realizando dois estudos empíricos, visando responder às seguintes questões ainda em aberto: (i) A adoção de CI está associada à evolução do código de teste dos projetos? (ii) O nível de aderência às práticas de CI está relacionada a uma melhoria da qualidade do código fonte dos projetos? Por fim, nós apresentamos um estudo pioneiro, considerando o nosso contexto de pesquisa, que vai além dos testes de correlação e investiga o efeito causal entre a adoção de CI e testes automatizados. Para isto, aplicamos uma inferência causal, através da utilização de diagramas causais e métodos probabilísticos, para determinar a efeito de CI nos testes automatizados. Nossos resultados sugerem que, apesar dos trade-offs relacionados à adoção de CI, é provável que a prática esteja associada à melhorias na qualidade do software. Além disso, CI emprega um efeito causal positivo e considerável no volume de testes dos projetos.

Palavras-chave: integração contínua, qualidade de software, estudo empírico, inferência causal.

Understanding the relationship between Continuous Integration and its effects on software quality outcomes

Author: Gustavo Sizílio Nery
Supervisor: Prof. PhD. Uirá Kulesza
Co-supervisor: Prof. PhD. Daniel Alencar da Costa

Abstract

Continuous Integration (CI) is the practice of automating and improving the frequency of code integration (e.g., daily builds). CI is often considered one of the key elements involved to support agile software teams. It helps to reduce the risks in software development by automatically building and testing a project codebase, which allows the team to fix broken builds immediately. The adoption of CI can help development teams to assess the quality of software systems. The potential benefits of adopting CI have brought the attention of researchers to study its advantages empirically. Previous research has studied the impact of adopting CI on diverse aspects of software development. Despite the valuable advancements, there are still many assumptions in the community that remains empirically unexplored.

Our work empirically investigates the software quality outcomes and their relationship with the adoption of CI. This thesis provides a systematic literature mapping that presents a broad knowledge of how practitioners and researchers recognize the CI practice to affect software product-related aspects. Additionally, we improve some assumptions by performing two empirical studies that aim to answer the following open questions: (i) Does the adoption of CI share a relationship

with the evolution of test code? (ii) The adherence to CI best practices is related to the degree of code quality? Finally, we present a pioneering study that goes beyond the correlation tests to investigate the estimated causal effect of CI adoption and its impact on automated tests. Thereby, we apply a causal inference using directed acyclic graphs and probabilistic methods to determine the causal effect of CI in automated tests. Our results suggest that, despite the CI adoption trade-offs, it is likely to be associated with improvements in software quality. Additionally, it employs a considerable positive causal effect on the volume of automated tests.

Keywords: Continuous integration, software quality, empirical study, causal inference.

List of Figures

List of Tables

Contents

1 Introduction

Continuous Integration (CI) is the practice of automating and improving the frequency of code integration (e.g., daily builds) (FOWLER; FOEMMEL, 2006). The adoption of CI can help development teams to assess the quality and to reduce the risks (DUVALL; MATYAS; GLOVER, 2007) in software development.

Over history, software quality has been one of the main concerns of engineers as it represents an essential attribute for the success of every software project (CROWSTON; ANNABI; HOWISON, 2003; REEL, 1999; CHOW; CAO, 2008). In this matter, software engineers are constantly developing enhancements on development methods to produce better results. They assume that a high quality *software process* will likely produce a high quality *software product* (HUMPHREY, 1988; SOMMERVILLE, 2011). As software quality represents a broad area of study, and as it might be affected by multiple aspects in software development, this thesis focuses on the quality to witch regards the quality of the code and the tests. We acknowledge that software quality is more than that. However, we believe that it is a base for reliable and consistent CI usage.

Similarly to any business, quality becomes a big concern also in the software community. High-quality software provides a better overall experience to end-users in both open source and corporate solutions. The competition for better software enhance the market and become a mandatory concern in successful companies. A software that is built in an environment that neglect quality can cause extreme side effects on the business and can affect people's life. Critical software can also

put human life in risk (KNIGHT, 2002).

In this regard, software development methods have evolved over the time. Continuous integration become one of the most important practice within software processes and has gained considerable attention from both research and industry (MÅRTENSSON; STÅHL; BOSCH, 2019; PINTO; REBOUÇAS; CASTOR, 2017; BELLER; GOUSIOS; ZAIDMAN, 2017b; VASILESCU et al., 2014; HILTON et al., 2016a; VASILESCU et al., 2015a; aO et al., 2017; BELLER; GOUSIOS; ZAIDMAN, 2016; LUZ; PINTO; BONIFÁCIO, 2018; ZHANG et al., 2018; WIDDER et al., 2019a).

The adoption of CI can help development teams to assess the quality of the product by promoting the execution of automated tests (FOWLER; FOEMMEL, 2006). Such automated tests can help development teams to detect errors earlier in the project life cycle (FOWLER; FOEMMEL, 2006). Continuous integration is often considered one of the key elements involved to support agile software teams (STOLBERG, 2009). CI is also considered to reduce the risks (DUVALL; MATYAS; GLOVER, 2007) in software development by automatically building and testing a project codebase, which allows the team to fix broken builds immediately (FOWLER; FOEMMEL, 2006).

The potential benefits of adopting CI have brought the attention of researchers to study its advantages empirically. Previous research has studied the impact of adopting CI in diverse aspects of software development (BERNARDO; COSTA; KULESZA, 2018a; VASILESCU et al., 2015a; ZHAO et al., 2017; HILTON et al., 2017a; LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017). Vasilescu et al. (VASILESCU et al., 2015a) studied the quality and productivity outcomes with respect to CI on GitHub projects. The authors found that CI improves productivity without an observable diminishment in code quality. However, Vassallo et al. (VASSALLO et al., 2018) investigated a core principle behind CI, the Continuous Code Quality, and revealed a strong dichotomy between theory and practice, i.e., developers do not perform the continuous inspection. Instead, developers that use CI tend to control for quality only at the end of a sprint and, most of the times, only on the release

branch. Such research provides valuable insight into the impact of CI adoption in software quality outcomes.

1.1 Problem Statement

Continuous integration becomes a consensus in software engineering. Adopting CI is perceived by practitioners as something that reflects positive outcomes in software quality aspects. However, the relationship between CI and build systems might give the sensation that only automatizing builds might lead the projects to improve the outcomes. Nevertheless, recent research work have proven that this sensation represents something spurious (MÅRTENSSON; HAMMARSTRÖM; BOSCH, 2017; FELIDRÉ et al., 2019) and might influence decision-makers to execute wrong decisions, i.e., to neglect the efforts and practices necessary to experience the benefits. Thus, better understanding the real scenarios where CI is associated with quality improvement is something essential.

Despite the valuable advancements, there are still many assumptions in the community that remains empirically unexplored. Most decisions are made in common sense that the results are positive, but there is still an unclear idea of the trade-offs and the real impact on the quality outcomes. Additionally, most of the studies in the area are designed to determine associations between the variables (YU et al., 2016; VASILESCU et al., 2015b; WIDDER et al., 2019b; RAHMAN; ROY, 2017; BERNARDO; COSTA; KULESZA, 2018b; GUPTA et al., 2017a; YU et al., 2015; ZAMPETTI et al., 2019; GREN, 2017), leaving open questions about the causal relationships implied by CI. Our work empirically investigates the software quality outcomes and their relationship with the adoption of CI.

1.2 Thesis Proposal

This thesis provides a systematic literature mapping that presents a broad knowledge of how practitioners and researchers recognize the CI practice to affect software product-related aspects. Additionally, we improve some assumptions by performing two empirical studies that aim to answer the following open questions: (i) Does the adoption of CI share a relationship with the evolution of test code? (ii) The adherence to CI best practices is related to the degree of code quality? Finally, we present a pioneering study that goes beyond the correlation tests to investigate the estimated causal effect of CI adoption and its impact on automated tests. Thereby, we apply a causal inference using directed acyclic graphs and probabilistic methods to determine the causal effect of CI in automated tests. This investigation is essential for decision-makers (e.g., team leaders) to better understand whether CI can improve software quality in the long run.

1.3 Thesis Overview

In this section we provide an overview of the thesis scope. Below we show how chapters are organized, which studies we performed, and what are the results of the current studies.

Chapter 2: Background

In Chapter 2, we present the core concepts of our research to provide to the reader the basis to understand our empirical studies. We first explain in more detail the *continuous integration* practice and how it relates to the agile software processes and teams. Additionally, we provide background about *Software Testing & Coverage*, which is a core concept in CI that was investigated in one of our studies, presented in Chapter 3. Finally, we provide a base knowledge about *Continuous Inspection & Continuous Code Quality*, which is related to the quality outcomes investigated in Chapter 4 and an overview about *Causal Analysis with DAG's*.

Chapter 3: An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution

Continuous Integration (CI) is the practice of automating and improving the frequency of code integration. CI has been widely adopted by software development teams and has brought the attention of researchers to study its benefits. Existing research shows that CI can improve software quality by identifying the errors earlier in the software development life-cycle. One question that remains open, however, is whether CI increases the adoption of testing practices in software projects. In Chapter 3 we investigate the evolution of software tests and its relationship with the adoption of continuous integration. We set out to compare 82 projects that adopted CI (CI projects) and 82 projects that have never adopted CI (NOCI projects). In total, we studied 3,936 versions of our studied projects to investigate trends on the test code ratio and coverage.

Chapter 4: On the Continuous Code Quality Outcomes of Continuous Integration: An Empirical Study

Motivated by the research results that show that CI may not be related to a improve on quality outcomes (FREITAS, 2019) and that projects that adopt CI may not be using CCQ in practice (VASSALLO et al., 2018), in Chapter 4, we investigate whether adherence to CI best practices is related to the degree of code quality. Hence, we empirically study 184 open source projects that use TravisCI and SonarCloud to evaluate the relationship between the CI best practices and software quality outcomes.

Chapter 5: An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution

Our first results, among other valuable contributions in the community, show that CI might share a relationship with an improvement in some quality aspects. We also observed that the associations are not always only due to CI adoption. This scenario shows that CI is somehow linked to a gain in software quality, even that is

a weak association. This situation opens the discussion about the role of if CI in the association, i.e., if it might be considered the cause of such gainings. To explore this question, Chapter 5 presents a study that goes beyond the correlation tests and investigates the estimated causal effect of CI adoption and its impact on automated tests. Thereby, it shows a systematic literature mapping that draws the results of researches about CI impact. We use these assumptions to perform a causal inference using directed acyclic graphs and probabilistic methods to determine the causal effect of CI in automated tests.

Chapter 6: Related Research

In this Chapter 6, we position our work with respect to the previous and related research. We discuss the works about CI with regarding of *Continuous Code Inspection & Software Quality in CI* ((HILTON et al., 2016b; VASILESCU et al., 2015a; VASSALLO et al., 2018; FELIDRÉ et al., 2019; MARCILIO et al., 2019)) and *Software Test Evolution & Coverage in CI* ((ELBAUM; GABLE; ROTHERMEL, 2001; HILTON; BELL; MARINOV, 2018; ZAIDMAN et al., 2008, 2011; GRANO et al., 2019; BELLER; GOUSIOS; ZAIDMAN, 2016; LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017; ZHAO et al., 2017)).

1.4 Thesis Contributions

The studies developed for this thesis presents the following main results:

- CI projects have more projects with a rising test ratio trend. We found that 33 out of 82 (40.2%) CI projects have a rising test ratio trend, while only 14 out of 82 (17%) NOCI projects have a rising test ratio trend – (Chapter 3).
- We observe that the adoption of CI is associated with a consistent increase of test ratio (MWW $p - value = 2.908e - 10$ and a small Cliff's $delta = -0.1612838$), while NOCI have a negligible change on the test ratio overtime

(MWW $p-value = 0.0001842$ and a negligible Cliff's $delta = -0.09270482$) – (Chapter 3).

- Our results reveal that CI projects likely obtain a higher test ratio growth than NOCI projects. – (Chapter 3)

- Our analysis shows statistical evidences that most of the analyzed CI projects tend to increase or maintain the test coverage (9 out of 10 projects), while NOCI projects have a different tendency (5 out of 10 projects increase or maintain the test coverage). In fact, NOCI projects have more projects decreasing the coverage (5 projects) when compared to CI projects (1 project) – (Chapter 3).

- Our mixed-effect models reveal that test ratio is largely explained by the project inherent context rather than by code or process factors – (Chapter 3).

- Although the project size expresses the most powerful explanation power on the technical debt, our study shows that maintaining a short build duration and high code coverage is also important to reduce the technical debt. – (Chapter 4).

- Technical debt is largely explained by the project size, outpowering the positive effects of short build durations and high code coverage – (Chapter 4).

- Our Systematic Mapping results shows that the relationship between CI and software factors is not always straightforward, and there is a lack of empirical researches to explain some aspects.

- Despite the CI adoption trade-offs, ou causal modeling shows that it is likely to be associated with improvements in software quality. Additionally, it employs a considerable positive causal effect on the volume of automated tests (nearly 54% of probability).

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the background material to the reader. Chapter 3 presents our empirical study to investigate the evolution of software tests and its relationship with the adoption of continuous integration. Chapter 4 presents our empirical study to evaluate the relationship between the CI best practices and software quality outcomes. Chapter 5 presents our causal study to evaluate the if is possible to determine causality effect between CI and the evolution of software tests. Chapter 6 situates this thesis with respect to related research. Finally, Chapter 7 draws the conclusions and summarize the contributions of this thesis.

2 Background

2.1 Continuous Integration

Continuous Integration (CI) is the practice of automating and improving the frequency of code integration (e.g., daily builds) (FOWLER; FOEMMEL, 2006). The adoption of CI can help development teams to assess the quality and to reduce the risks (DUVALL; MATYAS; GLOVER, 2007) in software development.

The integration of software work is a problem that is directly proportional to the complexity of the system and the size of the team (FOWLER; FOEMMEL, 2006). As huger the team or more complex is the architecture (i.e., more external dependencies of components) the higher to probability to have integration problems. The integration problem becomes evident when multiple pieces of work performed by different people must work together. Get these works to connect seamlessly is a dream, problems often occur, and many adaptations are needed to get everything working properly. Thus, integrating is a problem that the community assumes that we cannot avoid. The philosophy of continuous integration claims to mitigate those problems by increasing the regularity and systematically checking for integrations problems and immediately fixing it.

The process of continuously integrate require habits that are not easy to adhere to. Automated builds, a thorough test suite, and commit to the mainline branch every day are essential practices that sounds simple at first, but they require a responsible team to implement and constant care (MEYER, 2014). What starts

with improved tooling can be a catalyst for long-lasting change in your company's shipping culture (MEYER, 2014).

As ilustrated in Figure 1, the base of the continuous integration practice consistins on, firstly, that developers work locally and commit frequently to a version control system (e.g., Git or SVN). The CI server (e.g., TravisCI, Jenkings, CircleCI, etc.) monitors the repository and checks out changes when they occur. After that, the CI server builds the system and runs the automated tests. If the tests pass, the CI server releases the deployable artefacts, assign a label to the release and inform the team about the new successful build. If the build or tests fail, the CI server alerts the team to fix that issue at the earliest opportunity. Finally, this process is repeatedly performed generating successful builds of broken builds that should last quickly (FOWLER; FOEMMEL, 2006).

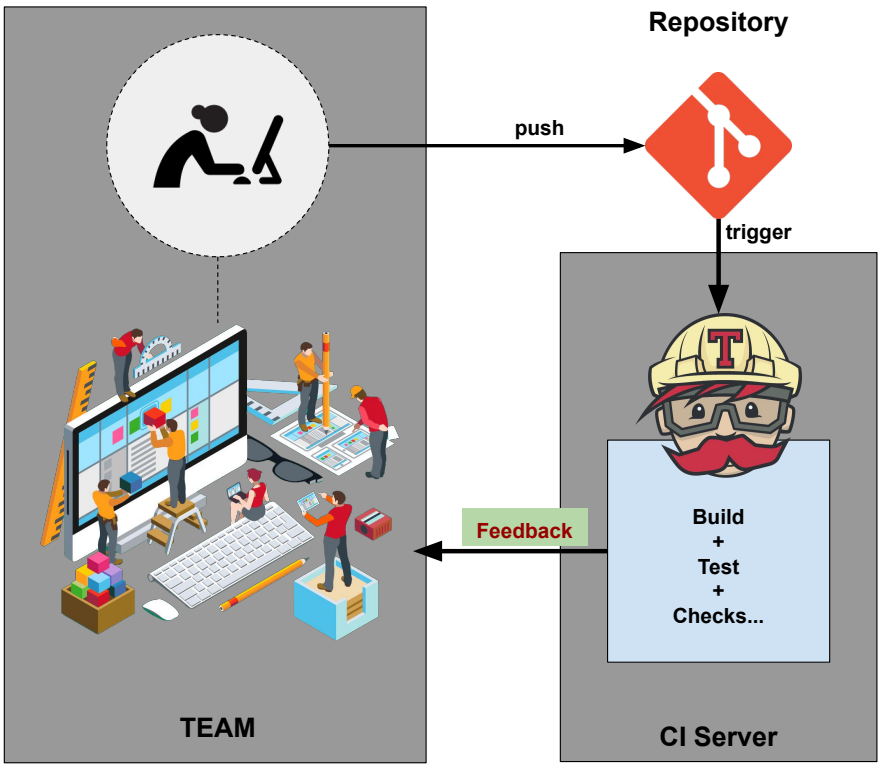


Figure 1: Continuous Integration process overview

2.2 Software Testing & Coverage

Automated tests play an important role in CI (FOWLER; FOEMMEL, 2006) as they represent a fundamental part of the build automation process to detect errors as quickly as possible. Automated tests consists in scripts that are executed in an automated test tool to verify test requirementes.

Although manual testing is also essential and can be used as an additional process to cover missed aspects on the automated testing, it is the automated tests that are responsible for catching integration errors that do not cause the build to break. It means that a build can be successfully generated but can also include errors if the automated tests are not appropriately implemented to capture these issues. Controlling for a high degree of code coverage is a way to ensure that the team is testing and that the process of continuous integration can accurately detect integration problems. Testing represents a central role in CI, and the team must employ the necessary effort to keep a good quality of the tests. In fact, Duvall (DUVALL; MATYAS; GLOVER, 2007) mentioned that without automated tests, a project should not be considered to be adopting CI at all (DUVALL; MATYAS; GLOVER, 2007).

Code coverage metrics are often used to identify whether an application is well-tested. It represents the degree to which the source code of a program is executed by automated tests. The larger the coverage the more of the source code is exercised by the tests and the greater the possibility of finding a latent problem. There are several metrics to represent code coverage (ELBAUM; GABLE; ROTHERMEL, 2001), such as statement coverage and branch coverage. Statement coverage represents the ratio of statements that are executed by automated tests over the total number of statements in the program. Branch coverage measures which possible program branches (e.g., if statements, loops) have been executed at least once during the tests. Collecting coverage metrics requires running automated tests from a test suite, instrumenting the code execution and observing the execution flow.

2.3 Continuous Code Inspection & Continuous Code Quality

As shown in the sections before, automated tests play an essential role in software quality. However, quality assurance is a more complicated process and must involve a systematic way to check for it (VASSALLO et al., 2018). Duvall et al. (DUVALL; MATYAS; GLOVER, 2007) advocated about the needs to check for software quality in every build. The author presents the Continuous Inspection, also known as Continuous Code Quality (CCQ), which consists of an additional step on the CI pipeline that specifically performs quality analyses and generates reports. In a properly configured CQQ environment, every build on the CI service triggers a quality inspect analysis that applies a specific level of criteria tho ensure the quality outcomes. If the quality result does not fulfill the criteria, the build is considered broken, and developers must fix it.

The continuous code inspection process performs automatic static analysis of the code to detect bugs, code smells, and others code issues. Static code analyses are the process to check the code without running it. Adversely to the automated tests, the static analyses does not execute any code. It just read the source to collect possible problems or defects. It is important to say that not all code issues can break a build. It is a configurable step, and teams must ensure the thresholds based on rules. The purpose of CQQ is to help developers to write code with improved quality.

SonarQube, or its online version called SonarCloud, is one of the most used tools to perform continuous code inspection. It scans the code and detects bugs, vulnerability nad code smell. A bug is considered a coding error that will break the code and needs to be fixed immediately. The vulnerability consists of a point in your code that is open to attack. Finally, code smell is an issue that makes your code confusing and difficult to maintain (SONARSOURCE, 2019). Those issues, if not controlled, can adversely affect the software maintainability and affect its

quality.

2.4 Modeling Causal assumptions

The Causal Modeling considers that any causal inferences from observational studies must ultimately rely on some kind of causal assumptions and gives an effective language for making those assumptions precise and explicit, so they can be isolated for deliberation or experimentation and, once validated, integrated with statistical data (PEARL, 2019).

The literature shows that Causal Modeling is a well-established approach and widely used in other areas (WINSHIP, 2007; VANDERWEELE, 2015; MORTON; FRITH, 1995; PEARL, 2019). For the best of our knowledge, all the current works that statistically investigate the adoption of CI rely on association relationship. None of them presented a more in-depth study to try to investigate causality. This chapter presents a study uses the Causal Modeling approach to investigate the causality between CI and software quality.

Structural-Equations Models (SCM) or simply Causal Systems Models consists in a mathematical equation to describe assumptions of how variables in a system interact with each other, describing the causal relationship between them. According to Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016), an SCM consists of a set of functions that defines the value for each variable in the system based on the values on the other variables. The causation inference uses the SCM model to define causation. According to the authors, a variable X is a direct cause of a variable Y if X appears in the $function$ that assigns Y's value. In other words, X is a cause of Y if it is a direct cause of Y, or of any cause of Y.

To illustrate the usage of SCM, consider the function $f_z : Z = X + Y$, where Z represents the overall number of bugs fixed in a project during the year, X the number of new features introduced during the period and Y represents the pressure of the client to deliver new features. In this case, both X and Y are present in the

function f_z, consequently both define Z and both might cause Z.

SCM may be an excellent approach to represent causal relations, just as we did in our example. However, we provide a simple example, and the causal relationships might consider more complex scenarios, i.e., it might have many variables, and the relation between them might not be so simple. In those cases, the SCM may represent a very complex equation, hard to express and worse to understand. To provide a more human-readable expression of the causal functions, the SCM is likely to be associated with graphical models. For this purpose, we express our SCM using directed acyclic graphs (DAGs).

A DAG is a graphical representation of the causal assumptions. Likewise, we can assume that a DAG expresses all the observed variables of the domain and how these variables correlate with each other, just like the SCM but in a visual approach. Fig.2 express our issue resolution SCM in the analogue DAG. The arrow indicate a direct causal effect, i.e., Z are directed influenced by X and Y.

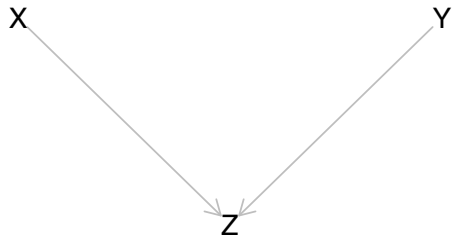


Figure 2: Sample causal dag of issue resolution

Analog to our previous example, consider the scenario where *the number of new features introduced during the period* (X) and *the pressure of the client to deliver new features* (Y) impact on textitthe overall number of bugs fixed (Z) but, additionally, *the pressure of the client to deliver new features* (Y) also shows a direct impact on the *the number of new features introduced during the period* (X). As shown in Fig.3, X might cause Z directly but this cause might be biased by the impact that Y also cause on Z. In that case, we say that X and Z share a common cause, and it represents a confounding bias. In cases where the causal effect between variables marked with some bias, we need to intervene on the model

and use data to test the causal implication.

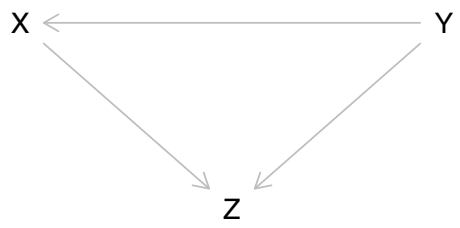


Figure 3: Sample causal dag of issue resolution with cofouding variables

2.4.1 Causal Discovery & Estimation

Computing causal effect requires an analysis of the underlying data behind the model. Consequently, our model must fit appropriately to the data. For example, if we have a model where an observed variable X causes a direct impact on Z, then X and Z must be correlated. Otherwise, our model does not correspond to reality, and we must enhance for a better model. Identifying causal effect with a model that fits the reality of the data is one condition to the analysis.

The term dependence in a graph, usually represented by connectivity, may refer to mathematical, causal, or statistical dependencies (GREENLAND; PEARL, 2011). All descendent node in the graph are dependent of its parent. In consequence, all node must be independent from its nondescendents. These constraints in the model must be held from the data. Dependence and independence are also present bwtween nodes that is not in direct association. It means that we also need to comprehend the dependency in the context of *chains*, *forks* and *coliders*. Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) defines *chains* as a sequence of three nodes and two edges, with one edge directed into and one edge directed ou of the middle variable ($A \rightarrow B \rightarrow C$); *forks* as three nodes with two arrows emanating from the mediator variable ($A \leftarrow B \rightarrow C$); and *colliders* as one node that receives edges from two other nodes ($A \rightarrow B \leftarrow C$).

In a *chain* $A \rightarrow B \rightarrow C$, the factors A and C are dependent but become

independent if we condition on B, i.e., A and C are *conditional independent*. Conditioning a variable is analog to filtering the data. A and C are conditional independent, in this case, because we can condition on B to consider only the values where A and C are independent. According to Greenland et al. (GREENLAND; PEARL, 2011), two variables, A and C, are **conditionally independent given $\boldsymbol{B}$**, if there is only one unidirectional path between A and C and B is any set of variables that intercepts that path. Regarding the *forks* $A \leftarrow B \rightarrow C$, the authors state that if a variable B is a common cause of variables A and C, and there is only one path between A and C, then A and C are **independent conditional on $\boldsymbol{B}$**. Finally, for *colliders*, If a variable B is the collision node between two variables A and C, and there is only one path between A and C, then A and C are **unconditionally independent** but are **dependent conditional on $\boldsymbol{B}$ and any descendants of $\boldsymbol{B}$**.

In a DAG, the sequence of nodes connecting two variables, disregarding the direction of the arrows, represents a *path*. For example, in Fig.2, there is a path from X to Y represented by the nodes $X - Z - Y$. A *directed path*, however, considers the paths that can be traced along with the arrows, i.e., there is no directed path between X and Y in Fig.2. According to Textor (TEXTOR, 2015), *causal paths* represents the directed paths from the exposure to the outcome and *biasing paths* contains all the other paths.

Conditioning on variables is an essential technique to control the bias. Since we cannot manipulate variables, i.e., we rely only on observational data, conditioning is used as a substitute for experimental control, in the hopes that with sufficient conditioning, X will be independent of uncontrolled influences. In other hand, is important to notice that conditioning variables might change the behavior of directed paths just like it changes the dependency of variables. For example, conditioning B a chain $A \rightarrow B \rightarrow C$, blocks that path from A to C. Generally, a path is blocked if conditioning Z when: (i) Z is in a chain $X \rightarrow P_z \rightarrow Y$ or a fork $X \leftarrow P_z \rightarrow Y$ where Z is in P_z or (ii) a collider $X \rightarrow P_z \leftarrow Y$ where P_z does not

contain Z or any successor of Z. In summary, conditioning a mediator in chains or forks blocks the open path, while conditioning a mediator (or any successor of a mediator) in a collider opens the blocked path. Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) says that two variables are d-separated if there is no open path connecting them, otherwise they are d-connected. Two d-separated variables are definitely independent while d-connected variables are likely dependent.

Causal inferences on model seek for model adjustment to provide a valid representation that minimizes the bias when estimating the causal effect of the exposure to the outcome. The adjustment sets can determine a model to *total effect* when all biasing paths are closed, leaving all causal paths opened. Additionally, the adjustment setting can also acquire a model to *direct effect* when all biasing and causal paths are closed, remaining only the direct and immediate arrow from the exposure to the outcome. The purpose of the analysis is to estimate the effect considering the *minimal adjustment set* possible to *total effect* or *direct effect*.

The estimation of causal effects consists in the execution of an effect measure on the underlying model, considering the minimal adjustment set utilized, which allows the reasoning of independence conditions implied by the assumptions. Probabilistic inference, used to obtain a probability distribution of variables of interest given the data from previous observations (SHACHTER, 1988), is one of the most used in the literature.

An quick example, considering an possible analysis performed on the bug fixing problem, presented on the previous section and illustrated int the Fig. 3, shows that to estimate the causal effect of X on Z we need to adjust for controlling Y (i.e., the minimal adjustment set), this is explained by Fig. 4. The figure shows that the problem have two open paths, one *causal path* ($X \rightarrow Z$) and one *biasing path* ($X \leftarrow Y \rightarrow Z$). Consequently, controlling Y blocks the path $X \leftarrow Y \rightarrow Z$ and eliminate the bias. Finally, the graphical representation allow the capture of the probabilistic information and can be expressd by the equation: $\sum_Y P(Z|Y,X)P(Y)$, i.e., the causal effect of X on Z is equal to the sum, for every value $Y = y$ of the probability

that Z occurs, given that Y and Z also occurs, plus the probability that Y occurs.

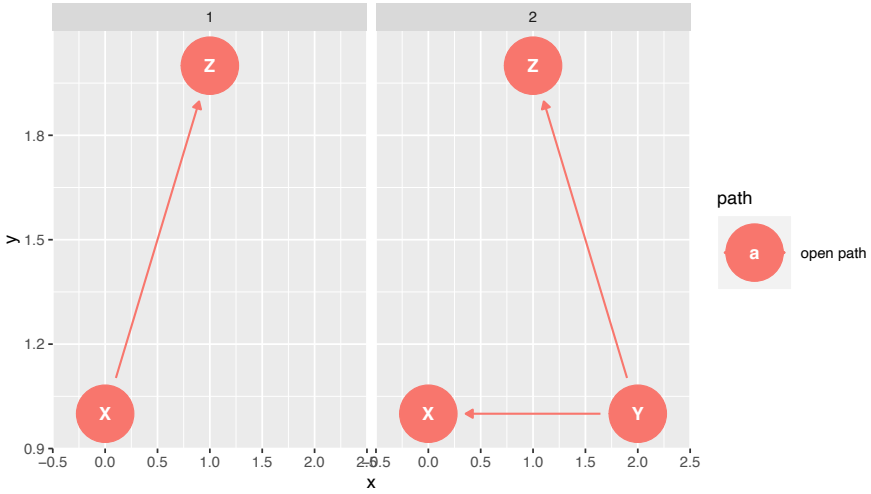


Figure 4: Open paths on the issue resolution problem from X to Z

2.4.2 The Backdor Criterion

As shown in the fundamentals explained in the previous sections, a DAG expresses pathways between variables. According to Textor (TEXTOR, 2015), *causal paths* represents the directed paths from the exposure to the outcome and *biasing paths* contains all the other paths. This subsection explores one specific case of biasing paths, the backdoor paths. A backdoor path is a biasing path that starts with an arrow pointing to the exposure. For example, if it is intended to estimate the causal effect of X in Y on the DAG presented by the Figure 3, it will be necessary to control the spurious association between X and Y that is expressed by the backdoor path $X \leftarrow Z \rightarrow Y$. Notice that the backdoor path might be an open, and spurious, path. In such cases, it is necessary to close the backdoor.

The adjustment set that closes all the backdoor paths is known as backdoor criteria. Peal et atl. (PEARL; GLYMOUR; JEWELL, 2016) presented the following definition: *given an ordered pair of variables (X, Y) in a directed acyclic graph G, a set of variables Z satisfies the backdoor criterion relative to (X, Y) if no node in*

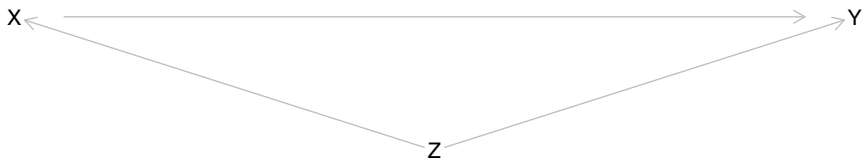


Figure 5: Sample causal dag with backdoor path

Z is a descendant of X, and Z blocks every path between X and Y that contains an arrow into X.

In summary, the backdoor criterion applies the following rules:

- Block all the spurious paths between the exposure and the outcome as they represent a bias on the causal effect.
- Leave all the directed path from the exposure to the outcome unperturbed as they are representative of the causal effect.
- Conditioning on variables in the backdoor might open new spurious paths. The backdoor criterion should not create new spurious paths when conditioning.

Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) state that if the set of variables Z satisfies the backdoor criterion for X and Y, then the causal effect of X in Y is given by the formula:

$$P(Y|do(X)) = \sum_{Z} P(Y|X,Z)P(Z) \tag{2.1}$$

Additionally, if Z represents an empty set, i.e., no backdoor adjustment is needed, then the causal effect of X in Y can be simply:

$$P(Y|do(X)) = \sum_{Z} P(Y|X) \tag{2.2}$$

2.4.3 The Front-door Criterion

The backdoor criterion is used to eliminate the effect of the spurious path and remove bias when estimating the causal effect. However, in some situations applying the backdoor criterion is not feasible. As a DAG, in the context of causal analysis proposed by Judea Pearl, is a representation of the relationships between variables in the world, it is important to represent the knowledge, including all the assumptions that we consider as truth. Hence, it is likely that our model contains variables that we do not observe. This scenario is illustrated in Figure 6. Notice that U opens the path $X \leftarrow U \rightarrow Y$. In order to estimate the causal effect of X in Y we might have to adjust U to close the spurious path. But it is not possible to adjust for something that you not control.

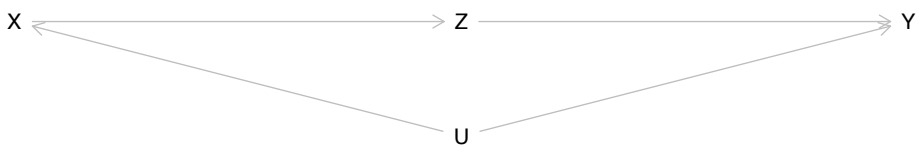


Figure 6: Sample causal dag with front-door path

Hopefully, Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) has proven that you can estimate the causal effect of the exposure on the outcome though the descendants of the exposure. It is true if the front-door criterion is satisfied. A set of variables Z satisfied the front-door criterion relative to X and Y if:

- Z intercepts all directed path from X to Y
- There is no unblocked path from X to Z
- All the backdoor paths from Z to Y are blocked by X

According to Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) theorem, if the front-door criterion is applied, then the causal effect of X to Y is expressed by the formula:

$$P(Y|do(X)) = \sum_{Z} P(Z|X) \sum_{X'} P(Y|X', Z)P(X') \tag{2.3}$$

3 An Empirical Study of the Relationship between Continuous Integration and Test Code Evolution

Earlier versions of the work in this chapter appears in the proceedings of the International Conference on Software Maintenance and Evolution (ICSME) (SIZíLIO; COSTA; KULESZA, 2019)

3.1 Introduction

Automated tests play an important role in CI (FOWLER; FOEMMEL, 2006) as they represent a fundamental part of the build automation process to detect errors as quickly as possible. In fact, Duvall (DUVALL; MATYAS; GLOVER, 2007) mentioned that without automated tests, a project should not be considered to be adopting CI at all (DUVALL; MATYAS; GLOVER, 2007). Zhao et al. (ZHAO et al., 2017) studied the transition of open-source projects to adopt Travis-CI[1]. The authors investigated how development practices (e.g., frequency of commits) have changed after the adoption of Travis-CI. The authors observed that the number of

[1]https://travis-ci.org/

automated tests tended to increase after the initial adjustments of adopting Travis-CI. Labuschagne et al. (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) studied the bug resolution process and its relationship with automated tests. Some of their findings reveal that 26% of non-flaky tests were due to incorrect or obsolete tests.

Despite the valuable advancements of previous research, the question as to whether CI **enhances the evolution** of software tests remains unexplored. This investigation is important for decision makers (e.g., team leaders) to better understand whether CI can improve the test evolution in the long run. Our work empirically studies the evolution of test code and its relationship with the adoption of CI. We set out to compare 82 projects that have adopted CI (CI projects) and 82 projects that have not adopted CI (NOCI projects). In total, we study 3,936 versions of our CI and NOCI projects to investigate trends on the test code and coverage. In particular, we address the following research questions:

- **RQ1 - What are the evolution trends of test ratio within CI and NOCI projects?** We find that 40.2% of the CI projects have a rising test ratio (i.e., the proportion of test code in a project) trend while only 17% of the NOCI have a rising test ratio trend.
- **RQ2 - Is there an association between the adoption of CI and the evolution of code test ratio?** We observe that the adoption of CI is associated with a consistent growth of test ratio, whereas NOCI projects have a negligible change of test ratio overtime.
- **RQ3 - Is there an association between the adoption of CI and the evolution of test coverage?** Our analysis shows statistical evidences that most of the analyzed CI projects tend to increase or maintain the test coverage (9 out of 10 projects), while NOCI projects show a different tendency (5 out of 10 projects increase or maintain the test coverage). In fact, NOCI projects have more projects decreasing the coverage (5 projects) when compared to CI projects (1 project)

- **RQ4 - What are the most important factors to model test ratio?** Our mixed-effect models reveal that test ratio is largely explained by the project inherent context (nearly 84% to 88% of the explanatory power) rather than by the code or process factors.

In overall, our results suggest that the adoption of CI can be empirically associated with a healthier test code evolution. However, the benefits of the CI adoption in terms of test code is likely more associated with the project inherent context (e.g., project culture), rather than process or code factors (e.g., number of contributors or number of commits that include test changes).

Chapter organization: The rest of this chapter is organized as follows. In Section 3.2 we explain the design of our empirical study. In Section 3.3, we present the results of our empirical study, while Section 3.4 presents the threats to the validity. Finally, we draw conclusions in Section 3.5.

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. We made all datasets discussed in this section available in our online appendix. [2]

3.2.1 Studied Projects

To perform our experiments, we analyze two groups of projects: (i) open source projects that have adopted CI at some point of their lifetime (CI projects) and (ii) open source projects that have never adopted CI during their lifetime (NOCI projects).

CI Projects. This dataset must consist of consolidated projects that eventually adopted CI at some point of their history. These projects also must have a

[2]https://ciqualityresearch.github.io/doctoral_thesis/

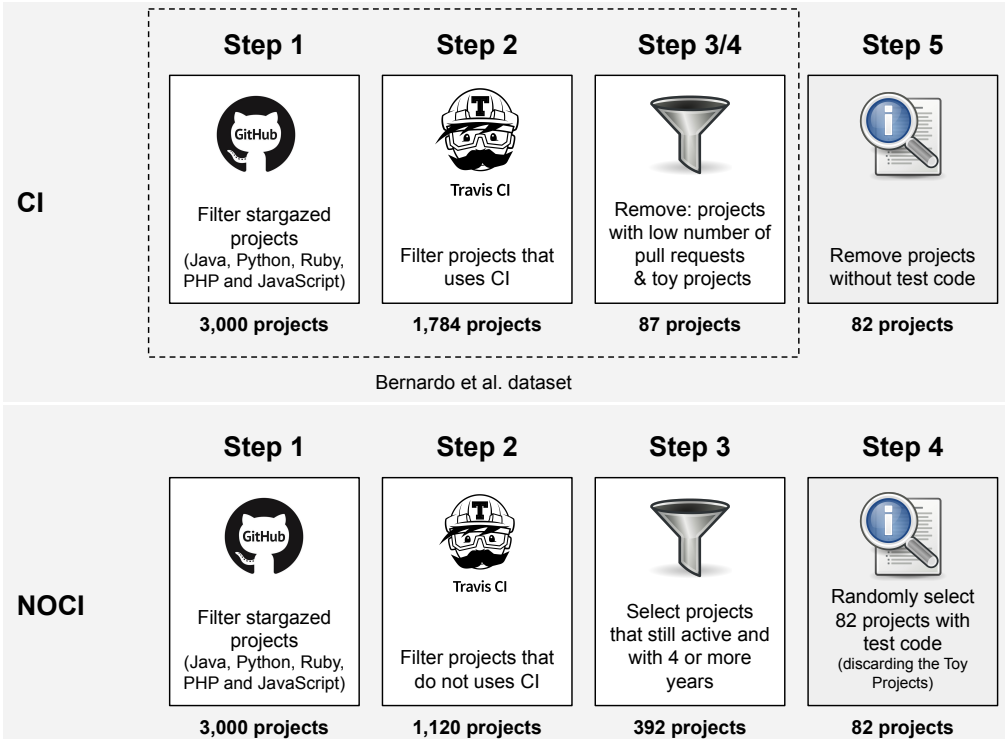


Figure 7: Overview of CI and NOCI projects selection process.

considerable data history (e.g., 2 years of data) before and after they have adopted CI. This is important because, in some of our analyses, we compare the evolution of tests before and after adopting CI. We leverage the dataset made available by Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a) to obtain our studied projects. Fig. 7 shows the process used by Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a) to collect the dataset by using the GitHub API[3]. Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a) selected the 3,000 most stargazed projects that are written in the five most popular programming languages on GitHub (Java, Python, Ruby, PHP and JavaScript). Afterwards, the authors filtered for quality criteria, such as the removal of toy projects and projects with low number of pull requests. To filter for projects that adopt CI, Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a) used the Travis-CI API[4] and found builds

[3]https://developer.github.com/v3/
[4]https://docs.travis-ci.com/api/

using the Travis-CI service. The authors considered the first build on Travis-CI as the date at which a project started using CI. Projects that have not had any build on Travis-CI were removed.

Fig. 7 also shows an additional filter that we apply onto the dataset provided by Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a) to select our projects. As our goal is to investigate the relationship between the adoption of CI and the test code evolution, we excluded 5 projects in which we did not find any test code (Step 5 in Fig. 7). Our online appendix shows the information regarding the 82 CI projects that remained in our modified dataset[5].

NOCI Projects. This dataset must consist of projects that have never adopted CI. This dataset is especially important for checking the presence of bias in our observations. For example, if we observe an increase in the test coverage after the adoption of CI, we also check whether the same increase of test coverage also occurs in the NOCI projects. If we observe the increase of test coverage in both datasets, then it is unlikely that the observed increase is associated with the adoption of CI.

To construct the NOCI projects dataset, we follow a similar approach as performed by Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a). Fig. 7 shows the process for building the dataset. We use the GitHub API to identify the 3,000 most stargazed projects that are written in the five most popular programming languages on GitHub (Java, Python, Ruby, PHP and JavaScript). However, in the NOCI projects dataset we select projects that have never adopted CI. We use the Travis-CI API to discard projects that have at least one build on Travis-CI. To avoid studying inactive and immature projects, in Step 3 of Fig. 7, we filter for projects that are still active in 2018 and have at least 4 years of activity. In step 4, we randomly select 82 projects from the 392 NOCI projects obtained from Step 3. This random selection is performed in several iterations. For example, for each randomly selected NOCI project we check (i) whether the project is a toy project (i.e., academic, books samples or data storage) and (ii) whether the project has

[5]https://ciqualityresearch.github.io/doctoral_thesis/#study1_datasets

tests in its development process (i.e., we search for test files and packages). If a given NOCI project does not fulfill both criteria, we discard the project and randomly select another one. We also manually check the projects to check whether they have configuration files of other popular CI services (e.g. Jenkins[6] and CircleCI[7]). If we identify that the project uses another CI service provider, we also exclude such a project from our analyses. To strengthen our certainty that our NOCI projects have never adopted CI, we contact the members of each project by using the development channels and developers e-mails. We repeat the process until we obtain 82 NOCI projects (i.e., to mach the number of CI projects). We received a NOCI confirmation reply for about 48% of the NOCI projects. For the remaining 52% NOCI projects, we confirmed the NOCI attribute by manually inspecting their code to find CI services configuration files. Finally, the reason to aim for 82 NOCI projects is to perform fair comparisons with the other 82 CI projects in our analyses (see our online appendix[5]).

3.2.2 Data Collection

Once we collect our projects, we also collect the versions of each project to perform our analyses. Since we aim to analyze the impact of adopting CI on the test ratio and coverage, we need to collect the history data from before and after the adoption of CI for our CI projects. In the dataset made available by Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018a), the moment at which a CI project has adopted CI is the moment when the first build on Travis-CI was generated. Therefore, the first build on Travis-CI is also the event that we consider as the moment at which a CI project has adopted CI. However, inspired by the work of Zhao et. al. (ZHAO et al., 2017), we disregard a period of 30 days around the event of adopting CI before collecting further data. The authors describe this period as an unstable period that might represent noisy data. Fig.8 shows how we collect the versions of our CI projects. Each red arrow indicates a checkout of the nearest

[6]https://jenkins.io/
[7]https://circleci.com/

snapshot (i.e., a commit in a project). We collect the 12 months historical data before the adoption of CI and 12 months after the adoption of CI for each of our CI projects.

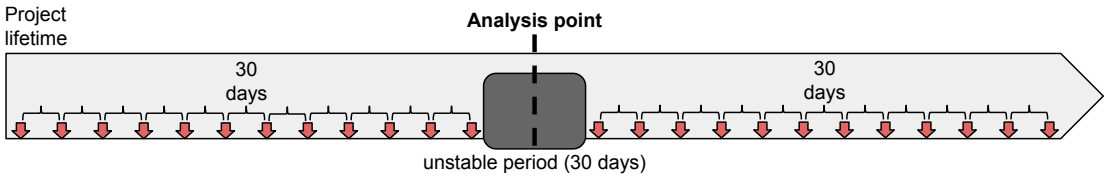


Figure 8: Overview of versions selection process.

With regards to our NOCI projects, we also need to establish a moment in time to divide their history in two sets of data. As stated before, having these two sets of data is important to double-check the observations that we obtain when we compare the data before and after in CI projects. If we observe the same evolution trend (e.g., an increase in test coverage) by comparing different time periods in both CI and NOCI projects, then it is unreasonable to consider the adoption of CI as the precursor of the observed trend. Considering that dividing the data history of NOCI projects is less obvious (i.e., there is no event such as the adoption of CI) we select a point of time that is proportional to a project's lifetime. First, we randomly select 10 projects from the CI projects dataset. Next, we compute the number of months prior to the adoption of CI. Then, we identify the proportion of time (in terms of percentage) that the 10 projects took to adopt CI with respect to the lifetime of the projects. For example, we observe that a particular project adopted CI at the 20% of its lifetime. Finally, we compute the median of the proportions. We find that, in the median, the sampled projects adopted CI at the 27,36% of their lifetime.

Subsequently, we clone the repositories of all CI and NOCI projects. We perform a checkout of 24 versions for all projects (i.e, both CI and NOCI projects) by following the process depicted in Fig. 8. For CI projects, the dividing event is the adoption of CI, whereas for NOCI projects, the dividing event is the 27% of their lifetime. For each version of a project, we compute the following metrics to

perform our study.

Test Ratio. The test code ratio is the proportion of test code in the project. To collect the test code ratio, we first define how to measure the source code. First (Step 1 in Fig. 9), we consider only the files related to the programming languages, i.e., we do not consider configuration files or HTML files. Afterwards, we select all code files (Step 2 in Fig. 9) for each project using the *find* unix command with a Regular Expression. Our command searches for Ruby (*.rb), PHP (*.php), Python (*.py), JavaScript (*.js) and Java (*.java). Given that we find a number of projects using different languages (e.g., Java & Scala), we also include TypeScript (*.ts), CoffeScrip (*.coffee), Scala (*.sc and *.scala) and Groovy (*.groovy, *.gvy, *.gy, *.gsh) in our search. The added languages represent a set of languages that may have a strong relationship with the main languages of our dataset (e.g., Java and Scala is likely a combination). We search for code files containing all the aforementioned extensions.

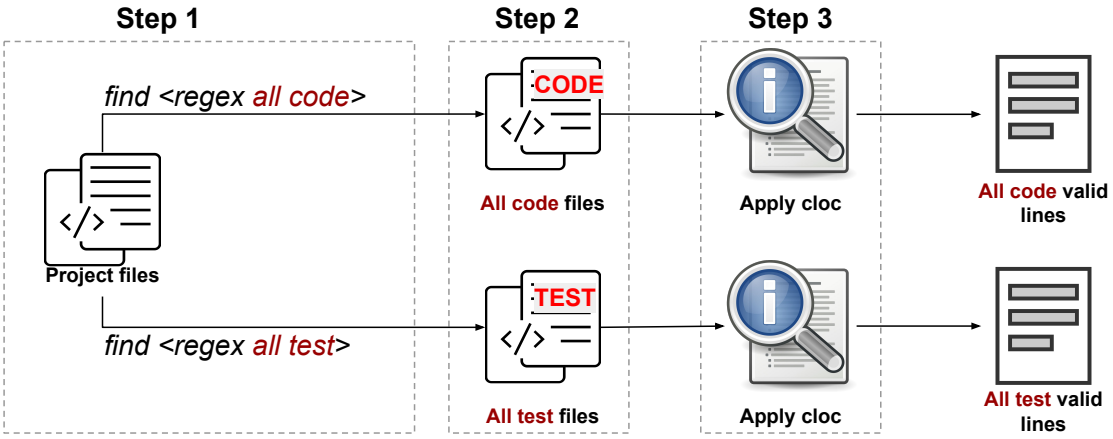


Figure 9: Overview of code lines counting.

In addition to identifying the source code files, we identify the test code. To this end, we apply another *find* command that, in addition to filtering for source files of our specific languages, also filters for the following conditions: (i) files inside test or spec folders; (ii) files that end with .test, -test, _test, Test, .spec, -spec, _spec, Spec and (ii) files starting with test., test-, test_, Test, spec., spec-, spec_,

Spec. All these rules were defined by manually studying the code conventions of our studied languages.

By applying the aforementioned filters, we obtain two sets of files: (i) code files and (ii) test files. We the apply the `cloc`[8] tool (Step 3 on Fig. 9) onto the two sets of files. The `cloc` tool counts blank lines, comment lines and physical lines of source code in many programming languages. Our analyses consider only the valid lines (i.e, excluding the comment and blank lines) to count the overall code and test lines of a project. Using these two metrics (i.e., all lines and test lines), we compute the **test ratio**, which is defined by the $\frac{test\_lines}{all\_lines}$ fraction.

In addition to the test ratio, we compute the **test ratio growth**. The test ratio growth measures the proportional growth of tests from a version to a subsequent version. The **test ratio growth** is defined as $test\_ratio\_growth = test\_ratio_i - test\_ratio_{i-1}$, where i is a given version and $i-1$ is the nearest previous version.

Collecting commit factors. In this study, we also investigate important software development factors that can be associated with the increase/decrease of the test ratio metric. CI is a practice that requires the team to adapt their workflow to embrace the good practices of *Source Code Management* (SCM) (FOWLER; FOEMMEL, 2006). Examples of guidelines that are usually introduced with the adoption of CI include: "Make small commits" and "Commit self-testing code." With respect to this matter, we explore factors in the SCM system (e.g., Git) that can influence the test ratio when adopting CI. For example, after adopting CI, the commits in a given project may be smaller and may contain more test code.

To collect the commit factors, we consider all the periods of 30 days presented in Fig.8. Each period contains the commits from one version to the next. In summary, a version comprises the set of commits that was recorded in the previous 30 days. We collect the log of each commit from each version and extract our studied factors from the commit logs. Table 1 shows the collected factors and the rationale behind each factor.

[8]https://github.com/AlDanial/cloc

Table 1: Commit Factors

Variable	**Rationale**
Total Committers	The count of every user that performed at least one commit in the version. More committers can contribute to more tests being developed, which can increase or maintain the test ratio.
Total Commits	The count of every commit performed in the version. A high number of commits in the version may decrease the test ratio if tests are not included.
Total Changes, Total Code & Test Changes	The sum of lines changed in all commits from the version. The higher the test lines that were introduced in one version, the higher the probability of increasing the test ratio.
Median Files & Median Code Files	The median number of files touched by the commits in the version. Small commits represent a good practice and can contribute to a more consistent test development and consequently increase or maintain the test ratio.
Median Changes & Median Code Changes	The median number of lines changed by the commits in the version. These factors also represent the size of commits. Small commits represents a good practice and can contribute to a more consistent test development and consequently increase or maintain the test ratio.
Test Changes per Commit	Defined by $\frac{total\_test\_lines\_changed}{total\_commits}$. This factor represents the test workload. A greater *Test Changes per Commit* can increase the test ratio.
Count Commits With Tests	Total number of commits that changed at least one test file in the version. More commits modifying test files can increase the test ratio.
Test Commit Ratio	Defined by $\frac{count\_commits\_with\_tests}{total\_commits}$. Represent the percentage of the commits that have worked on test files. A greater *Test Commit Ratio* can increase the test ratio.

Collecting Test Coverage. Although test ratio is a global measure of the amount of test code that a project has, it does not necessarily capture the amount of code that is exercised by the tests. Code coverage metrics are often used to identify if an application is well-tested (MALAIYA et al., 2002). In order to compute the test coverage for our projects, we face the challenge of configuring and building each project to run the tests, which is a non trivial process. As we study 2 years of historical data for each project, we face several changes in the project's design. These design changes require different libraries and settings to be configured. A tremendous effort would be required to configure and run all the 2 years of historical data of all of our studied projects in order to compute the test coverage of each project's snapshot. Notwithstanding the use of automated build tools (such as maven, gradle or ant), we still face the challenge of unavailability of libraries and other required assets to build the legacy code of our studied projects. Therefore, we further sample our CI and NOCI datasets. We randomly select 10 CI projects and obtain 4 Python, 3 Java and 3 JavaScript projects. We then randomly select 10 NOCI projects and obtain 4 Python, 3 Java and 3 JavaScript projects. With a set of 20 projects (10 CI and 10 NOCI), we have a total of 480 versions to configure, build and run the tests.

Collecting the coverage is a process that requires instrumenting the code. For this purpose, we use popular tools such as `Coverage.py`[9] to instrument Python projects, `Open Clover`[10] for Java projects and `Istanbul`[11] for JavaScript projects. After running the tests with these tools, we collect the **test coverage** for each version of each project. `Open Clover` computes the coverage percentage using the following formula: $PC = (BT+BF+SC+MC)/(2*B+S+M)*100\%$. BT stands for the branches that evaluated to "true" at least once, BF stands for the branches that evaluated to "false" at least once, SC stands for the covered statements, MC stands for the entered methods, B represents the total number of branches, S the

[9]https://coverage.readthedocs.io/en/v4.5.x/
[10]https://openclover.org/
[11]https://istanbul.js.org/

stands for the total number of statements and M is the total number of methods[12]. `Istanbul` and `Coverage.py` compute the coverage percentage using the following formula: $PC = EL/VL$ where EL stands for *executed lines* and VL stands for *valid lines* (i.e., ignoring comments and blank lines).

In addition to the test coverage, we compute the **test coverage growth**, which measures the proportional growth (or decrease) of test coverage from a version to the subsequent version.

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 test ratio within CI and NOCI projects?

3.3.1.1 Motivation

Recent research has studied the characteristics of test code in projects that use CI. For example, Beller et al. (BELLER; GOUSIOS; ZAIDMAN, 2016) investigated the central role that testing has in CI. However, previous research has not explored whether the adoption of CI can improve the test ratio in software projects. In this RQ, we first study the trends of test code ratio in NOCI and CI projects. This preliminary investigation is important to first identify whether there is an apparent difference in the trends of test ratio evolution when comparing NOCI and CI projects. If significant differences in the test ratio trends are apparent, we have further motivation to deepen our analyses regarding the impact of adopting CI on the test ratio and coverage metrics.

[12]http://openclover.org/documentation

3.3.1.2 Approach

We study two datasets of 82 CI projects and 82 NOCI projects. We analyze the evolution trends of test ratio in each dataset (as illustrated in Fig. 8).

The evolution of the test ratio overtime can be interpreted as a time series data. Therefore, to identify the dominant trends of test ratio evolution, we use the Dynamic Time Warping (DTW) algorithm. DTW minimizes the differences between two time series by aligning their offsets (BERNDT; CLIFFORD, 1994; SALVADOR; CHAN, 2007). This characteristic of DTW allows us to better group the test ratio trends into clusters. For example, if we use the euclidean distance to cluster our time series, we may group similar trends into different clusters due to the offsets in the time series. Although DTW supports the cluster discovering, we must first set the optimal number of clusters (MILLIGAN; COOPER, 1985) to obtain high quality clusters, i.e., clusters that better represent the trends of test ratio. To this end, we use the gap statistics approach (TIBSHIRANI; WALTHER; HASTIE, 2001) by varying the number of clusters from 2 to 20. According to the gap statistic, the optimal number of clusters for the NOCI projects is 4, whereas the optimal number of clusters for CI projects is 18. After running the DTW algorithm we analyze the obtained clusters to identify the increasing, decreasing or maintaining trends of test ratio in CI and NOCI projects. It is important to mention that the level of test ratio is not in analyses, we focus on the trend, i.e., how it behaves along the time. The difference of scale on the test ratio is the focus of the next research question and will be discussed further in this chapter.

3.3.1.3 Results

Observation–1. We observe more projects with rising test ratio trends in the CI projects (40%) than NOCI projects (17%). Fig. 10 shows the test ratio clusters of NOCI projects. Each graph on the figure contains the distribution of test ratios (Y axis) and the versions of the NOCI projects (X axis). We denom-

inate as *raising trends*, the groups of which the centroids represent a continuous growth in the analyzed period of time. We denominate as *decreasing trends*, the groups of which the centroids represent a continuous decrease in analyzed period of time. Finally, the *maintaining trends* are represented by the groups of which the centroids represent a negligible growth or decrease in the analyzed period of time.

We observe that the centroids (i.e., the gray dashed lines) of clusters 1, 3, and 4 denote a maintaining test ratio trend. However, cluster 2 denotes a slightly rising trend of test ratio. We observe that 68 CI projects have a maintaining test ratio trend (nearly 83%) and 14 projects have a rising test ratio trend (nearly 17%).

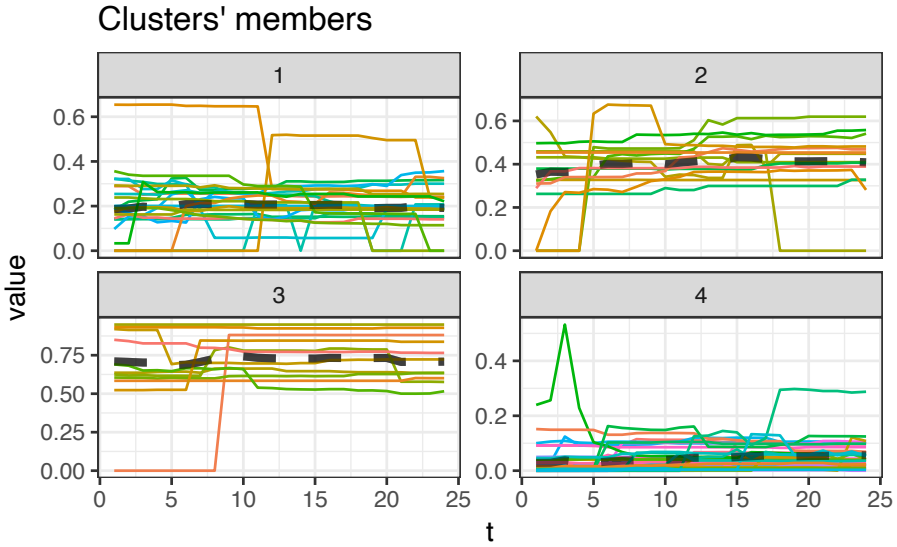


Figure 10: NOCI projects - clusters of test ratio.

Regarding the CI projects, Fig. 11 shows that the centroids of clusters 5, 6, 10, 12, 14, 16 and 17 represent maintaining test ratio trends. However, we also observe that clusters 1, 2, 3, 7, 8, 9, 11, 13, 15 and 18 represent rising test ratio trends. The number of projects that falls within each cluster denotes that 49 projects have maintaining test ratio trends (nearly 59.8%) and 33 projects have rising test ratio trends (nearly 40.2%).

Our results reveal that CI projects have considerably more rising trends (nearly 40.2%) than NOCI projects (nearly 17%), as shown in the Table 2. The result of

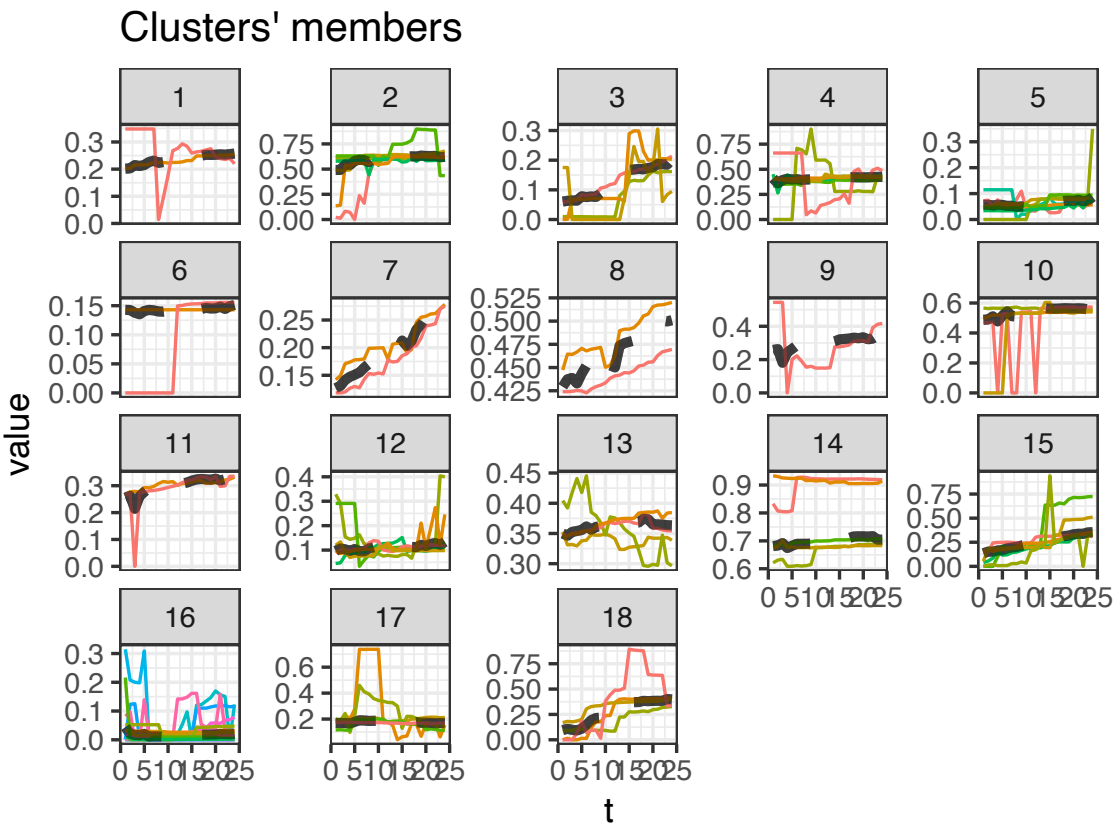


Figure 11: CI projects - clusters of test ratio.

this RQ motivates us to further investigate whether the adoption of CI was a main precursor of the rising test ratio trend within the CI projects.

Table 2: Sum of projects in clusters

	Decreasing	**Maintaining**	**Raising**
CI	0	49	33
NOCI	0	68	14

3.3.2 RQ2 - Is there an association between the adoption of CI and the evolution of code test ratio?

3.3.2.1 Motivation

RQ1 reveals that CI projects have considerably more rising test code ratio trends than NOCI projects. Hence, in this RQ, we empirically investigate whether the adoption of CI has a significant influence on the evolution of test ratio within CI projects. This investigation is important to better understand the impact of adopting CI in software development in terms of the growth of software tests.

3.3.2.2 Approach

To address RQ2, we further split our two datasets (i.e., CI and NOCI projects) into two subdatasets: *before-CI*, *after-CI*, *early-NOCI* and *late-NOCI* (Fig. 8). The *before-CI* dataset contains the data of our CI projects before the adoption of CI, whereas the *after-CI* dataset contains the data after the adoption of CI. Conversely, the *early-NOCI* contain the first 27% of the lifetime of our NOCI projects, whereas the *late-NOCI* dataset contains the remaining data of the NOCI projects (i.e., the remaining 73% of the projects' lifetime, see Section 3.2.2). We collect 12 versions for both before and after the dividing events for both CI and NOCI projects. Afterwards, we perform comparisons between the *before-CI* and *after-CI* with respect to the test ratio to check whether there exist significant differences on the test ratio after adopting CI. We also perform comparisons between the *early-NOCI* and *late-NOCI* to double check if the results obtained for the comparisons between *before-CI* and *after-CI* datasets can simply occur due to the projects' natural growth (i.e., and not mainly because of the CI adoption). In addition to comparing the differences in the test ratios, we study the test ratio growth metric to verify whether CI influences the overall increase of test ratios across the projects' versions.

To check whether the distributions of test ratio are statistically different when

comparing *before-CI* vs *after-CI* and *early-NOCI* vs *late-NOCI*, we use Mann-Whitney-Wilcoxon (MWW or Wilcoxon rank sum test) tests (WILKS, 2011). The MWW is a non-parametric test which checks the null hypothesis (H0) that two distributions come from the same population ($\alpha \geq 0.05$), i.e., two distributions are not significantly different.

Although the MWW test shows whether two distributions are statistically different, MWW does not indicate the magnitude of such a difference (e.g., large or small). Therefore, we use the Cliff's delta metric to check the magnitude of the statistical differences between distributions (MACBETH; RAZUMIEJCZYK; LEDESMA, 2011). A positive Cliff's delta indicates that the first distribution is higher than the second distribution (and a negative Cliff's delta indicates otherwise). The higher the Cliff's delta the larger the difference between the values of two distributions. We use the default thresholds from the `effsize R` package[13]. The default thresholds are the same as provided by Romano et al. (ROMANO et al., 2006). The thresholds are set as the following: delta < 0.147(negligible), delta < 0.33 (small), delta < 0.474 (medium), and delta $>= 0.474$ (large). First, we compare the distributions of test ratio between the *before-CI* and *after-CI* datasets. Afterwards, we compare the distributions of test ratio between the *early-NOCI* and *late-NOCI* datasets to double-check our observations. We repeat the same comparison process for analyzing the test growth metric.

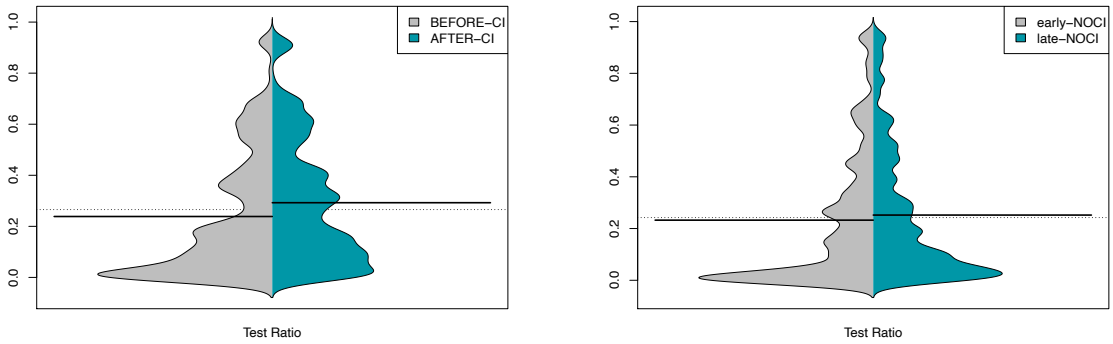
3.3.2.3 Results

Observation–2. CI projects tend to increase the test ratio in a constant growth, while NOCI projects have a negligible change in the test ratio overtime. The beanplot in Fig 12a shows the distribution of test ratios for the *before-CI* and *after-CI* datasets. We observe an increase in the median test ratio after the CI adoption. We obtain a Wilcoxon rank sum $p-value = 2.908e-10$, which indicates a statistical difference between the test ratios in the *before-CI*

[13]https://cran.r-project.org/web/packages/effsize/

and *after-CI* datasets. We also obtain a small (non-negligible) Cliff's delta of -0.1612838.

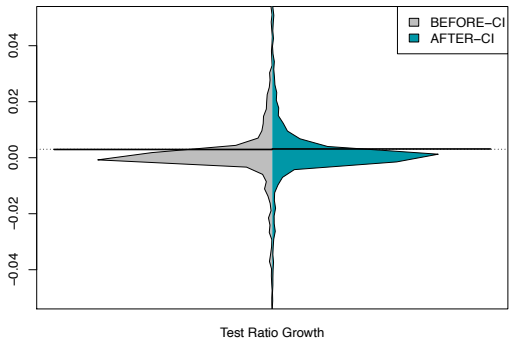
The beanplot of Fig 12b shows the distribution of test ratios of the *early-NOCI* and *late-NOCI* datasets. We observe an increase in the median test ratio in the *late-NOCI* dataset ($p - value = 0.0001842$). Nevertheless, we obtain negligible Cliff's delta of -0.09270482.



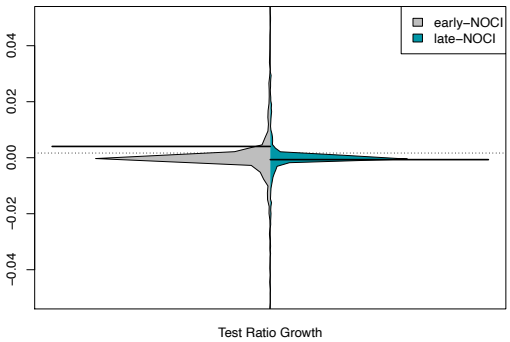
(a) CI projects (b) NOCI projects

Figure 12: Beanplot comparisons of test ratio within CI and NOCI datasets

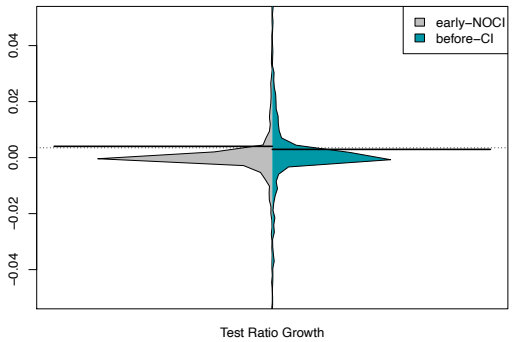
Our results suggests that the adoption of CI is likely to have an impact on the test ratio, since we observe a negligible difference in test ratios when analyzing the NOCI projects (and non-negligible difference when analyzing the CI projects). To further check whether the increase in test ratio is unrelated to the natural grow of the projects, we analyse the test ratio growth metric, which is the degree of change in the test ratio between one version and another. The beanplot in Fig 13b shows the distribution of test ratio growth for the *early-NOCI* and *late-NOCI* projects. Although we observe a decrease in the test ratio growth in the *late-NOCI* dataset, we obtain a negligible Cliff's delta (MWW $p - value = 0.3861$ and Cliff's $delta = 0.02301242$). The beanplot on Fig 13a shows the distribution of test ratio growth for the *before-CI* and *after-CI* datasets. We observe that, after adopting CI, the median of test ratio growth is also maintained (MWW $p - value = 0.1164$ and a negligible Cliff's $delta = -0.04144399$).



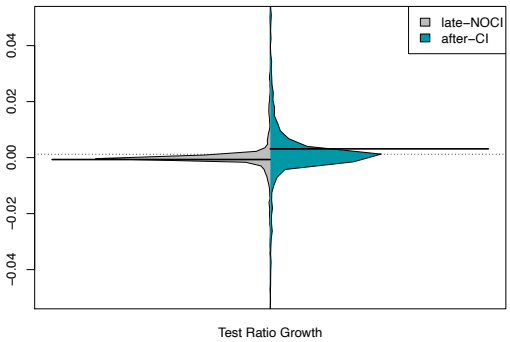
(a) CI projects



(b) NOCI projects



(c) early-NOCI vs before-CI



(d) late-NOCI vs after-CI

Figure 13: Beanplot comparisons of test ratio growth of CI and NOCI datasets

Furthermore, we compare the *before-CI* dataset against the *early-NOCI* dataset and the *after-CI* dataset against the *late-NOCI* dataset. This analysis helps us to further understand how the projects that adopted CI have evolved.

The beanplot in Fig 13c shows the distribution of test ratio growth for the *before-CI* and *early-NOCI* datasets. We obtain a non significant difference between distributions (Wilcoxon rank sum test $p - value = 8.892e - 07$ but a negligible Cliff's delta of -0.1321945). Moreover, the beanplot in Fig 13d shows the distributions of test ratio growth for the *after-CI* and *late-NOCI* datasets. We observe a significant difference in the test growth distributions (Wilcoxon rank sum test $p - value = 2.566e - 15$ and non-negligible small Cliff's delta -0.2138018).

Our results suggest that the adoption of CI may have a small and positive

effect on the test ratio. However, by analyzing the *test ratio growth*, our results reveal that, after adopting CI, the project maintains the same ratio growth as before the adoption of CI. Our results suggest that the CI period has a superior test ratio, but it is likely a consequence of a growth that has already existed before adopting CI.

Finally, the non-significant differences observed between the *before-CI* and *early-NOCI* datasets, as well as the significant differences observed between the *after-CI* and *late-NOCI* datasets, may suggest that the adoption of CI is associated with a positive maintenance of the test ratio growth. Conversely, not adopting CI may incur a decrease in the test ratio growth. Therefore, projects that adopt CI may value testing more than projects that do not adopt CI.

3.3.3 RQ3 - Is there an association between the adoption of CI and the evolution of test coverage?

3.3.3.1 Motivation

In RQ1 and RQ2, we study the test ratio in CI and NOCI projects. However, as much as the test ratio provides an idea of the proportion of test code in a project, the test ratio does not show how much code is actually exercised by the tests. For example, two distinct projects may have 50% of test ratio. However, one project can concentrate the majority of the tests to exercise few functionalities (i.e., low test coverage), while the other project may exercise almost all the functionalities in its tests (i.e., high test coverage). We investigate the test coverage of CI and NOCI projects to better understand the impact of the adoption of CI on the test coverage of the studied projects.

3.3.3.2 Approach

To address this RQ, we randomly select 10 CI and 10 NOCI projects as explained in Section 3.2. We compare the CI and NOCI projects to study whether

CI has a significant influence on the test coverage. We also analyze the impact of CI on the test coverage growth. Similar to the previous RQs, we use the MWW tests and the Cliff's delta (MACBETH; RAZUMIEJCZYK; LEDESMA, 2011) in our analyses.

3.3.3.3 Results

Observation–3. Most of the analyzed CI projects tend to increase or maintain the test coverage (9 out of 10 projects) while for NOCI projects only 5 out of 10 increase the test coverage. Fig. 14 and Fig. 15 shows the evolution of test coverage for each studied project. The graph shows the percentage of test coverage in the Y-axis and the project version in the X-axis. The vertical dotted line in the middle represents the point of transition from *early-NOCI* and *late-NOCI* for NOCI projects and *before-CI* and *after-CI* for NOCI projects. The red lines represent the test coverage evolution, and the blue line represents the linear regression in each period.

By analyzing the NOCI projects, we observe that, in its majority, NOCI projects experience a decrease or maintenance in the test coverage in the *late-NOCI* dataset. The Table 3 shows the statistical tests results of the coverage comparison of the projects periods (before-CI/after-CI and early-NOCI and late-NOCI). The results of our Wilcoxon rank sum tests show significant differences of test coverage between the *early-NOCI* and *late-NOCI*. Our Cliff's delta measurements reveal (i) a large negative effect size in 5 out of 10 projects (red rows), (ii) a positive effect size for 4 projects (green rows) and (iii) only 1 negligible effect size (yellow rows).

Regarding the CI projects, we observe that the majority of the projects experience a significant increase in the test coverage. Our Wilcoxon rank sum tests (with alternative="less") reveal significant differences between the *before-CI* and *after-CI* periods and our Cliff's delta measurements are significantly positive (large or small) for 6 out of the 10 projects (green rows). Our results suggest that the projects that eventually adopt CI may likely experience an increase in the test

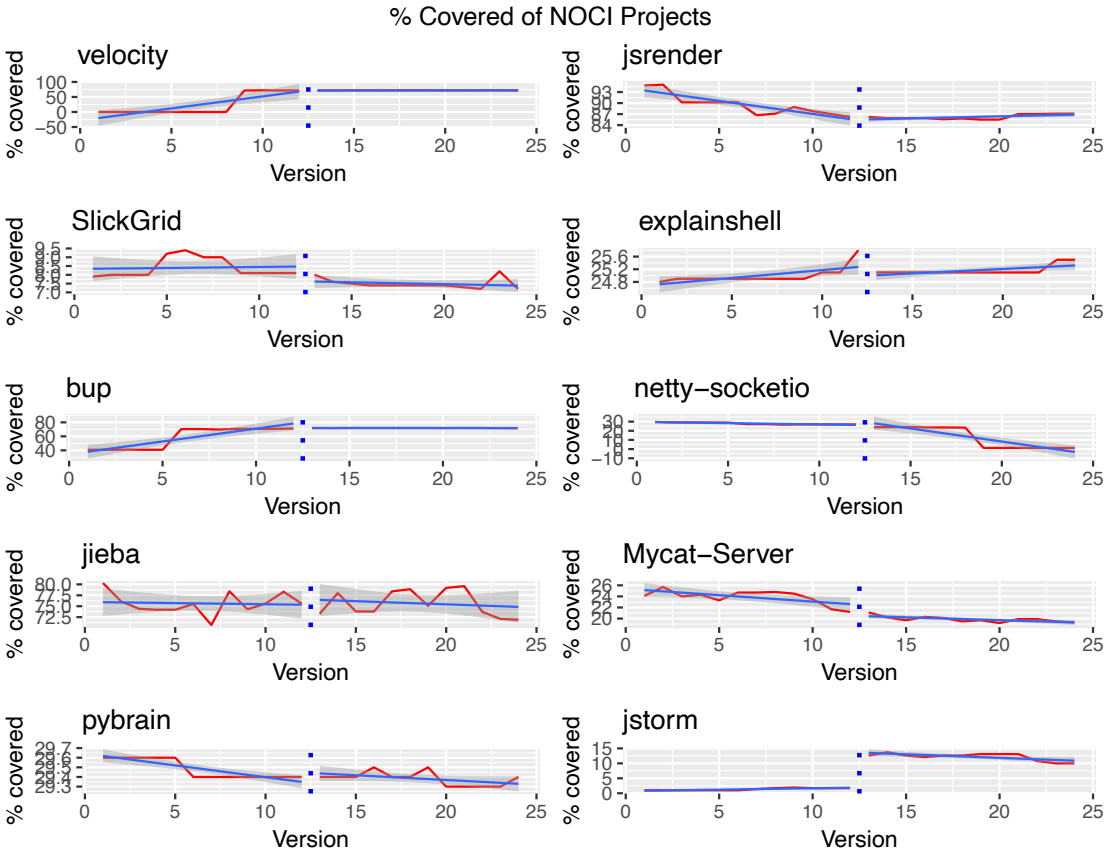


Figure 14: Percent coverage evolution of NOCI projects

coverage.

Similarly to RQ2, we also analyse a measure of growth for the test coverage. The measure of growth is important to check whether the increase or decrease of the test coverage occurs due to a natural aging process of a project. Our results show that both NOCI projects (Fig. 16b) and CI projects (Fig. 16a) have negligible results in terms of test coverage growth. NOCI projects obtain a MWW $p-value = 0.3032$ and a Cliff's $delta = 0.0385124(negligible)$, while CI projects obtain a MWW $p - value = 0.383$ and Cliff's $delta = -0.02322314(negligible)$. However, by comparing the *late-NOCI* and *after-CI* periods (Fig. 16c), our data reveals a small, but positive, effect size for the *after-CI* period (MWW $p-value = 0.003509$ and Cliff's $delta = -0.2068595, small$). Our results suggest that CI projects tend

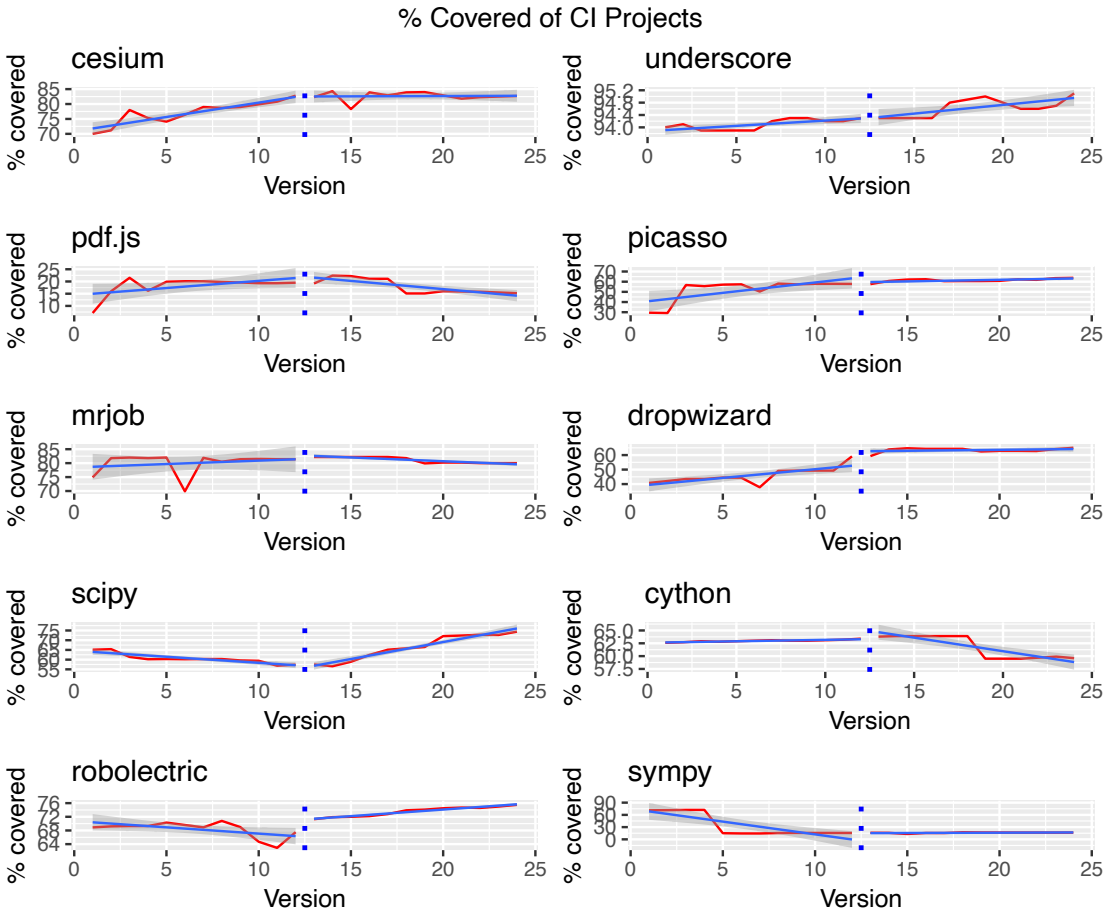


Figure 15: Percent coverage evolution of CI projects

to maintain the test coverage growth while NOCI projects may likely experience a decrease.

3.3.4 RQ4 - What are the most important factors to model test ratio?

3.3.4.1 Motivation

In addition to analyzing the impact of CI on test ratio, we aim to study which software development factors are important to explain the test ratio on CI projects. This is important because, as we have observed, not all CI projects improve the

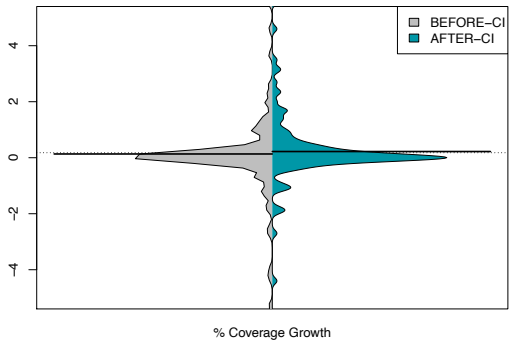
Table 3: % Test Coverage statistical tests

NOCI projects	**MWW $p-value$**	**Cliff's delta**
velocity	0.0001185	$-0.8680556(large)$
jsrender	0.9998	$0.8263889(large)$
SlickGrid	0.9998	$0.8541667(large)$
explainshell	0.0009023	$-0.6944444(large)$
bup	$1.369e-05$	$-1(large)$
netty-socketio	1	$1(large)$
jieba	0.6985	$0.1180556(negligible)$
Mycat-Server	1	$1(large)$
pybrain	0.9913	$0.5138889(large)$
jstorm	$1.439e-05$	$-1(large)$
CI projects	**MWW $p-value$**	**Cliff's delta**
cesium	0.0002361	$-0.8472222(large)$
underscore	$6.095e-05$	$-0.9166667(large)$
pdf.js	0.8225	$0.2152778(small)$
picasso	$4.816e-05$	$-0.9444444(large)$
mrjob	0.3313	$-0.1111111(negligible)$
dropwizard	$1.759e-05$	$-1(large)$
scipy	0.01871	$-0.5069444(large)$
llow cython	0.5116	$0(negligible)$
robolectric	$1.815e-05$	$-1(large)$
sympy	0.3322	$-0.1111111(negligible)$

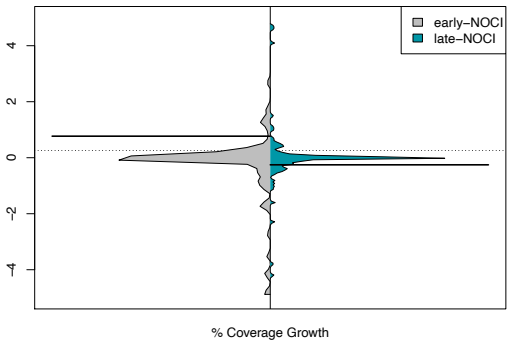
test ratio after adopting CI. Therefore, understanding factors that explain the increase of test ratio can help developers to leverage the full potential of adopting CI.

3.3.4.2 Approach

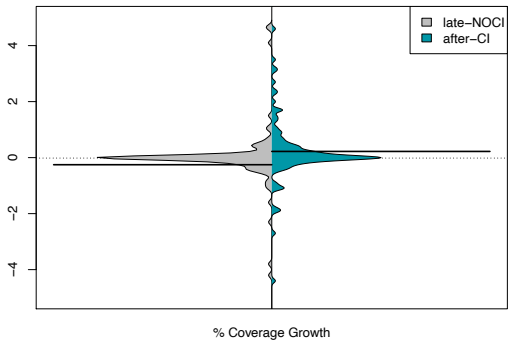
We use the CI projects dataset to observe the periods *before* and *after* the CI adoption. The dataset contains 82 projects with 24 versions each (12 versions before CI and 12 versions after CI), totalling 1,968 versions. As shown in Section 3.2, we retrieve the git commit history and extract a set of factors (Table 1), which



(a) CI Projects



(b) NOCI Projects



(c) late-NOCI vs after-CI

Figure 16: Comparison of growth of percent coverage of CI and NOCI datasets

capture aspects for the projects' evolution (e.g., size, frequency, and nature of the changes). We then use a linear mixed-effects regression to model the test ratio in our CI projects at the version level, i.e., each observation in our model is a version with its respective test ratio. Linear mixed-effects models are statistical regression models that contain both fixed and random effects (JULIAN, 2016). Fixed effects are variables with constant coefficients and intercepts for every observation (i.e., version), while random effects may vary. Random effects are variables that are used to control the variances between observations across different groups. Our linear mixed-effects models assume that each project represents a group (i.e., a source of random variation), therefore we consider a different intercept for each project. This is reasonable, since a software project may have a huge source of external influences that are specific to the project (e.g., the complexity of the project domain or the

development culture).

Regression models can be adversely affected by the existence of highly correlated and redundant independent variables (DOMINGOS, 2012). We perform correlation analyses for the studied factors (henceforth referred as independent variables) used in our models. We use the `varclus` function from the `rms`[14] `R` package. For each pair of independent variables within all correlation clusters that have a correlation $|\rho| > 0.7$, we remove one variable and keep the simpler variable by following the principle of parsimony (VANDEKERCKHOVE; MATZKE; WAGENMAKERS, 2015). Afterwards, we use the *redun* function from the `rms R` package to perform the redundant variable analysis for the remaining variables. The redundant analysis verifies whether an independent variable can be estimated by other independent variables. We discard a variable that can be estimated with an $R^2 >= 0.9$, since such variables do not add explanatory power to the model and can distort the relationship between other variables (JR, 2015).

After removing correlated and redundant variables, we use the `lmer` function from the `lme4`[15] R package to fit a linear mixed-effects model (LMM) to our data. We divide our dataset in two: *before* and *after* CI adoption. We then perform the linear mixed-effects regression in both datasets to construct the LMM that fits to the two periods of versions (before and after CI). We evaluate the performance of the models using the marginal R^2 and the conditional R^2 as computed by the `squaredGLMM` function from `MuMIn`[16] `R` package. To identify the significant independent variables and their explanatory power, we use the *Wald* χ^2 metric. The higher the χ^2 for a given independent variable, the stronger the shared relationship between the variable and the test ratio. All the p-values, correlated and redundant variables, χ^2 values and coefficients of our models are available in our online appendix.

[14]https://cran.r-project.org/web/packages/rms/index.html
[15]https://cran.r-project.org/web/packages/lme4/index.html
[16]https://cran.r-project.org/web/packages/MuMIn/index.html

3.3.4.3 Results

Observation–4. Test ratio is largely explained by the project inherent context rather than code or process factors. This result shows that it might exist some uncontrolled factor that coevolves among the project version evolution that might better explain the test ration evolution.

We call *Model I* the model that fits the period before the adoption of CI and *Model II* the model that fits the period after the adoption of CI. We obtain a good fit for *Model I* with a marginal $R^2 = 0.3098444$ and a conditional $R^2 = 0.9621811$ and *Model II* with a marginal $R^2 = 0.2596061$ and a conditional $R^2 = 0.985688$. The χ^2 values and coefficients are summarized in the Table 4 and Table 5

Table 4: *Model I* - Results of the LMM

Factor	**Coef**	$\boldsymbol{\chi^2}$	$\boldsymbol{Pr(>\chi^2)}$	**Sign**
Intercept	$2.200e-01$	64.5024	$9.642e-16$	***
Median Files	$-8.492e-03$	6.6942	0.0096728	**
Total Commits	$-1.662e-04$	20.8298	$5.020e-06$	***
Test Changes / Commit	$7.463e-05$	25.6239	$4.149e-07$	***
Count Commit with Tests	$5.604e-04$	11.0556	0.0008842	***
Version:Project		1002.0471	$< 2.2e-16$	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 5: *Model II* - Results of the LMM

Factor	**Coef**	$\boldsymbol{\chi^2}$	$\boldsymbol{Pr(>\chi^2)}$	**Sign**
Intercept	$2.594e-01$	58.0176	$2.598e-14$	***
Total Commits	$-1.348e-04$	11.8034	0.0005912	***
Test Changes / Commit	$-4.410e-05$	5.0549	0.0245569	*
Count Commit with Tests	$2.872e-04$	7.0145	0.0080854	**
Total Committers	$7.241e-04$	4.7638	0.0290636	*
Version:Project		461.4989	$< 2.2e-16$	***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Our model reveals that, instead of being explained by process or code factors (as described in Table 1), test ratio is largely explained by the project inherent context. The *project* independent variable obtains the vast majority of the explanatory power in our models. The proportional χ^2 of the project variable corresponds to nearly 88% of the explanatory power in *Model I* and nearly 84% of the explanatory power in *Model II*. We also observe that *Test Changes per Commit* correspond to only 3% of the explanatory power before the adoption of CI and $0,9\%$ after the adoption of CI.

Our results suggest that the influence of the project inherent context is still the dominant influence on the test ratio when comparing to other metrics that can be computed by the SCM system. This result might signal that a high test ratio might depend more on the cultural aspects of a given project rather than the number of commiters or the number of test code per commit.

3.4 Threats to the Validity

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

Construct Validity. The construct threats to validity are concerned with errors caused by the methods that we use to collect our data. We use the GitHub and Travis-CI APIs to collect our projects samples. We also use existing popular tools to collect the coverage and total lines of test code. Any bias in these tools may influence our results. However we analyze the results in a dataset sample to test and assess these tools. Another threat is that the selection of the NOCI projects was accomplished by analyzing projects that are not supported by Travis-CI and other (most used) CI services. However, we recognize that projects may use other tools for continuous integration. We mitigate this threat by contacting the community to reinforce that a given project does not use any CI service. 39 projects out of the 82 NOCI projects have replied and confirmed that they do not use any CI service. For the remaining 43 projects, we performed a manual analysis

to check any evidence of CI usage, such as configuration files of other CI services or tools (e.g., Circle-CI). Additionally, we assume that a project with Travis-CI builds is using continuous integration. However we did not ensure that they adopt the CI practices or use the Travis-CI only to automate the build. We plan to conduct future researches that evaluate the degree of adoption of CI and software quality. Finally, there is a threat in the way we define the project lifetime. We consider that a project lifetime consists in the time that the project uses GitHub, however, it could exist projects that have migrated from another CVS and we disregard the time before GitHub adoption.

Internal Validity. Internal threats are concerned with the ability to draw conclusions from the relationship between the dependent variable (test code ratio & test coverage) and independent variables (e.g., the variables collected in Table 1). We recognize that our independent variables described in Table 1 are not exhaustive. The use of additional independent variables (e.g., number of pull requests submitted in a version) may produce different results in our model. We plan to collect a more exhaustive list of independent variables in future work.

External Validity. External threats are concerned with the extent to which we can generalize our results (PERRY; PORTER; VOTTA, 2000). In this work we analyzed 3,936 versions of 164 popular open source projects from GitHub. All CI projects adopt the most popular CI server on GitHub, i.e., Travis-CI. We excluded projects that do not use the most used CI services for obtaining the NOCI dataset. However we recognize that our results is restricted to our projects' setup and further analysis (e.g., additional projects) should be performed in future work.

3.5 Conclusion

In this study, we investigate the impact of adopting CI on the evolution of test ratio and coverage. We set out to compare 82 CI and 82 NOCI projects. In total, we studied 3,936 versions of our studied projects. Our results reveal that:

- CI projects have more projects with a rising test ratio trend. We found that 33 out of 82 (40.2%) CI projects have a rising test ratio trend, while only 14 out of 82 (17%) NOCI projects have a rising test ratio trend.

- We observe that the adoption of CI is associated with a consistent increase of test ratio (MWW $p - value = 2.908e - 10$ and a small Cliff's $delta = -0.1612838$), while NOCI have a negligible change on the test ratio overtime (MWW $p - value = 0.0001842$ and a negligible Cliff's $delta = -0.09270482$).

- Our results reveal that CI projects likely obtain a higher test ratio growth than NOCI projects.

- Our analysis shows statistical evidences that most of the analyzed CI projects tend to increase or maintain the test coverage (9 out of 10 projects), while NOCI projects have a different tendency (5 out of 10 projects increase or maintain the test coverage). In fact, NOCI projects have more projects decreasing the coverage (5 projects) when compared to CI projects (1 project).

- Our mixed-effect models reveal that test ratio is largely explained by the project inherent context rather than by code or process factors.

In overall, our work demonstrates that continuous integration can be empirically associated with a healthier test code evolution (in terms of test ratio and coverage).

4 On the Continuous Code Quality Outcomes of Continuous Integration: An Empirical Study

4.1 Introduction

Automated Tests is a core concept in the CI environment. Our previous study (Chapter 3) among other works (ZHAO et al., 2017; VASILESCU et al., 2015a; DUVALL; MATYAS; GLOVER, 2007; LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) explored the relationship between continuous integration and aspects of tests. However, Duvall et al. (DUVALL; MATYAS; GLOVER, 2007) advocate that teams should not only of running automated tests but also of performing static and dynamic code analyses at every build. This practice is called *Continuous Inspection*, which is used as a tool for *Continuous Code Quality* (CCQ), which is a practice that aims to continually check and improve (or maintain) the software quality.

Vassallo et. al. (VASSALLO et al., 2018) investigated whether projects using CI were also performing *continuous inspection*. The authors studied data from TravisCI and SonarCloud. SonarCloud [1] is a cloud service based on SonarQube [2] and is used to continuously inspect the code quality by detecting (for ex-

[1] `https://sonarcloud.io/`
[2] `https://www.sonarqube.org/`

ample) bugs, vulnerabilities, and code smells. The authors find that projects using TravisCI may not be using CCQ in practice. Freitas (FREITAS, 2019) complemented the work by investigating if the CI adoption is related to a change on the quality outcomes (from the continuous inspections). The author studied a dataset that includes both projects that have adopted CI and projects that never adopted CI to employ empirical tests. The work reveals that adopting CI has little to no association to improved quality outcomes.

Motivated by the research results that show that CI may not be related to a improve on quality outcomes (FREITAS, 2019) and that projects that adopt CI may not be using CCQ in practice. (VASSALLO et al., 2018), we investigate whether adherence to CI best practices is related to the degree of code quality. Hence, we empirically study 184 open source projects that use TravisCI and SonarCloud to evaluate the relationship between the CI best practices and software quality outcomes. In particular, we address the following research questions:

- **RQ1: Does the degree of CI adoption share a relationship with improved quality outcomes?** Existing research has shown that not all projects using TravisCI fully adhere to the CI best practices (FELIDRÉ et al., 2019). Therefore, in this RQ, we investigate whether the degree of CI adherence has an association with the studied quality outcomes. For this purpose, we analyze the adoption of 4 well-known CI practices (i.e., avoiding long-running builds; fixing a broken build as soon as possible; controlling for acceptable test coverage and integrating at least daily). Our main hypothesis is that the better the degree of CI adherence the better might be the quality outcomes. However, our analyses show that only the practices of *controlling for acceptable test coverage* and *avoiding long-running builds* share a relationship with the improvement in the studied quality outcomes.
- **RQ2: Which practices of CI share a relationship with technical debt?** To better understand which specific practices of CI may share a strong relationship with improved quality outcomes, we fit a regression model in

which the *amount of technical debt* is the response variable. We use the amount of technical debt in particular because this metric factors in all of the other studied metrics (as it is explained in Section 4.2). The project size (i.e., median_ncloc) metric obtains the most of the explanatory power of our model (as determined by its proportional χ^2 of %85).

In the next section (Section 4.2), we explain the design of our empirical study. The results and discussions are presented in Section 4.3. The threats to the validity are discussed in Section 4.4. Finally, we draw conclusions and venues for future work in Section 4.5.

4.2 Empirical Study

In this section, we explain the design of our study. We (i) describe our studied quality outcomes, (ii) motivate each investigated RQ, and (iii) describe the data collection process and methodology for each investigated RQ. Our datasets are available in our online appendix[3].

4.2.1 Studied Quality Outcomes (Metrics)

To perform all of our analyses, we collect metrics from SonarCloud and TravisCI. The metrics that we collect from SonarCloud are the metrics that we refer to as *quality outcomes*. We motivate each of the collected metrics below (the source of the metric is indicated in parentheses).

Project Size or simply "ncloc" (SonarQube). SonarCloud performs quality inspections through static code analysis (e.g., it detects bugs and code smells without running the software (evans; larochelle, 2002)) We extract the *project size* metric from SonarCloud as a *support metric* in our experiments. We normalize our other metrics by the *project size*. For example, if two distinct

[3]https://ciqualityresearch.github.io/doctoral_thesis/#study2_datasets

projects have the same amount of code smells, but also have a huge difference in size, the code smells should be analyzed proportionally to the size of the projects. As for RQ2, we use the *Project Size* to account for a possible *confounding factor* in our regression model (i.e., to avoid the problem of observing spurious correlations solely because the projects differ in size). We measure *project size* through the *ncloc* metric (i.e., number of non-comment lines of code) that is available on SonarQube.

Bug Density (SonarQube). SonarQube defines several rules that are used in the static code analyses. The Quality Model of SonarQube divides these rules into four categories: *Bugs*, *Vulnerabilities*, *Security Hotspots*, and *Code Smells*. The rules related to *Bugs* detect pieces of code that are demonstrably wrong (or more likely wrong than not). For example, the *Bugs* category includes rules for detecting a `NullPointerException` in the code. In our work, we measure the *Bug Density* (SONARSOURCE, 2019) by using the following formula $\frac{\text{Number of Bugs}}{ncloc}$. *Interpretation:* The lower the *Bug Density* the better the quality outcome in a quality inspection.

Code Smells Density (SonarQube). The rules related to Code Smells detect the pieces of code that are hard to maintain (SONARSOURCE, 2019). According to Martin Fowler and Kent Beck[4] (FOWLER, 2018; TUFANO et al., 2015), code smells are indications that usually represent deeper problems (e.g., a very long method may harder to maintain or more likely to contain bugs). *We measure the Code Smells Density* by using the following formula: $\frac{\text{Number of Code Smells}}{ncloc}$. *Interpretation:* The lower the code smells density the better the quality outcome of a quality inspection.

Duplicated Lines Density (SonarQube). Duplicated Lines represents the number of lines involved in duplicated code. Duplicated code has been associated to lower maintainability over the years (RAHMAN; BIRD; DEVANBU, 2012). For example, having several duplicated methods may be troublesome, since a developer

[4] `https://martinfowler.com/bliki/CodeSmell.html`

would probably have to propagate a maintenance change to all of the clones (i.e., if one clone has been forgotten, a bug would likely occur). We measure the *Duplicated Lines Density* through the following formula: $\frac{\text{Number of Duplicated Lines}}{ncloc}$. *Interpretation:* The lower the duplicated lines, the better the quality outcome of a quality inspection.

Technical Debt (SonarQube). Technical debt reflects the implied costs when developers choose a suboptimal solution to fix a problem without considering the long terms consequences (SURYANARAYANA; SAMARTHYAM; SHARMA, 2014). SONARQUBE provides the technical debt in terms of the required minutes that would be necessary to fix all the detected *Code Smells*. We use this metric as the *response variable* in our regression model in RQ4. *Interpretation:* The lower the *Technical Debt* the better the quality outcome.

Technical Debt Ratio (SonarQube). This metric is a derivation of the aforementioned *Technical Debt* metric. SONARQUBE provides the *Technical Debt Ratio* as a comparable indicator, which can be used to compare different projects or different versions of the same project. The *Technical Debt Ratio* is denoted by the $\frac{\text{Technical Debt}}{\text{Development Cost}}$ fraction. The numerator (i.e., Technical Debt) is the estimated time (indicated by SONARQUBE) that would be necessary to fix all the detected *Code Smells*. The denominator (i.e., the development cost) is an estimation (by SONARQUBE) of the required effort to develop the whole source code of the project. The development effort is computed by multiplying the *ncloc* of the project by a cost estimation (expressed in terms of time, e.g., the *default cost value* is 0.06 days). Ultimately, the development cost is defined by the following algebra: $DevelopmentCost$ = Cost of each line $\times$ number of lines. *Interpretation:* The lower the *Technical Debt Ratio* the better the quality outcome in a quality inspection.

Code Coverage (SonarQube). Code Coverage is the degree to which automated tests exercise the source code of a program. There are several metrics to represent code coverage (ELBAUM; GABLE; ROTHERMEL, 2001). For example,

statement coverage measures the number of statements that are exercised by automated tests, whereas *branch coverage* measures which "program branches" (e.g., if statements, loops) have been exercised at least once by the automated tests. The coverage metrics are often used to indicate whether a software is well-tested (HILTON; BELL; MARINOV, 2018). Collecting coverage metrics requires (i) running automated tests from a test suite, (ii) instrumenting the code execution and (iii) observing the execution flow (as performed by Sizilio et al (SIZíLIO; COSTA; KULESZA, 2019)). Unfortunately, a local instance of SonarQube does not perform such a measurement process automatically. In addition, following the same approach as performed on our previous work (Chapter 3) would compromise the scalability of this study (i.e., the effort to configure, run, and instrument hundreds of projects would be herculean). Therefore, to compute the *Code Coverage* metric, we rely on the SonarCloud platform, which readily provides this metric for many projects that are hosted on SonarCloud. The *Code Coverage* metric is especially important as we strive to understand the relationship between the best practices of CI and the quality outcomes. *Interpretation:* The higher the value of the *Code Coverage* metric, the better the quality outcome in quality inspection.

Build Duration (Travis-CI). Continuous Integration employs an automated process to generate software builds. This automated process clones the code repository into a virtual environment and executes a pipeline of necessary tasks for generating the builds (such as compiling the code, running the automated tests and running the static code analyses). A CI pipeline can be customized to include/exclude different tasks depending on the requirements of the project. A duration of 10 minutes to build a project has been considered as an acceptable threshold to separate an acceptable build duration from a long build duration (BROOKS, 2008a; HILTON et al., 2017a; GHALEB; COSTA; ZOU, 2019b). We compute the *Build Duration* for our analyses, since having long build durations is considered a bad practice when using CI (FELIDRÉ et al., 2019; GHALEB; COSTA; ZOU, 2019b). *Interpretation:* a lower build duration would likely share a relationship with improved quality outcomes.

Required Time to Fix a Build (Travis-CI). Frequently integrating code while delivering successful builds is a core concept in CI. Indeed, CI dissipates its purpose if builds are constantly broken in the repository. Therefore, the ability to quickly fix broken builds is essential in a healthy CI environment (FELIDRÉ et al., 2019). We collect this metric by computing the duration in seconds between the time when a build was broken the next successful build. To aggregate this metric at the project level, we use the median *Required Time to Fix a Build*. We use to assess whether the adherence to CI best practices share a strong relationship with improved quality outcomes. *Interpretation:* The quicker a team fixes the broken builds the better should be the quality outcomes.

Build Activity Ratio (Travis-CI). Another good practice in CI is to integrate builds frequently (FELIDRÉ et al., 2019; DUVALL; MATYAS; GLOVER, 2007) (i.e., at least daily). Integrating builds frequently helps the development team to identify errors earlier, which is the purpose of CI (BELLER; GOUSIOS; ZAIDMAN, 2016). We compute this metric by using the following formula: $\frac{\text{CI Active Days}}{\text{SonarCloud Time Period}}$. The *CI Active Days* is the number of days where at least one build was generated in TravisCI, whereas the *SonarCloud Time Period* is the number of days of a project's historical information available on SonarCloud. We use the *Build Activity Ratio* to assess whether the adherence to CI best practices share a strong relationship with improved quality outcomes. *Interpretation:* the higher the *Build Activity Ratio* the better should be the quality outcomes.

CCQ Activity Ratio (SonarCloud/Travis-CI). The *CCQ Activity Ratio* is a metric inspired by the study from Vassalo et al. (VASSALLO et al., 2018). The authors observed that although projects were using CI, just a few of them were performing CCQ on SonarCloud. Therefore, if projects that use CI generate many more builds than the number of builds submitted to quality inspections on SonarCloud, such projects do not perform a proper CCQ. Hence, we compute the *CCQ Activity Ratio* by using the following formula: $\frac{\text{Number of TravisCI Builds}}{\text{Number of Builds Checked on SonarCloud}}$. *Interpretation:* A CCQ Activity of 1 indicates that there exists one quality check

(in a build) for every CI build (which signals a more proper CCQ usage). We compute the *CCQ Activity Ratio* for two purposes: (i) to ensure that we analyze only the projects that perform a proper CCQ (i.e., we filter out projects with a *CCQ Activity Ratio* lower than 1) and (ii) as a control variable in our regression model (in RQ4). Having the *CCQ Activity Ratio* as a control variable is important because the higher the *CCQ Activity Ratio* the higher the likelihood of observed problems (e.g., a project may report more Technical Debt because it performs a more intense CCQ). Hence, we keep the *CCQ Activity Ratio* as a control variable to ensure that the observed relationships do not occur solely because of a higher *CCQ Activity Ratio* in the projects.

4.2.2 Research Questions

The aim of our study is to empirically investigate whether adherence to CI best practices is related to the degree of *quality outcomes*. In our study we collect *Continuous Code Quality* (CCQ) metrics that are available on the SonarCloud platform (a online instance of SonarQube). We consider these metrics as the *quality outcomes* of our analyses, i.e., our response variables. In this study, we investigate two exploratory RQs and provide the motivation for each RQ below.

RQ1: Does the degree of* CI *adoption share a relationship with improved quality outcomes? Motivated by the research results that show that CI may not be related to a improve on quality outcomes(FREITAS , 2019) and that projects that adopt CI may not be using CCQ in practice, we further investigate this lack of relationships may be related to the level of adherence to the CI best practices (FELIDRÉ et al., 2019; DUVALL; MATYAS; GLOVER, 2007). Recent studies have shown that not all projects using CI actually employ the CI practices to the fullest (FELIDRÉ et al., 2019). For example, although CI has been adopted, projects may not necessarily produce frequent builds. Therefore, it is important to verify whether the degree of adherence to the CI best practices shares a relationship with improved quality outcomes.

Data collection & approach: In RQ1, we analyze whether the following CI practices share a significant relationship with improved quality outcomes: (i) *integrating at least daily*; (ii) *avoiding long-running builds*; (iii) *fixing a broken build as soon as possible* and (iv) *controlling for acceptable test coverage*. These investigated practices are inspired in the study performed by Felidré et al. (FELIDRÉ et al., 2019). We use the *Build Activity Ratio*, *Build Duration*, *Required Time to Fix a Broken Build*, and *Code Coverage* (see Section 4.2.1) to respectively study each of the aforementioned CI practices.

Figure. 17 shows an overview of the criteria to select the projects for RQ1. **Step 1.** We select all projects that have the *Test Coverage* information available. This criterion results in 3,053 projects. **Step 2.** Next, we filter for the projects that use GitHub. We attempt to locate the respective GitHub repositories of the projects. In this step, we discard the projects for which we cannot find the respective (and public) GitHub repository, (the project may use another version control system or there is no explicit link between SonarCloud and GitHub for such projects). **Step 3.** Afterwards, for the projects that we could properly find the GitHub repository, we verify these projects use TavisCI Steps 2 and 3 ensure that the studied projects use TravisCI and is published as an open source repository on GitHub. After step 3 we obtain 910 projects.

Step 5. Vassalo et al. (VASSALLO et al., 2018) observed that not all CI projects hosted on SonarCloud indeed perform CCQ. Therefore, we use the criteria proposed by Vassalo et al. (VASSALLO et al., 2018) to select only the projects that perform CCQ systematically. First, Vassalo et al. (VASSALLO et al., 2018) state that at least 20 quality inspections on SonarCloud are required to consider that a project actively uses SonarCloud. We apply this criterion in our data, which results in 798 projects. **Step 5.** Vassalo et al. (VASSALLO et al., 2018) also observed that projects with a low *CCQ Activity Ratio* may not be performing quality inspections properly. The authors define the *CCQ Activity Ratio* as the following fraction: $CQQ\_activity\_ratio = \frac{\text{Number of CI Builds}}{\text{Number Builds Inspected on SonarCloud}}$ (see Section 4.2.1).

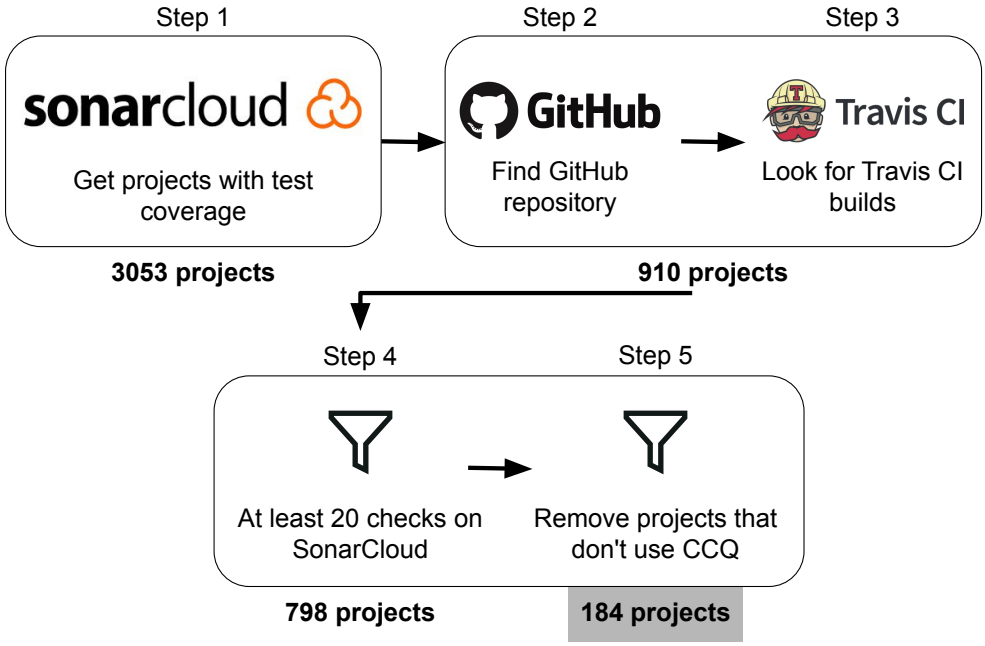


Figure 17: Overview of RQ3 and RQ4 projects selection.

A *CCQ Activity Ratio* close to 1 indicates that for each CI build there exists one SonarCloud build check, which indicates a proper CCQ usage. In step 5, we filter out the projects with a *CCQ Activity Ratio* below 1. Finally, 184 projects survive our project selection criteria.

To analyze each CI practice (e.g., *avoiding long-running builds*), we identify the projects that adhere to that practice (e.g., projects with acceptable build durations) and the projects do not adhere to that practice (e.g., projects with long build durations). To this end, we use an approach similar to the approach employed by Ghaleb et al (GHALEB; COSTA; ZOU, 2019b). First, we compute the *median value* of each project with respect to a given CI practice. For example, with respect to the *avoiding long-running builds* practice, we compute the *median* build duration (in seconds) for each project. We generate a distribution of 184 median values of build duration. Afterwards, we use this distribution to generate the quantiles shown in Table 6. We use the `dplyr`[5] R package to compute the

[5] `https://www.rdocumentation.org/packages/dplyr/versions/0.7.8`

Table 6: Explanatory variables expressed in quantiles

	Build Duration (sec.)	Time to Fix (sec.)	Coverage (%)	CI Build Activity ($\frac{\#CIDays}{\#SonarDays}$)
Q1	59.0 to 146.0	0.0 to 1048.0	0.00 to 52.65	0.01 to 0.21
Q2	146.5 to 231.0	1052.0 to 2172.0	53.30 to 78.35	0.21 to 0.44
Q3	233.0 to 417.0	2179.0 to 7694.0	79.10 to 89.50	0.44 to 0.73
Q4	428.0 to 3110.0	8289.0 to 490133.0	89.60 to 100.00	0.73 to 1.50

quantiles. We consider the *median* in our analyses because the median is more robust against outliers than the *mean* (LEYS et al., 2013).

The projects for which the median build duration falls under the first quantile are considered as *adhering* the *avoiding long-running builds* practice. Conversely, the projects for which the build duration falls under the fourth quantile are considered as *not adhering* the *avoiding long-running builds practice* practice. The projects which fall under the second or third quantiles are excluded from our analyses, since we wish to compare the extreme sides of adherence level for each CI practice. This same process is repeated for the other CI practices (i.e., *integrating at least daily*, *fixing a broken build as soon as possible* and *controlling for acceptable test coverage*). For each CI practice, we interpret the quantiles accordingly. For example, while a higher build duration is detrimental, a higher test coverage is indeed positive. Hence, we adjust our interpretation of *adhering* and *not adhering* depending on the specific analyzed CI practice. To perform the statistical comparisons between the quantiles, we use MMW tests and Cliff's delta measurements.

RQ2: Which practices of* CI *share a relationship with technical debt? In RQ2, we deepen the analyses performed in RQ1 and investigate which practices of CI are mostly important with respect to positive/negative quality outcomes. It is essential to understand which CI practices should be prioritized in case not all CI practices can be adopted by the team at a certain period in time. For instance, not all best practices may be adopted (yet) due to cultural or organizational barriers, e.g., a development team whose build frequency was one build per week may feel

an initial friction before shifting to producing hourly builds.

Data collection & approach: We use the same dataset used in RQ1 to address RQ2. In RQ2, we build a linear regression model to study the importance of each CI practice with respect to the *Technical Debt Ratio* (see Section 4.2.1). A linear regression model fits a linear equation that explains the relationship between a set of *explanatory variables* $X = x_1, ..., x_n$ and a *response variable* Y. In RQ2, each studied project is considered an observation. Therefore, the number of observations in our dataset is 184. To represent the *explanatory variables*, we use the per-project median values of each CI practice studied in RQ3 (i.e., *avoiding long-running builds*, *integrating at least daily*, *fixing a broken build as soon as possible* and *controlling for acceptable test coverage*). We also add two *confounding variables* in our set of explanatory variables. Our confounding variables are the *Project Size* and the *CCQ Activity Ratio* metrics (see Section 4.2.1). The motivation to add these confounding variables is explained in Section 4.2.1. As the *response variable*, we use the per-project median value of *Technical Debt Ratio* (explained in Section 4.2.1). We study the *Technical Debt Ratio* in our regression model as this metric factors in all the other quality outcomes. Ultimately, we obtain an initial setup of 184 observations (i.e, projects) and 6 explanatory variables per observation. According to Harrell Jr. (JR, 2015), because our ratio of observations to explanatory variables (i.e., $\frac{184}{6} = 30$) is above 15, we have sufficient *degrees of freedom* to fit a linear regression models without running into the risk of overfitting.

Before fitting our linear regression model, we perform a correlation analysis because correlated variables may impair the ability to understand the relationships between the explanatory variables and the response variable (JR, 2015). To this end, we use the `varclus` function of the `rms` R package.[6] We observe no pair of explanatory variables within all correlation clusters with a correlation $|\rho| > 0.7$. Therefore, we keep all the explanatory variables (JR, 2015). Next, we check for *redundant* variables, as they may also distort the shared relationships between the

[6] `https://cran.r-project.org/web/packages/rms/index.html`

explanatory and response variables (JR, 2015). We use the `redun` function of the `rms` R package to check for *redundant* variables. We do not observe any redundant pair of variables between our explanatory variables.

Next, we use the `ols` function ("ols" stands for Ordinary Least Squares) of the `rms` package R package to fit our linear regression model (henceforth referred as LM). We evaluate the goodness-of-fit of the model by analyzing the R^2 metric. We also use the `validate` function of the *rms* R package to perform a resampling validation of our model with a number of iterations of $B = 1,000$. This validation is important to show how stable is the fit of our model (JR, 2015).

Finally, we evaluate the *explanatory power* of our explanatory variables through the Wald χ^2 metric. The higher the χ^2 value for a given explanatory variable the stronger the shared relationship between the explanatory variable and the response variable (*Technical Debt Ratio*).

4.3 Results

In this section we explain the approach and findings for each of our research questions.

RQ1: Does the degree of CI adoption share a relationship with improved quality outcomes?

Observation–1. Projects with longer build durations tend to have worse quality outcomes. Long-running builds are undesirable as developers would waste precious time waiting for the builds. This matter worsen in CI, where builds are expected to be generated more frequently (GHALEB; COSTA; ZOU, 2019b). A build duration of 10 minutes is often used as a threshold for determining whether a build has a long duration (BROOKS, 2008a; GHALEB; COSTA; ZOU, 2019b; HILTON et al., 2017a). Table 7 shows our obtained $p - values$ and

Cliff's deltas when comparing the first and fourth quantiles of *Build Durations*. Our results suggest that longer build durations are associated with worse quality outcomes. We also show violin plots for these comparisons in Fig. 18.

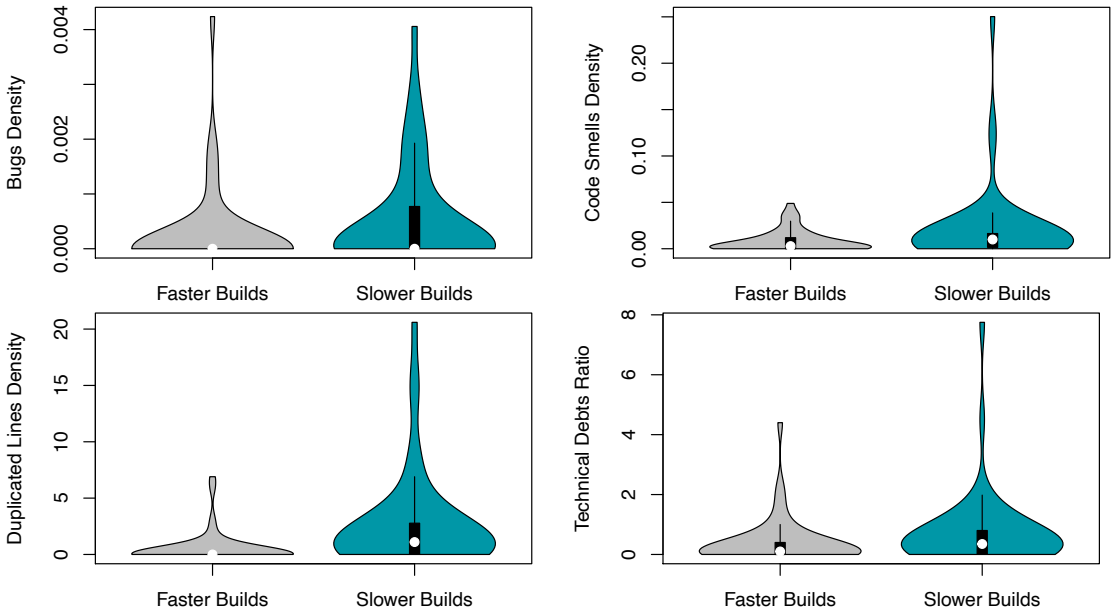


Figure 18: Comparison of CI projects build duration and independent variables

Table 7: Build Duration vs. Quality Outcomes

Metrics	Wilcoxon	Cliff's Estimate
Bug Density	0.003349	−0.2896975 (small)
Code Smells Density	0.01462	−0.294896 (small)
Duplicated Lines Density	$8.275e-07$	−0.5500945 (large)
Technical Debts Ratio	0.01967	−0.2759924 (small)

Observation–2. The required time to fix broken builds does not share a strong relationship with the studied quality outcomes. Another CI practice that we evaluate is the required time to fix a broken build. The CI guidelines state that a team "should not go home before fixing a broken build" (FOWLER; FOEMMEL, 2006).

Table 8 shows the obtained $p-values$ and Cliff's deltas when comparing the first and fourth quantiles of the *Required Time to Fix Broken Builds* metric. Our

results reveal no apparent relationship between the time required to fix broken builds and our studied quality outcomes (i.e., all $p-values > 0.05$). We also show violin plots for these comparisons in Fig. 19

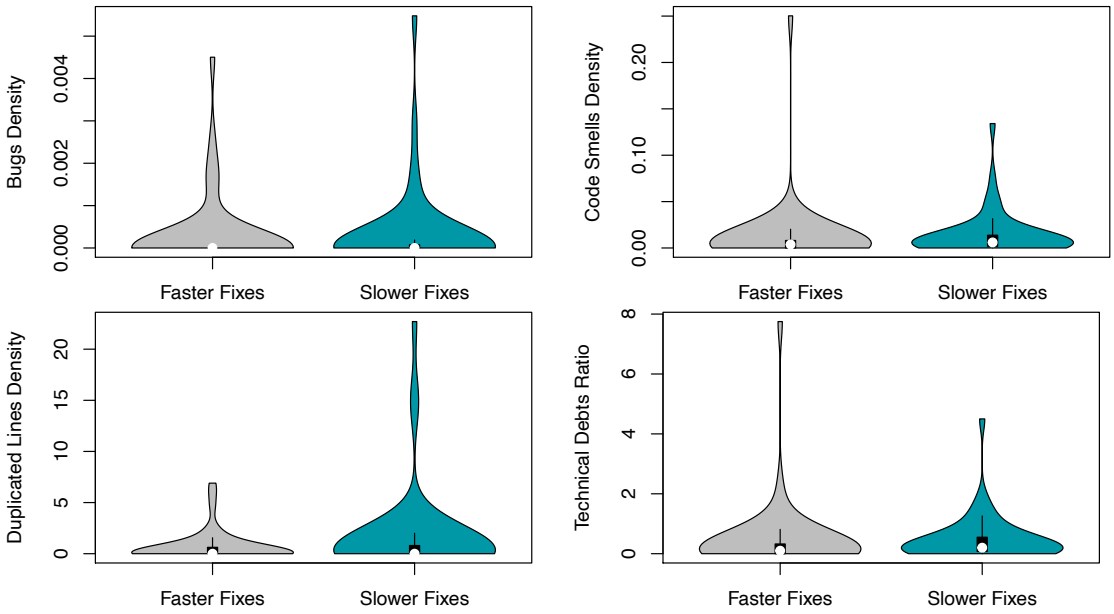


Figure 19: Comparison of CI projects time to fix and independent variables

Table 8: CI Time to Fix vs. Quality Outcomes

Metrics	Wilcoxon	Cliff's Estimate
Bug Density	0.3659	−0.08350951 (negligible)
Code Smells Density	0.09904	−0.2040169 (small)
Duplicated Lines Density	0.6525	−0.04756871 (negligible)
Technical Debts Ratio	0.1191	−0.1908034 (small)

Observation–3. A higher test coverage shares a significant relationship with improved quality outcomes. Continuous Integration is intrinsically related to the best practices of software tests (DUVALL; MATYAS; GLOVER, 2007) (GRANO et al., 2019) (BELLER; GOUSIOS; ZAIDMAN, 2016). Test coverage is often used to assess how well an application is tested (GRANO et al., 2019) (HILTON; BELL; MARINOV, 2018).

Table 9 shows the obtained $p-values$ and Cliff's deltas when comparing the first and fourth quantiles of *Code Coverage*. Our results suggest that, in overall, projects with a higher test coverage share a significant relationship with improved quality outcomes. We also show violin plots for these comparisons in Fig.20.

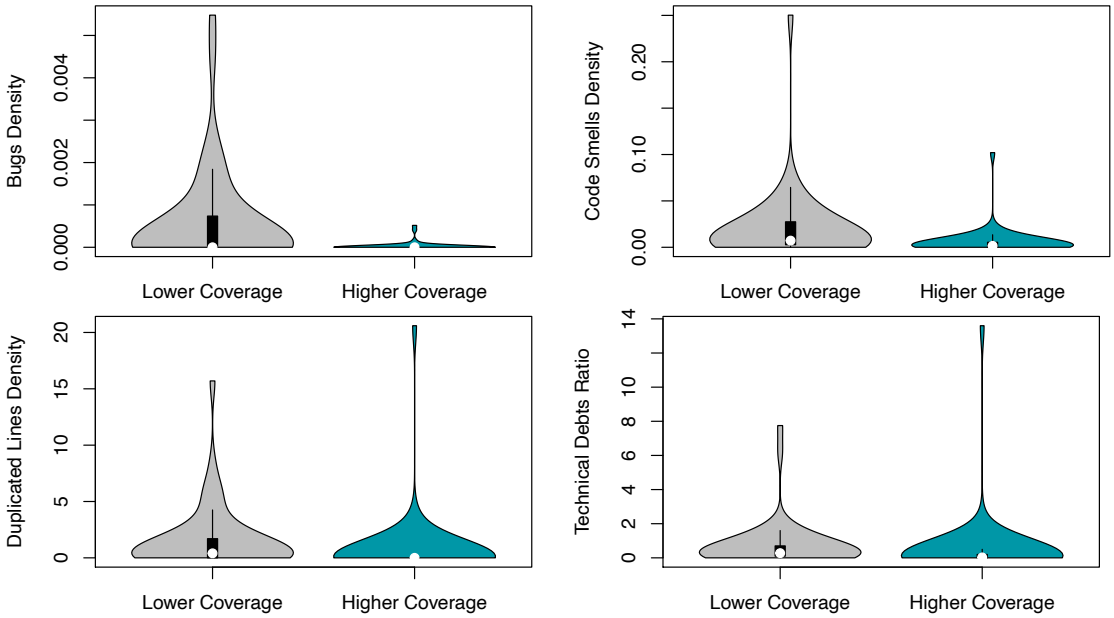


Figure 20: Comparison of CI projects coverage and independent variables

Table 9: Coverate Effect

Metrics	Wilcoxon	Cliff's Estimate
Bug Density	$2.211e-05$	0.4083176 (medium)
Code Smells Density	0.0005421	0.4149338 (medium)
Duplicated Lines Density	0.0001984	0.3903592 (medium)
Technical Debts Ratio	0.0006362	0.4031191 (medium)

Observation–4. The* Build Activity Ratio *does not share a significant relationship with the studied quality outcomes. Building and integrating code frequently (at least daily) is a core principle of CI (DUVALL; MATYAS; GLOVER, 2007) (FOWLER; FOEMMEL, 2006). As explained in Section 4.2, we calculate the *Build Activity Ratio* from our projects' history available on SonarCloud. Table 10 shows the obtained $p-values$ and Cliff's deltas when comparing the first

and fourth quantiles of *Build Activity Ratio*. Our results reveal no apparent relationship between the frequency of builds and the studied quality outcomes. We also show violin plots for these comparisons in Fig.21.

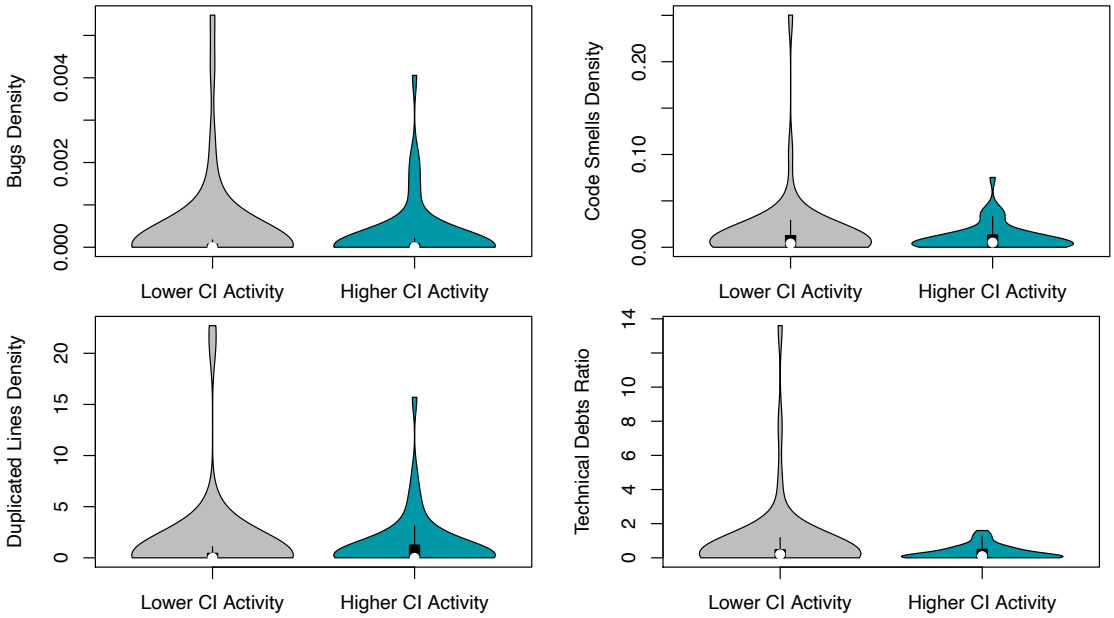


Figure 21: Comparison of CI Build Activity and independent variables

Table 10: CI Build Activity Effect

Metrics	Wilcoxon	Cliff's Estimate
Bug Density	0.9074	0.01134216 (negligible)
Code Smells Density	0.8353	0.025519852 (negligible)
Duplicated Lines Density	0.3081	−0.1063327 (negligible)
Technical Debts Ratio	0.615	0.0600189 (negligible)

Discussion: Our results suggest that only *Test Coverage* and *Build Duration* share a strong relationship with the studied quality outcomes (positively in overall). It is surprising to observe that the (i) *Required Time to Fix Broken Builds* and the (ii) *Build Activity Ratio* do not share strong relationships with the studied quality outcomes. Regarding the *Required Time to Fix Broken Builds*, our observation may be explained by the fact that not all build breakages on TravisCI may be important or they may even be noisy (Gallaba et al., 2018). Therefore,

despite observing a breakage, developers may not rush to fix them in case the broken jobs (which are responsible for the breakage) are not really perceived as important. In fact, TravisCI has a feature called "allowed failures", which allows developers to state which jobs are "allowed" to break without influencing the build outcome. In a future study, we plan to investigate the perceived importance of build breakages and preemptively warn the development team about important breakages. As for the lack of a significant relationship between the *Build Activity Ratio* and the quality outcomes, this could be explained by the fact that, although building frequently, the development team might not aggregate quality to every build (DIGKAS et al., 2018). Therefore, our results suggest an addendum to the core CI principle of *building frequently* (DUVALL; MATYAS; GLOVER, 2007). This addendum would be to *not only build frequently, but also strive to aggregate quality to every build* (e.g., by addressing technical debt whenever possible).

RQ2: Which practices of CI share a relationship with technical debt?

Observation–5. Technical debt is largely explained by the project size. Table 11 summarizes the results of our model, presenting the model coefficients, standard errors and statistical significance of each explanatory variable (as indicated by the asterisks). We observe that our model fits well the data ($R^2 = 0.744$ and $AdjustedR^2 = 0.734$).

Table 11 also shows the overall χ^2 and the $p-value$ of our model. The *Project Size* metric (i.e., median_ncloc) obtains the most of the explanatory power of our model (as determined by its proportional χ^2 of %85). As expected, our confounding variable *CCQ activity ratio* also obtains a significant role (i.e., the more the performed CCQ, the more the identified Technical Debt). However, the explanatory power of the *CCQ activity ratio* (proportional χ^2 of %11) is far superseded by the explanatory power of the *Project Size*. The explanatory power of our variabels is expressed in the plot in Fig. 22. Interestingly to note is that, similar to RQ1, our

Table 11: Regression Results

	Coef	S.E.	χ^2	Prop. χ^2
Median Coverage	-21.718^{*}	12.500	3.0	≈1%
Median Build Duration	1.859^{**}	0.861	4.7	≈1%
Median Required Time to Fix	0.002	0.005	0.2	≈0%
CI Build Activity Ratio	$-1,037.885$	1,172.193	0.8	≈0%
CCQ Activity Ratio	638.799^{***}	111.347	32.9	≈11%
Project Size	0.203^{***}	0.013	247.7	≈85%
Constant	-789.148	1,183.778		
R^2				0.744
Adjusted R^2				0.734
Note:				*p<0.1; **p<0.05; ***p<0.01

model indicates that the effect of *Build Duration* and *Code Coverage* are significant (although outpowered by the *Project Size*).

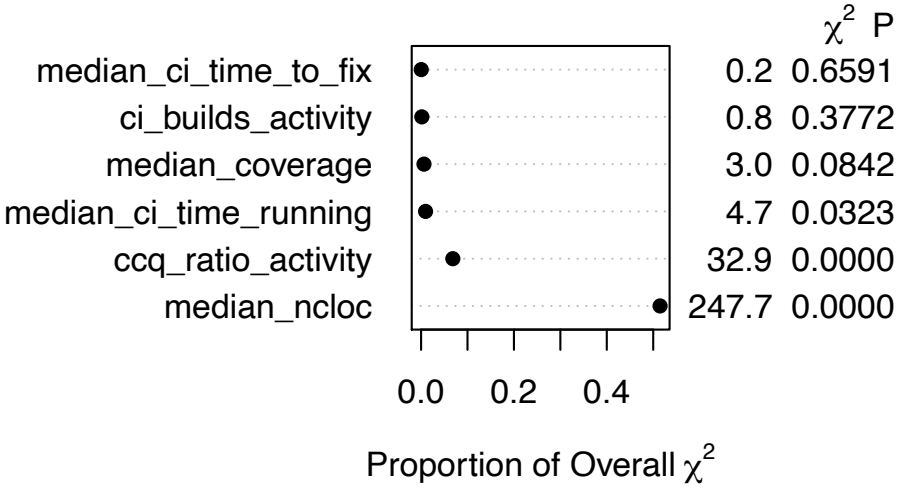


Figure 22: Independent variables explanatory power

Discussion: The fact that the *project size* far supersedes the explanatory power of the remaining variables, corroborates with the hypothesis that software systems tend to drop their quality as they grow bigger (HERRAIZ et al., 2013). Therefore, our results suggest that, unless a deliberate effort is invested to increase the software quality (i.e., by deliberately removing the Technical Debt in our case), such improvement in quality will unlikely be ensured just by the adherence to certain

CI practices. In terms of CI practices, our results also suggest that, if not all practices can be adopted, it is crucial to at least maintain a short build duration and a significant code coverage (as they have shown some significance in relation to Technical Debt).

4.4 Threats to the Validity

Construct Validity. The construct threats to validity are concerned with errors caused by the methods that we use to collect and model our data. For this study, we rely on data collected from the TravisCI, and SonarCloud APIs. Especially regarding the metrics collected from SonarCloud, we may have contracted some bias. For example, most probably, all of the studied projects have used the default values provided by SonarCloud when computing the *Technical Debt* metric. Further studies should investigate the impact that optimizing such default values might have on the quality outcomes.

In addition, although the *Code Coverage* metric is stored SoarCloud in the form of percentages, the responsibility for providing this metric relies on the projects. Therefore, it is challenging to determine whether some projects have informed the, for example, the *branch coverage* or the *methods coverage*. Nevertheless, we believe that our analyses regarding the *Code Coverage* metrics are still important given that, universally, the higher the *Code Coverage* percentage the better the quality outcome.

Moreover, in many phases of our experiments, it was required to take decisions regarding thresholds. For example, the utilized *CCQ Activity Ratio*. Researchers that replicate our work using different thresholds in different steps may obtain different results. Nevertheless, we strived to always take these decisions by consulting the existing and highly regarded empirical research.

Internal Validity. Internal threats are concerned with the ability to draw conclusions regarding the relationship between the explanatory variables (i.e., in

our case, the CI best practices) and the response variable (e.g., the metrics explained in Section4.2.1).

We acknowledge that the metrics to express the CI practices in this study is not exhaustive. It is always possible to customize and polish the metrics presented in this study or add extra metrics. We plan to extended and customized the our metrics in future work.

External Validity. External threats are concerned with the extent to which we can generalize our results. Although 184 projects, we cannot generalize our results to other projects with different domains, skill-sets and sizes. We also cannot generalize our results to corporate settings as the practices enforced in a corporate setting might differ from those performed in an open source project. Nevertheless, we strive to perform our analyses applying criterias of highly regarded empirical research (VASSALLO et al., 2018) (for example, by analyzing projects that perform CCQ systematically).

4.5 Conclusion

Motivated by the lack of evidence about the relationship between continuous integration and quality oucomes (FREITAS, 2019), we study the degree of CI adherence. We investigate the relationship between CI good practices and software quality. Concerning this aspect, we rely on online data of 184 projects that monitor their quality using SonarCloud. We set out to compare metrics that represent four CI good practices (i.e., avoid long-running builds; fix a broken build as soon as possible; control for acceptable test coverage and integrate at least daily). Our study compares the collected values of the CI good practices metrics for the best and worse projects against the quality outcomes. We also create a linear regression model to estimate the technical debt of projects. The results show that:

- Maintaining a short build duration and a high code coverage are essential to

reduce technical debt (see Observations–1, 2, and 5)

- technical debt is largely explained by the project size, outpowering the positive effects of short build durations and high code coverage (see Observations–5).

In overall, our results lead us to conclude that, unless a deliberate effort is invested to increase the software quality (i.e., by deliberately removing the Technical Debt in our case), such improvement in quality will unlikely be ensured just by the adherence to certain CI practices.

5 Estimating causal effects of Continuous Integration in software quality outcomes

5.1 Introduction

Previous works empirically study the relationship between continuous integration and software outcomes in many aspects. Our last chapters focused on answering which factors associated with continuous integration may have a statistically perceptive correlation with quality attributes. Determine the correlation relationship is essential to give the practitioner an overview of what results come with CI and in what conditions. However, the results cannot be used to determine the causation relationship. Mahdavi et al. (DAMGHANI et al., 2012) explain correlation as a statistical measure that describes the size and direction of a relationship between two or more variables. However, a change in one variable does not necessarily cause a difference in the value of the other variable. Regarding our previous work, we may assume that CI has a *association* with quality outcomes. We can not state, however, that CI *causes* the effect on quality.

Correlation analysis might lead researchers to wrong conclusions if there is no clear understanding of the data behind it. The *sympson's paradoxes* is one example of how data can suggest contradictory findings according to the perspective given to it. As Judea (PEARL, 2019) shows, *sympson's paradoxes* represents a correlation

Table 12: Example of Sympson's Paradox

	Drug	**No Drug**
Total Sample	273 out of 350 recovered (78%)	289 out of 350 recovered (83%)
Men	81 out of 87 recovered (93%)	234 out of 270 recovered (87%)
Women	192 out of 263 recovered (73%)	55 out of 80 recovered (69%)

that can be observed in the population, but it inverts if we stratify into sub-groups. This scenario can be observed in the Table 12. It represents a scenario where a total of 350 patients chose to take the drug, and 350 patients did not take it. It also shows that the recovery rates of the patients who were given access to the drug. When we observe the population, the results show that more patients recovered by not using the drugs. If we observe each gender separately, we can see that both men and women have better recovering rated by taking the drug. Observing the population, we conclude that the drug does not improve the recovery rate, but observing the gender shows otherwise. Although this scenario might appear weird at the first look, if we understand the history behind the data, we can understand the causal relationship. In the example, being a woman represents a bias. Women are more propensity to take the drug, but at the same time, estrogen decreases the drug effect, i.e., women have their own scenario and must be observed separately. This example shows how important it is to understand the data behind the correlations and to study the causal mechanism.

In this matter, this chapter aims to answer whether and how CI *causes* an effect on quality. The understanding of the cause-effect relationship is fundamental to avoid bias when making a decision. Our results suggest, for example, that CI has a relationship with software quality. The current thesis statement says, however, that it is not ensured just by the adherence to certain CI practices. What factors or practices in a CI environment cause the change in the quality is an open question to be answered statistically.

We propose to use the Causal Modeling as the method to infer the causation relationship. Explaining causal relationship is not trivial. Judea (PEARL, 2019) exposes that scientist was unable to determine mathematical equations to explain simple facts like "Mud does not cause rain". The affirmation is evident to human perception, but how to put this in a theorem and test it mathematically and statistically? Judea (PEARL, 2019) claim that Causal Modeling permits scientists to express causal questions formally, codifying their existing knowledge in both diagrammatic and algebraic forms, and then leveraging data to estimate the answers. The method is used in other areas, especially social science, and epidemiology (WINSHIP, 2007; VANDERWEELE, 2015; MORTON; FRITH, 1995), helping scientists extract causal relations from associations and deconstruct paradoxes that have baffled researchers for decades (PEARL, 2019).

The Causal Modeling considers that any causal inferences from observational studies must ultimately rely on some kind of causal assumptions and gives an effective language for making those assumptions precise and explicit, so they can be isolated for deliberation or experimentation and, once validated, integrated with statistical data (PEARL, 2019).

The literature shows that Causal Modeling is a well-established approach and widely used in other areas (WINSHIP, 2007; VANDERWEELE, 2015; MORTON; FRITH, 1995; PEARL, 2019). For the best of our knowledge, all the current works that statistically investigate the adoption of CI rely on association relationship. None of them presented a more in-depth study to try to investigate causality. This chapter presents a study uses the Causal Modeling approach to investigate the causality between CI and software quality.

In the next section (Section 5.2) we explain the design of our study. The results and discussions are presented in Section 5.3. The threats to the validity are discussed in Section 5.4. Finally, we draw conclusions and venues for future work in Section 5.5.

5.2 Study Design

Most of the practitioners adopt CI with the expectation of gaining quality in software outcomes. However, there is no common sense of what CI causes in the results. The software development community advocates CI as an excellent tool for helping the development of software with more quality. However, when and how that occurs, using the science of empirical studies remains an open matter. This lack of assay leads the community, among our previous studies presented in the earlier chapters, to perform multiple studies to test CI relationships in many contexts.

The current scenario of studies in this area shows that CI has a relationship with many aspects of software quality. Many assumptions were, however, proven to be not the absolute truth. Nevertheless, the techniques employed focused on the seek for associations and not necessarily represent causation relationships. For example, the first study presented in this thesis shows that CI shares a relationship with the test ratio growth. However, we keep asking ourselves if CI was the cause of such growth. To answer this question, we perform a **causal study**, applying the method presented by Pearl et al. (PEARL, 1995) (PEARL; GLYMOUR; JEWELL, 2016).

The technique consists of expressing the causal assumptions of the real world in a DAG that contains the known factors and the causation between them, representing how each phenomenon relates to each other with the semantic of cause. Furthermore, the proposed DAG is tested with data collected from real samples to estimate the causal effect between observed factors considering its covariants and confoundings. An overview of the approach shows that we first need to construct the DAG and represent the relationship between the variables. Furthermore, we must collect real data to determine if the DAG represents the truth or if it must be adapted to another scenario. With valid assumptions expressed in the DAG, we can finally process the theorems presented by Pearl et al. (PEARL; GLYMOUR;

JEWELL, 2016) to compute the probabilistic equation that represents the causal effect between two variables in the graph. How we conduct these steps is expressed in Fig. 23.

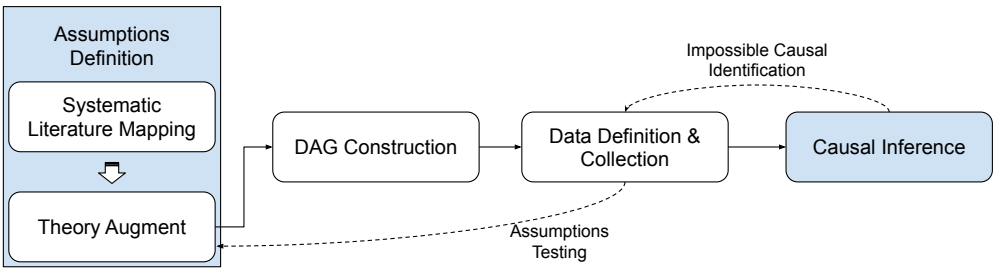


Figure 23: Causal Study Overview

Systematic Literature Mapping: Following the proposal presented by Pearl et al. (PEARL, 1995), our study must start with the construction of the causal knowledge in the area. In this regard, as illustrated in Fig. 23, we start our process with the Assumptions Definition. This step focuses on the creation of a bigger picture of the CI relationships. In other words, we start the study by gathering information on how practitioners and researchers recognize the CI approach as to affect software product-related aspects. To collect this knowledge, we first rely on the assumptions reported in the literature by performing a systematic literature mapping focused to obtain any causal relationship between CI and software factors.

The Systematic Mapping performed in this study follows the guidelines proposed by Wohlin et al. (WOHLIN et al., 2012). We first performed the **identification of research**, which is the step that focuses on specifying search strings and applying them to databases to collect the candidate papers from the literature. Then we focused on the **selection of primary studies** and **study quality assessment**, using inclusion and exclusion criteria. As the inclusion criterion, we focused on the selection of papers that exposes a clear intention to evaluate CI effects. To which regards the exclusion criterion, we exclude the non-peer reviewed papers. We decided to design a very inclusive criterion since we intended to collect

opinions even though it was not statistically evaluated. With the papers selected, we perform a **data extraction and monitoring** and **data synthesis** to prepare the data to the analyses and to answer the research questions:

- **RQ1** - What are the impacts of the Continuous Integration adoption on the software outcomes?

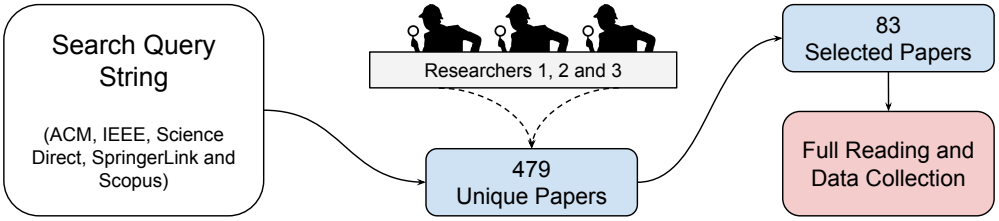


Figure 24: Systematic Mapping process overview

The process executed is expressed in Fig. 24. We first query for candidate papers on the main digital repositories for the technical literature (ACM[1], IEEE[2], Science Direct[3], SpringerLink[4] and Scopus[5]). We perform a query using the search string as follows:

TITLE-ABS-KEY ("continuous integration" AND ("impact" OR "outcome" OR "evaluation" OR "effect" OR "relationship" OR "influence" OR "importance" OR "consequence" OR "study"))

The search process results in 479 unique candidate papers. Two researchers analyze each article by applying the inclusion criteria. At this point, we ensure that the included documents show an evaluation of CI in any software aspect. We exclude documents that explicitly introduce any new element in the CI technique that represents a modification of the practice. If there is no consensus in the inclusion between the two readers, a third researcher performs the final judgemnent,

[1]https://dl.acm.org/
[2]https://ieeexplore.ieee.org/
[3]https://www.sciencedirect.com/
[4]https://link.springer.com/
[5]https://www.scopus.com/

this is the case of 34 of the 479 candidate papers. We compute the Cohen's Kappa Coefficients that results in an agreement of 0.7071. After the inclusion criteria applied to the 479 candidate papers, the process results in 83 papers for full reading and data collection.

The full reading is a process performed by three researchers. We split the dataset, so each paper is read once. However, to reduce bias, the results of the data collection were reviewed and discussed by pairs. Each reading focused on answering the questions presented and explained in Table13.

Table 13: Questionaire of data collection in the Systematic Mapping

Question	**Rationale**
What are the assumptions presented on the paper?	This question aims to collect each assumption that expresses the causality of Continuous Integration in any software aspect.
What are the variables related to the assumptions?	We collect this answer to understand the criterion and to determine if it is possible to collect the data used to determine the assumption.
How the assumption was validated?	This question collects how the assumption was determined and if it was statistically tested.
How many projects were involved in the study?	This aim to determine if the study might represent a significant dataset of observations.
There are open-source projects involved in the study?	This question is related to reproducibility aspects and aims to determine if the assumption can be extended to open source projects.
The study is based on domain-specific projects area? Which one?	This question is also related to reproducibility aspects and aims to determine if the assumption can be extended to other areas or if it is exclusive to an specific domain.
Does the study have the potential for replication?	In this question we determine if the study has clear potential for replication considering the reproducibility to another projects datasets.

The resulting data of the collection was furthermore encoded to assumptions and analyzed by a pair of researchers. In some cases, we might join assumptions in

a more general concept, i.e., some papers may conclude similar assumptions with different but comparable theories. In such cases, the pair of researchers decide if it expresses the same assumption and can, consequently, joined together.

Theory Augment: One of the main contributions of our systematic literature mapping is an overview of assumptions related to CI adoption. The resulting data shows every possible relationship between CI and aspects of software presented in the included papers. Despite the broad outlook provided by that, our study focuses on investigating the relationship between two factors, i.e., continuous integration and its effects on automated tests. The study presented in this chapter enhances the correlation already expressed by our first analysis shown in Chapter 3.

To which regards the effect of CI on automated tests, we use the domain knowledge of the assumptions resulted from the systematic mapping to derive a causal theory about this relationship. The theory represents the assumptions that might lead to an impact on the automated tests, considering the factors produced in the systematic mapping. To construct the theory, we consider the factors that are impacted by CI and enrich the knowledge with assumptions on how these factors might also affect on automated tests, resulting in a representation of how CI affect automated tests among its covariants. This theory is further expressed in a DAG that will be analyzed with the causal inference technique.

DAG Construction: The causal inference presented by Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) is based on a graphical representation, an DAG, of the relationship among factors that represents the behavior of the real world. To estimate the causal effect, we must apply the theorems presented by the authors to determine a probabilistic equation. In our context, the real-world relationship is obtained by the results of the systematic mapping as collected assumptions. As we have shown in the previous section, we enhance the collected assumptions with the knowledge that links CI and its covariants that might also lead to an impact on automated tests. At this point in the process, we construct a DAG that must represent our theoretical assumptions and, hopefully, the real world.

Data Definition & Collection: Construct a valid dag is a premise in the causal theory proposed by Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016). It means that our assumptions must represent a valid relationship between the variables in the real-world. At this point in the process, we have a DAG constructed by the theory. However, we cannot proceed to the analysis without validating if that DAG consists of an acceptable representation of the data. Thus, we define the variables to represent each factor in our dag and collect the corresponding data to validate our dag.

Nevertheless, our DAG may represent some variables that might be difficult or impossible to measure. For example, it would be improbable to find a variable feasible to collect, and that might represent the pressure that a team is subjugated at some period of data. Although we can find a way to express that factor, it most challenging to have it for all the projects. Thus, we define the common data that we can collect for all projects and disregard the others. Although we do not collect some variables it is still very important to the causal theory, i.e., the variables that we do not expect to collect remains at the model, we only set them as unobserved and this is crucial to compute if the causal effect still possible to compute.

To perform our analyses we consider the same project dataset presented in the first study of this thesis. We analyze two groups of projects: (i) 82 open source projects that have adopted CI at some point of their lifetime (CI projects) and (ii) 82 open source projects that have never adopted CI during their lifetime (NOCI projects). Accordingly, we also collect the 12 months historical data before the adoption of CI and 12 months after the adoption of CI for each of our CI projects and the data corresponding to the proportional period in the NOCI projects (Section 3.2.2). In this case, however, we encode our variables to represent dichotomy events, i.e., binary variables. For example, if the test ratio increases from one version to another, it will receive *one*. Conversely, if the test ratio does no increase, it will receive *zero*. The dichotomy events represent if some phenomena occur. For instance, if within versions the variable X causes Y, both must receive 1 because

both events occur.

Considering the collected variables, we further test the validity of our model. A valid model represents only associations that hold a semantic link with the data. In other words, every causal assumption in the DAG must express correlated data in the real-world. In this regard, we build a correlation matrix between the variables using the *Spearman Correlation Coefficients Test* (MYERS; SIROIS, 2004). Every association in the DAG that is not supported by the *p-value* in the correlation tests must be excluded, and the DAG must be reconstructed.

As the variable definition and the data collection is a process that depends on the DAG construction, i.e., we mostly need to draw our assumptions before performing our analyses, we present the collected data and the rationale further in the results section.

Causal Inference: To check if our DAG can lead to an estimated value of causality between the exposure and the outcome, we apply the *do-calculus* method presented by Pearl et al. (PEARL, 1995) and implemented by the *causaleffect*[6] R package. The method consists of a set of inference rules, which can be used to express the interventional probability distribution using only observational distributions (TIKKA; KARVANEN, 2018). As the algorithm implemented in *do-calculus* is considered to be complete, so there are only two possible returns of its execution: it will return a conditional probabilistic equation that represents the causal effect between the exposure and the outcome. If the causal effect could not be estimated, otherwise, it will return an error.

In a complete and fully observed model, the causal effect will always be identifiable. Nevertheless, we might have unobserved variables that we decide to do not collect. In such cases, it is possible that causal effect cannot be identifiable and we must come back to the model *Data Definition & Collection* and probably to the *Dag Construction*. If the Causal Inference is possible, we use our data to compute the conditional probability to represent the causal effect.

[6]https://cran.r-project.org/web/packages/causaleffect/index.html

5.3 Results

In this section, we will present the results of our study according to the planning explained in the previous section. Here we show the results of the systematic mapping as the assumptions that represent the overviews of the CI causality expressed in the Literature. Furthermore, we show the resulting theory that maps the causality between CI and automated tests. Finally, we analyze our theory to estimate the causal effect between CI and automated tests.

5.3.1 Mapping Causal Assumptions

One of the main contributions of this work is the overview of the assumptions established about CI and its causality. The main focus of this subsection is to determine and draw a broader image of how the community, including researchers and practitioners, considers the CI causality. The first question aimed to be answered in the systematic mapping is to determine **what are the impacts of the Continuous Integration adoption on the software outcomes?** As shown in previous section, in order to answer this inquiry, we map all the assumptions in our canditate papers about factors that CI might impact on. The resul of this investigation can be summarized in the Fig.25.

Fig.25 represents all the causality assumptions, direct or indirect, that we found following the systematic mapping process. We draw a graph to present an overview of the results that represent the causality landscape; however, that is not our DAG yet. We will cover the DAG development in further sections. Nevertheless, the causal analysis with DAG requires an understanding of real-world relationships. The Appendix A presents a summarized view of the relationships. The $\nearrow$ represents a direct relationship between the variables and the $\searrow$ an indirect relationship. Additionally, to explain the rationale behind the assumptions, we provide a swift description of each relation in the graph to provide shared knowledge to researchers that may enhance our study in the near future.

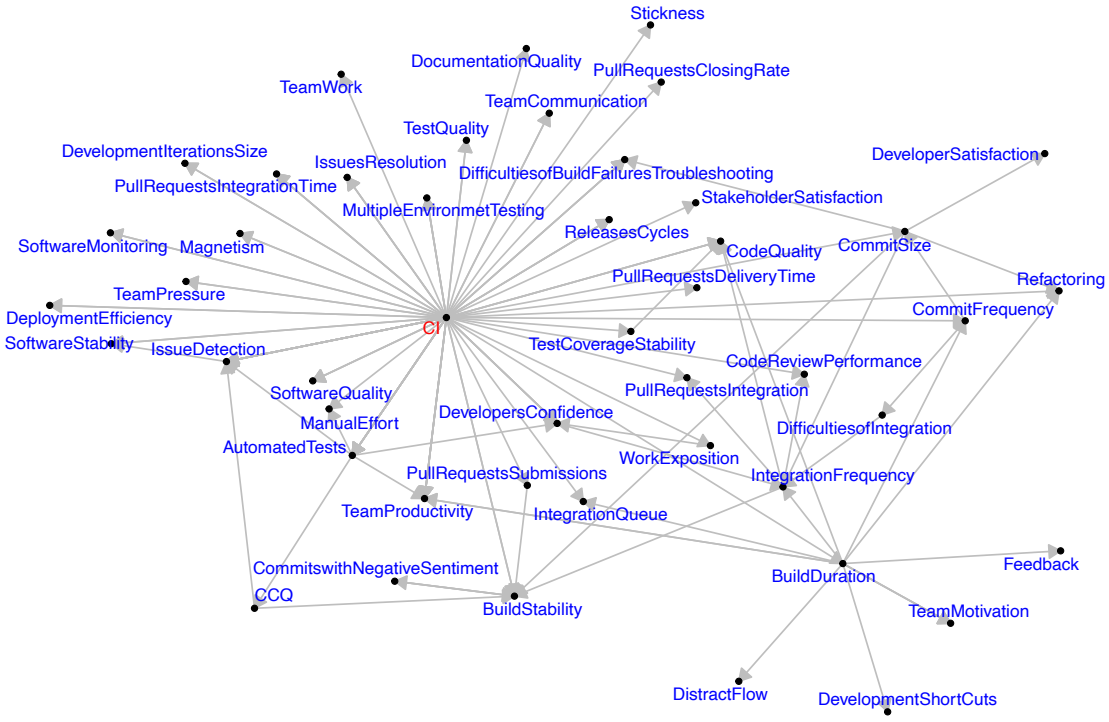


Figure 25: Systematic Mapping assumptions overview

CI ↗ AutomatedTests: Among our first study (Chapter 3), where we investigate the relationship between CI and the volume of tests in an application, other studies also investigated this question. Johnson et al. (JOHNSON; EKSTEDT, 2016) proposed a theory which states that CI increases the effort on test development and consequently improve the automated tests. Gupta et al. (GUPTA et al., 2017a) performed a survey and found that, in terms of testing, after some (expected) initial adjustments, the amount (and potentially the quality) of automated tests seems to increase. Additionally, multiple cases studies (STÅHL; BOSCH, 2013; YUKSEL et al., 2009; WILSON, 2009), applied to different contexts, found that there is a relationship between continuous integration and the increase of automated tests. Our first study complements the previous and related work by introducing a broader approach with substantial projects dataset and introducing statistical methods to compare CI and automated test evolution.

CI ↘ ManualEffort • AutomatedTests ↘ ManualEffort: Pinto et al. (PINTO et al., 2018) performed a survey with 158 CI users, and, among other conclusions, the authors expressed that adopting CI helps to automate manual steps on the development. Additionally, the theory proposed by Johnson et al. (JOHNSON; EKSTEDT, 2016) argues that manual effort also decreases an effect on the increase in automated testing.

CI ↗ SoftwareQuality • CI ↗ CodeQuality • CI ↗ TestQuality • CI ↗ DocumentationQuality: With respect to software quality, studies have proposed the assumptions that represent a positive relationship between CI and the gaining of quality in software aspects (PINTO et al., 2018; KAYNAK; ÇILDEN; AYDIN, 2019; WILSON, 2009). More specifically, to which regards the code quality, the results of the survey presented by Pinto et al. (PINTO et al., 2018) improves software quality in various aspects, including by improving the quality of code. Embury et al. (EMBURY; PAGE, 2018) and Hilton et al. (HILTON et al., 2017b) also validated this assumption in a case study with students and a survey, respectively. The work of Pinto et al. (PINTO et al., 2018), also studied the software quality

in a more specific concern, including the assessment of Documentation Quality and stating that CI improves the software in this aspect. The assumption that CI also improves the test quality also appears in the survey proposed by Pinto et al. (PINTO et al., 2018) and (HILTON et al., 2017b).

BuildDuration $\searrow$ ***CodeQuality*** $\bullet$ ***TestCoverageStability*** $\nearrow$ ***CodeQuality*** $\bullet$ ***IntegrationFrequency*** $\nearrow$ ***CodeQuality***: Regarding the impact of CI in the Code Quality, additionally to works that studied the direct relationship, some works investigated the effect through its covariances. Our second study (Chapter 4), where we explore the degree of CI adoption and code quality, shows that Build Duration shares a negative effect on the Code Quality, i.e., longer builds are related to worse code quality in terms of issues in SonarCloud. The same study also shows that maintaining a high Test Coverage Stability is likely to be associated with an increase in Code Quality. Furthermore, although our research concludes that code quality is not related to Integration Frequency, the work proposed by Rahman et al. (RAHMAN; ROY, 2017) shows a significant relationship between Integration Frequency and the quality of code reviews. This relationship might be expressed as an enhancement on the code quality control process and, consequently, influence the code quality.

CI $\nearrow$ ***DevelopersConfidence*** $\bullet$ ***AutomatedTests*** $\nearrow$ ***DevelopersConfidence*** $\bullet$ ***DevelopersConfidence*** $\searrow$ ***IntegrationFrequency*** $\bullet$ ***CI*** $\nearrow$ ***WorkExposition*** $\bullet$ ***WorkExposition*** $\searrow$ ***DevelopersConfidence***: Developers Confidence is the degree of how secure the developers feel to contribute by coding in a project. Multiple surveys evidentiate that CI may improve the Developers Confidence (PINTO et al., 2018; SU et al., 2013; HILTON et al., 2017b, 2016c). This assumption might relate to the interview results shown by Mårtensson et al. (MÅRTENSSON; STÅHL; BOSCH, 2017) that the increase in the degree of automated tests also leads to an improvement in the Developers Confidence and, consequently, increases the Integration Frequency. Adversely, these assumptions contradict what is exposed by the interviews presented by Debbiche et al (DEBBICHE; DIENÉR; SVENS-

SON, 2014), which shows that CI may also increase the Work Exposition, i.e., CI developers are worried about integrating low-quality code that could be questioned by experts and managers. Thus, the authors conclude that Work Exposition may reduce Developers' Confidence.

CI ↘ DevelopmentIterationsSize • CI ↘ ReleasesCycles •: Pinto et al. (PINTO et al., 2018) show that practitioners believe that CI faster development cycle. This assumption relates to the survey proposed by Hilton et al. (HILTON et al., 2017b), which concludes that CI allows faster iterations and shorter releases cycles (KAYNAK; ÇILDEN; AYDIN, 2019; HILTON et al., 2016c; LAUKKANEN et al., 2018).

CI ↗ SoftwareStability • CI ↗ TestCoverageStability • CI ↗ IssueDetection • AutomatedTests ↗ IssueDetection • IssueDetection ↗ SoftwareStability• CI ↗ IssuesResolution: One of the most benefits advocated by CI practitioners is that it helps in building more stable projects. Our first study, presented in Chapter3 shows that this stability is related to an consistent testing and test coverage stability. Lai et al. (LAI; LEU, 2015) relate this phenomenon with the fact that CI help to reduce various risk (e.e., testing, integration, communication, quality, and productivity risks). Yuksel et al. (YUKSEL et al., 2009) presented a case study that complies with this assumption, mostly relating the stability with a consistent presence of automated tests. Additionally, the early and more efficient issue detection is one of the most advocated benefits of CI adoption (PINTO et al., 2018; KAYNAK; ÇILDEN; AYDIN, 2019; HILTON et al., 2017b; VASILESCU et al., 2015b; HILTON et al., 2016c; AMRIT; MEIJBERG, 2017; KOCHHAR et al., 2019; MELLEGARD et al., 2018; STÅHL; BOSCH, 2013), causing an improvement in the software stability. Finally, according to Beller et al. (BELLER; GOUSIOS; ZAIDMAN, 2017a), testing is the single most important reason for integration to break, sharing a direct relationship of improvement between issue detection. This improvement of the issue detection process is also followed by a relationship among CI and an improvement in the issue resolution process (RAHMAN

et al., 2018; GUPTA et al., 2017a; ZAYTSEV; MORRISON, 2013).

CCQ ↗ IssueDetection • CCQ ↘ BuildStability • AutomatedTests ↗ CCQ: Adversely to what presented before, where Beller et al. (BELLER; GOUSIOS; ZAIDMAN, 2017a) considers that improving issue detection may improve software stability, some may consider that detection more issues will lead to more instability on the build. In other words, detecting issues only improves software stability if there is an effort to fix those issues. This aspect is evident by the assumptions proposed by Zampetti et al. (ZAMPETTI et al., 2017), which says that Continuous Code Quality (CCQ) improves the issue detection, which is explained by the new verification rules attached to the build pipeline and that will raise more errors. However, according to the authors, this leads to a decrease in software stability, mostly because with more rules the builds will break more often. This point of view shows that CCQ must be followed by a continuous refacoring environment (VASSALLO; PALOMBA; GALL, 2018) and, as shown in Chapter4, with a high disciplined CI environment (e.g., by performing efficient testing) in order to perceive its benefits.

CI ↗ TeamProductivity • CI ↗ DeploymentEfficiency • CI ↗ PullRequestsDeliveryTime • CI ↘ PullRequestsIntegrationTime • CI ↗ PullRequestsClosingRate • CI ↗ PullRequestsSubmissions • CI ↗ PullRequestsIntegration • IntegrationFrequency ↗ PullRequestsIntegration • PullRequestsSubmissions ↘ BuildStability: Another impact of CI that is perceived by practitioners and researchers is related to the productivity of the project. According to the literature, CI is related to an increase in team performance in various ways (KAYNAK; ÇILDEN; AYDIN, 2019; HILTON et al., 2017b; VASILESCU et al., 2015b; STÅHL; BOSCH, 2013; ZAYTSEV; MORRISON, 2013). Surprisely, this productivity was reported as negativelly related to the degree of automated tests in a project (GREN, 2017). It was also reported as an aspect that leads to an improvement in the deployment efficiency (SU et al., 2013; HILTON et al., 2017b). More specifically, multiple works investigate the productivity regarding

aspects related to Pull Requests. Bernardo et al. (BERNARDO; COSTA; KULESZA, 2018b) show that CI is related to an increase in the pull request delivery time, i.e., delivering slowly after the adoption. Despite this, the author, among other reports (YU et al., 2016; HILTON et al., 2016c), shows that the pull request integration time is reduced (BERNARDO; COSTA; KULESZA, 2018b), meaning that CI integrates pull requests quickly even though it is delivered slowly to the releases. Additionally, the improvement trend is also perceptive when considering pull requests closing rate (GUPTA et al., 2017a). Conclusively, CI projects receive and integrate more pull requests (BERNARDO; COSTA; KULESZA, 2018b; VASILESCU et al., 2015b). When considering the indirect effects, the increased integration frequency also relates to an increase in the pull requests integration (ZAMPETTI et al., 2019). Finally, when perceived an increase in the pull requests submissions, it might cause some significant build instability (RAUSCH et al., 2017).

CI ↗ IntegrationQueue • BuildDuration ↗ IntegrationQueue • IntegrationFrequency ↗ CodeReviewPerformance • CI ↗ CodeReviewPerformance • CI ↗ Refactoring: In addition to the previous assumptions about CI and the aspects of team performance, we might consider the side effect that CI may also cause an increase in integration queue (YU et al., 2016). In other words, projects with long builds may cause team performance loss since the developers tend to wait until the build to pass to progress with their work (BROOKS, 2008b). The integration frequency is also considered to be related to code review performance, i.e., the most frequent are the integrations, and consequently, the higher the degree of CI, the better the process of code review (RAHMAN; ROY, 2017; YU et al., 2015). Analog to that, CI is considered to improve the code refactorings (YUKSEL et al., 2009).

CI ↗ BuildStability • IntegrationFrequency ↗ BuildStability • CommitSize ↘ BuildStability: The mapping of the literature assumptions shows multiple references to the fact that CI may affect build stability. It is possible to perceive that this factor is related to CI in many ways. Some studies reported

that there are perceptions that CI increases the build stability (HILTON et al., 2017b, 2016c), just like an increase in the integration frequency is also perceived to be associated with this factor (HILTON et al., 2016c). Notwithstanding, it is crucial to notice that we have many covariants that are improved by CI, and that impacts negativelly to build stability. For example, among other examples aforementioned, big commits are likely to be associated with instability on build (LAUKKANEN; MÄNTYLÄ, 2015).

CI ↗ CommitSize • CI ↗ CommitFrequency • CommitFrequency ↗ CommitSize • CommitFrequency ↗ DifficultiesofIntegration • DifficultiesofIntegration ↘ IntegrationFrequency • CommitSize ↗ IntegrationFrequency • CommitSize ↗ Refactoring • CommitSize ↗ DeveloperSatisfaction • CommitSize ↗ DifficultiesofBuildFailuresTroubleshooting: As observed above, commit size is also a key concept that is associated with CI and its covariances. Small commits are considered to be a good practice in software development, although it is hard to define this metric. Rahman et al. (RAHMAN et al., 2018) perceived that CI projects are associated with big commits and alto to an increase in the project commit frequency. In parallel, Laukkanen et al. (LAUKKANEN; MÄNTYLÄ, 2015) show that low commit frequency may be related to the big commits that might lead to difficulties of integration, which, consequently, leads to low integration frequency (MÅRTENSSON; HAMMARSTRÖM; BOSCH, 2017). Complementary, projects with small commits are associated with higher integration frequency (STARON et al., 2018), projects with more code refactorings, developer satisfaction, and a decrease of build failures troubleshooting (LAUKKANEN; MÄNTYLÄ, 2015).

CI ↗ TeamCommunication • CI ↗ TeamWork • CI ↗ StakeholderSatisfaction • CI ↗ TeamPressure • CI ↘ Magnetism • CI ↘ Stickness: CI is directly linked to agile teams and has been used as a tool to support some of the best practices of agile development. Team communication is crucial to the agile process, and CI is observed to be associated with an increase in that

aspect. CI is seen by practitioners as a methodology that helps to improve team communication and team Work (KAYNAK; ÇILDEN; AYDIN, 2019; STÅHL; BOSCH, 2013; RAHMAN et al., 2018). Additionally, it is recognized to improve stakeholders' satisfaction (FERREIRA; COHEN, 2008). However, CI might adversely increase the team pressure to deliver more often and with more quality (DEBBICHE; DIENÉR; SVENSSON, 2014). Yet it may be related to the magnetism and stickiness of developers, meaning that it can be related to the fact that the project attracts and retains fewer developers (GUPTA et al., 2017b).

CI ↗ SoftwareMonitoring • CI ↘ DifficultiesofBuildFailuresTroubleshooting • CI ↗ MultipleEnvironmetTesting: The overview of the CI practice shows that it goes beyond the build system; it includes a set of practices and premisses that must be observed. However, the tooling around CI is crucial to the success of the practice. This aspect must bring automation to its fullest. Regarding this automation, additionally to the previous assumptions, practitioners claim that CI improve the software monitoring (KAYNAK; ÇILDEN; AYDIN, 2019; EMBURY; PAGE, 2018). And, conversely, to what conclude Laukkanen et al. (LAUKKANEN; MÄNTYLÄ, 2015), with success on the monitoring, it might lead to a decrease in the difficulties of build failures troubleshooting. The tooling associated with the CI environment also might facilitate the multiple environment testing, which may be a complex setup without the proper intruments (PINTO et al., 2018; HILTON et al., 2017b; BELLER; GOUSIOS; ZAIDMAN, 2017a; ZAYTSEV; MORRISON, 2013).

CI ↘ BuildDuration • BuildDuration ↘ TeamMotivation • BuildDuration ↘ Feedback • BuildDuration ↘ IntegrationFrequency • BuildDuration ↘ CommitFrequency • BuildDuration ↗ DistractFlow • BuildDuration ↘ Refactoring • Finally, despite the clear evidence that CI is far more than only a build system tooling, it is obvious that CI has an intrinsic relationship with builds. Yuksel et al. (YUKSEL et al., 2009) presented evidence in a case study that CI may improve builds and decrease the build duration if it is properly con-

figured. However, this relationship with build duration might consider many other variables, and this may not always be true. It might depend on the proper build configuration, the project size, and the volume of test cases (GHALEB; COSTA; ZOU, 2019a). Nevertheless, keeping the build faster is a good practice. In fact, a long build duration may share a relationship with a low team motivation (DEBBICHE; DIENÉR; SVENSSON, 2014; WIDDER et al., 2019b; LAUKKANEN; MÄNTYLÄ, 2015) and to an impediment on feedback and to integration frequency (LAUKKANEN; MÄNTYLÄ, 2015). Additionally, long build duration may impact to a decrease in team productivity (HILTON et al., 2017b; BROOKS, 2008b) and might lead developers to commit less frequent (LAUKKANEN; MÄNTYLÄ, 2015) and to implement more shortcuts (i.e., workarounds) (LAUKKANEN; MÄNTYLÄ, 2015). Finally, long build duration difficulties the code refacoring (BROOKS, 2008b) and also distract the developer's focus (LAUKKANEN; MÄNTYLÄ, 2015).

This section presented an overview of CI and its relationship with multiple software aspects. We draw a broad scenario with all the hypothesis on how CI might impact on software. In the next sections, we focus on the relationship between CI and automated tests to present a theory about how CI may cause an improvement o the volume of tests.

5.3.2 On the continuous integration effect in Automated Tests, an causal theory

The previous section provides a broad overview of the relationships between CI and factors on software development. The literature shows that many research works have been studying CI and its benefits, however, many of the results are based only on the practitioner's perception and some does not provide statistical tests or substantial dataset analysis to test the findings statistically. However, we consider the dataset of hypotheses presented in the literature as a true representation of real-world knowledge, even though we know that some assumptions hold only for specific domains or scenarios.

In this chapter, we dive deeper into a specific association between CI and automated tests. with the expectation of finding a causal relationship between them. Our previous study (Chapter3) shows that CI shares a small relationship with an increase in the test volume in projects that have adopted CI. Notwithstanding, our first study disregards the complex relationships among CI presented by our systematic mapping. A causal inference should consider that covariants and confounding might represent bias and any correlation may lead to a wrong inference of causal relationship.

To perform our causal analysis, we must first consider how CI is correlated to automated tests. For that, we consider not only the assumptions presented before. We further increase it with theory assumptions. In this section, the theory means that we considered every association between CI and the factors presented in the systematic mapping study increased with new relationships that may represent the truth and is absent from the review. Figure 26 shows how we encode our theory. Grey arrows represent the assumptions founded in the literature, while green arrows were introduced to complete the causal relationships with wisdom knowledge. We keep from the systematic mapping only the assumptions that might affect automated tests. Tables 14 and 15 provides the rationale behind each edge on the graph.

As Figure 26 represents a causal theory, and every causal effect must hold a correlation between the variables, testing the assumptions to obtain a valid DAG is crucial. Moreover, we must collect real-world data to check if our DAG might represent the truth. It would be necessary, therefore to determine variables that may represent each one of our factors.

In our data collection, we do not propose to collect data for all of our variables. Consider, for example, that it would be improbable to find a scalable manner to represent how frequently projects perform integration if it is related to projects that do not use CI. Defining the integration frequency on CI projects might be consensus using Travis CI API but improbable to NOCI projects. As the approach

Table 14: Theory Assumptions

Assumption	**rationale**
Complex Changes (+-) Automated Tests	Complex changes may be hard to test and may decrease the test volume
Developers Experience (++) Automated Tests	Core developers, the ones that contribute frequently, may be more test-driven than sporadic developers and may increase the test volume
CCQ (++) Automated Tests	As much as the project checks for quality, it may check also for tests (e.g., coverage check)
Integration Frequency (++) Automated Tests	The integration frequency is an degree of CI activity and more active periods may increase Automated Tests
Issues Resolution (++) Automated Tests	Solving an issue may include new test-case to ensure the issue fixing
Project Activity (++) Automated Tests	More code activity on the project may represent more developers coding and consequently more tests being developed
Project Age (++) Automated Tests	We consider that increasing tests is a trend over the time
Project Size (+-) Automated Tests	Big projects are hard to test may difficult the test to co-evolve with production code
Pull Requests Integration (++) Automated Tests	Pull Requests politcy may include tetsts verification and may increase test volume
Team Size (++) Automated Tests	Big teams may include more test policies and may increase test volume
Time Pressure (+-) Automated Tests	Time pressure may decrease the effort on tests due to priority to deliver production code
Project Activity (++) Integration Frequency	More coding activity on the project may increase the integration frequency
CCQ (++) Code Quality	More quality checks may increase the code quality
Automated Tests (++) Issue Detection	Automated tests tend to detect issues early and may increase the number of issue detection
CI (++) CCQ	CI enable the integration with CCQ tools and may increase the frequency of quality checks
Project Age (+-) Commit Frequency	Old projects tend to have fewer commits
Commit Frequency (++) Project Activity	More commits may cause more coding activity
Issue Detection (+-) Build Stability	More issue detection may cause more instability on the builds
CI (++) Integration Frequency	CI increases the integration frequency

Table 15: SLR Assumptions

Assumption	**rationale**
CI (++) Automated Tests	CI increases the volume of tests
Complex Changes (+-) Build Stability	Complex changes has more potential to break builds
CI (++) Build Stability	CI increase build stability, i.e., builds tend to break less with CI
Developers Experience (++) Build Stability	Experienced Developers commits break less builds
CCQ (+-) Build Stability	CCQ tools increase the build checks which increase build breakage
CCQ (++) Issue Detection	CCQ tools increase the build checks with new issue types
Integration Frequency (++) Build Stability	As projects integrate more frequently, it increases the build stability
Build Duration (+-) Integration Frequency	Long builds decreases integration frequency
CI (++) Issues Resolution	CI helps team to fix more issues and early
Project Age (+-) Project Activity	As project age the coding activity decreases
Project Age (++) Build Stability	As project age the build stability increases
Project Size (++) Build Duration	Bigger projects tend to have longer build duration
Team Size (++) Build Duration	Projects with bigger teams tend to have longer build duration
Build Duration (+-) Commit Frequency	Long build duration decreases the commit frequency since the devs tend to wait the build to pass
CI (+-) Build Duration	CI decrease build duration
Build Duration (+-) Code Quality	Longer build duration generates low code quality
CI (++) Pull Requests Integration	CI increases the number of pull requests integration
Integration Frequency (++) Pull Requests Integration	More integration frequency also cause more pull requests integration
Team Size (++) Commit Frequency	Projects with bigger teams tend to have more frequency of commits
Team Size (+-) Build Stability	Projects with bigger teams tend to have more builds instability
Time Pressure (+-) CCQ	Teams tend to not do code quality checks on time pressuring

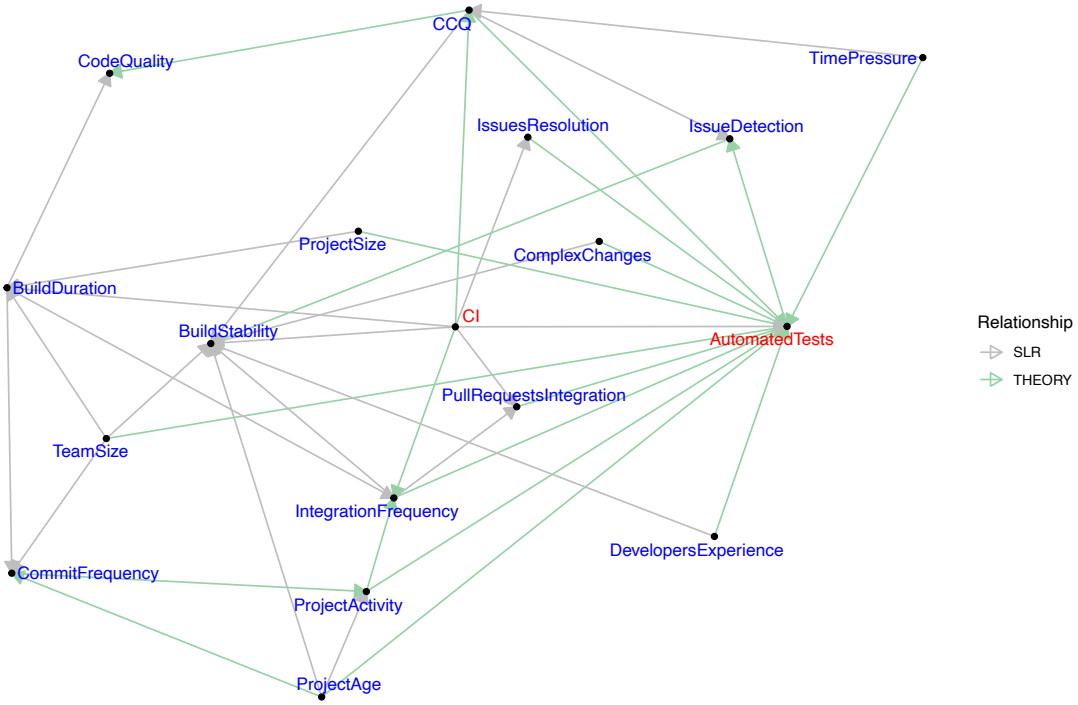


Figure 26: Causal theory of CI effect on the Automated Tests

proposed by Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016), through the theorems of backdoor and front-door adjustment, shows that it is possible to determine causal effect even with unobserved variables, so we decide to collect only the variables that are feasible to collect for all projects. Thus, we mark *TimePressure*, *BuildDuration*, *BuildStability*, *CCQ*, *IntegrationFrequency*, *CodeQuality*, *IssueDetection*, and *IssuesResolution* as unobserved. If some of these are proven to be indispensable to the causal inference, we might come back and refactor our definitions. Table 16 shows the remaining variables used to instrument the data of our DAG and to perform our analyses.

Even though we chose to not observe some variables due to the unfeasibility of data collection, our DAG is considered by our analyses as the representation of the truth. Therefore, we further need to test the associations with the collected

Table 16: Theory Factors Variables

Factor	**variable**
Automated Tests	Test ratio in the end of the period
CI	CI / NOCI period (Binary variable that represents if the project was using CI during the period)
Commit Frequency	# commits in the period
Complex Changes	Median code chrun of the commits in the period
Developer Experience	To compute the experience, we attach a score for each contributor in the project, which is the number of commits that each one made during the overall period (i.e., two years). This metric computes the overall experience attached to the version which represents the sum of all dev experience scores
Project Activity	Total code frequency, in terms of commits, in the period
Project Age	Repository age, in months, at the end of the period
Project Size	CLOC at the end of the period
Pull Requests Integration	# of Pull Requests integrated during the period
Team Size	# total of contributors in the version

data.

Our data collection considers the settings proposed in the first study of this thesis (Chapter3). Thus, we collect data of two years for 82 CI and 82 NOCI projects considering monthly versions. In the causal analysis proposed in this chapter, we encode the variables as binary, i.e., dichotomic *zero* or *one*. *One* represents an increase of that variable when compared to the previous version and *zero* otherwise. For example, if the project activity increases from one version to another, and the automated test ratio decreases, it would be represented a pair of 1 and 0, respectively.

Finally, we perform a spearman correlation tests, implemented by the function

rcorr of the *Hmisc*[7] R package, to generate a matrix of correlation p-values between our variables.

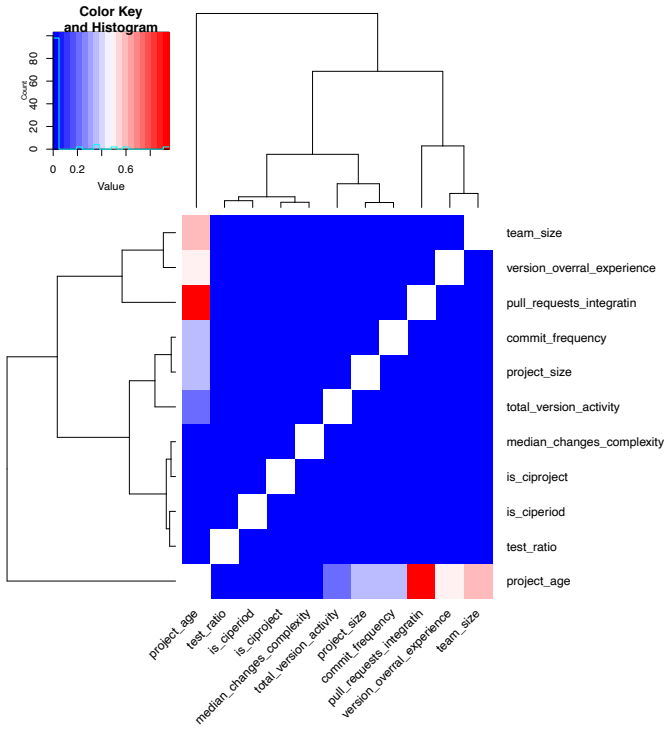


Figure 27: Correlation tests between theory factors

Figure 27 shows that our assumptions holds statistically significant correlations. However, the variable *ProjectAge* shows non-significant p-value to reject the null hypothesis. One explanation for that might be the fact that we consider only two years of data on the activity period of the projects. It may not represent enough period for the projects to be affected by time issues. Consequently, we discard all edges relate to Project Age from our theory since it does not carry a relationship to the real-world collected data. Figure 28 draws out a theory containing the remaining causal assumption and disregarding the rejected assumptions. In the next section, we check if the DAG is suitable to perform causal inferences.

[7]https://www.rdocumentation.org/packages/Hmisc/versions/4.3-0/topics/rcorr

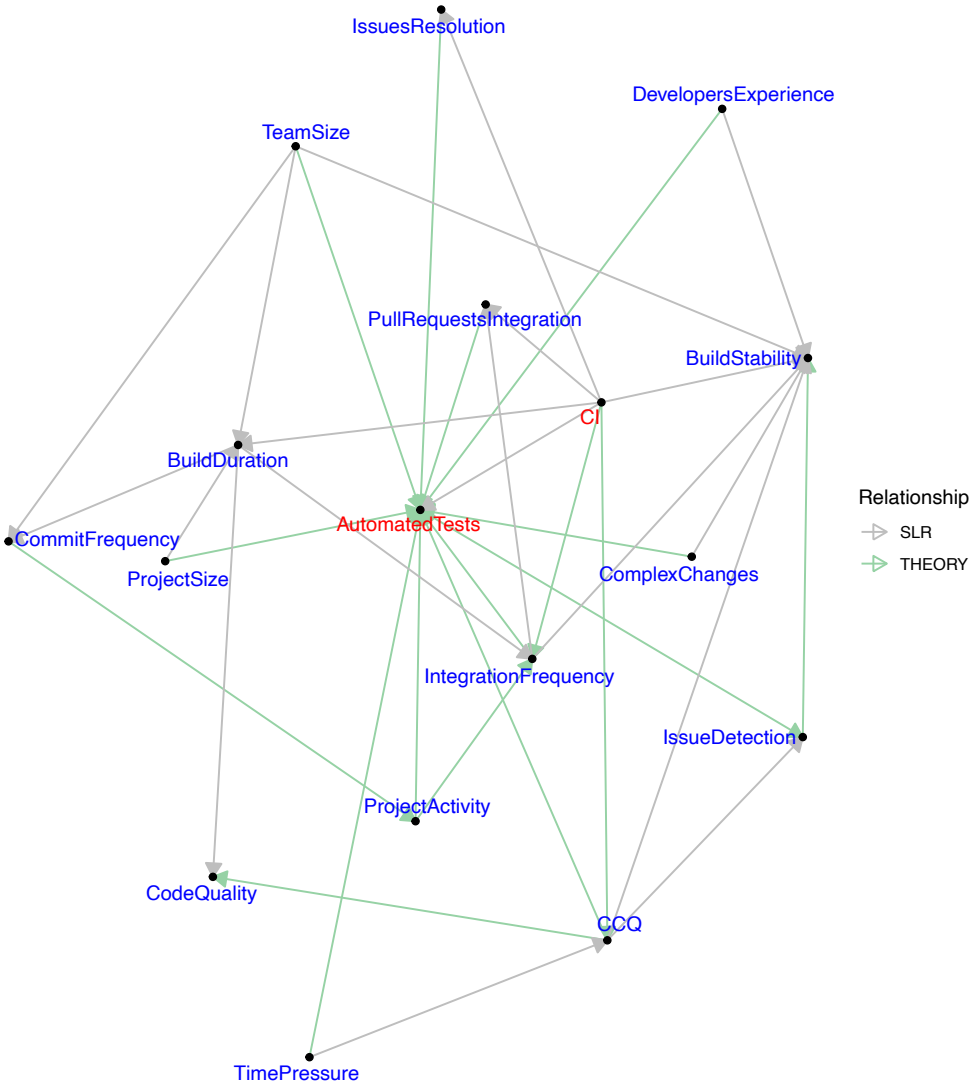


Figure 28: Final theory DAG after the correltion data rejection

5.3.3 Causal Inference

In this section, we provide an analysis of the total causal effect of CI in the automated tests. As an important aspect of the analysis and fundamental to the application of the theorems proposed by Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016), we first identify the open paths, causal and biasing, between the exposure and the outcome. We use the function *ggdag_paths* from the *ggdag*[8] R package to compute the open paths and draw the graph as shown in Figure 29. As illustrated in the figure, our DAG contains six open paths between CI and automated tests, all causal paths, and none biasing paths. Surprisingly, despite the apparent complexity of the relationships in the DAG, we do not need to adjust for any variable to express the causal effect. This is also manifested by the result of the *ggdag_adjustment_set*, also from the *ggdag* R package, which computes the minimal adjustment set as an empty set for our scenario.

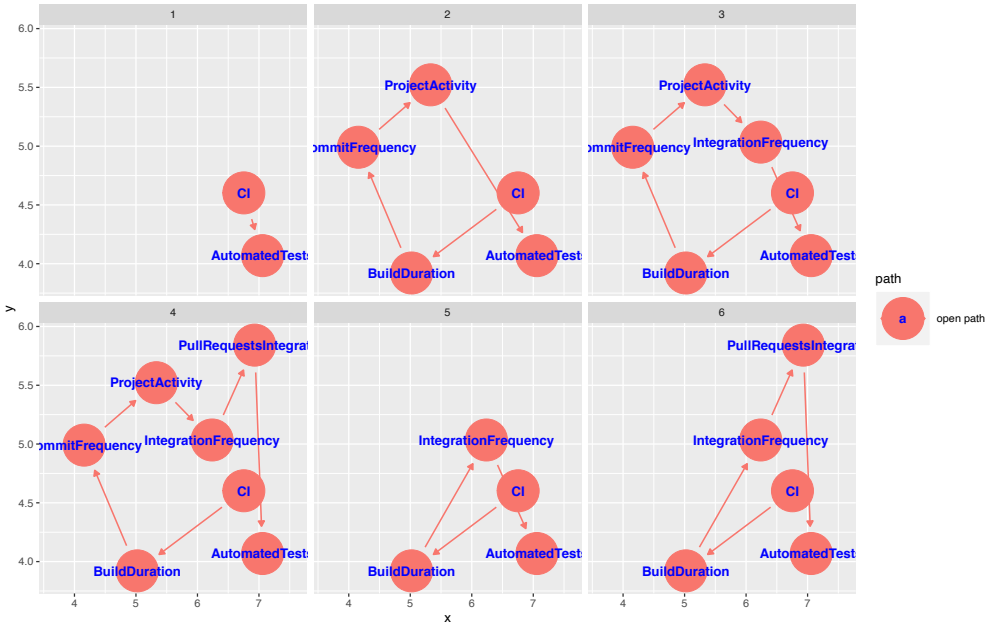


Figure 29: Final theory DAG after the correltion data rejection

Computing the causal effect consists in the discovery of the conditional prob-

[8]https://cran.r-project.org/web/packages/ggdag/index.html

ability equation that explains the occurrence of the outcome phenomena. Pearl et al. (PEARL; GLYMOUR; JEWELL, 2016) proposed the *do-calculus* function that comprehends the estimation using the proposed theorems (i.e., applying the front-door and the backdoor adjustment). The *do-calculus* will return an error if the causal effect might not be estimated and will return the probability function otherwise. To compute the *do-calculus* theorems, we rely on function *causal.effect* from the *causaleffect*[9] R the package. The function outcomes the following equation when we provide our DAG and asks for the causal effect of CI in the AutomatesTests:

$$\sum_{CC,DE,PS,TS,CF,PRI,PA} P(AutomatedTests|CC,DE,PS,TS,CI,CF,PRI,PA) \\ P(PA|TS,CF) \\ P(PRI|CI) \\ P(CF|TS) \\ P(TS) \\ P(PS) \\ P(DE) \\ P(CC) \tag{5.1}$$

We shorten the terms in the equation as CC (ComplexChanges), DE (DevelopersExperience), PS (ProjectSize), TS (TeamSize), CF (CommitFrequency), PRI (PullRequestsIntegration) and PA (ProjectActivity). The fact that *do-calculus* returned the equation represents that the causal effect may be estimated. However, this is not the best equation possible since the package does not simplify the result and it computed the backdoor criterion considering unnecessary data adjustment. As our exposure variable CI is exogenous, we can follow the backdoor adjustment

[9]https://cran.r-project.org/web/packages/causaleffect/vignettes/causaleffect.pdf

criterion with the empty set to the adjustment formula, and we can estimate it as a univariate regression. Thus, our equation can be expressed as the probability of the joint distribution like followed:

$$P(AutomatedTest|CI) \tag{5.2}$$

Computing our conditional probability, the effect of CI on the automated tests results in 0.5465769. This result shows that adopting CI expresses nearly 54% of probability to consequently cause an increase in the test ratio of the project versions. As a complement to the understanding of this value, we can filter our dataset to select the events where continuous integration was not present, i.e., NOCI periods. In such cases, the probability of having an increase in test ratio, even though there is no continuous integration associated, is nearly 32%. We argue that an improvement of nearly 22% is a considerable effect, given our context.

According to Pearl (PEARL; GLYMOUR; JEWELL, 2016), causation studies go beyond the statistics as it allows to uncover workings of the world that traditional statistics methods alone cannot. Our previous results show that CI is indeed associated with improvements in the test ratio, for that we compare two samples, CI and NOCI projects. In this chapter, we enhance our analyses by introducing the causal perspective and showing how the real-world variables interact between them and constructing a causal model that leads us to conclude that, considering the goodness of our model, CI causes a positive and substantial impact on the test ratio of projects.

5.4 Threats to the Validity

Construct Validity. The construct threats to validity are concerned with errors caused by the methods that we use to collect and model our data. For

this study, we rely on data collected from the GitHub APIs. We propose a set of variables to represent each aspect that CI might impact, these variables may have multiple interpretations, and many were not represented in our definitions. To minimize this bias, we ensure in our analyses that our variables are really correlated with their possible cause, meaning that our specification is feasible. To the variables that do not hold the correlation, i.e., the causal effect was not expressed in the data, we discard the relationship.

Internal Validity. Internal threats are concerned with the ability to draw conclusions regarding the relationship between the exposure variable (i.e., in our case, the CI adoption) and the outcome variable (e.g., the automated tests volume). We acknowledge that the associations resulting in the systematic mapping might not be complete to express the CI and all its effects. Although we perform a systematic approach to gather all information in the literature, we acknowledge that some assumptions might not be represented either because we miss some paper or because there are open questions to be answered by the community. In future works, we plan to extend our systematic mapping, including Snowballing papers, and enhance our DAG with more knowledge.

External Validity. External threats are concerned with the extent to which we can generalize our results. We cannot generalize our results to other projects with different domains, skill-sets and sizes. We also cannot generalize our results to private companies settings as the practices enforced in a corporate setting might differ from those performed in an open source project. Nevertheless, we collected a consistent dataset that is representative to the open-source community.

5.5 Conclusion

In this thesis, our previous works invested in the relationship between CI and software quality. The essence of these works investigates how CI is observed to be associated with some software quality aspects. Our first results show that CI has a

association with quality outcomes. However, despite the meaningful contribution that helps researchers and practitioners to understand this relationship, we can not state that CI causes better quality. The reason for this limitation is that our previous work does not observe the power of covariances and confoundings on the CI and quality associations.

The work presented in this chapter is designed to complement the results gathered before. In this case, we perform a wider outlook on the CI environment to observe the factors outcoming from it and the interrelationship between these factors. In this chapter, we present a pioneering study that goes beyond the correlation tests to investigate the estimated causal effect of CI adoption and its impact on automated tests. Thereby, we apply a causal inference using directed acyclic graphs and probabilistic methods to determine the causal effect of CI in automated tests. We highlight the following results:

- We perform a Systematic Mapping that helps researchers and developers to understand CI and its effects in a broader view. Our results show that the relationship between CI and software factors is not always straightforward, and there is a lack of empirical research studies that explain some aspects.
- Despite the CI adoption trade-offs, our causal modeling shows that it is likely to be associated with improvements in software quality. Additionally, it employs a considerable positive causal effect on the volume of automated tests (nearly 54% of probability).

Overall, our results lead us to conclude that, considering the scenario of our theory, CI causes an enhancement on the quality with a high likelihood to cause an increase in the volume of automated tests.

6 Related Research

In this chapter, we position our work with respect to the related research. We discuss the works about CI with regarding of ***Continuous Code Inspection & Software Quality in CI*** (Section 6.1) and ***Software Test Evolution & Coverage in CI*** (Section 6.2).

6.1 Continuous Code Inspection & Software Quality in CI

Hilton et al. (HILTON et al., 2016b) analyzed 34,544 open source projects from GitHub to understand which CI tools are mostly used by developers. They analyzed 1,529,291 builds from the mostly used CI tools to better understand how developer use CI. They observe that CI helps projects to release new versions more often and the overall percentage of projects using CI continues to grow. Our research differs from the work by Hilton et al. (HILTON et al., 2016b), since our focus is to understand the relationship between CI and quality outcomes.

Vasilescu et al. (VASILESCU et al., 2015a) studied the quality and productivity outcomes regarding CI on GitHub. The authors analyzed the code quality (e.g., number of bug reports raised in a project each month) and team productivity (e.g., efficiency on pull requests integrations). Their findings reveal that CI improves the productivity without reducing code quality. Vassallo et al. (VASSALLO et al., 2018) investigated a core principle behind CI, the Continuous Code Quality, which in-

cludes automated testing and automated code inspection. Their results reveal a strong dichotomy between theory and practice: developers do not perform continuous inspection. Instead, developers control the code quality only at the end of a sprint and, most of times, only on the release branch. Our work addresses quality by analyzing trends in test code. We use the test size (*n_test_loc*) to study the test ratio metric, which is the number of lines of code of automated tests over the total lines of the analyzed system. We use the test ratio overtime to discover trends. We compare the test ratio trends between CI and NOCI projects (Chapter 3). We also complement the observations regarding quality and CI by studying the relationship between CI and CCQ (Chapter 4).

Felidré et al. (FELIDRÉ et al., 2019) inspected 1,270 open-source projects that use TRAVISCI to quantitatively study how frequently CI has been misused. Felidré et al. (FELIDRÉ et al., 2019) study projects that perform (i) infrequent commits, (ii) poor test coverage, (iii) slow fix of broken builds, and (iv) long-running builds. Marcilio et al. (MARCILIO et al., 2019) studied whether the code violations reported by projects on SONARCUBE tool were indeed fixed over time. They analyze two open-source projects (Eclipse and Apache), and two Brazilian companies: the Federal Court of Accounts (TCU) and the Federal Police (PF) and observe a low resolution rate of code violations (only 13%) in all studied organizations. We also study code violations (e.g., bugs and code smells). However, inspired by the work of Felidré et al. (FELIDRÉ et al., 2019), we research how the adherence to the CI best practices are related to software quality outcomes (i.e., code violations metrics).

6.2 Software Test Evolution & Coverage in CI

Code coverage is the degree to which the source code of a program is executed by automated tests. The larger the coverage the more of the source code is exercised by the tests and the greater the possibility of finding a latent problem. There are several metrics to represent code coverage (ELBAUM; GABLE; ROTHERMEL, 2001),

such as statement coverage and branch coverage. Statement coverage represents the ratio of statements that are executed by automated tests over the total number of statements in the program. Branch coverage measures which possible program branches (e.g., if statements, loops) have been executed at least once during the tests. Collecting coverage metrics requires running automated tests from a test suite, instrumenting the code execution and observing the execution flow.

Code coverage metrics are often used to identify whether an application is well-tested. Hilton et al. (HILTON; BELL; MARINOV, 2018) performed a large-scale evaluation of code coverage in 7,816 builds of 47 projects written in popular languages including Java, Python, and Scala. The authors observed that the lines which are covered vary widely in a project even when the overall coverage appears to remain the same. However, the experiments of Hilton et al. (HILTON; BELL; MARINOV, 2018) considered only the last 250 commits of the projects when analyzing code coverage. The 250 last commits may not substantially represent the evolution of code coverage overtime. Complementing the prior work, our study aims to investigate the relationship shared by CI and the code coverage evolution in a time window of 2 years. Studying 2 years of code coverage allows us to provide a wider overview of the test evolution.

Zaidman et. al. (ZAIDMAN et al., 2008, 2011) studied whether the production code and the accompanying test code co-evolve. They investigated the versioning system, code coverage reports and code size-metrics of software projects. The findings reveal that the test code within the source code tends to increase with increasing coverage. Our research complements the study performed by Zaidman (ZAIDMAN et al., 2008, 2011) and colleagues by investigating the relationship that is shared between test code evolution and the adoption of CI.

Grano et al. (GRANO et al., 2019) highlight the importance of test evolution in software engineering and emphasize that test evolution is one of the central concerns in projects that have adopted CI. Their research advocates that having a previous knowledge about branch coverage (which can be achieved with test-

data generation tools) can help developers to select the subset of classes that must be tested and/or addressed by the test-case generation. Their work built and evaluated machine learning models to predict the achievable branch coverage by test-data generation tools. The authors studied 79 factors belonging to four different categories that might be correlated with branch coverage. Our work also aims to evaluate the code coverage of CI projects. However, instead of predicting a possible branch coverage for a specific software version, we study the impact of adopting CI on the test coverage of software projects.

Beller et al. (BELLER; GOUSIOS; ZAIDMAN, 2016) investigated the central role that testing has in CI. The authors found that testing is the most important reason as to why CI builds fail. Their observations suggest that the selected programming language has a strong influence on the number of (i) executed tests, (ii) the time to run the tests, and (iii) the test failure proneness. Finally, the authors advocate that testing in CI must not be a surrogate for running tests in the IDE itself. Similarly to Beller et al. (BELLER; GOUSIOS; ZAIDMAN, 2016), we study test metrics in CI context. However, instead of studying the number of tests per CI build, we analyze how test ratio and coverage evolve when adopting CI. In addition, we also compare CI and NOCI projects to better understand the impact of CI on the test ratio and coverage.

Labuschagne et al. (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) investigated the costs and benefits of automated regression testing in CI and identified how developers solved the failures that were encountered. The failures were then classified as (i) a flaky test, (ii) a bug in the system under test, or (iii) a broken or obsolete test. The authors found that 18% of the test suite executions have failed and 13% of the failures were flaky. A total of 74% of the non-flaky failures were caused by a bug in the system under test; the remaining 26% were due to incorrect or obsolete tests. Labuschagne et al. (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) mentioned that the test code grew faster than the production code in a CI environment. Differently from our work, their experiments stud-

ied the test code evolution in terms of medians only. In addition, Labuschagne et al. (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) did not empirically study whether the faster growth of test code and coverage may share a relationship with the adoption of CI. Our work complements Labuschagne et al. (LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017) work by empirically investigating the shared relationship between the adoption of CI and the evolution of test ratio and coverage.

Zhao et al. (ZHAO et al., 2017) studied open-source projects that have adopted Travis-CI. The authors found that the number of tests per build has likely increased after the adoption of Travis-CI. Complementary to the work of Zhao et al. (ZHAO et al., 2017), we study the impact of adopting CI on the test ratio and coverage metrics. We also perform comparisons between the time periods before and after the adoption of CI for our test metrics (as opposed to Zhao et al. (ZHAO et al., 2017)). Finally, we also analyze the test ratio and coverage growth in order to better understand if the test ratio and coverage have been increasing/decreasing since the period before the adoption of CI.

The aforementioned research has focused on better understanding the relationship between CI and tests. However, to the best of our knowledge, we are the first to investigate the evolution of tests by (i) comparing CI and NOCI projects; and (ii) comparing the period before the adoption of CI against the period after adoption the CI. We also complement the existing research by focusing on identifying the trends of test code ratio and test coverage evolutions in both NOCI and CI projects.

7 Conclusions

The potential benefits of adopting CI have brought the attention of researchers to study its advantages empirically. Previous research has studied the impact of adopting CI in diverse aspects of software development (BERNARDO; COSTA; KULESZA, 2018a; VASILESCU et al., 2015a; ZHAO et al., 2017; HILTON et al., 2017a; LABUSCHAGNE; INOZEMTSEVA; HOLMES, 2017). Vasilescu et al. (VASILESCU et al., 2015a) studied the quality and productivity outcomes with respect to CI on GitHub projects. The authors found that CI improves productivity without an observable diminishment in code quality. However, Vassallo et al. (VASSALLO et al., 2018) investigated a core principle behind CI, the Continuous Code Quality, and revealed a strong dichotomy between theory and practice, i.e., developers do not perform the continuous inspection. Instead, developers that use CI tend to control for quality only at the end of a sprint and, most of the times, only on the release branch. Such research work provides valuable insight into the impact of CI adoption in software quality outcomes.

Despite the valuable advancements, there are still many assumptions in the community that remains empirically unexplored. Most decisions are made in common sense that the results are positive, but there is still an unclear idea of the trade-offs and the real impact on the quality outcomes. Our work empirically investigates the software quality outcomes and their relationship with the adoption of CI. In the remainder of this chapter, we outline the contributions of this thesis and disclose promising venues for future work.

7.1 Contributions and Findings

This thesis aims at studing the relationship between continuous integration and its impacts on software qualities outcomes. We run two empirical studies that aim to answer the following open questions: (i) Does the adoption of CI share a relationship with the evolution of test code? (ii) The adherence to CI best practices is related to the degree of code quality? The results show that CI might have association with quality improvement, open space to investigate if these association might be caused by CI adoption. To answer this, we present a pioneering study that goes beyond the correlation tests to investigate the estimated causal effect of CI adoption and its impact on automated tests. Thereby, we apply a causal inference using directed acyclic graphs and probabilistic methods to determine the causal effect of CI in automated tests.

Below, we reiterate the main findings of this thesis:

- CI projects have more projects with a rising test ratio trend. We found that 33 out of 82 (40.2%) CI projects have a rising test ratio trend, while only 14 out of 82 (17%) NOCI projects have a rising test ratio trend – (Chapter 3).
- We observe that the adoption of CI is associated with a consistent increase of test ratio (MWW $p - value = 2.908e - 10$ and a small Cliff's $delta = -0.1612838$), while NOCI have a negligible change on the test ratio overtime (MWW $p - value = 0.0001842$ and a negligible Cliff's $delta = -0.09270482$) – (Chapter 3).
- Our results reveal that CI projects likely obtain a higher test ratio growth than NOCI projects. – (Chapter 3)
- Our analysis shows statistical evidences that most of the analyzed CI projects tend to increase or maintain the test coverage (9 out of 10 projects), while NOCI projects have a different tendency (5 out of 10 projects increase or

maintain the test coverage). In fact, NOCI projects have more projects decreasing the coverage (5 projects) when compared to CI projects (1 project) – (Chapter 3).

- Our mixed-effect models reveal that test ratio is largely explained by the project inherent context rather than by code or process factors – (Chapter 3).

- Although the project size expresses the most powerful explanation power on the technical debt, our study shows that maintaining a short build duration and high code coverage is also important to reduce the technical debt. – (Chapter 4).

- Technical debt is largely explained by the project size, outpowering the positive effects of short build durations and high code coverage – (Chapter 4).

- Our systematic mapping study shows that the relationship between CI and software factors is not always straightforward, and there is a lack of empirical researches to explain some aspects.

- Despite the CI adoption trade-offs, our causal modeling shows that it is likely to be associated with improvements in software quality. Additionally, it employs a considerable positive causal effect on the volume of automated tests (nearly 54% of probability).

Our main findings suggest that, although continuous integration can be empirically associated with some improvements in software quality (e.g., healthier test code evolution and a decrease of technical debts), this quality will unlikely be associated just by the adherence to certain CI practices. The quality outcome still depends on the effort to adhere to process good practices. Additionally, our causal analysis shows that CI affects software quality (expecifically on the automated tests volume) by multiple pathways, which are aligned with first resutls. Furthermore, this causal pathways lead us to conclude that CI indeed causes an improvement on automated tests.

Bibliography

AMRIT, C.; MEIJBERG, Y. Effectiveness of test driven development and continuous integration–a case study. *IT professional*, IEEE, 2017.

aO, K. V. R. P. et al. On the interplay between non-functional requirements and builds on continuous integration. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. [s.n.], 2017. (MSR '17), p. 479–482. ISBN 978-1-5386-1544-7. Disponível em: <https://doi.org/10.1109/MSR.2017.33>.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. *Oops, my tests broke the build: An analysis of travis ci builds with github*. [S.l.], 2016.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Oops, my tests broke the build: An explorative analysis of travis ci with github. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 356–367.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. [S.l.: s.n.], 2017. p. 447–450.

BERNARDO, J. H.; COSTA, D. A. da; KULESZA, U. Studying the impact of adopting continuous integration on the delivery time of pull requests. In: IEEE. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2018. p. 131–141.

BERNARDO, J. H.; COSTA, D. A. da; KULESZA, U. Studying the impact of adopting continuous integration on the delivery time of pull requests. In: IEEE. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2018. p. 131–141.

BERNDT, D. J.; CLIFFORD, J. Using dynamic time warping to find patterns in time series. In: SEATTLE, WA. *KDD workshop*. [S.l.], 1994. v. 10, n. 16, p. 359–370.

BROOKS, G. Team pace keeping build times down. In: IEEE. *Agile 2008 Conference*. [S.l.], 2008. p. 294–297.

BROOKS, G. Team pace keeping build times down. In: IEEE. *Agile 2008 Conference*. [S.l.], 2008. p. 294–297.

CHOW, T.; CAO, D.-B. A survey study of critical success factors in agile software projects. *Journal of systems and software*, Elsevier, v. 81, n. 6, p. 961–971, 2008.

CROWSTON, K.; ANNABI, H.; HOWISON, J. Defining open source software project success. *ICIS 2003 Proceedings*, p. 28, 2003.

DAMGHANI, B. M. et al. The misleading value of measured correlation. *Wilmott*, Wiley Online Library, v. 2012, n. 62, p. 64–73, 2012.

DEBBICHE, A.; DIENÉR, M.; SVENSSON, R. B. Challenges when adopting continuous integration: A case study. In: SPRINGER. *International Conference on Product-Focused Software Process Improvement*. [S.l.], 2014. p. 17–32.

DIGKAS, G. et al. How do developers fix issues and pay back technical debt in the apache ecosystem? In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.: s.n.], 2018. p. 153–163.

DOMINGOS, P. M. A few useful things to know about machine learning. *Commun. acm*, v. 55, n. 10, p. 78–87, 2012.

DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous integration: improving software quality and reducing risk*. [S.l.]: Pearson Education, 2007.

ELBAUM, S.; GABLE, D.; ROTHERMEL, G. The impact of software evolution on code coverage information. In: IEEE COMPUTER SOCIETY. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. [S.l.], 2001. p. 170.

EMBURY, S. M.; PAGE, C. Effect of continuous integration on build health in undergraduate team projects. In: SPRINGER. *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. [S.l.], 2018. p. 169–183.

EVANS, D.; LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE software*, IEEE, v. 19, n. 1, p. 42–51, 2002.

FELIDRÉ, W. et al. Continuous integration theater. *arXiv preprint arXiv:1907.01602*, 2019.

FERREIRA, C.; COHEN, J. Agile systems development and stakeholder satisfaction: a south african empirical study. In: *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*. [S.l.: s.n.], 2008. p. 48–55.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018.

FOWLER, M.; FOEMMEL, M. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, v. 122, p. 14, 2006.

FREITAS, G. *On the Continuous Code Quality Outcomes of Continuous Integration: An Empirical Study*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Natal, 2019. Unpublished.

GALLABA, K. et al. Noise and heterogeneity in historical build data: an empirical study of travis ci. In: ACM. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. [S.l.], 2018. p. 87–97.

GHALEB, T. A.; COSTA, D. A. D.; ZOU, Y. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, Springer, v. 24, n. 4, p. 2102–2139, 2019.

GHALEB, T. A.; COSTA, D. A. da; ZOU, Y. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, Springer, p. 1–38, 2019.

GRANO, G. et al. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2158, 2019.

GREENLAND, S.; PEARL, J. Causal diagrams. *International encyclopedia of statistical science*, Springer, p. 208–216, 2011.

GREN, L. The links between agile practices, interpersonal conflict, and perceived productivity. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. [S.l.: s.n.], 2017. p. 292–297.

GUPTA, Y. et al. The impact of the adoption of continuous integration on developer attraction and retention. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 491–494.

GUPTA, Y. et al. The impact of the adoption of continuous integration on developer attraction and retention. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 491–494.

HERRAIZ, I. et al. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, ACM, v. 46, n. 2, p. 28, 2013.

HILTON, M.; BELL, J.; MARINOV, D. A large-scale study of test coverage evolution. In: ACM. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. [S.l.], 2018. p. 53–63.

HILTON, M. et al. Trade-offs in continuous integration: assurance, security, and flexibility. In: ACM. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2017. p. 197–207.

HILTON, M. et al. Trade-offs in continuous integration: assurance, security, and flexibility. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. [S.l.: s.n.], 2017. p. 197–207.

HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2016. (ASE 2016), p. 426–437. ISBN 978-1-4503-3845-5.

HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: KHURSHID, S.; LO, D.; APEL, S. (Ed.). *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. [S.l.]: Association for Computing Machinery, Inc, 2016. (ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering), p. 426–437.

HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: IEEE. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2016. p. 426–437.

HUMPHREY, W. S. Characterizing the software process: a maturity framework. *IEEE software*, IEEE, v. 5, n. 2, p. 73–79, 1988.

JOHNSON, P.; EKSTEDT, M. The tarpit–a general theory of software engineering. *Information and Software Technology*, Elsevier, v. 70, p. 181–203, 2016.

JR, F. E. H. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. [S.l.]: Springer, 2015.

JULIAN, J. F. *Extending the linear model with R: Generalized linear, mixed effects and nonparametric regression models*. [S.l.]: CRC press, 2016.

KAYNAK, İ. K.; ÇILDEN, E.; AYDIN, S. Software quality improvement practices in continuous integration. In: SPRINGER. *European Conference on Software Process Improvement*. [S.l.], 2019. p. 507–517.

KNIGHT, J. C. Safety critical systems: challenges and directions. In: ACM. *Proceedings of the 24th international conference on software engineering*. [S.l.], 2002. p. 547–550.

KOCHHAR, P. S. et al. Moving from closed to open source: Observations from six transitioned projects to github. *IEEE Transactions on Software Engineering*, IEEE, 2019.

LABUSCHAGNE, A.; INOZEMTSEVA, L.; HOLMES, R. Measuring the cost of regression testing in practice: a study of java projects using continuous integration. In: ACM. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2017. p. 821–830.

LAI, S.-T.; LEU, F.-Y. Applying continuous integration for reducing web applications development risks. In: IEEE. *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. [S.l.], 2015. p. 386–391.

LAUKKANEN, E.; MÄNTYLÄ, M. Build waiting time in continuous integration–an initial interdisciplinary literature review. In: IEEE. *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*. [S.l.], 2015. p. 1–4.

LAUKKANEN, E. et al. Comparison of release engineering practices in a large mature company and a startup. *Empirical Software Engineering*, Springer, v. 23, n. 6, p. 3535–3577, 2018.

LEYS, C. et al. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, Elsevier, v. 49, n. 4, p. 764–766, 2013.

LUZ, W. P.; PINTO, G.; BONIFÁCIO, R. Building a collaborative culture: a grounded theory of well succeeded devops adoption in practice. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*. [S.l.: s.n.], 2018. p. 6:1–6:10.

MACBETH, G.; RAZUMIEJCZYK, E.; LEDESMA, R. D. Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, Pontificia Universidad Javeriana, v. 10, n. 2, p. 545–555, 2011.

MALAIYA, Y. K. et al. Software reliability growth with test coverage. *IEEE Transactions on Reliability*, IEEE, v. 51, n. 4, p. 420–426, 2002.

MARCILIO, D. et al. Are static analysis violations really fixed?: A closer look at realistic usage of sonarqube. In: *Proceedings of the 27th International Conference on Program Comprehension*. Piscataway, NJ, USA: IEEE Press, 2019. (ICPC '19), p. 209–219. Disponível em: <https://doi.org/10.1109/ICPC.2019.00040>.

MÅRTENSSON, T.; HAMMARSTRÖM, P.; BOSCH, J. Continuous integration is not about build systems. In: IEEE. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. [S.l.], 2017. p. 1–9.

MÅRTENSSON, T.; STÅHL, D.; BOSCH, J. Continuous integration impediments in large-scale industry projects. In: IEEE. *2017 IEEE International Conference on Software Architecture (ICSA)*. [S.l.], 2017. p. 169–178.

MÅRTENSSON, T.; STÅHL, D.; BOSCH, J. Test activities in the continuous integration and delivery pipeline. *Journal of Software: Evolution and Process*, Wiley Online Library, p. e2153, 2019.

MELLEGARD, N. et al. Contrasting big bang with continuous integration through defect reports. *IEEE Software*, IEEE, 2018.

MEYER, M. Continuous integration and its tools. *IEEE software*, IEEE, v. 31, n. 3, p. 14–16, 2014.

MILLIGAN, G. W.; COOPER, M. C. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, Springer, v. 50, n. 2, p. 159–179, 1985.

MORTON, J.; FRITH, U. Causal modeling: A structural approach to developmental psychopathology. John Wiley & Sons, 1995.

MYERS, L.; SIROIS, M. J. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences*, Wiley Online Library, v. 12, 2004.

PEARL, J. Causal diagrams for empirical research. *Biometrika*, Oxford University Press, v. 82, n. 4, p. 669–688, 1995.

PEARL, J. The seven tools of causal inference, with reflections on machine learning. *Commun. ACM*, v. 62, n. 3, p. 54–60, 2019.

PEARL, J.; GLYMOUR, M.; JEWELL, N. P. *Causal inference in statistics: A primer*. [S.l.]: John Wiley & Sons, 2016.

PERRY, D. E.; PORTER, A. A.; VOTTA, L. G. Empirical studies of software engineering: a roadmap. In: ACM. *Proceedings of the conference on The future of Software engineering*. [S.l.], 2000. p. 345–355.

PINTO, G. et al. Work practices and challenges in continuous integration: A survey with travis ci users. *Software: Practice and Experience*, Wiley Online Library, v. 48, n. 12, p. 2223–2236, 2018.

PINTO, G.; REBOUÇAS, M.; CASTOR, F. Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users. In: *10th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2017, Buenos Aires, Argentina, May 23, 2017*. [S.l.: s.n.], 2017. p. 74–77.

RAHMAN, A. et al. Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In: *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. [S.l.: s.n.], 2018. p. 8–14.

RAHMAN, F.; BIRD, C.; DEVANBU, P. Clones: What is that smell? *Empirical Software Engineering*, Springer, v. 17, n. 4-5, p. 503–530, 2012.

RAHMAN, M. M.; ROY, C. K. Impact of continuous integration on code reviews. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 499–502.

RAUSCH, T. et al. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 345–355.

REEL, J. S. Critical success factors in software projects. *IEEE software*, IEEE, v. 16, n. 3, p. 18–23, 1999.

ROMANO, J. et al. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In: *annual meeting of the Florida Association of Institutional Research*. [S.l.: s.n.], 2006. p. 1–33.

SALVADOR, S.; CHAN, P. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, IOS Press, v. 11, n. 5, p. 561–580, 2007.

SHACHTER, R. D. Probabilistic inference and influence diagrams. *Operations research*, INFORMS, v. 36, n. 4, p. 589–604, 1988.

SIZíLIO, G.; COSTA, D. A. da; KULESZA, U. An empirical study of the relationship between continuous integration and test code evolution. In: *Proceedings of the 35th International Conferenceon Software Maintenance and Evolution*. New York, NY, USA: IEEE, 2019. (ICSME '19).

SOMMERVILLE, I. Software engineering 9th edition. *ISBN-10137035152*, 2011.

SONARSOURCE. *SonarQube 7.9 Documentation*. 2019. `https://docs.sonarqube.org/latest/`. Accessed: 2019-08-21.

STÅHL, D.; BOSCH, J. Experienced benefits of continuous integration in industry software product development: A case study. In: *The 12th IASTED International Conference on Software Engineering*. [S.l.: s.n.], 2013. p. 736–743.

STARON, M. et al. Measurement and impact factors of speed of reviews and integration in continuous software engineering. *Foundations of Computing and Decision Sciences*, Sciendo, v. 43, n. 4, p. 281–303, 2018.

STOLBERG, S. Enabling agile testing through continuous integration. In: IEEE. *Agile Conference, 2009. AGILE'09.* [S.l.], 2009. p. 369–374.

SU, T. et al. Continuous integration for web-based software infrastructures: Lessons learned on the webinos project. In: SPRINGER. *Haifa Verification Conference*. [S.l.], 2013. p. 145–150.

SURYANARAYANA, G.; SAMARTHYAM, G.; SHARMA, T. *Refactoring for software design smells: managing technical debt*. [S.l.]: Morgan Kaufmann, 2014.

TEXTOR, J. Drawing and analyzing causal dags with dagitty. *arXiv preprint arXiv:1508.04633*, 2015.

TIBSHIRANI, R.; WALTHER, G.; HASTIE, T. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, Wiley Online Library, v. 63, n. 2, p. 411–423, 2001.

TIKKA, S.; KARVANEN, J. Identifying causal effects with the r package causaleffect. *arXiv preprint arXiv:1806.07161*, 2018.

TUFANO, M. et al. When and why your code starts to smell bad. In: IEEE PRESS. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. [S.l.], 2015. p. 403–414.

VANDEKERCKHOVE, J.; MATZKE, D.; WAGENMAKERS, E.-J. Model comparison and the principle. In: *The Oxford handbook of computational and mathematical psychology*. [S.l.]: Oxford Library of Psychology, 2015. v. 300.

VANDERWEELE, T. *Explanation in causal inference: methods for mediation and interaction*. [S.l.]: Oxford University Press, 2015.

VASILESCU, B. et al. Continuous integration in a social-coding world: Empirical evidence from github. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE Computer Society, 2014. (ICSME '14), p. 401–405. ISBN 978-1-4799-6146-7.

VASILESCU, B. et al. Quality and productivity outcomes relating to continuous integration in github. In: ACM. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2015. p. 805–816.

VASILESCU, B. et al. Quality and productivity outcomes relating to continuous integration in github. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. [S.l.: s.n.], 2015. p. 805–816.

VASSALLO, C. et al. Continuous code quality: are we (really) doing that? In: ACM. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. [S.l.], 2018. p. 790–795.

VASSALLO, C.; PALOMBA, F.; GALL, H. C. Continuous refactoring in ci: A preliminary study on the perceived advantages and barriers. In: IEEE. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2018. p. 564–568.

WIDDER, D. G. et al. A conceptual replication of continuous integration pain points in the context of travis ci. In: ACM. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.], 2019. p. 647–658.

WIDDER, D. G. et al. A conceptual replication of continuous integration pain points in the context of travis ci. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2019. p. 647–658.

WILKS, D. *Statistical Methods in the Atmospheric Sciences*. Elsevier Science, 2011. (Academic Press). ISBN 9780123850225. Disponível em: <https://books.google.com.br/books?id=IJuCVtQ0ySIC>.

WILSON, J. Agile testing: The power of continuous integration builds and agile development. 2009.

WINSHIP, C. *Counterfactuals and causal inference: Methods and principles for social research*. [S.l.]: Cambridge University Press, 2007.

WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012.

YU, Y. et al. Wait for it: Determinants of pull request evaluation latency on github. In: IEEE. *2015 IEEE/ACM 12th working conference on mining software repositories*. [S.l.], 2015. p. 367–371.

YU, Y. et al. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, Springer, v. 59, n. 8, p. 080104, 2016.

YUKSEL, H. M. et al. Using continuous integration and automated test techniques for a robust c4isr system. In: IEEE. *2009 24th International Symposium on Computer and Information Sciences*. [S.l.], 2009. p. 743–748.

ZAIDMAN, A. et al. Mining software repositories to study co-evolution of production & test code. In: IEEE. *2008 1st international conference on software testing, verification, and validation*. [S.l.], 2008. p. 220–229.

ZAIDMAN, A. et al. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, Springer, v. 16, n. 3, p. 325–364, 2011.

ZAMPETTI, F. et al. A study on the interplay between pull request review and continuous integration builds. In: IEEE. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. [S.l.], 2019. p. 38–48.

ZAMPETTI, F. et al. How open source projects use static code analysis tools in continuous integration pipelines. In: IEEE. *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2017. p. 334–344.

ZAYTSEV, Y. V.; MORRISON, A. Increasing quality and managing complexity in neuroinformatics software development with continuous integration. *Frontiers in neuroinformatics*, Frontiers, v. 6, p. 31, 2013.

ZHANG, Y. et al. One size does not fit all: an empirical study of containerized continuous deployment workflows. In: *Proceedings of the 2018 ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. [S.l.: s.n.], 2018. p. 295–306.

ZHAO, Y. et al. The impact of continuous integration on other software development practices: a large-scale empirical study. In: IEEE PRESS. *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2017. p. 60–71.

8 Appendix A

Table 17: Assumptions

Facor I	**Factor II**	**Factor II**
Automated Tests	CCQ	↗
Automated Tests	Developers Confidence	↗
Automated Tests	Issue Detection	↗
Automated Tests	Manual Effort	↘
Automated Tests	Team Productivity	↘
Big Travis Conf Size	Build Duration	↗
Build Duration	Code Quality	↘
Build Duration	Commit Frequency	↘
Build Duration	Development ShortCuts	↗
Build Duration	Distract Flow	↗
Build Duration	Feedback	↘
Build Duration	Integration Frequency	↘
Build Duration	Integration Queue	↗
Build Duration	Refactoring	↘
Build Duration	Team Motivation	↘
Build Duration	Team Productivity	↘
Build Stability	Commits with Negative Sentiment	↘
CCQ	Build Stability	↘

CCQ	Issue Detection	↗
CI	Automated Tests	↗
CI	Build Duration	↘
CI	Build Stability	↗
CI	Code Quality	↗
CI	Code Review Performance	↗
CI	Commit Frequency	↗
CI	Commit Size	↗
CI	Deployment Efficiency	↗
CI	Developers Confidence	↗
CI	Development Iterations Size	↘
CI	Difficulties of Build Failures Troubleshooting	↗
CI	Difficulties of Build Failures Troubleshooting	↘
CI	Documentation Quality	↗
CI	Integration Queue	↗
CI	Issue Detection	↗
CI	Issues Resolution	↗
CI	Magnetism	↗
CI	Magnetism	↘
CI	Manual Effort	↘
CI	Multiple Environmet Testing	↗
CI	Pull Requests Closing Rate	↗

CI	Pull Requests Delivery Time	↗
CI	Pull Requests Integration	↗
CI	Pull Requests Integration Time	↘
CI	Pull Requests Submissions	↗
CI	Refactoring	↗
CI	Releases Cycles	↘
CI	Software Monitoring	↗
CI	Software Quality	↗
CI	Software Stability	↗
CI	Stakeholder Satisfaction	↗
CI	Stickness	↘
CI	Team Communication	↗
CI	Team Pressure	↗
CI	Team Pressure	↘
CI	Team Productivity	↗
CI	Team Productivity	↘
CI	Team Work	↗
CI	Test Coverage Stability	↗
CI	Test Quality	↗
CI	Work Exposition	↗
Commit Frequency	Commit Size	-↗
Commit Frequency	Difficulties of Integration	-↗
Commit Size	Build Stability	↘
Commit Size	Developer Satisfaction	↘

Commit Size	Difficulties of Build Failures Troubleshooting	↗
Commit Size	Integration Frequency	-↗
Commit Size	Refactoring	↘
Commits Mentioning CI	Commits with Negative Sentiment	↘
Commits with Negative Sentiment	Build Stability	↘
Complex Changes	Build Stability	↘
Complex Setup of Tools	Build Stability	↘
Complex Setup of Tools	CCQ	↘
Complex Setup of Tools	Difficulties of Build Failures Troubleshooting	↗
Core Developers	Build Stability	↗
Developers Confidence	Integration Frequency	↗
Developers Popularity	Build Stability	↗
Difficulties of Integration	Integration Frequency	↘
Integration Frequency	Build Stability	↗
Integration Frequency	Code Quality	↗
Integration Frequency	Code Review Performance	↗
Integration Frequency	Pull Requests Integration	↗
Issue Detection	Software Stability	↗
Lack of Proper Tools	Build Stability	↘
LEAN	Feedback	↗
LEAN	Test Coverage Stability	↗
Local Testing	Build Stability	↗

Modeling CI Architecture	Complex Setup of Tools	↘
Project Age	Build Stability	↘
Project Age	Project Activity	↘
Project Age	Team Productivity	↘
Project Age	Team Size Stability	↗
Project Planning	Integration Frequency	↗
Project Size	Build Duration	↗
Properly Build Configuration	Build Duration	↘
Pull Request Size	Code Review Performance	-↗
Pull Requests Submissions	Build Stability	↘
Pull Requests with Successfull Builds	Pull Requests Integration	↗
Quick Interaction	Code Review Performance	↗
Scepticism	Integration Frequency	↘
Screening Submissions	Gating Performance	↗
Team Size	Build Duration	↗
Team Size	Build Stability	↘
Team Size	Commit Frequency	↗
Test Case Density	Build Duration	↗
Test Coverage Stability	Code Quality	↗
Time Pressure	CCQ	↘
Work Exposition	Developers Confidence	↘