Exploring the Socio-technical Impact of Continuous Integration: Tools, Practices, and Humans

by

Omar M. Elazhary

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Exploring the Socio-technical Impact of Continuous Integration:
Tools, Practices, and Humans

by

Omar M. Elazhary

Supervisory Committee

Dr. Margaret-Anne Storey, Supervisor
Department of Computer Science

Dr. Neil Ernst, Departmental Member
Department of Computer Science

Dr. Andy Zaidman, Outside Member
Department of Software and Computer Technology
Faculty of Electrical Engineering Mathematics and Computer Science
Delft University of Technology

ABSTRACT

Continuous software engineering is a rapidly growing discipline in software engineering. Among its many reported benefits is increased development velocity, faster feedback for developers, and better software quality. It also comes with its own share of challenges, most of which are centered on making automated builds more efficient or detecting problems with build configuration. However, the majority of literature in this area does not take into account software developers, which are arguably the cornerstone of software development.

Software development is still a human-driven endeavour. It is a developer who writes the code, tests it, makes the final decision while factoring in the build results, and so on. Furthermore, software development does not happen in a vacuum. Development takes place within the context of practices dictating how it should be done, and perceived benefits that drive practice adoption and implementation. Software development, and by extension continuous software development, is a socio-technical endeavour that features interactions between human aspects (developers, testers, etc.), technical aspects (automation), and environmental aspects (process, project-specific characteristics, infrastructure, etc.). While the software engineering field has its share of theories, frameworks, and models, or borrows them from other fields, we still do not have a human-centric framework for software engineering that takes into account other socio-technical aspects (technical and environmental).

My dissertation addresses this need for a socio-technical framework by illustrating a series of studies that ultimately resulted in the creation of a socio-technical theory of continuous software engineering that focuses on phenomena involving both humans and automation. In particular, I focus on the role of continuous software engineering tools (automation) in the software development process and how they displace existing tools, disrupt existing workflows, and feature in software developer decision making. This theory will enable further research in this area as well as allow researchers to make more grounded recommendations for industrial applications.

Contents

List of Figures

List of Tables

Part I

Introduction

1 Introduction

There's an old saying, "Fortune favors the bold." Well, I guess we're about to find out.

Cpt. Benjamin Sisko
Star Trek: Deep Space 9

Continuous integration is a software development paradigm that advocates integrating small, frequent changes to an application's codebase [9] in order to reduce the impact of integration issues and provide rapid feedback to developers so they can identify deviations if necessary. One key component practice continuous integration uses to achieve rapid feedback is the use of automation to run tests and other development tasks. However, continuous integration is still contingent on developers writing code to be integrated and making decisions on whether that code should be merged. Furthermore, while automation is what continuous integration is best known for, it also advocates various other practices (e.g., frequent small commits, testing, etc.). As such, continuous integration is a socio-technical endeavour that combines human interaction (human aspect) with automation (technical aspect) within the context of practices that are meant to support a project (environment). However, there has yet to be a study of the full socio-technical nature of continuous integration.

Continuous software engineering research has mostly focused on the technical aspects of this development process, and at times has included process aspects in the form of higher level system and process metrics. For instance, the use of automation is associated with increased software quality [10], more frequent releases [11], and higher development velocity [12]. However, aspects such as software quality, release frequency, and higher development velocity are often only considered in relation to automation (technical aspect). Soft-

ware development is a human-centric activity, and it does not occur in a vacuum. Developers are bound by the processes and practices they follow which are put in place to maximize some desired aspects of the development workflow. The practices and tools are also dependent on the project they are meant to support and the infrastructure limitations within which they must operate. By focusing mostly on automation, researchers and practitioners risk not capturing the entire picture of the development process, particularly *why* practices are implemented the way they are and *why* automation is configured the way it is. This dissertation addressed the need for a socio-technical exploration of continuous integration by conducting a series of studies that eventually led to building a theory of socio-technical phenomena in continuous integration.

1.1 Why Is Continuous Integration Important?

In 2000, Martin Fowler popularized the paradigm of continuous integration (CI) [13] as an offshoot of eXtreme Programming. The paradigm has since garnered a large following [14]; continuous integration is now one of the most popular development paradigms within the software engineering community. It is praised for its improvement of quality, flexibility (via the adoption of a continuous mindset), and rapid releases [10, 15]. The existence of version control systems like `git` and `svn` has facilitated the adoption of continuous integration as a development paradigm. GitHub[1] and other software repository service providers have contributed greatly as well.

Along with the paradigm's popularity, tools that support continuous integration also gained popularity. For instance, in 2011 Travis CI[2] rose to the top of a wave of automated tools and services that act in concert with repository service providers while requiring little effort in terms of configuration. For instance, once a commit is pushed to a Travis-enabled branch on GitHub, a build will execute whatever scripts it has been configured to run. Travis then provides a build log that details the operations performed and their output. Developers are then free to act on that output as they see fit based on their interpretation of the tool's output.

Travis CI is an example of a generic automated build tool that essentially provides a platform as a service to run various build scripts, automated

[1] `https://github.com/`

[2] `https://travis-ci.org/`

tests, packaging tools, and deployment configurations. There are other more targeted tools that focus on particular software engineering aspects, such as software quality (e.g., BetterCodeHub[3]) or code coverage (e.g., Codecov [4]). These tools usually feature more intricate interfaces that provide metrics regarding their area of focus separately from generic tools (e.g., Travis CI, Jenkins, etc.). As such, these tools can automate portions of the contribution (e.g., building, fetching dependencies, etc.), review (e.g., linting, testing, static code analysis, etc.), integration (e.g., merge strategies), release (e.g., packaging, cross-compilation, etc.), and deployment processes upon which the collaborative foundation of software development platforms (e.g., GitHub) are built. These tools form the *technical* aspect of the socio-technical nature of continuous integration and constitute the majority of what researchers have explored over the past decade. However, the remaining socio-technical aspects (humans and environment) have not been as thoroughly explored, neither individually or in connection to the technical aspects of automation. In comparison to automation-centric studies, there is little work on how automation facilitates collaboration, how it features in developer decision making, or how project characteristics and practices come into play.

1.2 Research Goal and Scope

Most literature has explored only the technical aspects of continuous integration (build efficiency, build outcome prediction, build configuration, ...etc.), which means the focus of the studies has tended to be on the automation used to support the development process. For instance, Vasilescu et al. [10] explored the positive effect continuous integration has on the open source development process on GitHub in terms of quality and productivity. To select projects that employed the continuous integration paradigm, they focused on those that use Travis CI as a form of automation. However, the non-technical aspects of continuous integration were not considered (project age, contributor team size, development workflow), which begs the question: do non-technical factors such as human interpretation, workflow, and project characteristics have an impact on the quality and productivity benefits derived from automation? There is work that indicates that non-technical fac-

[3] `https://github.com/marketplace/better-code-hub`
[4] `https://github.com/marketplace/codecov`

tors have an impact on both quality and productivity. For instance, Storey et al. [16, 17, 18] show that productivity as well as satisfaction can be impacted by several non-technical factors such as rewards, team culture, how impactful developers perceive their work to be, and many more. Tsay et al. [19] investigate how social factors impact the likelihood of pull request acceptance. Petre et al. [20] make a case for the socio-technical nature of software development by using a behavioral science lens to illustrate the wide array of approaches available to software engineering researchers.

It is developers who implement continuous practices, and developers who interact frequently with automation. While there has been some investigation of how this automation impacts developers [21, 22], it is not clear how much of an impact automation has on developer-centric phenomena (the social aspects of continuous integration). Questions like how does automation impact developer decision making, or how do development practices and automation inform developer behaviour and decision making are seldom discussed in the continuous software engineering research literature. When similar questions are discussed, contextual factors such as project characteristics or development practices are typically not included in the analysis. While there is a body of work in literature that addresses general automation and its relationship with humans, this work is lacking when it comes to the context of software engineering, particularly collaborative software engineering. There do exist some studies in the realm of automation that examine how much trust a human places in automation, but these occur in a generic context, and thus do not take into account all the possible factors specific to software developers.

As it now stands, research in the continuous integration area is largely oriented towards automation, without taking into account the socio-technical nature of the development process. Most of the literature focuses on the technical aspect (i.e., automation) and how it can be optimized, or enhanced by adding new functionality. While some research takes into account automation's impact on the environment (i.e., development practices) or humans (i.e., software developers), it only focuses on two aspects at a time (automation-human, or automation-environment) but rarely considers all three. Not capturing the socio-technical nature of continuous integration can limit the ability to reason about why automation-related phenomena occur and how they impact developers. To that end, I proposed viewing continuous integration through a socio-technical lens by addressing the following overarching goals:

- the study of workflow-centric continuous practice phenomena while considering the context of collaborative software engineering to map the relationship between the environmental and technical aspects of continuous software engineering,
- and the study of developer-centric phenomena while considering the continuous context within which they occur to map the relationship between the social and technical aspects of continuous software engineering.

The results of these studies will help in framing further research into socio-technical phenomena specific to continuous integration. In addition, these studies will provide a suitable guide for industry to conduct a viable evaluation of their status quo and determine the possible effects continuous practices and automation have on both their development workflow and their developers. As such, the high level goals of this dissertation can be formulated as the following research questions:

1. How does the presence of automation impact software development workflow?
2. How does automation feature in software developer decision-making?

Based on these previous research questions, the purpose of this dissertation is to formulate a theory that captures how automation interacts with both developers and the processes within which it is employed. In doing so, it will also shed some light on how the presence of such tools impacts the behaviour of software developers.

1.3 Dissertation Contributions

This dissertation makes the following contributions:

Empirical studies of human-centric phenomena in a continuous software engineering context

In this dissertation, I discuss the empirical studies we conducted to explore and observe socio-technical phenomena related to continuous practices. We performed two separate empirical studies (Chapters 4 and 6) that shed light on how automation impacts developer workflow and decision making in a continuous context. I also collaborated on an additional empirical study

(Chapter 5) that added to the insights we gathered on how automation impacts developer decision making.

A socio-technical theory of continuous software development

The empirical studies we conducted allowed us to build an explanatory theory of how **A**utomation, **D**ocumentation, **E**nvironment, **P**rocess, and **T**eam members interact with each other in a continuous context (Chapter 7). ADEPT can be used in an interpretive capacity to reason about *why* continuous software engineering phenomena occur while taking into account socio-technical factors. Furthermore, ADEPT can also be used in an exploratory capacity to *generate* hypotheses for further research.

A roadmap for socio-technical research in continuous software engineering

The literature mapping study we conducted allowed us to use ADEPT in an exploratory capacity to identify the less explored aspects of the socio-technical nature of continuous software engineering. These less explored aspects became the cornerstones of a socio-technical research roadmap for continuous software engineering, which will allow researchers to explore this field while taking into account the full socio-technical nature of the phenomena they observe.

1.4 Dissertation Outline

This dissertation consists of ten chapters. A high-level overview of the dissertation can be found in Figure 1.1.

1.4.1 Part I – General Introduction

In this part, I give an introduction into my topic of focus and establish a shared vocabulary. Chapter 1 explains the motivation behind this dissertation and clarifies that most literature does not consider the socio-technical aspects of continuous integration. It then establishes the overarching research goal and questions.

Chapter 2 explores the literature surrounding continuous integration. It discusses the history of continuous software engineering, along with the frequently researched areas within the topic, and maps the benefits and challenges to their individual continuous practices. Then, it categorizes the tools frequently used for task automation in continuous software engineering in commonly used automation hierarchies and establishes the case that human-centric automation phenomena can transfer to a software engineering domain. Finally, it discusses human-centric automation phenomena and maps them to phenomena already observed in a software engineering context.

Chapter 3 establishes the context and scope for this dissertation. In this chapter, I establish the vocabulary that will be used throughout the remainder of this dissertation. Afterwards, I describe the context within which I planned to conduct the empirical studies. Finally, I articulate how and why I planned on using mixed methods to answer the overarching research questions, as well as my biases.

1.4.2 Part II – Empirical Studies

In this part, I discuss the three empirical studies I conducted to answer the overarching research questions and goals. In Chapter 4, I discuss the study we conducted in the context of open source projects where we explored the role of continuous integration automation in project contribution workflow by investigating project contribution guidelines and development logs [7]. This study revealed that we cannot make assumptions about continuous practices in an open source context beyond the fact that they use automation.

In Chapter 5, I discuss the study I collaborated on with Colin Werner, Ze Shi Li, Derek Lowlind, Neil Ernst, and Daniela Damian which explored developers' interpretations of non-functional requirements in a continuous practice context [8]. We discovered that developer perceptions of non-functional requirements were impacted by the functionality the automation provided. The major takeaway here was that developers tended to delegate non-functional requirements to the tools they were using, and their perceptions were impacted as a result.

Finally, in Chapter 6, I discuss the study we conducted on individual continuous practices in a multiple case-study with three organizations [6]. Through interviews and development log mining, we examined the extent to which continuous development practices were being applied and why, while taking into account the socio-technical aspects of the development process.

As a result, we uncovered several human-centric phenomena, and our observations led to building the ADEPT theory.

1.4.3 Part III – Synthesis: ADEPT

In this part, I discuss how we built our explanatory theory that combined our findings from the previous three studies into ADEPT. I also explain how it was applied in a literature mapping study to generate a socio-technical research roadmap for continuous software engineering.

In Chapter 7, I discuss why we need a socio-technical theory of software developer interaction with automation. I then elaborate on how we constructed ADEPT in collaboration with Elise Paradis [23], its various constructs and propositions, and the extent of its validity. Finally, I establish the need to explore how well the theory can be used in interpreting existing phenomena as well as generate hypotheses for future research.

In Chapter 8, I discuss our yet-to-be-published study on the utility of the ADEPT theory. I explain how we conducted the study and how well ADEPT interprets the phenomena studied in the past five years of continuous practice literature. Furthermore, I illustrate two important findings: most literature tends to abstract development contexts away when focusing on the technical aspect of continuous practices, and that most literature focuses on technical, automation-centric studies. Finally, I chart possible research paths based on the aspects of ADEPT that were less explored in the literature to date.

1.4.4 Part IV – General Discussion

In this part, I synthesize all my findings so far to answer the overarching research goal and questions.

Chapter 9 discusses the implications of considering continuous software engineering from a socio-technical lens. I also interpret all our studies' findings through the ADEPT theory as well as establish a socio-technical roadmap of continuous software engineering research.

Chapter 10 summarizes the dissertation thus far, going over what was discussed in previous chapters and it lists possible future research paths based on the ADEPT theory, including human-centric hypothesis generation and testing, codebase integration into the ADEPT theory, and using ADEPT for different types of automation (chatbots, data processing pipelines, ...etc).

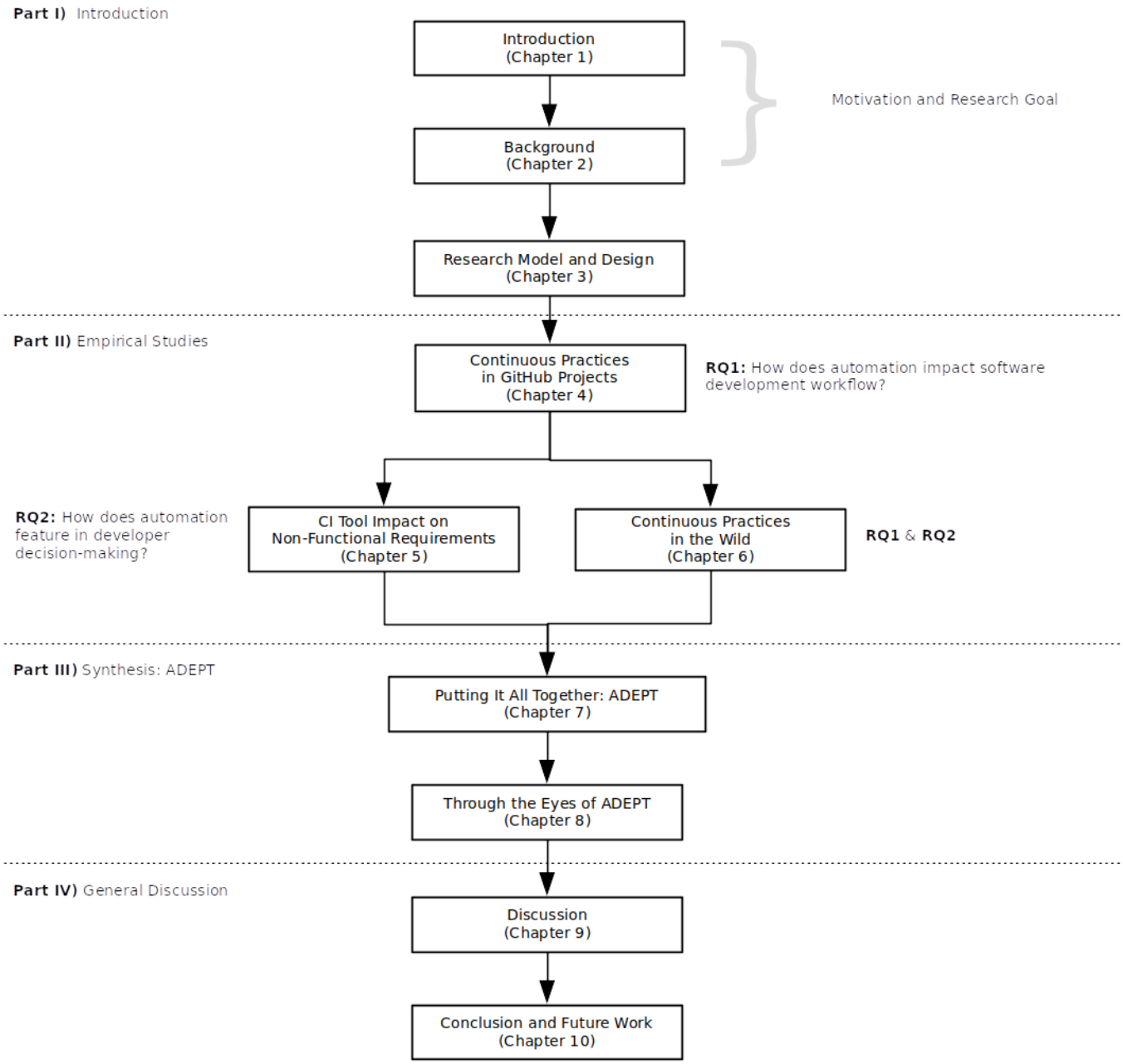

Figure 1.1: An outline of this dissertation.

2 Background

Insufficient facts always invite danger.

Cmdr. Spock
Star Trek: The Original Series

In this chapter, I provide background information on the domain of continuous software engineering and its current areas of research. I then further describe its supporting tools as forms of automation and whether those tools can have negative effects on both the collaborative nature of modern software development and the developers themselves.

2.1 The History of Continuous Software Engineering

The concept of continuous integration was proposed by Booch et al. [24] in 1990 as a way of emphasizing the need for continually integrating and testing new code. It was later incorporated into the eXtreme Programming (XP) paradigm by Kent Beck in 1999 [25], then separated into its own paradigm by Martin Fowler in a 2000 blog post [13]. In 2006, Fowler and Foemmel laid down what would essentially constitute the ten core practices of continuous integration [9]. Later, Humble and Farley would elaborate on this paradigm as they introduce the concept of continuous delivery [26]. Researchers would build their own definitions of what constituted continuous integration, delivery, or deployment later, and would evolve it into what Fitzgerald and Stol refer to as Continuous Software Engineering (CSE).

To best highlight the different ways this concept is defined, this section focuses on the umbrella term of continuous software engineering and

breaks it down to examine its different paradigms. I start by introducing the paradigms and the sub-practices that form them. Then, I focus on the tools themselves, the various ways they support the practices, and the body of research that has been built around them. Finally, I decompose the various benefits and challenges that have been assigned to the higher-level paradigms and assign them to their corresponding practices.

2.1.1 The Continuous Software Engineering Paradigm

In their work, Shahin et al. [4] refer to CSE as the frequent development and deployment of software. The purpose of these frequent releases is to elicit rapid feedback from customers which would minimize development divergence from client requirements. This overarching concept is also featured in Fitzgerald and Stol's work [1] where they imply that not only should the development activities be continuous, but also the business strategy and operations activities that occur both before and after development, respectively. They further decompose CSE into three separate phases: Business Strategy, Development, and Operations within which continuous activities occur, as can be seen in their continuous * (pronounced continuous-star) framework in Figure 2.1.

The cycle starts with a business strategy phase, where planning and budgeting occur. Continuous planning addresses how plans for the current cycle are formed and evolved from the previous cycle in order to adapt to changing business and software requirements (as discussed by Lehtola et al. [27]). Continuous budgeting refers to the use of the Beyond Budgeting model as discussed by Lohan [28], whereby the nature of the budgeting process becomes continuous, and easily responsive to emerging changes instead of only occurring once a year.

The next phase in the cycle is the development phase. This is where most of the technical activities take place: integration, deployment, delivery, verification, and so on. Continuous verification implies the use of activities such as formal methods and code inspections continuously throughout the development life cycle, as opposed to conducting them in a consolidated test phase after development. This was proposed as an attempt to formally verify embedded software by Cordeiro et al. [29]. Continuous testing involves automating test case execution in an effort to detect problems earlier in the development life cycle, as proposed by Saff and Ernst [30]. Continuous compliance refers to continuously evaluating software for regulatory compliance,

instead of doing that in its own phase prior to product release, as shown by Fitzgerald et al. [31]. Continuous security operates on the premise that security is no longer a non-functional requirement, rather, it becomes a critical piece of functionality that must be constantly tested and evaluated, as discussed by Merkow and Raghavan [32]. Continuous evolution refers to constantly evolving assumptions about software as well as architectures being used, to avoid the accumulation of technical debt, as proposed by Del Rosso [33]. I will discuss continuous integration, delivery, and deployment later in this section.

Finally, the operations phase represents the activities needed to ensure the smooth running of an application. The operations phase includes activities that focus more on how an application is operated by a development team and used by a customer: use, trust, and run-time monitoring. Continuous use involves more of a focus on user retention than on initial adoption as a strategy, as highlighted by Gebauer et al. [34]. Continuous trust refers to the trust built up by users as a consequence of dealing with the software/service provider, as discussed by Zhou in the case of mobile payment services [35]. Continuous run-time monitoring refers to keeping an eye on running software behavior in order to guarantee quality of service, as presented through the Kieker framework by Van Hoorn et al. [36].

There are also ongoing activities throughout the entire cycle: continuous improvement, innovation, and experimentation. Continuous improvement refers to the incremental optimization across all phases based on performance data gathered in previous cycles, as proposed through the Continuous-SPA prototype by Chen et al. [37]. Continuous innovation involves responding to rapidly and constantly fluctuating market conditions, as demonstrated by Olsson Holmström et al. [38]. Continuous experimentation is the development methodology of building an application, measuring its important features, and learning from the tests while involving stakeholders, as emphasized by Fagerholm et al. [39].

In previous work, Fitzgerald and Stol [14] also proposed bridging the business strategy and development phases into what they refer to as BizDev (also featured in continuous *) in order to increase the adaptability of the process in a similar manner to DevOps. The BizDev approach aims to mitigate the disconnect that exists between an organization's planning and development activities.

The remainder of this dissertation focuses on continuous integration, delivery, and deployment. I discuss these three sub-paradigms because they

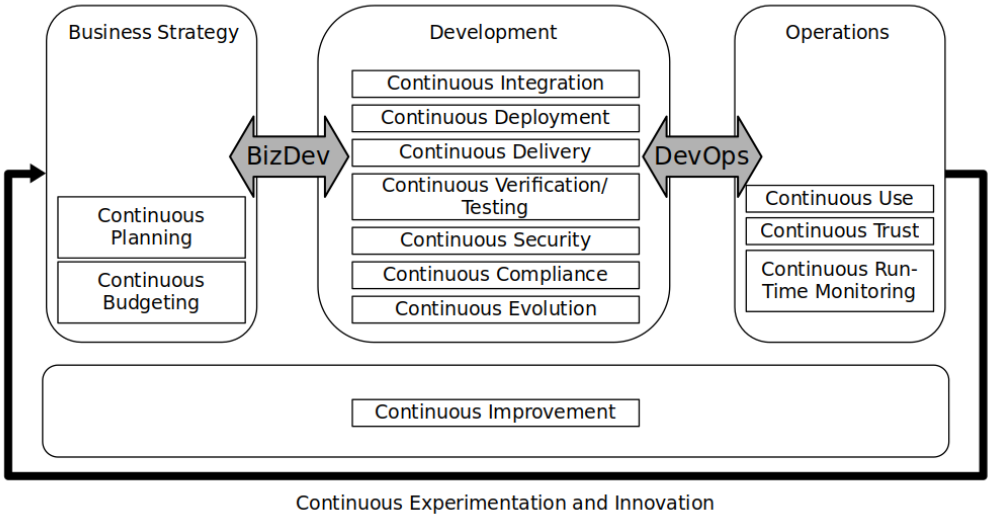

Figure 2.1: Continuous * as conceptualized by Fitzgerald and Stol [1]

generally revolve around or include automation in some form or another, which is the main focus of this dissertation. According to Figure 2.1, this places the focus squarely in the development section of the continuous cycle. Additionally, continuous integration is considered an integral component of a successful DevOps methodology and, along with continuous delivery, helps a team achieve continuous deployment, as noted by Lwakatare et al. [40].

Continuous Integration

Fowler [13] initially discussed continuous integration (CI) in 2000, where he emphasized the core building block of a successful CI implementation, the build. According to him, a contribution to a code repository should kick off a build of the latest version of that source code. This would include compiling/building the code, running tests on that code, and reporting on the results. He further highlighted that tests are also crucial to a build, and the more comprehensive they are, the more benefits that can be reaped from implementing CI.

CI as a practice, however, stems from the agile practice of *eXtreme Programming (XP)* according to Paulk [41] and Beck [25] as one of its 10 core

principles. In that article, Paulk emphasizes two aspects of CI that would guide the supporting tool infrastructure required by CI. These are:

- frequently integrating into and building the system (usually several times a day), and
- regression testing on a continuous basis to prevent functionality regressions.

Five years later, Fowler [9] re-popularized the term and discussed it in more detail. He also outlined a few key principles that CI should include:

- committing to and maintaining a single source repository,
- automating the build (usually by build scripts),
- executing tests as part of the build,
- committing to the main branch daily,
- every commit should result in a build of the main branch,
- optimizing builds for speed of execution (10 minutes is reasonable according to XP guidelines),
- testing the code in a production environment clone, and
- communicating results.

There are two more principles that pertain to continuous delivery and deployment, which I discuss later.

Fowler also highlighted the two major benefits of continuous integration, which are reduced risk and fewer bugs. Reduced risk refers to what the software development community has come to call *integration hell* as illustrated by Lindstrom and Jeffries [42]. This is the situation where integrating with the main branch has been deferred so long that it becomes painfully difficult to merge code back into the main branch, and when it does merge, everything breaks and no one knows what is going on. Applying CI reduces that risk by ensuring that integrations happen frequently, resulting in smaller deltas, and are thus relatively painless. Fewer bugs is a side-effect of the builds executing tests, and because builds are conducted frequently, the tests are also run frequently. This leads to a significantly smaller accumulation of integration bugs [13].

More recently, and due to the widespread variation in the way CI practices are adopted, Fowler and Humble [2] employ an informal test to distinguish whether or not a team implements a CI methodology during development. The first question they ask is whether every developer in the team commits at least daily to the main branch. Those that pass are asked whether every commit triggers an automated build and test. Finally, for those that remain, they are asked about when a build and test fails, whether it is repaired

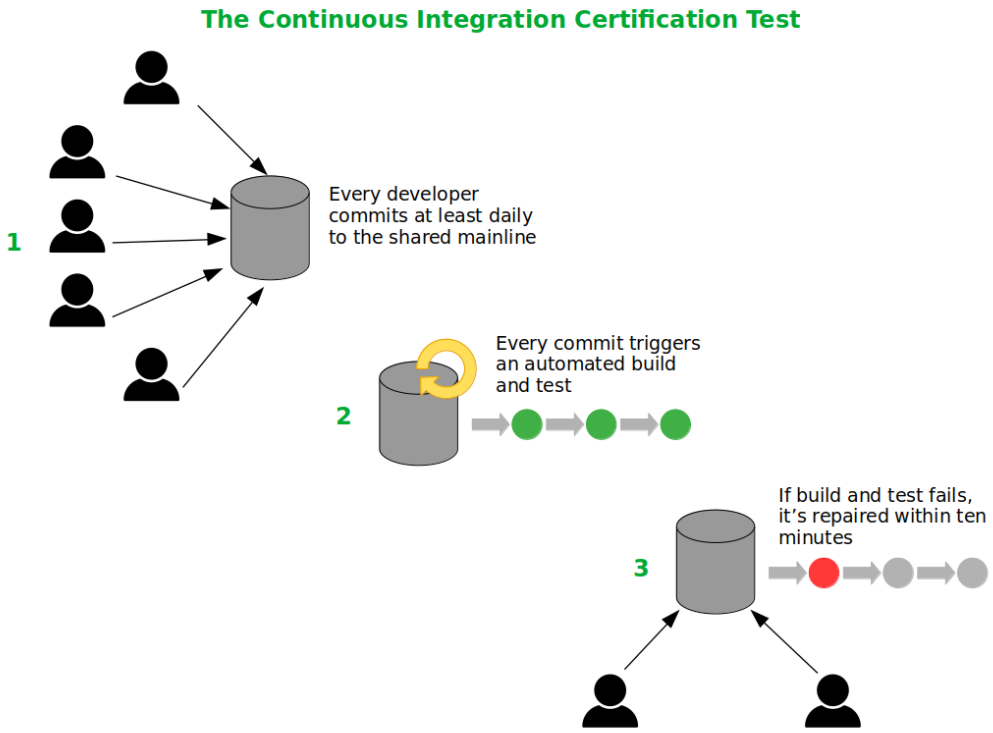

Figure 2.2: The CI certification test by Fowler and Humble [2]

within ten minutes. Those that pass all three questions are then considered to implement CI. An overview of the test is given in Figure 2.2. This further reinforces the fact that automation is a core practice of CI.

Continuous Delivery

The first of the practices mentioned earlier pertains to continuous delivery. Specifically, Fowler [9] states that it should be *easy for anyone (developer, tester, or client) to get the latest executable*. This practice makes it possible for other team members to run tests on the finished product, as well as demo it or examine any changes. This practice forms the core of what becomes continuous delivery (CDE). In 2011, Humble and Farley [26] published a book solely about continuous software engineering practices, mainly CDE. In it they discuss several principles of CDE:

- the release process must be repeatable and reliable,
- the release process must be mostly, if not completely, automated,
- everything should be kept in a version control system (VCS),
- doing difficult operations more frequently and as early as possible,
- fixing issues as early as possible,
- only when a feature is released is it *done*,
- the delivery process is everyone's responsibility, and
- the delivery process should be continuously improved.

Nowadays, CDE is largely defined by Laukkanen et al. [43] as being a software development practice whereby developers keep software in a good and releasable state. Most existing literature uses that definition when studying the CDE practice. However, it is often conflated with continuous deployment, and vice versa.

Continuous Deployment

The second of the continuous integration practices mentioned above is that of continuous deployment (CD): the automated deployment of applications to a production environment. Fowler also mentions the added benefit of applying CD at a relatively low cost and the need for automated rollback.

CD as a practice is best represented by the deployment pipeline as presented by Humble and Farley [26]. It is the culmination of automating the majority of a software project's processes, from CI, to CDE, to CD. This can be best described using Figure 2.3. A developer implements a functionality and commits their code to the VCS, which then triggers a build (as represented by the CI portion of the diagram). Within the build, the system provisions an environment and fetches dependencies. The application is then built from scratch, and existing test suites are executed. Once the integration step completes free of errors, a release is packaged for deployment to a customer's production system. This is where canary testing [44] or blue/green deployment [45] is done.

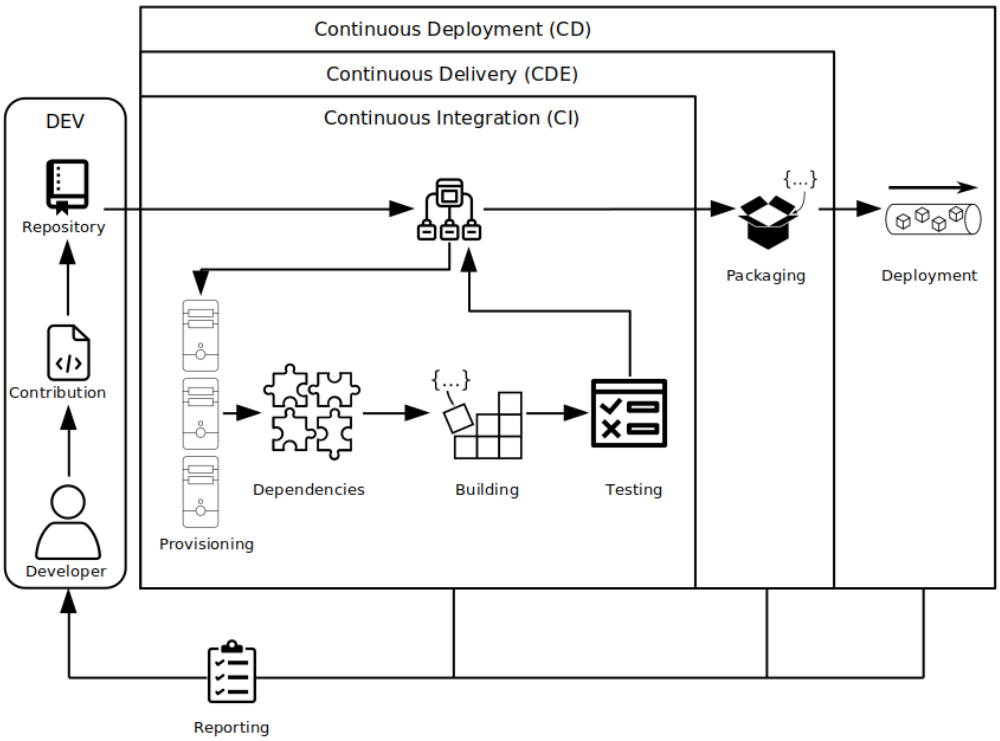

Figure 2.3: The structure of a completely automated pipeline

As a final step, a report would be compiled and sent back to the development team regarding the performance of the build and any possible errors that might have occurred. This report forms the basis for the continuous improvement principle of CSE.

The core premise of continuous software engineering is the feedback developers receive on a continual basis throughout the development process. Developers no longer operate in a vacuum; Their activities are driven by a product vision, and they receive feedback across all stages of the development workflow. Beller [3] proposed a feedback-driven development model that illustrates how ingrained feedback has become in the modern software development process. In the model depicted in Figure 2.4, they illustrate the types of feedback a developer can interact with, as well as the effects of particular practices and tools on developer behaviour. They examine both static and dynamic analysis, as well as testing habits, builds, and debugging.

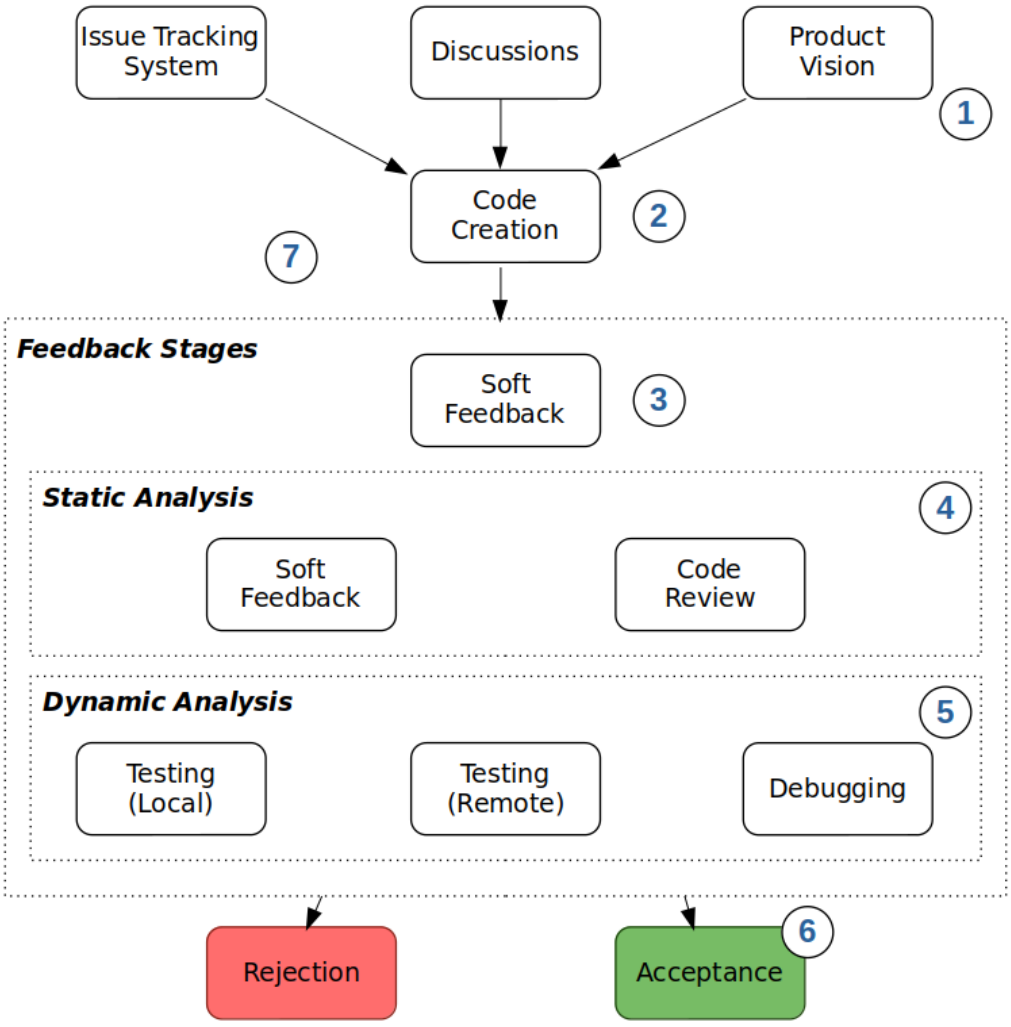

Figure 2.4: Feedback-driven Development by Beller [3]

2.1.2 The Tools

In order to support such an automation-heavy practice, tools and automated build service providers have begun to offer such functionality in concert with social coding platforms. These tools advertise themselves as "continuous integration" tools and to some extent draw from Fowler's principles [13] and his vision of what an automated build tool should be able to do. Generally speaking, there are two types of tools that support automation in a software development environment. The first is configurable or generic "CI" tools. The second can be referred to as more *targeted* CI tools. Both types of tools share in the provisioning and fetching of source code steps, but diverge afterwards in what they can do.

Configurable CI Tools

Configurable CI tools, or CCTs, are usually generic platform-as-a-service providers that offer high customizability in terms of what they can and cannot do. These tools essentially execute whatever scripts they have been configured to run by the developer. As such, a developer can configure them to run a wide array of build commands as well as test suites (to enable continuous testing). Examples of this type of tool include Travis CI[1], CircleCI[2], and Jenkins[3]. While there are other similar tools, these are by far the most used in both open source communities and industry projects hosted on GitHub.

Targeted CI Tools

Targeted CI tools, or TCTs, are more of a service provided by a third party that a developer can configure, but they only address one aspect of the CSE practices. As mentioned previously, the initial steps targeted CI tools follow include running when a developer commits code, then provisioning an environment, and finally fetching that code are similar to those followed by configurable CI tools. What comes next, however, is highly dependent on the tool itself. For instance, SonarCloud[4] is geared towards providing code quality analyses, and Codecov[5] focuses on checking test coverage. There is

[1] `https://travis-ci.org/`
[2] `https://circleci.com/`
[3] `https://jenkins.io/index.html`
[4] `https://sonarcloud.io/about`
[5] `https://codecov.io/`

a large ecosystem of such tools that only focus on a single aspect and work in tandem with software coding platforms (GitHub, GitLab, etc.). However, these targeted CI tools do not offer as much configuration options as CCTs do in the sense that they only focus on the aspect their vendors decided to include as opposed to CCTs that can be configured to accommodate a multitude of tasks a developer needs.

2.2 Mapping the Research Landscape

In 2015, Shahin et al. [4] explored the CSE literature in terms of the types and number of publications in an effort to identify benefits and challenges in the research area. Their plot in Figure 2.5 shows a steady increase in the number of studies in the area of CSE, reflecting the research community's interest in the topic.

Year	2004	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016
Workshop	0	0	0	0	0	0	0	1	3	0	3	0
Conference	2	1	2	3	3	1	3	5	3	12	9	4
Journal	0	0	0	0	0	0	0	2	1	0	10	1

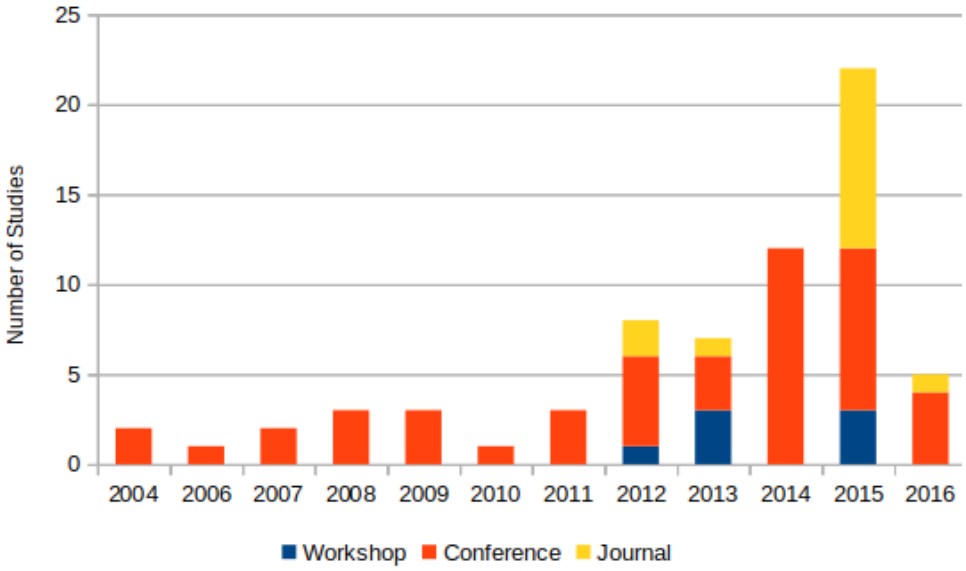

Figure 2.5: Types of continuous-related publications per year, by Shahin et al. [4]

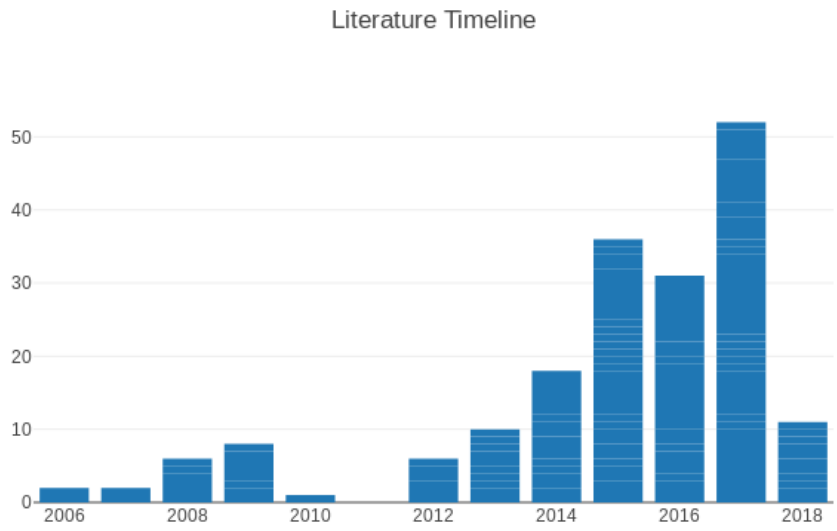

Figure 2.6: Results of my thematic classification of the literature

Following a similar process, I conducted a thematic classification of the literature that corresponds to the following keywords on DBLP[6]: `continuous integration`, `continuous delivery`, `continuous deployment`. The keywords I used were the ones used by Laukkanen et al. [43] with the exception of the "`software`" keyword as DBLP is a computer science-specific database. DBLP is a successful effort by the University of Trier[7] and Schloss Dagstuhl[8] to collect and provide access to *"open bibliographic information on major computer science journals and proceedings"*.

These results are represented in Figure 2.6. In 2017, the annual Mining Software Repositories (MSR) conference challenge featured a dataset compiled via the TravisCI API, called travistorrent[9]. This would account for the high number of publications regarding CSE (and CI in particular) that year [46]. For the purposes of providing a readable background, I have as-

[6] `https://dblp.uni-trier.de/`
[7] `https://www.uni-trier.de/?L=2`
[8] `https://www.dagstuhl.de/en`
[9] `https://travistorrent.testroots.org/`

signed these publications to the various sub-domains they addressed, as listed below:

- Publications tagged as "continuous integration"
  - CI Experiences
  - Build Mining, Monitoring, and Analysis
  - Requirements Traceability in CI
  - CI Systems Architecture
  - Applying CI in Other Domains
  - CI Information Consolidation
  - CI as a Software Development Practice
  - CI Tool Contribution
  - CI Build Optimization
  - Novel Uses in CI
- Publications tagged as "continuous delivery"
  - Modelling and Understanding CDE as a Practice
  - CDE Tool Contribution
  - CDE Tool Architecture
  - CDE Experiences
  - Adoption of CDE Practices
  - Testing in the Context of CDE
  - Guides to Practicing CDE
  - Knowledge and Resource Dissemination of CDE Artifacts
- Publications tagged as "continuous deployment"
  - CD Tool Contributions
  - CD Experiences
  - Adoption of CD Practices
  - Standards of CD Systems
  - Effects of CD Systems on Other Systems
  - The Practice of CD
  - CD in Non- Software Development Contexts
  - CD and System Security

The long list of subtopics shows how widespread the research areas in the three main CSE domains are. While there are some common trends like developer experiences with implementing a CSE system, some areas are wholly unique to their domain (e.g., CD and system security). However, the most commonly addressed topic in the literature is that of the automated continuous integration tool itself, with little or no publications addressing how it impacts developers or their workflow. For the remainder of this chapter,

however, I focus on topics that involve the relationships between developers and the tools they use, and topics that focus on the developer as a unit of analysis.

2.2.1 Developer Experiences with CSE Tools and Practices

Publications under this category group publications from the three subtopics: CI experiences, CDE experiences, and CD experiences. These publications vary based on the experiences of applying CSE practices/tools to their specific contexts but the one thing they all claim is that *using CSE techniques increases release frequency*. The increased release frequency makes sense, especially considering what Fowler [13] had initially proposed: the ease of integration brought about by adopting CSE practices leads to more frequent releases. However, what these publications mainly focus on is the effect of the CSE practice/tool adoption on system and process metrics (e.g., number of bugs filed, development velocity as counted by issues closed, etc.), as opposed to effects CSE practices might have on the developers themselves. They focus on automation-process relationships while not accounting for human and other environmental factors. The focus on the tool and associating it with system and process metrics is a missed opportunity because each of those publications focuses on the adoption or study of CSE practices, which means they have access to the developers involved in those practices as well and could focus more on the socio-technical aspect of these practices.

These publications, however, do bring valuable insights to the table. For instance, the work done by Meyer [47] highlights how CI is practiced in industry and proposes it as a solution to *Integration Hell*, a situation where developers work in isolation and then attempt to merge their work into the master branch but run into integration problems because their branch is out of date. The same purpose is addressed by Kim et al. [48], but the context is no longer generic software development, rather mobile applications. There are also several studies that promote the use of CI (and in some cases, CSE practices and tools) as a way to streamline the development process and make integration easier [49, 50, 15]. In fact, Chen [15] makes the following claims about CDE, which summarize most of this category's findings:

1. CDE increases efficiency.
2. CDE accelerates time to market.

3. CDE accelerates feedback from customers.
4. CDE improves productivity and reliability.
5. CDE reduces developer stress.
6. CDE is related to open issue reduction.
7. CDE is related to customer satisfaction.

Their findings are based on observations of CDE adoption at the Paddy Power organization. The majority of the claims shown above are geared towards organizational and process goals, and only claim 5 focuses on developers and human-centric phenomena.

A note on productivity

The majority of the studies conducted regarding open source software development, and subsequently those that focus on continuous software engineering practices and tools, have different ways of measuring productivity. The most common way of doing so is by measuring pull request merges similar to Zhao et al.'s work [12]. Another common way in industry is to measure issue closure [15]. There are also some studies that use lines of code (LoC and KLoC) to measure productivity [11]. These metrics reflect process-level metrics and/or do not take into account the creative nature of the software development process from the human perspective, making them unsuitable for a socio-technical investigation of CSE. Furthermore, these measures are too abstract and involve a large number of assumptions, generally making them unsuitable for comparing multiple projects in terms of performance. For example, comparing projects based on pull request merges assumes that each project's pull request review criteria are the same. Similarly, comparing projects based on issue closure rate assumes that each project's issues are defined at the same level and are equally complex (or simple). And using LoC does not adequately reflect the amount of effort a developer puts into a contribution because this might also involve creative activities and problem solving before writing code.

As a way of capturing the effort and creativity software development entails, Meyer et al. [51] use a developer's perceived productivity as a result of the argument that productivity is inherently something personal to a developer. Productivity in this case is a subjective construct that can reflect the various socio-technical factors involved in either increasing or decreasing it.

2.2.2 CI Information Consolidation

Information consolidation is a problem faced by most developers. CSE tools, especially the more generic ones, are notorious for providing walls of logs as their sole output.

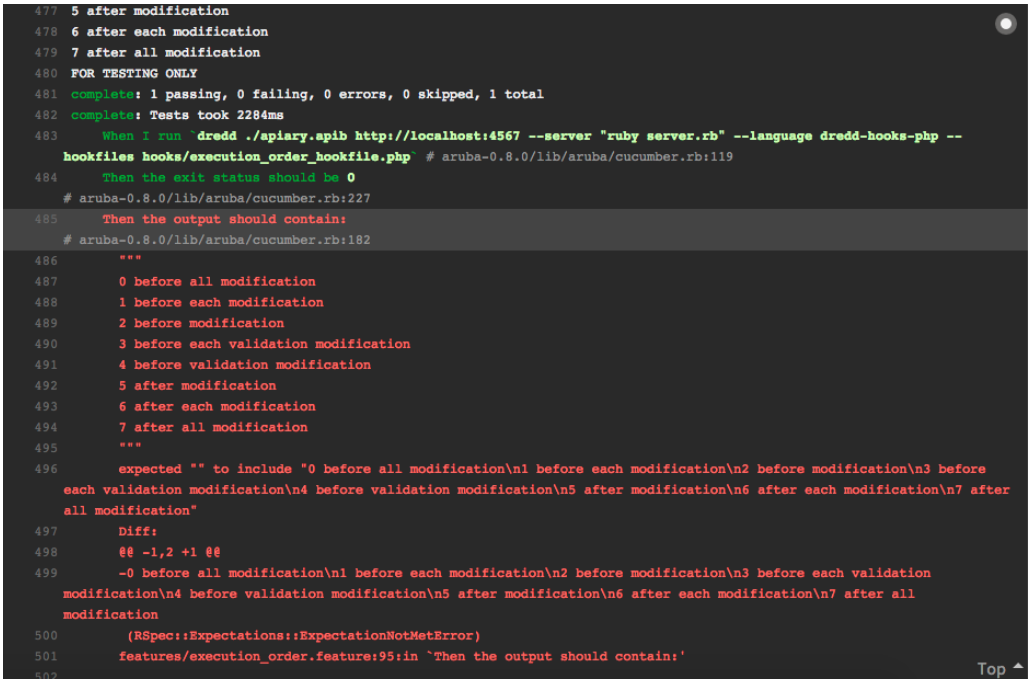

Figure 2.7: Travis CI example log output

Figure 2.7 depicts an example log output for a typical Travis build that runs some tests on an application. As shown, the logs can be extremely long (around 500 lines in the example) only to arrive at errors. This is mainly because generic build tool logs also include output messages for provisioning and dependency management, as well as any intermediate build steps. And this problem is only exacerbated by the use of several automated tools, as indicated by Brandtner et al. [52]. In that case, a developer is expected to gather information from all the various systems to infer enough information about the project's state. Naturally, the more automated build tools a project uses, the more information is generated. This places the developer in an environment where they can suffer information fragmentation, as defined by Storey et al. [53].

Brandtner et al. [54, 52] attempt to address this problem by creating what they refer to as profiles. These profiles represent different developer personas and create dynamic dashboards that consolidate CI tool output based on the persona requesting the information. They further evaluate their dynamic dashboard generation tool (which they refer to as SQA-Mashups) via a user study, which yields positive results. However, beyond Brandtner et al.'s work on CI information consolidation [55], there is very little literature on the usability of CI build logs.

2.3 Practice Benefits and Challenges

Researchers and practitioners attribute several benefits to CI as a paradigm, and while some of these align with what Fowler and Foemmel [9] expected as a result of applying particular practices, others are unexpected. However, it is worth noting that Fowler and Foemmel [9] (and later, Humble and Farley [26]) do not mention any possible adoption risks or post-adoption backlash regarding implementing continuous practices. Similarly, most systematic literature reviews on CI practices focus on several impediments to adoption [43] but not risks or backlash from using them. Practitioners, however, have indicated that backlash does indeed exist based on anecdotes, but still discuss CI as a high-level paradigm without delving into its individual practices [56, 57, 58].

In the coming sub-sections, I list the claimed benefits and challenges per continuous practice and how the literature has examined them (where relevant). I use Fowler and Foemmel's ten core CI practices [9] as a skeleton to map the various benefits and challenges. I also discuss negative phenomena per CI practice and augment them with Duvall's catalog of CI practice anti-patterns [59]. Similarly to Zampetti et al. [60], I use Duvall's anti-patterns to categorize incorrect applications of a practice in the sense that negative outcomes resulting from these anti-patterns could be considered risks or backlash. For the remainder of this section, benefits refer to positive effects attributed to particular practices, while challenges refer to impediments to practice application as well as negative effects resulting from practice application. The benefits and challenges per practice listed below are summarized in Table 2.1.

2.3.1 Maintain a single source repository

Having a single source repository (i.e., a repository that contains everything a developer needs to build the project locally) is the first among Fowler and Foemmel's ten core CI practices [9]. In theory, a developer would check out the repository on a machine with the bare minimum tools required and be able to build it with minimal effort. Fowler and Foemmel even state that a single source repository is more likely to reduce friction in identifying and accessing relevant sources and tools.

Benefits:

Jaspan et al. [61] discovered that a single source repository offers enhanced codebase visibility since all of the application code is in a central location. Jaspan et al. also found that developers suffer reduced cognitive load resulting from the aforementioned enhanced visibility. Developers who report a reduction in cognitive load say that it is due to the presence of the API within the same repository as the source code, which facilitates code understanding. This case notwithstanding, CI in the literature has mostly been investigated in an open source context [10], where projects tend to organize themselves in a microservice-like architecture and heavily use external dependencies [62]. On the other hand, industry studies that examine CI do not clarify or mention how the project codebase is structured, what scheme it follows, or whether this has an impact on their findings [11]. Therefore, it is unclear if and how a project's repository structure impacts how continuous practices are implemented, and whether it impacts the benefits derived from implementing continuous practices.

Challenges:

Single source repositories mean that all the application code is located in the same place. This also includes any dependencies the project may require. Thus, it comes as no surprise that this codebase structure results in lengthy build times [61, 63]. And while Zampetti et al. [60] do discuss issues related to codebase structure, these are typically mentioned in a design context as opposed to how it may affect build time or complexity.

2.3.2 Automate the build

Build automation is the second of the ten core CI practices Fowler and Foemmel mention [9]. It involves automating portions of the testing and review processes using an automated build server to run tests and checks in a sterile, reproducible, and visible environment. Fowler and Foemmel claim that build automation reduces mistakes that would normally occur from manual steps.

Benefits:

Zhao et al. [12] observe an increase in commit frequency correlated with the presence of a CI tool based on their investigation of commit and pull request rates. This, in turn, increases development velocity. Additionally, Hilton et al. [64] indicate that the developers they surveyed claimed that build automation gives rise to better quality software. The claim about fewer mistakes has not been investigated in literature nor, for that matter, whether automation gives rise to a completely new category of mistakes.

Challenges:

With respect to challenges, Laukkannen et al. [43] and Duvall [59] mention that a common risk–and sometimes impediment–to adopting and maximizing the benefits of CI is high build complexity. In Laukkannen's case, however, this was due to the build being customized so heavily that it included several special cases required by different teams working on the same project. This resulted in a higher build complexity that was more difficult to change and maintain later. While Laukkanen's case is specific to the organization the study was investigating, given the variety of tools serving different purposes (linting, testing, static analysis, etc.) that can be included in a build, high build complexity is not an impossible situation for most projects.

There are also no studies that investigate whether build automation introduces mistakes. For instance, developers may trust the tool resulting in what Parasuraman et al. [65] refer to as complacency, which means that developers may overlook hidden issues as long as the build result is successful.

Since build automation involves introducing a new highly generic tool into the development workflow, it is possible that mistakes related to its configuration may arise. Zampetti et al. study such anti-patterns or smells, but such mistakes are based on existing anti-pattern catalogues [60]. In the case of Gallaba et al. [66] they inductively build an anti-pattern catalogue

from formal and informal documentation. However, Zampetti et al. consider deviations according to Duvall [59] as anti-patterns, which does not take into account a singular project's context. Similarly, Gallaba et al. target Travis CI specifically as an automated build tool, which may emphasize tool-specific mistakes, as opposed to general automated build tool mistakes.

2.3.3 Make your build self-testing

The claim for this practice is that testing within the build catches bugs quickly and efficiently, which was corroborated by several studies [67] [64].

Benefits:

In the study conducted by Ståhl and Bosch [67], they found that their developers felt a sense of predictability with automated testing because it identifies problems faster. Similarly, Hilton et al. [64] report that their respondents found the automated tests that ran within a build helped catch bugs. However, Fowler and Foemmel [9] note that the benefit of catching bugs is primarily dependent on the tests used.

Challenges:

Challenges arise if there are no proper testing strategies in place or if the test quality is poor, as reported by Shahin et al. [4]. Tests need to be planned out, given adequate infrastructure resources, and effort needs to be invested into test labour. Most of the problems associated with testing can be considered impediments preventing the application of this practice. Furthermore, test quality, flaky tests [43], low test coverage, and long-running tests [59] are all risks resulting from halfheartedly applying this practice.

2.3.4 Everyone commits to the mainline every day

Fowler and Foemmel [9] claim this practice facilitates conflict detection and resolution in a speedy manner. This is largely because when every developer is expected to commit to the mainline (master branch) at least once daily, the overall divergence tends to be small enough to minimize integration issues. It is also a mechanism to enforce running tests frequently. Furthermore, they also claim that smaller commits help with tracking progress and providing developers with a sense of accomplishment.

Benefits:

Ståhl et al. [68] found that it is code properties (i.e., complexity) that impact whether developers commit more often—the more complex the area of code being changed, the more likely developers are to commit often. Additionally, Laukkannen et al. [43] found that *not integrating often* results in long-lived branches which, in turn, result in more integration issues.

Challenges:

Duvall [59] classifies the practice of long-lived branches as an impediment to this practice. Long-lived branches indicate developers work in isolation, and while they may commit daily to their separate branches, merging once their work is complete creates the issue of "integration hell" that continuous integration was meant to solve [9]. These integration issues, or merge conflicts, can block work [43, 4]. Work blockage can also be due to a slow approval process for commits.

2.3.5 Every commit should build the mainline on an integration machine

Fowler and Foemmel [9] claim that running automated builds whenever a commit is pushed is essentially similar to frequently running tests since the tests are contained within the build. This practice should help ensure that new changes do not break the existing codebase.

Benefits:

There is some work on how commit size and change complexity impact build success. However, no correlation is made between the level at which a build is run (e.g., the commit level, the pull request level, etc.) and build success. Islam et al. [69] only find that the more complex changes are more likely to result in broken builds. Builds that run at the commit level will be executed more frequently on smaller changes (depending on the commit size). Conversely, builds that run at the pull request level will be executed less frequently on larger changes (because one pull request can comprise many commits). Therefore, employing a finer build grain (i.e., executing on the commit level) may result in less complex builds, and thus fewer broken builds.

Challenges:

Limited infrastructure resources have a strong impact on what level a build is run at [4], and thus pose a challenge for teams who wish to run frequent builds but have limited resources. For instance, open source projects on GitHub that use the basic version of Travis CI have limited build resources (i.e., they can only run a specified number of builds concurrently). Therefore, GitHub projects tend to run builds at the pull request level as opposed to the commit level.

2.3.6 Keep the build fast

Fowler and Foemmel [13] claim this practice helps provide developers with *rapid feedback*. The tighter the feedback loop, the faster developers can respond to issues and adapt to customer requests.

Benefits:

Rapid feedback has been classified as a benefit [11] and finds its place nicely within the current trend of agile adaptable software development that emphasizes faster velocity, as indicated by Fitzgerald and Stol [1]. Furthermore, Hukkannen [70] purports that faster feedback cycles (typically by minimizing build durations) helps keep developers focused on the task at hand and minimizes context switching. Faster builds can also mitigate work blockages [43].

Challenges:

The literature does not explicitly investigate whether there are negative side effects to minimizing build duration. However, not minimizing build durations can have negative effects [60]. For instance, Laukkannen et al. [43] report that work blockage is a risk that can result from lengthy and complex builds.

2.3.7 Test in a clone of the production environment

With this practice, Fowler and Foemmel [9] advocate making the typical development environment as close to the application's intended production

environment as possible. This allows developers to spot any production-related problems before the application is actually deployed to a client's production environment.

Benefits:

Environment clones have not been extensively explored in the context of continuous integration. Outside of the CI context, there has been work on digital twins and their use in industry 4.0 [71, 72]. If one were to consider a software application in its production environment a real-world artifact, then that application in its development environment would be tantamount to a digital twin, with the same concepts, benefits, and challenges applying here as well.

Challenges:

Conversely, waiting until a project's release milestone to test it in a production environment is the antithesis of this practice [59]. While there are no reports of impediments, risks, or backlash, issues related to testing strategy and flaky tests can potentially impact the implementation of this practice. Flaky tests make it harder to reliably test an application, and introduce a confounding variable when testing across different environments.

2.3.8 Make it easy for anyone to get the latest executable

Part of the agile nature of continuous practices is to involve end-users in the development cycle as much as possible, and this practice aims to do just that [9]. It allows the product's users to give feedback before it is deployed to their systems. It also facilitates the testing process internally within the development team by giving everyone access to the latest version of the product.

Benefits:

Ståhl and Bosch [67] report that allowing easy access to an executable (or a running version of the application) facilitates agile testing, which relies on

client testing and evaluation (mostly a non-technical individual). An argument can also be made that this facilitates constantly keeping the application in a releasable state, which Rossie et al. [11] have shown significantly shortens the development cycle.

Challenges:

A manual build process makes it harder for individuals to obtain an executable, and thus poses a challenge to this practice [59]. Another challenge is build complexity where the more complicated the build process is, regardless of whether it involves manual steps, the harder it becomes for developers to implement this practice since part of a developer's workflow is dedicated to building the application so they can test it locally [43].

2.3.9 Ensure that system state and changes are visible

Fowler and Foemmel [9] advocate keeping all developers and other team members in the loop by making information about the system, its changes, and its state visible to everyone involved. This is meant to facilitate communication between team members, even if they are not co-located.

Benefits:

Ståhl and Bosch [67] indicate that using build system interfaces or dashboards helps this process achieve its claimed benefits to an extent. However, they do not elaborate on what information developers used to ensure the system and its changes are visible, as the accounts they received diverged more often than not with respect to what the dashboards and interfaces contained.

Challenges:

Conversely, unsent, ignored, or otherwise uninformative notifications are indications that this practice is not being fully applied [59, 43]. There are also situations where only a portion of the team is privy to such information and notifications, which runs counter to this practice [60]. Ignored or uninformative notifications lower visibility into a project's development workflow.

2.3.10 Automate deployment

In a similar vein to practice 2 (automate the build), Fowler and Foemmel [9] claim this practice reduces errors introduced by manual steps and speeds up the deployment process.

Benefits:

This practice has been shown to lead to faster release cycles, as demonstrated by Hilton et al. [64] when they compared projects that use an automated build tool that is configured to deploy software against those that do not. They found that the projects that automate their deployments release about twice as often as those that do not, which Rossi et al. [11] show allows for more frequent customer feedback.

Challenges:

Dependency centralization, the use of a common scripting language, externalizing configuration variables that differentiate between environment-specific details, and using the same deployment script but allowing for variable configurations constitute risks to this practice being successfully applied [59]. These risks generally pertain to deployment build complexity and script bloat, which contribute to overall build complexity and have been shown to have a negative impact on build success [43]. And the less tolerable downtime is to a customer, the more important this risk becomes.

2.3.11 Practice Summary

To sum up, there is a significant body of research that has been conducted on continuous integration. This work is geared primarily towards two aspects: either applying a higher level CI practice and identifying issues/antipatterns/smells with respect to that practice, or the optimization and detection of possible issues with the underlying automation tool, similar to the work done by Gallaba and McIntosh [66]. Thus, I have attempted to associate the benefits discussed in the literature with their corresponding practices as defined in Fowler's original article, upon which most modern CI principles are built. I have also done the same regarding possible impediments to applying the practices and the risks resulting from partially applying them. A summary is given in Table 2.1. On the practice level, however, it remains

Table 2.1: Literature mapping of practices to claimed benefits and possible challenges from Elazhary et al. [6].

Practice	Claimed Benefits	Possible Challenges
Maintain a single source repository	• Facilitates project building [9] • Enhanced codebase visibility [61] • Centralized dependency management [61]	• Slower build speed due to dependency building [61]
Automate the build	• Reduces mistakes resulting from manual steps [9] • Higher commit rate [12] • Higher pull request processing rate [12] • Higher software quality [10]	• Build complexity increases [43, 59]
Make the build self-testing	• Faster and efficient bug capture [9, 67, 64]	• Inadequate testing strategies [4] • Ambiguous test results [43] • Flaky tests [43] • Manual testing reduces utility from CI [59]
Everyone commits to the mainline every day	• Quick conflict detection and resolution [9, 68]	• Large commits and long-lived branches cause merge conflicts [43, 59]
Every commit should build the mainline on an integration machine	• Ensures new changes do not break existing functionality or introduce new bugs [9]	• Work blockage may occur due to merge conflicts [43, 59]
Keep the build fast	• Provides rapid feedback [9] • Reduces effects of context switching [70]	• Work blockage as a result of lengthy and complex builds [43]
Test in a clone of the production environment	• Recognize problems in production early before deployment [9]	• Creating production clone too close to a release date [59]
Make it easy for anyone to get the latest executable	• Gives users access to feedback before deployment [9] • Facilitates agile testing [67]	• Build complexity impacts developer ability to build locally [43] • Developers have to perform manual build steps [59]
Ensure that system state and changes are visible	• Facilitates communication between project team members [9, 67]	• Not sending or ignoring build notifications [59] • Build information and notifications are not available to the entire team [59] • Lack of discipline when monitoring build status [43]
Automate deployment	• Reduces errors resulting from manual steps [9] • Speeds up the deployment process [9] • Faster release cycle [64]	• Deployment complexity affects its success [43, 59]

unclear whether applying all CI practices is necessary to derive all the benefits attributed to the methodology since most of the literature focuses on the automated tools that support CI practices. While Duvall [59] and later Zampetti et al. [60] both attempted to create and augment a catalog of *bad practices* with respect to CI, they do not elaborate on why these fixes are not made or whether the remaining practices impact the feasibility of these fixes. It also remains unclear if non-adherence to CI guidelines/practices

has a negative impact on developer workflow as evidenced by either risks or backlash.

2.4 CSE Tools as Automated Systems

In this section, I focus on both generic (CCTs) and targeted (TCTs) CSE tools. I review the core functionality of CSE tools, establishing that they are indeed highly automated systems and thus share their properties. Then I place them in the most common automation system hierarchy based on their functionality to establish that phenomena that occur around generic automation systems can also transfer to CSE tools.

2.4.1 CSE Tool Functionality

From Section 2.1.2 as well as Fowler's [9] outline of what such a tool should be able to do, a tool that supports frequent integration, delivery, and deployment in a software context should be able to perform the following tasks:

1. Fetch the current software version from a version control system.
2. Build, compile, and run said software.
3. Enforce quality standards by applying tests as well as other types of analyses to the code.
4. Bundle the code in an easy-to-execute package and make it available to those who need it.
5. Install and run the latest version of the application in a production environment.
6. Report on build status and any problems it might have encountered throughout the process.

Naturally, these steps differ based on whether the tool provides a specific targeted function (e.g., static code analysis, linting, etc.) or is generic (i.e., can be configured by developers to suit their needs). If the tool is targeted, it is very likely that the build will terminate after step 3. If the tool is generic but only used for integration purposes, it will skip steps 4 and 5. If the tool is generic but used for integration and delivery, it will skip step 5. If the tool is generic and meant for deployment, it will execute all 6 steps. This generic CSE tool workflow is further reinforced by Humble and Farley's work [26].

Now that I have established the basic functionality a CSE tool provides, the next section will demonstrate how that functionality can fit into the

most common automation taxonomies that classify automation based on its involvement in the human decision making process.

2.4.2 CSE Tools in Automated System Hierarchies

Sheridan and Verplanck built a classification scheme that assigned levels of automation control based on the extent of the automation's involvement in the human decision process [73]. Save and Feuerberg [74] build on Sheridan and Verplanck's work, as well as Parasurman et al. [75] to produce a more recent version of the different levels of automation and their involvement in human decision making. The Level of Automation Taxonomy (LOAT) classifies automation based on four criteria: information acquisition, information analysis, decision and action selection, and action implementation.

- **Information Acquisition:** Refers to the degree an automation is configured to retrieve and acquire information about specific tasks. This criterion consists of six levels, with A0 indicating that information acquisition is a fully manual process, and A5 indicating that it is fully automated.
- **Information Analysis:** Refers to the degree an automation is capable of combining, comparing, and analyzing information about the task it is configured to run. This criterion also consists of six levels, with B0 indicating that the analysis is fully conducted by a human, and B5 indicating that it is fully automated.
- **Decision Selection:** Refers to the degree an automation is capable of generating suggestions, choices, or courses of action and recommending them to the human. This criterion consists of seven levels, with C0 indicating that choice generation is completely done by the human, and C6 indicating that the automation selects the most appropriate course of action without informing the human.
- **Action Implementation:** Refers to the degree to which an automation is capable of implementing pre-configured actions with or without the involvement of the human. This criterion consists of nine levels, with D0 indicating that the human is in charge of manually executing tasks, and D8 indicating that the automation is in charge of task execution without human interruption.

An automation system can be classified on a discrete scale per activity depending on how much of its functionality is geared towards that particular activity type. However, since these aspects are heavily dependent on a tool's

configuration, classifying tools according to that model will yield fairly similar results across most generic tools, with specialized tools such as SonarQube or Codecov having different rankings due to the analysis they perform.

Based on their functionality and their degree of agency, CSE tools can be assigned the following ratings depending on how they are configured.

- **Information Acquisition (A5 - Full automation support of information acquisition):** CSE tools, particularly generic automated build systems, can be configured to fetch source code, data from external sources, external dependencies, and many other artifacts in a completely automated manner. For example, an empty Travis CI build fetches the project repository and actively attempts to build the project based on the language and tools used to implement it, which entails fetching dependencies.
- **Information Analysis (B4 - High-level automation support of information analysis/B5 - Full automation support of information analysis):** CSE tools can perform a variety of analyses. Jenkins and similar tools (B4) can be configured to run any type of static or dynamic analysis to provide in-depth insights about an application, whereas more specific tools such as SonarQube or BetterCodeHub (B5) use expert knowledge to indicate to the developer the various quality issues their code could suffer from.
- **Decision Selection (C2 - Automated decision support):** Excluding automated program repair, CSE tools are generally not capable of implementing their own recommendations since they act on a separate version of the codebase. They generate possible fixes for the developer to choose from, and the developer is free to implement a recommended solution or generate one by themselves.
- **Action Implementation (D2 - Step-by-step action support):** CSE tools automate some tasks, but have yet to fully automate a review or development activity. They are capable of building, testing, and deploying applications, but a human is typically required throughout the various steps to make decisions as to whether the results are satisfactory.

Now that CSE tools have been classified within a human-centric automation hierarchy, I can safely make the following assumptions:

1. Automation, by its nature, interacts with humans to varying degrees (based on the above classification scheme), and as such, fits within my interpretation of the socio-technical nature of continuous practices.

2. CSE tools are automation, which means that human-centric phenomena that impact automation systems may transfer to a continuous software engineering context.

In the next section, I discuss the various human-centric phenomena that occur in relation to automation and illustrate how they can apply to CSE tools as well.

2.5 Automation and Humans

The previous section established CSE tools as automation. This section explores the effects automation has had on its human operators in the past as a starting point for understanding human-automation phenomena that could impact software developers. It also examines the newer effects that have surfaced in the context of software development, particularly those focused around CSE tools.

2.5.1 The Perils of Automation

When Parasuraman et al. [76] studied how humans and automation interact, they concluded that there are three different phenomena that can affect human operators of automated systems. These are:

- the abuse of automation,
- the disuse of automation, and
- the misuse of automation.

These three phenomena generally stem from the human operator's relationship with the automated system and how much trust they have in it.

Abuse of automation

Abuse of automation refers to automation for the sake of automation. This generally manifests itself by automating processes that would normally yield more efficient results were they done by the human operator.

Disuse of automation

Disuse of automation refers to the operator no longer depending on the automation system, usually due to mistrust. The operator would then perform all automated processes as if the automated system no longer existed, thereby

decreasing process efficiency and wasting the investment made in automation in general.

Misuse of automation

Misuse of automation refers to the operator taking the automated system's report at face value and trusting it completely. This is typically unrealistic since automated systems are only as good as their internal mechanisms and can be error prone. This can lead to operators either wasting time and resources following wrong directives given by the system, or simply causing more damage than good.

Misuse was later addressed in by Parasuraman et al. [65] in the form of complacency, which refers to operators growing complacent and overly relying on system output.

2.5.2 Perils Related to CSE Tools

Although there is a large abundance of literature when it comes to CSE (as evidenced by this chapter), there is fairly little literature concerning the interplay between automated software processes and software developers in the CSE context. This is particularly distressing because what little work that has been done in this area is showing some alarming results. I will attempt to use the categories identified by Parasuraman et al. [76] (as discussed above) to categorize these studies, however, some phenomena are specific to the software engineering domain and do not fit under any of the three categories.

Abuse of Automation

Pinto et al. [77] note that developers suffer an increase in cognitive load in projects that use a CI tool. This manifests itself as a lack of focus. Their respondents interpreted this lack of focus as a result of having to worry about the build system's code as well as their projects' source code. Their respondents also brought up the issue of building across multiple operating systems and how this increases debugging difficulty. Abuse refers to the overuse of automation to an extent where it becomes burdensome. In this case, developers are struggling with build script and configuration maintenance, most likely because it is an additional task they have to perform. Furthermore, it

is also possible that building across multiple operating systems has increased build complexity, which in turn affects maintenance.

Disuse of Automation

Gupta et al. [22] observed that projects that use Travis CI experience a slight drop in developer attraction and retention. They measure attraction as the ratio between new developers and existing developers (with respect to a project) in a particular timeframe. Similarly, they measure retention as the ratio between the number of contributors who made a contribution previous to a particular time window versus all contributors who made a contribution within that time window. They note a slight decrease in those measures after the adoption of Travis CI. They do not, however, indicate or speculate as to why this happens.

Misuse of Automation

Zolfagharina et al. [78] discuss the phenomenon of *build inflation*, which is when builds are run on multiple platforms. So many, in fact, that it shifts the developer's perception of the build from whether it indicates quality to whether it indicates portability. As a result, developers may spend time addressing issues that may not be of high priority or that may introduce technical debt. Misuse refers to placing too much trust in the automation such that its results are assumed to be correct. In this case, developers interpreted portability related errors as indicators of application quality, causing them to conflate both concepts. In this case, trusting the automation to deliver the right result resulted in developers focusing on lower priority issues. Similarly, Pinto et al. [77] note that believing blindly in test results has a negative effect on developer performance. The issue was raised by their survey respondents.

Uncategorizable Phenomena

The work done by Souza and Silva [21] shows that CSE tools can affect team morale, and consequently productivity. They illustrate that broken commits result in broken builds, which increase the team's overall negative sentiment. This negative sentiment is then more likely to produce future broken commits, which result in broken builds, and so on.

Additionally, there is tool-centric work that attempts to investigate issues with existing CSE tools. The work done by Widder et al. [79] attempts to test whether pain points reported in the literature apply to Travis CI by surveying developers. They compiled a rather extensive list of phenomena that pertain to various aspects of Travis CI, including usability. For instance, their respondents indicated difficulties with build configuration as a result of having to use a YaML configuration file. However, the context within which Travis CI is used is not considered, which indicates that the exploration focused mainly on the human and technical aspects of the automation.

Throughout this chapter, I have introduced the concepts of continuous software engineering with its most common practices. I illustrated how these practices overlap and how they all incorporate feedback throughout their workflow. Then I discussed the various benefits and challenges of the different continuous practices and how these claims have been handled by the literature. I explored how CSE tools fit into the generic automation literature and classified them according to the most recent version of a human-centric automation hierarchy to establish my assumptions that automation must be studied with the human in the loop. Finally, I discussed the possible risks CSE tools as a form of automation may bring software developers. The next chapter outlines this dissertation's scope based on the concepts discussed in this chapter and sets the overall research goals.

3 Concepts and Research Model

Instruments register only through things they're designed to register. Space still contains infinite unknowns.

Cmdr. Spock
Star Trek: The Original Series

In this chapter, I articulate the main concepts that I use throughout the remainder of this dissertation. I present the definitions I use and the context within which I operate. Finally, I showcase the overarching methodology used throughout this dissertation.

3.1 Concepts and Scope

As I discussed in Chapter 2, there are several ways of defining and interpreting CSE practices. CI in particular is generally used to indicate the presence of an automation in conjunction with the code repository, be it a Continuous Configurable Tool (CCT) or a Targeted Continuous Tool (TCT). Furthermore, there are several ways of contributing to a codebase that uses a collaborative software development platform. Because collaborative software development is largely dependent on team and developer preferences and the practices they already have in place, it becomes very hard to make sweeping generalizations. The same can be said of automated build tools, particularly the generic type of tools or CCTs. These are tools that are purposefully made as generic as possible in order to appeal to a large audience. As such, it is necessary to define what falls within this dissertation's scope and what does not. To that end, I am defining the terms and constructs that I will be

using for the remainder of this dissertation.

3.1.1 Pull Request-Driven Software Development

There are several ways to contribute to a repository. For instance, depending on whether review occurs before or after a change is merged in, a team could be following the commit-then-review workflow or the review-then-commit workflow [80]. However, irrespective of when the review takes place, most software development teams that use a collaborative software development platform to host their code and maintain versioning use the pull request model [81].

The pull request model, as popularized by software coding platforms like GitHub and BitBucket, operates under the premise that changes are to be developed separate from the codebase until such a time that they are complete and one reviewer (or more) approves the merge. Only then are the changes integrated into the codebase to become part of the overall product. In 2013, Gousios et al. [81] estimated 14% of the repositories on GitHub used pull requests as a software development workflow. Nowadays, it is the default workflow according to GitHub's Octoverse reports in both industry and open source projects that use a collaborative software development platform as evidenced by the metrics used [82]. Furthermore, thanks to its integration into platforms such as GitHub, BitBucket, and GitLab, the pull request model offers an easily navigable structure when it comes to investigating software development workflows. Therefore, I will be considering software development as conducted via the pull request model throughout the remainder of this dissertation.

3.1.2 Automation

Automation is a very generic concept. It can be something as small as an automated test suite that runs locally on a developer's workstation, or something as sophisticated as a deployment pipeline meant to install new changes directly to a customer's system. Then there is the distinction I made earlier in Chapter 2 between configurable automation (CCTs) and targeted automation (TCTs). However, because of the sheer variety of TCTs and their specific use cases, it is more generalizable to focus on CCTs instead. Therefore, for the remainder of this dissertation, I will focus on configurable automation.

Configurable automation in the context of the pull request model refers to the *automated build* referenced by Fowler [13, 9]. In essence, it is a mechanism for developers to automate running certain tasks that reduce their workload, and in some cases, ensures that some tasks are not skipped. In that sense, and as was described earlier in Figure 2.3, this would include the integration, delivery, and deployment phases. Configurable automation can then build software, test it, produce a packaged version, and potentially deploy it to a customer's system. Additionally, this form of automation can send notifications back to developers, either via email, or through communications tools such as Slack or Teams. Finally, due to its configurability, configurable automation is often referred to as an *automated pipeline*, which I will use for the remainder of this dissertation interchangeably with automation.

3.1.3 Continuous Practices

Continuous practices can theoretically span any practice featured in Fitzgerald and Stol's Continuous* framework [14]. However, the pull request model scopes this considerably to the development phase of that framework. To narrow it down further, I will be referring to Fowler and Foemmel's articulation of the core ten CI practices as continuous practices [9]. These practices form the basis upon which modern continuous practices are built, along with their extensions (delivery and deployment):

1. Maintain a single source repository.
2. Automate the build.
3. Make your build self-testing.
4. Everyone commits to the mainline every day.
5. Every commit should build the mainline on an integration machine.
6. Keep the build fast.
7. Test in a clone of the production environment.
8. Make it easy for anyone to get the latest executable.
9. Everyone can see what is happening.
10. Automate deployment.

This confines the scope of this dissertation to these ten practices, how automation is used in congruence with them, and how these practices and automation function within the pull request model. They also form the basis for the context within which we conduct our investigation.

3.1.4 Developer Decision Making

There are two perspectives relevant to developer decision making that fall within the scope of this dissertation:

- tool interpretation,
- and tool configuration.

Tool interpretation takes a more human-computer interaction perspective of how developers interpret and understand an automated pipeline's result, and how that result influences decisions they make. For instance, one part of the pull request review process is checking whether or not a pipeline build was successful. This check informs the decision a reviewer makes when contemplating merging a code change. Factors that impact such a decision can include how readable its output is, how ambiguous the results are, and how much trust a developer can place in that tool which impacts how thoroughly they examine the tool's results.

Tool configuration relates to how a developer configures an automated tool to perform specific functions. For instance, a developer may choose to include unit tests in the pipeline, but not integration or usability tests. They may choose to fully automate deployment to external systems, or to make that process manual. This perspective captures the rationale behind why a pipeline is configured they way it is, and what trade-offs a developer had to make when configuring it.

3.2 Research Context

There are two contexts in which I explore these practices and tools: public projects within a collaborative software development environment (i.e., one that uses a collaborative platform such as GitHub or GitLab), and industry teams that also operate using some form of the pull request model (usually they would be using or implementing a version control system within the context of their organization). These types of projects were selected for two reasons:

1. The pull request model's workflow allows us to make assumptions about how developers behave on its platform, since projects on GitHub, BitBucket, or GitLab use that workflow. Confirming that sampled projects actually use the pull request workflow is an additional inclusion criterion that will be included in the individual studies.

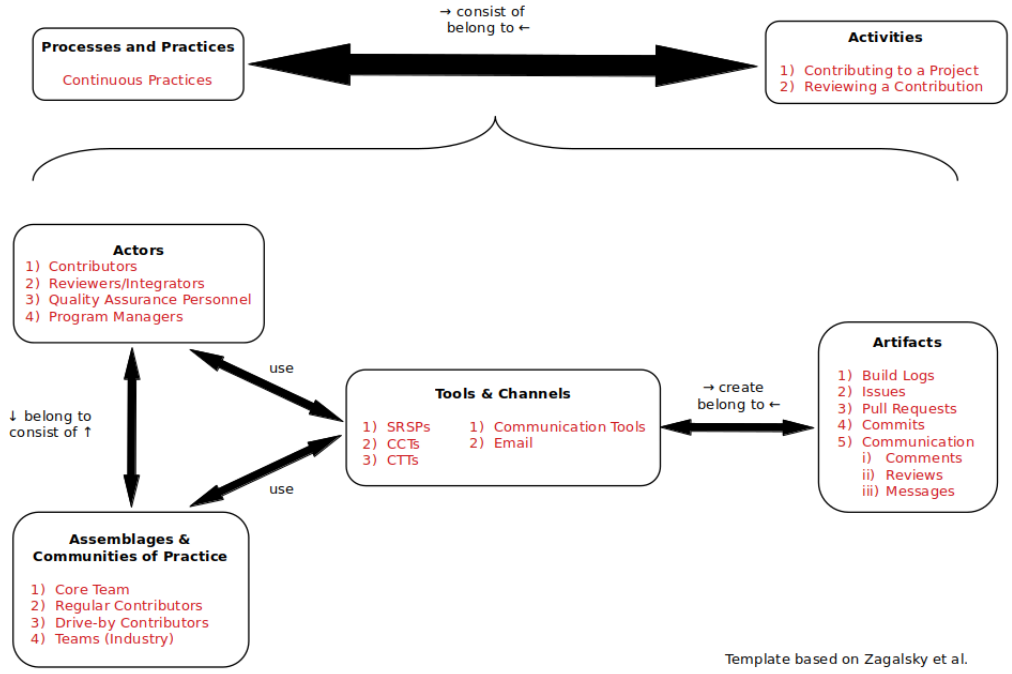

Figure 3.1: Research Context

2. Their data is easily obtainable via the software coding platform's APIs, be they public or private (provided we have consent to do so). Data availability allows us to triangulate using different data sources.

The project selection criteria allowed me to establish the rudimentary context surrounding these projects.

Combining configurable automation, collaborative software development platforms that operate using the pull request model, and the different types of team members who are likely to affect and be affected by that automation yields a more concrete version of this context. Continuous practices constitute a unique paradigm by which developers contribute to a project that involves automation, among other things. The paradigm is unique in the sense that it advocates for automation, but does not *require* it [9]. On the other side of the contribution equation is the reviewer who also interacts with the results such automation provides [16]. The overall research context can be found in Figure 3.1.

3.3 High-Level Methodology

Throughout this section, I explain the process I used to answer the research questions proposed in Chapter 1. I discuss my personal biases, how they impacted my choice of research design, and elaborate on what steps were necessary to better understand automation's place in the software development ecosystem.

3.3.1 Epistemological Bias

As discussed earlier, there is a multitude of elements from which data can be collected and inferences made. However, it was necessary to reflect inward on my own biases so as to minimize their impact on the methods I chose.

Referred to as a "worldview" by Creswell [83], and sometimes either a paradigm or epistemology, is the inherent bias a researcher carries with them when embarking on a research project. It influences, to some extent, the methods they choose, the way they interpret their research goals, and the validity of their results [84]. And while it is not always made explicit, I feel it is necessary that I do so here to better justify my methodological choices.

My epistemological bias is that of a pragmatist. According to Creswell, a pragmatist is as follows:

> *"Another position about worldviews comes from the pragmatists. [...] There are many forms of this philosophy, but for many, pragmatism as a worldview arises out of actions, situations, and consequences rather than antecedent conditions (as in postpositivism). There is a concern with applications-what works-and solutions to problems (Patton. 1990). Instead of focusing on methods, researchers emphasize the research problem and use all approaches available to understand the problem (see Rossman & Wilson, 1985). As a philosophical underpinning for mixed methods studies, Tashakkori and Teddlie (1998), Morgan (2007), and Patton (1990) convey its importance for focusing attention on the research problem in social science research and then using pluralistic approaches to derive knowledge about the problem."* [83, p. 10]

Kaushik and Walsh [85] define a pragmatist as a believer in an objective reality that is heavily grounded in the environment and cannot be attained

except via human experience. In that sense, reality is what works at the time. A pragmatist would then let the research goal determine the most appropriate method of achieving it, all the while not forgetting that the context within which the analysis is conducted is what lends it validity. That context-derived validity, however, may not transfer once the context is changed.

3.3.2 Research Goal

Continuous practices advocate automation, and when investigated in the literature they are typically reduced to the second and tenth practices: automate the build, and automate deployment. However, there is more to continuous practices than just automation: Automation is advocated in only two of the 10 proposed continuous practices, namely build and deployment automation [9].

Another approach taken in the literature is to focus on a continuous paradigm such as CI or CD from a high-level perspective, similar to Vasilescu et al. [10] and Hilton et al. [64]. This approach tends to assume that the projects they examine employ the CI paradigm because of their use of automation. As such, the remaining practices tend to go overlooked.

It is only recently that researchers have started looking at specific continuous practices, like Ståhl et al. [68] with commit sizes and frequencies, or Zampetti et al. [60] and continuous integration anti-patterns. Similarly, more human-centric research has emerged in this area as well, such as Widder et al. [79] and their investigation of possible human-related issues with TravisCI, a popular continuous integration build automation tool. However, in the quest for generalizability, the context within which these practices are applied tends to be abstracted away.

The main objective of this dissertation is to better understand the role automation plays in the software development ecosystem. So far, the literature has explored automation while seldom taking into account its impact on developers. And when it has been studied in conjunction with developers, it rarely takes into account the surrounding factors, such as process, project context, and other possibly confounding factors opting for generalizability at the cost of overlooking how all these constructs interact in their individual contexts. To that end, I propose a theory that attempts to combine these constructs together in such a way as to illustrate how they interact with each other and why it is necessary to keep the context in mind when investigating human-automation phenomena in software development.

3.3.3 Overall Approach

Based on the context and goal described previously, we found it necessary to revisit and more crisply define the questions mentioned in Chapter 1. Our initial research goals were defined as follows:

1. How does the presence of automation impact the software development workflow?
2. How does automation feature in software developer decision-making?

Throughout this chapter, I have narrowed down what each of these concepts and constructs are. This dissertation's more specific research questions are now:

1. How does the presence of an automated pipeline impact the software development process projects use in a continuous context?
2. How does the presence of an automated pipeline impact developer decision-making in a continuous context?

These questions, along with my worldview, informed my choice of research methods, and eventually the limitations my results are subject to.

This dissertation involved both quantitative and qualitative methods, resulting in a mixed methods design. By mixed methods, I refer to the definition synthesized by Johnson et al. [86] (later referred to by Creswell [83]) which states that such a research design incorporates elements of both qualitative and quantitative domains to better study the depth and breadth of the problem in question. I chose a mixed methods approach because the continuous practices phenomenon with all its factors and intricacies has not been sufficiently explored on the developer level. This would involve qualitative probing. However, it cannot be solely examined from a purely qualitative perspective since it also needs to take into account the quantitative factors within which the developers operate as a result of using the pull request model [12, 87, 88].

While a mixed methods design helped address the phenomenon from multiple perspectives, ultimately leading to a comprehensive view of the usage of CI, there were still associated risks. For instance, Terrell [89] outlines how mixed methods research can be a time consuming task when giving equal priority to both the quantitative and qualitative aspects of the design. However, in this case, the focus was more on the qualitative side as I relied on the quantitative insights already explored in literature, and analyzed quantitative data purely to triangulate and understand the context around the qualitative data I gathered. Another issue he raises is the difficulty of

data comparison (both quantitative and qualitative) as well as reconciling any discrepancies that may arise. Once again, since the main focus was on the qualitative aspect, the quantitative data was modally qualitized [90] for further qualitative analysis by identifying and verbally encoding the most frequent attributes.

Regarding triangulation, it occurred on two different levels: the data gathering and analysis levels. Greene and McClintok [91] recommend using separate analysis techniques and checking whether the findings converge. And on the data gathering level, Yin [92] proposes using multiple sources of data in order to improve result robustness.

The studies we conducted (mentioned later in Chapters 4, 5, and 6) followed a convergent design as discussed by Creswell [93]. I collected data from open source projects hosted on GitHub, which amounted to both quantitative and qualitative data. Both types of data were analyzed separately, then merged together and further analyzed. As recommended by Driscoll et al. [94], a database with unique identifiers per data point was maintained for each data set (qualitative and quantitative), and the quantitative data was modally qualitized.

In addition to the data mining approach we used for our study in Chapter 4, we opted to use case studies in Chapters 5 and 6. According to Yin [92], a case study is an empirical inquiry that explores a phenomenon in its realistic context, particularly if it is hard to distinguish where the phenomenon ends, and the context begins. Case studies are especially useful when the number of variables exceeds that of data points, and when there are multiple data sources that can be used for triangulation. Case studies were particularly well suited here because of the amount of context they manage to capture, as well as the in-depth investigation that was necessary to examine how developers interact with these systems. They were also not too granular in detail so as not to lose sight of the original objective, which is to *explore* and *understand* how automation interacts with the developer and the process within the confines of the project it is meant to serve.

In Part II, I discuss how we used these research strategies to conduct our empirical studies. In Chapter 4, I discuss how we used both process mining and contribution guideline mapping to explore how automated pipelines were being used in GitHub project workflows. In Chapter 5, I discuss how we used a multiple case study to explore how developers handled non-functional requirements, and how dealing with these requirements illustrated aspects of how developers made configuration-related decisions. In Chapter 6, I discuss

how we used another multiple case study to conduct an in-depth exploration of continuous practices and how these results illustrated the different factors that impacted the differences between continuous practices across different contexts.

Part II

Empirical Studies

4 Continuous Practices in GitHub Projects

The computer is notorious for not volunteering information.

Cmdr. Geordi La Forge
Star Trek: The Next Generation

I recount the study I conducted to investigate the impact of CI tools on project development workflow in collaboration with Margaret-Anne Storey, Neil Ernst, and Andy Zaidman. What started out as an exploratory study quickly grew into a bit of a cautionary tale about what assumptions I could and could not make regarding open source software projects hosted on platforms such as GitHub. The results of this study were accepted and present at ICSME 2019 [7].

4.1 Motivation

In Chapter 3, I established the crisper, more precise version of the overall research questions of this dissertation. This study [7] attempted to answer the first overarching research question:

> How does the presence of an automated pipeline impact the software development process projects use in a continuous context?

To investigate the impact of an automated pipeline on project development workflow, we followed a similar approach to Prana et al. [95]. We used project contribution guidelines as a proxy for development workflow knowledge with the assumption that these guidelines reflected how core team members expected contributions to be made to their projects. This process knowledge would include how reviews were conducted, whether automated

tools were a part of the review process, and how developers interpreted the results of automated tools as a criterion for contribution acceptance. Therefore, we structured the study-specific research questions differently from the overarching research question, as follows:

RQ1: What is the content of contribution guidelines for projects on GitHub?
RQ2: Do projects that use CI tools mention these tools in their contribution guidelines?
RQ3: To what extent do the actual processes in projects that use CI tools match their guidelines?

We gathered workflow-specific artifacts from project GitHub repositories and used them as a basis for our investigation. These artifacts included project contribution guidelines and project activity workflow telemetry mined from the GitHub API. The rationale was that contribution guidelines convey how core team members perceive the team workflow, and the project activity workflow telemetry would provide triangulation in addition to showing the actual workflow contributions follow. In that sense, observing these two artifacts helped us model GitHub project workflows in terms of the activities contributors were performing.

Investigating the workflow of GitHub projects was, at the time, the first logical step in understanding the role automation played in that workflow. When researchers investigated the impact automation had on a software project, the literature had either focused on GitHub projects as a unit of analysis [87, 10, 12, 66] or GitHub developers as an alternative unit of analysis using surveys and interviews [64]. Additionally, GitHub in particular offered a large variety of automated tools that supported projects hosted there. These tools were featured in what GitHub calls its *Marketplace*[1] and were referred to as *Integrations*, which later evolved into *Apps*. Furthermore, the popular GHTorrent project [96] and studies that used it increased my confidence at the time that the primarily artifact-focused approach to investigating projects was a "*research community best practice*".

In the following sections, I outline how we conducted the study in terms of collecting GitHub data, what the results were, and how they answered the study-specific research questions. Then, I discuss the limitations that applied to this study. Finally, I put the study into my dissertation's higher-level context and illustrate how it helped answer the first overarching research

[1] `https://tinyurl.com/pjbka9aj`

question as well as inform the studies that followed.

4.2 Study Design

The goal for this study was to investigate how automation featured in project workflows. Following the trend of using artifact-centric research methods to investigate GitHub projects in the literature, we examined a sample of GitHub projects to answer the study's questions. First, we discuss how that sample was selected, how we coded their contribution guidelines, and then how we mined and visualized their contribution workflows.

4.2.1 Project Selection Criteria

To obtain a sample of projects from GHTorrent (about 37 million projects), we used an amalgamation of criteria from Vasilescu et al., Tsay et al., and Munaiah et al. [10, 19, 97]. These criteria were checked against the guidelines set by Kalliamvakou et al. [98] to confirm we were not making any erroneous assumptions about how GitHub (and consequently, GHTorrent) structured their data. These criteria are as follows:

- **Ignore forks:** Forks are duplicates of a repository created when a contributor cannot push directly to the main code repository. In addition to being duplicates of project history, this history is incomplete and not representative of the development that occurs on the main repository. As such, these repositories were not included in the sample [19, 98].
- **Ignore deleted projects:** GitHub and GHTorrent retain *all* repositories, even those that have already been deleted. Development on these repositories is no longer active and they are no longer accessible via the GitHub API. Thus, deleted repositories were excluded from this sample [98].
- **Ignore projects without recent commits:** Recent commits mean that a project is active. To operationalize "recency", we specified a time window of one week before the sampling period. If a project had at least one commit made in the week prior to sampling, then it would be considered active [19, 98].
- **Ignore projects that have less than 10 recent pull requests:** Pull requests indicate a project is using the pull request model and may or may not incorporate automation at the pull request level. They are also

indicators of activity [16, 10]. Pull requests may also indicate whether a project accepts contributions from external contributors. Recency was operationalized in a manner similar to the previous point, where a project had to have at least 10 open pull requests in the week prior to sampling.

- **Ignore projects that have fewer than three unique contributors:** This criterion imposed a lower bound on the team size. It indicated that a project had a developer community that was actively working together [97].
- **Ignore projects that do not have at least one recently merged pull request:** Having pull requests alone does not mean that development related to that pull request is active or that it was merged [98]. To that end, we specified this criterion as a way of operationalizing whether a project was actively accepting contributions.

The above criteria reduced the population size to 41,642 projects. This version of the population was active, did not include duplicates, used the pull request model, and had an attached team or community of developers.

After narrowing down the population to active projects that involved a team of developers, the next step was to determine which of these projects used automation in the form of a CI tool. I cloned all 41,642 projects and mined their repositories for common CI tool configuration files (e.g., .travis.yml). Based on that process, we separated the projects into two sets: those that used a common CI tool (28,904 projects), and those whose repositories did not contain common CI tool configuration files (12,738) similar to Zampetti et al. [99].

Given the large number of projects, there was no guarantee that all of them were maintained by a reasonably large community, or that they would have enough activity to accommodate a mining and qualitative coding approach. Thus, we sorted the projects similarly to how GitHub ranks open source repositories [100] by unique contributors and selected the top 100 projects.

Following the assumption that process knowledge would be available in contribution guidelines, for the projects that used CI tools, we coded their contribution guidelines. First, we looked for a *CONTRIBUTING.md* file, and if that was not available, we looked for a *README.md* file. I excluded 28 of these 100 projects due to the following reasons:

- The file size was too small to contain useful information. Files less than 2KB in size were ignored [95].

- The file did not contain actual guidelines, rather, it linked to external sources (usually style guidelines for various languages) [95]. We used this criterion because the norm was to use files that GitHub initializes automatically, and guidelines that exist apart from the repository are far less common.

At the end of this process, we were left with a final sample of 72 projects with reasonably high contribution activity, that use CI tools, and have meaningful guidelines in their repositories.

4.2.2 Contribution Guideline Coding

To investigate how project team members perceived their project's contribution process, we decided to examine their contribution guidelines. If the *CONTRIBUTING.md* file did not exist in their project repository, we selected the *README.md* file instead.

For each of the 72 projects remaining in the sample, we coded the contents of their process documentation as indicated by their contribution guidelines. I labelled each statement based on what topic it addressed. For instance, "*If the code change needs to be applied to other branches as well (for example a bugfix needing to be backported to a previous version), one of the team members will either ask you to submit a PR with the same commit to the old branch, or do this for you.*" - w. as given the "How to Submit Bugfixes" category. Similarly, "*Please sign our Contributor License Agreement (CLA) before sending PRs. We cannot accept code without this.*" - w. as assigned the "Signing a CLA" category. Using this process, we built a coding scheme that reached saturation after 50 files. I coded all 72 files, however, no new codes emerged after the first 50. The codebook and other relevant artifacts are available as part of the reproducibility package [101]. Using thematic coding as described by Creswell and Creswell [83] allowed the themes to emerge inductively from the data.

As a sanity check, we compared our list of codes to the ones Prana et al. [95] generated when they labelled README file contents for machine learning purposes. We also compared it against the information about the contribution process accumulated by Gousios et al. [16, 17] when they surveyed GitHub reviewers and contributors about the contribution process. Our codes were of a finer grain than those Prana et al. [95] created, which allowed me to fit them into their higher-level categories. Our codes also aligned with what Gousios et al. [16, 17] found regarding contributing via

pull requests.

4.2.3 Project Workflow Mining and Visualization

To capture the projects' workflows in a manner that reflected how contributions were being made in real time, we mined the data from the GitHub events API. Unfortunately, only 53 out of the 72 projects were accessible via this API endpoint. For a period of four weeks, we mined the 53 projects by capturing repository events as they occurred. At the four-week mark, we reached a saturation point where the data collected did not indicate any variation from the patterns we had already seen. Over the four weeks, we queried each project's API for events pertaining to contribution workflow. The events included, but were not limited to:

- opening/closing an issue,
- opening/closing a pull request,
- pushing a commit, and
- commenting on an issue/pull request/commit.

To better grasp the contribution workflows of the projects we were mining, we attempted to connect the various activities together. I used, where possible, the references developers made in the each entity's text (e.g., issue title or description, pull request title or description, etc.) to link issues to their corresponding pull requests and pull requests to their corresponding commits. Once the mapping was accomplished, we visualized the workflows using a process mining tool named Disco[2] and generated process maps from the log data to facilitate comparisons to the actual documentation as well as to each other.

In Figure 4.1, I give an excerpt from the contribution process of the Apache Camel project. For instance, a contribution can be made directly to the master branch, as indicated by the *Push Commit(s)* step. Or, it can take the form of a pull request, where one is typically opened and reviewed. If a pull request is found to meet the project's acceptance criteria, it is approved and closed. If it does not, the contributor is encouraged to make amendments to their contribution, after which it is reviewed again, then either approved, returned, or rejected.

[2] `https://tinyurl.com/wk743d`

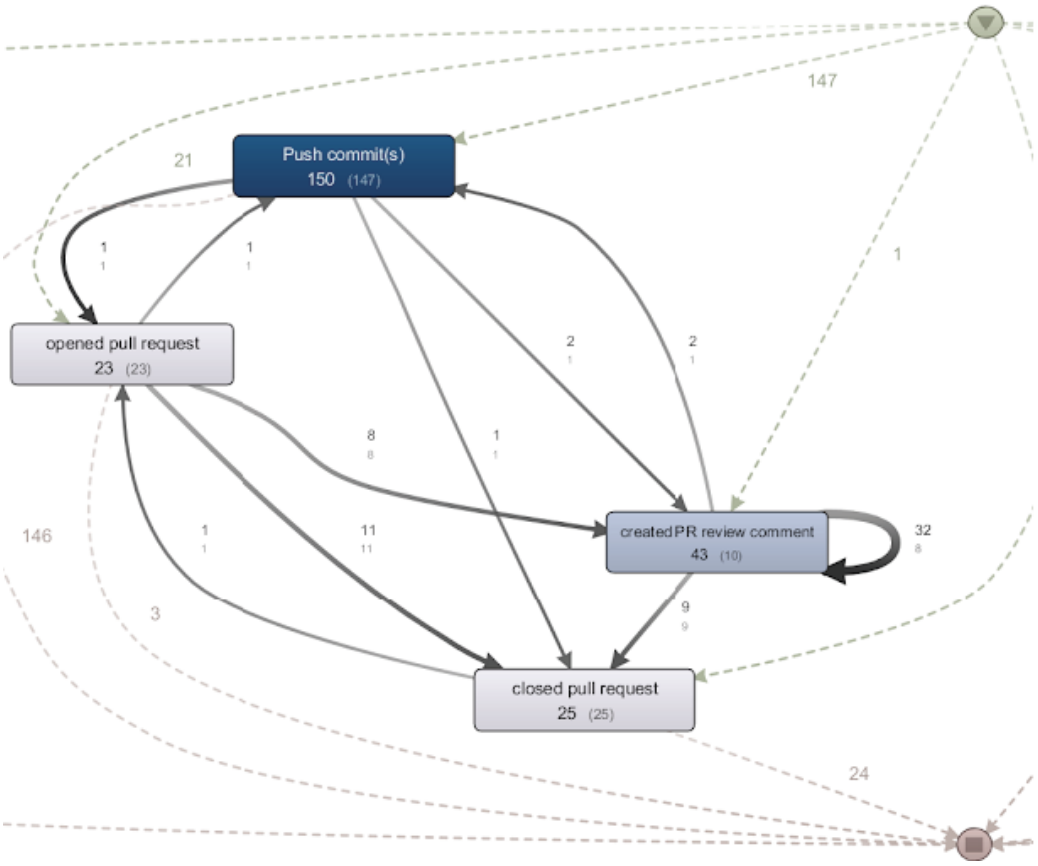

Figure 4.1: Contribution workflow visualized by Disco

4.3 Results

Following the methods described in the previous section, I managed to extract contribution guideline content patterns. I also compared the prescribed contribution process in those guidelines to those we visualized from real-time data for the 53 projects that were accessible over the API and had substantive guideline documents. This section answers the study-specific research questions posed in Section 4.1 and is organized according to those questions.

4.3.1 RQ1: What Is the Content of Contribution Guidelines for Projects on GitHub?

Contribution guidelines reflect how core team members envision how the contribution process *should* work. They typically provide contributors with details on the steps they need to take to contribute to an open source project, what criteria their changes will be evaluated against, and other relevant details. As a result of our investigation, we found five high-level categories of information communicated through these guideline documents: project orientation, contribution workflow, pull request acceptance criteria, continuous integration tools, and traceability. An overview of these categories can be found in Table 4.1, and the full version can be found in our reproduction package [101].

Table 4.1: Most frequent contribution guideline categories [7].

Content Category	Featuring Projects
Pull Request Acceptance Criteria	
Contribution Style	72.22%
Contribution Includes Test Cases	52.78%
Contribution Documentation	47.22%
Project Navigation	
How to Open an Issue	69.44%
How to Set up a Local Development Environment	48.61%
General Technical Knowledge	38.89%
Contribution Workflow	
Submitting a Pull Request	73.61%
How to Branch in a Repository	56.94%
How to Fork/Clone a Repository	52.78%
Continuous Integration Tools	
Testing by CI Tool	30.56%
Traceability	
Artifact Linking for Traceability	19.44%

Pull request acceptance criteria

Pull request acceptance criteria offer information on the standard of quality (and other aspects) a contribution should strive to meet for a reviewer to accept it. The criteria embody core maintainer expectations from contributions made to their projects, as well as standards in terms of quality,

testability, maintainability, and other project-related aspects. Examples of sub-categories that fit here include, but are not limited to: contribution size, types of tests to include with a contribution, and even what constitutes acceptable contribution documentation.

According to Prana et al.'s categories [95], this category falls under the *Contribute* category since it involves contributing to a project. Furthermore, information in this category aligned with reports that Gousios et al. [16, 17] collected from both contributors and reviewers regarding reviewer expectations about the contribution process.

Project navigation

The project navigation category typically contains guidelines for introducing developers to a project, familiarizing them with the various internal processes and common task workflows. Examples of sub-categories that fit here include, but are not limited to: how to open issues, project setup and execution, and what constitutes adequate documentation.

Information that fell into this category aligned with what Prana et al. [95] consider to be the *What*, *Why*, *How*, *When*, and *References* categories. For instance, project setup and execution was similar to the information they classified under the *How* category because it describes how a user would set up a project and run it locally.

Contribution workflow

This category explains to a new contributor how to make a contribution to a project, be it code or otherwise. Some projects consider changes or additions to documentation a form of contribution. Examples here of sub-categories include, but are not limited to: how to fork or clone a repository, how to submit a pull request and how to document it, how the review process works, and how contributors should pick issues.

Prana et al. [95] classified such information under the *Contribute* category because it pertains to contribution. This category, however, covers a wider area and includes what we chose to classify into a separate category: pull request acceptance criteria. We chose to separate the two categories (pull request acceptance criteria and contribution workflow) because the contribution workflow is almost solely contributor specific, whereas the pull request acceptance criteria pertain to both contributors and reviewers. Information

under this category also aligned with contributor reports that Gousios et al. [17] collected from contributors regarding the contribution process.

Traceability

A common theme across project contribution guidelines is the aspect of traceability. Contribution guidelines ask contributors to reference the issues their contribution addresses in both their commit messages and their pull request titles and/or descriptions. Linking issues to pull requests and commits as contribution artifacts establishes a chain of traceability that facilitates the review proces, and future investigation into the overall contribution process similar to the work done by Ståhl et al. [102, 103]. Traceability via contribution artifact linkage was not included in the categories identified by Prana et al. [95].

4.3.2 RQ2: Do Projects That Use CI Tools Mention These Tools in Their Contribution Guidelines?

To discern the role of continuous integration tools in the development process, we separated excerpts addressing the use of continuous integration tools into their own category nested within the overarching contribution category. Whenever continuous integration tools were discussed in a project's contribution guidelines, it was only in reference to a form of review automation, where passing tests (i.e., green builds) were a criterion a reviewer considered when reviewing a contribution. Automated continuous integration tools were only ever treated as a vehicle for running tests.

In terms of documentation, there was little to no information about what the automated tools did beyond testing. There was little to no documentation indicating what scripts they ran or how the build process was handled within the automated continuous integration tool itself compared to the dense amount of documentation found on other topics.

With respect to continuous integration practices, there was no evidence to indicate that the projects in our sample followed continuous practices. It was not mentioned if developers were expected to commit at least once a day, the extent to which the development environment was similar to the production environment, or how automation was featured in developer communications.

4.3.3 RQ3: To What Extent Do the Actual Processes in Projects That Use CI Tools Match the Processes in Their Guidelines?

Upon comparing the projects' contribution guidelines to the real-time processes visualized using Disco, we found three distinct deviations.

- Some projects used contribution practices that were not discussed in their contribution guidelines. For instance, 51% of the projects in our sample reopened closed issues despite the contribution guidelines not mentioning when an issue should be reopened. Another example found in 68% of the projects in our sample was reopening pull requests, which was also not documented.
- 19.5% of the projects in our sample explicitly asked contributors to link contribution process artifacts (issues, pull requests, and commits) together to enhance traceability. However, we rarely observed such links which resulted in issues, pull requests, and commits occurring arbitrarily.
- 68% of the projects whose activity we had access to prescribed a pull request-based contribution workflow, however, all but one (52 out of 53 projects) featured direct commits to the master branch that were not linked to a pull request.

4.4 Discussion

In this section, I discuss the significance of this study's three major contributions. I also relate them to this thesis' overarching research goal and illustrate how it led to the next two studies.

4.4.1 Study Contributions

Contribution guidelines are, for many, the first point of interaction with a project's contribution process [104]. Their purpose is to guide new contributors and familiarize them with the project. They also include information about the tools needed to effectively and efficiently contribute to a project. However, our study of 53 active and accessible GitHub projects that use a continuous integration tool reveals two issues in contribution guideline documents: they do not discuss *all* agreed upon project contribution workflows,

and they tend to focus more on details that could be handled by automation than they do on communicating the tacit knowledge necessary to contribute to a project.

For instance, the majority (72%) of the project contribution guidelines we studied contain information about code style, rudimentary GitHub workflow, and other technical information for developers. Code style and other technical information can be automated via linters and other similar automated code analysis tools.

As it now stands, contribution-specific tacit knowledge regarding team workflows is not being communicated sufficiently in project contribution guidelines. Furthermore, Steinmacher et al. [105] found that this lack of tacit process knowledge constitutes a barrier to newcomers, effectively making it harder for newer contributors to join a project.

4.4.2 Implications for My Dissertation

I conducted this study aiming to answer the first overarching research question:

> How does the presence of an automated pipeline impact the software development process projects use in a continuous context?

However, the inferences I could make based on this study regarding the overarching research question were limited. Aside from the lack of tacit process knowledge, another observation we made throughout our study was the limited amount of information contribution guidelines communicated about continuous integration and its associated tools. While contribution guideline documents might include instructions on how to run a continuous integration tool locally or a continuous integration tool being used for testing, there was little to no information regarding the build process. There was also no mention of whether project maintainers expected contributors to follow a continuous development paradigm.

The lack of workflow and tool information meant that I would not be able to make assumptions about whether open source projects used a continuous paradigm or the capacity in which they used continuous integration tools. As a consequence, future studies aimed at understanding development workflow had to be conducted with the precondition that the development team followed a continuous paradigm. And since this precondition is not verifiable

on platforms such as GitHub or GitLab without being closely involved with the development process (especially its tacit aspects), I decided to focus on an industry context with a small sample size where it would be more feasible to acquire such tacit knowledge.

4.5 Limitations and Threats to Validity

In this section, I discuss the limitations of this study and how we attempted to mitigate them. I use the total quality framework by Roller and Lavrakas [106] and consider credibility, analyzability, transparency, and usefulness.

4.5.1 Credibility

Credibility refers to the accuracy and completeness of the data we collected. Accuracy and completeness generally apply to two aspects of the data: the scope and the process of data gathering.

Scope

The scope of our study was limited to the top 100 projects on GitHub that used their own metrics [100] and following the guidelines set by the literature [98, 97, 10, 19]. Out of those, we were only able to mine 72 projects via the GitHub API. These were all open source, publicly available projects, which implies the context is limited to open source projects. Because of the open source nature of the projects in our sample, it is possible that our results may not transition/generalize to a non-open source context.

Data Gathering

We mined the GitHub API in real time for events the projects were experiencing related to issues, pull requests, and commits. Our sampling window was four weeks. As such, it is possible that we collected incomplete event chains (i.e., full progression of issues to pull requests to commits) in our sample. There are three possible event chains that are included in our sample based on their progression relative to our sampling window:

- **Ideal Sampling:** Ideal sampling occurs when an event chain starts and ends within a sampling window. No special measures were needed for this case.

- **Late Sampling:** Late sampling occurs when an event chain starts before sampling is initiated. For instance, an issue may have been opened and a pull request initiated before our sampling process started. As such, we may have missed earlier links (opening an issue, opening a pull request). To mitigate this threat, we used the relevant API when querying the artifacts to determine the event chain links. When a pull request or commit appeared in the event stream, we examined its title and body for references to issues, and retrieved that information via the issue or pull request API instead of the events API as it does not return more than 300 distinct events per query.
- **Early Sampling:** Early sampling occurs when an event chain ends after a sampling window is over. For instance, an issue or pull request may have been opened during the sampling window, but it was closed after the window elapsed. Early sampling did not have an impact on forming event chains since we had the issue information from the beginning. As such, no special measures were needed for this case.

The differences between the sampling windows can be seen in Figure 4.2.

4.5.2 Analyzability

Analyzability refers to the accuracy of the analysis that was conducted throughout the study and the interpretations drawn from it. Analyzability involves making sure the way the data was processed is valid, and that the conclusions we drew from the data are valid.

Processing

Upon fetching event data from the GitHub API, it was checked for duplicates. If any duplicates were included in the event stream, they were removed. We visualized the event data using Disco, an established process mining tool, to examine developer workflows.

Regarding the contribution guidelines, we made sure to follow criteria established in the literature regarding which guidelines to include and which to exclude [95]. After we completed the coding, the categories were checked against similar works in the literature [95] as well as accounts of the contribution process [16, 17].

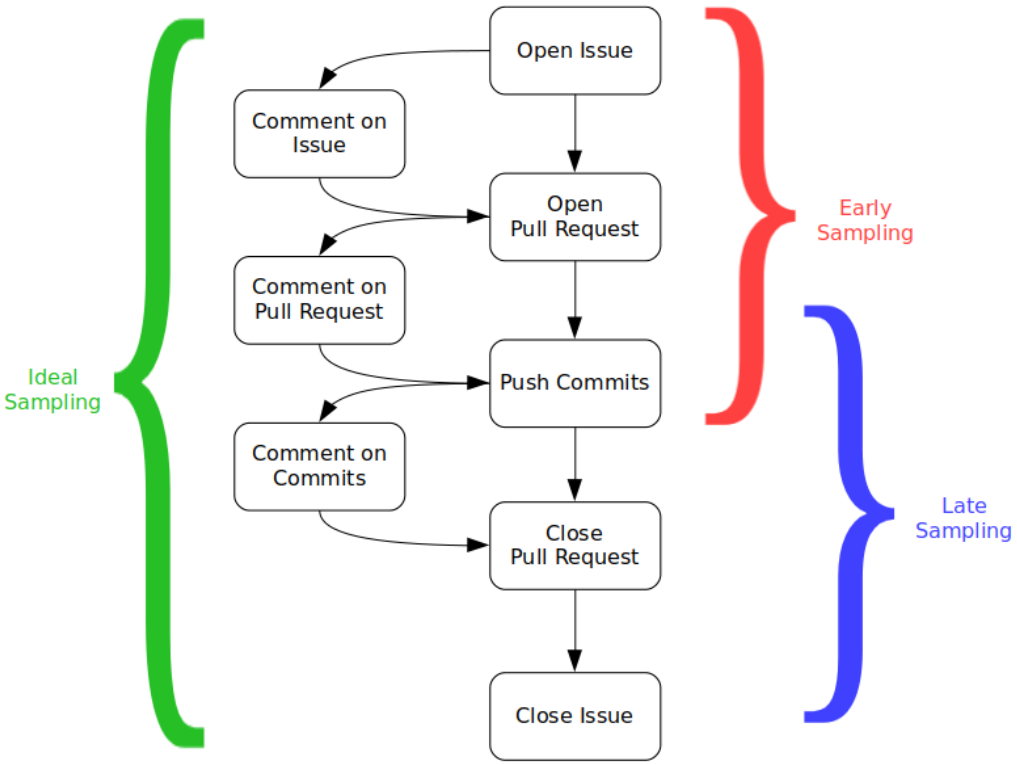

Figure 4.2: Early, ideal, and late sampling windows in the event stream

Verification

As mentioned earlier, we triangulated our coding results by comparing against categories and development workflows already examined in the literature [95, 16, 17] to enhance our construct and internal validity.

With respect to process mining, to make sure the results were consistent with reality, the visual representations (including metrics on how many instances of a particular event occurred) were compared against the actual development activity on GitHub about how many issues, pull requests, and commits were made during the sampling window. We found our event stream to be consistent with development activity on GitHub corresponding to the projects in our sample.

4.5.3 Transparency

With respect to transparency, we provide all our data as part of our replication package [101]. The package includes the different event streams and process maps per project, as well as the coded project-specific contribution guidelines, and our codebook. In theory, the same process may be applied to projects that use GitHub as a platform (irrespective of whether they are open source or not) to generate similar artifacts, which can be used for comparison or to make further inferences.

4.5.4 Usefulness

Usefulness indicates the degree to which this study's results are actionable and how well they can transfer to other contexts (external validity). This study's contribution can be summarized in the following points:

Contribution 1: We map the contents of open source project contribution guidelines and use process mining techniques to visualize GitHub project contribution workflows for comparison. Our process can be used to guide similar research in an industry context, and our results indicate which topics are not well discussed in contribution guidelines.

Contribution 2: We investigate the involvement of continuous integration tools in an open source project's workflow. The result of that investigation indicates that they are only used in a testing capacity, and that there is no guarantee that open

source projects use continuous practices beyond automation.

Contribution 3: We highlight the discrepancies between project contribution guidelines (which serve as entry points to new contributors) and the actual development workflow. These discrepancies indicate tacit contribution knowledge that is not documented, and opens up a research avenue regarding what the impact of such tacit process knowledge has on the contribution process.

5 Automation Impact on Non-Functional Requirements

Computers make excellent and efficient servants, but I have no wish to serve under them.

Cmdr. Spock
Star Trek: The Original Series

This chapter is based on a study I conducted in collaboration with Colin Werner, Ze Shi Li, Derek Lowlind, Neil Ernst, and Daniela Damian [8]. While the main focus of this study was understanding how non-functional requirements are treated in a continuous practice context and how some non-functional requirements were automated, it also shed light on how developers perceive automation when dealing with non-functional requirements. Thus, in keeping with this dissertation's topic, I focus only on the aspects relevant to automation and its impact on developers and the development process. Appendix A includes supplementary material for this study, including the ethics certificate we received for conducting the study. This study was published in Transactions on Software Engineering (TSE) 2021 [8].

5.1 Motivation

With the previous study (Chapter 4) in mind, it was clear that we could not depend on artifact-centric approaches to investigating automation's impact on developers in a continuous context. Not only were we unable to establish whether projects on code hosting platforms were using continuous practices (save for automation), by focusing solely on the artifacts, we would be capturing only a single (or at most two) aspect(s) of the socio-technical nature

of continuous software development. Our inability to capture an in-depth human perspective on both the technology and the environment would mean that we would be forced to make assumptions about the nature of those aspects. We wanted to capture as realistic a picture as possible of how developers interact with automation in a continuous context via a developer's handling of non-functional requirements. Thus, we designed and conducted this study in such a way as to maximize realism, at the expense of generalizability and control [107, 108] to capture the continuous context within which developers were operating and explore the socio-technical aspects of their development process.

Because this study was a collaboration with another research group whose focus was on non-functional requirements, we adopted a view focused on non-functional requirements when establishing the research questions.

RQ1: How do organizations that adopt continuous software engineering practices manage non-functional requirements?

RQ2: What challenges do continuous software engineering practices introduce when managing non-functional requirements?

While the focus was on non-functional requirements, some of the behaviours we identified shed light on how developers treated automation, how they used it to make decisions, and how automation was used to implement these decisions. Thus, this study contributed to answering the second overarching research question:

> How does the presence of an automated pipeline impact developer decision-making in a continuous context?

Non-functional requirements, also referred to as quality attribute requirements, typically refer to intangible aspects of a software system that impose some constraints on its operation [109]. Due to the multifaceted nature of the term "non-functional requirement", there are several types of non-functional requirements in software engineering [110]: Some of them can address issues related to usability (e.g., consistent interface design), others focus on reliability (e.g., site downtime thresholds), and many more.

However, non-functional requirements are often not at the forefront of a developer's mind when building software resulting in non-functional requirements having a lower priority than functional requirements. Consequently, there is a lack of investment in properly testing and documenting

them [111, 112]. Non-functional requirements have two interesting properties. First, developers have more latitude when it comes to making decisions about non-functional requirements than their functional counterparts, and thus they can be used as an indicator of how developers go about making decisions with a reasonable amount of free will. Second, while most aspects of a system (including non-functional requirements) can be continuously monitored by automated means (e.g., server up-time, connection status, etc.) [14], non-functional requirement testing cannot be easily automated [113], requiring creative solutions by the developer.

In the following sections, I outline how we conducted the study in terms of investigating how non-functional requirements are handled in a continuous context, and highlight the strategies developers used to accomplish that goal. Then, I relate this study to the overarching dissertation research goals and illustrate how the study helps answer the second overarching research question. Finally, I discuss the limitations that apply to this study.

5.2 Study Design

The goal of this study was to investigate how non functional requirements were being treated in a continuous context. To achieve that objective, we employed qualitative methods to obtain an in-depth understanding of *why* developers were making the choices they did. This section discusses how we recruited our participants, how we conducted our interviews, how we analyzed the data they provided us, and finally how we validated that data.

5.2.1 Recruitment

We used personal contacts at organizations who claimed to use continuous practices. We gauged each organization's suitability to participate in our study based on two factors:

1. They claimed to be continuous or employ some fashion of continuous process.
2. They were interested in participating in the study to understand more about how non-functional requirements were treated in a continuous context.

These criteria helped us narrow down our initial list of seven candidates to three: organizations alpha, beta, and gamma.

Alpha specializes in large data collection and analytics projects, where they process large amounts of data (usually via automated means) frequently. Beta develops and maintains an online e-commerce system focused on E-Bookings with multiple customers worldwide. Gamma's business model is focused on providing online content, and they also manage online advertisements. However, all organizations have a few characteristics in common. Each organization grew from a small startup to an established leader in their own business domains. Furthermore, when we validated the results of our analysis with them (via survey), 92% of the respondents from these three organizations held the belief that their organizations managed non-functional requirements well. All three organizations were eight years old at the time of this study and had 30-60 employees. All three ran automated builds that featured testing and deployment to an online cloud provider such as Amazon web services or Google cloud. From these three organizations, we managed to recruit 18 participants willing to be interviewed across a variety of roles within each organization. An overview of the participants is included in Table 5.1.

Table 5.1: Participants and their roles at the three studied organizations [8].

Org.	P#	Role	Gender	Exp.at Org.	Overall Exp.
Alpha	P1	Developer	Male	<2 years	<20 years
	P2	Developer	Male	<10 years	<20 years
	P3	Manager	Male	<10 years	<20 years
	P4	Manager	Male	<5 years	<20 years
	P5	Manager	Male	<10 years	<10 years
Beta	P6	Developer	Female	<2 years	<20 years
	P7	Manager	Female	<5 years	<20 years
	P8	Manager	Male	<10 years	<10 years
	P9	Developer	Male	<5 years	<5 years
	P10	Developer	Male	<5 years	<20 years
	P11	Developer	Male	<2 years	<20 years
	P12	Developer	Female	<2 years	<2 years
	P13	Developer	Male	<2 years	<5 years
Gamma	P14	Developer	Male	<2 years	<2 years
	P15	Manager	Female	<2 years	<20 years
	P16	Developer	Male	<2 years	<5 years
	P17	Developer	Male	<5 years	<20 years
	P18	Developer	Female	<2 years	<5 years

5.2.2 Interviews

Once we had willing interview participants, we proceeded to conduct our interviews. The cross-section of interviewees who volunteered offered a minor form of triangulation for the results we received from the different roles within the organization. For the participants' anonymity, we list them simply as developer or manager in Table 5.1 based on how involved with the project's technical aspects they were. The interviews were semi-structured in nature, beginning with a script of prepared questions, and we asked follow-up questions as necessary. A list of our questions can be found in appendix A. For each of the 18 interviews, we made sure to have at least two of the authors present to minimize the bias brought about by a single interviewer. I participated in 13 of the total 18 interviews, and acted as a primary interviewer for the gamma interviews, and as a secondary interviewer for the beta interviews. Each interview was recorded for later transcription and processing with the interviewee's permission.

5.2.3 Thematic Analysis of Interview Data

Since we used an automated service to transcribe the interviews, each interview transcript was later verified by a human. Once the transcripts were complete, we used inductive coding to label the different transcripts, with the codes eventually leading to overarching themes [114].

To generate our initial codebook, we picked the interview richest in detail (P5) that was coded by the first four authors. Once we had established a codebook, we coded several interviews in pairs (P3, P4, P9, P12, P13, P16, and P18) to examine the extent to which the coders were aligned. After each pair of authors had finished coding a transcript, we conducted an agreement session to evaluate how well they aligned with respect to the codes used. Because of the rich nature of the data, we assigned multiple codes to the same excerpt. However, this meant we were not able to use either Cohen's Kappa or Krippendorff's alpha to measure agreement. As such, we defined agreement as an instance where two coders agreed on at least a single code per excerpt. Because the agreement sessions involved a lot of discussion about why disagreements happened and how to remedy them, all four coders developed a shared understanding of the codes used. Once we reached a satisfactory agreement level (85%), we opted to code the interviews separately in parallel to maximize efficiency, while having each coder's work reviewed by

another coder. Objections were addressed in a similar fashion to agreement sessions.

Once the coding process was complete, we proceeded with the thematic analysis. We divided the codes into clusters based on how similar they were, with each cluster representing a higher-level theme. The resulting themes were sorted into two categories based on our research questions: practices (RQ1), or challenges (RQ2). Only the themes relevant to this dissertation are discussed in Section 5.3.

5.2.4 Member Checking

To validate our themes and make sure we were accurate in interpreting our interviewee responses, we conducted a two-step member checking process [8]. We first sent our interviewees a survey of the themes we discovered to confirm whether the themes resonated with them. Once that was complete, we sent them a copy of our paper for them to read and give us feedback. The feedback we obtained from this two-step process was used to revise our themes.

5.3 Results

As discussed in the previous section, two main categories of themes emerged from our analysis: practices and challenges pertaining to how developers perceive and handle non-functional requirements in a continuous context. We discovered four practices in total:

1. **Measure and monitor the non-functional requirement:** Developers monitored non-functional requirements by assigning them metrics and incorporating them into a monitoring tool (e.g., dashboard).
2. **Let someone else manage the non-functional requirement:** As a side-effect of using a particular tool suite or infrastructure provider, developers relied on a third party to ensure a non-functional requirement was met.
3. **Write a customized tool to check the non-functional requirement:** Developers built tools and scripts specifically designed to test for and measure a non-functional requirement to ensure it was met.
4. **Put the non-functional requirement in source control:** Developers added any customized tool configurations, scripts, or tests that

dealt with non-functional requirements to a project's version control system.

With respect to challenges, three emerged as being consistent across the organizations:

1. **Not all non-functional requirements are easy to automate:** Some non-functional requirements were not as easy to test for by automated tools and developers had to perform these tests manually. Usability and user experience were a common theme discussed in this challenge.
2. **Functional requirements get prioritized over non-functional requirements:** Product owners frequently prioritized functional requirements over non-functional ones, which meant developers could only address non-functional requirements when time was available. Developers were seldom able to treat non-functional requirements as high-priority issues.
3. **There is a lack of a shared understanding of non-functional requirements:** Each organization, in fact each team (and sometimes developer within a team) had a different understanding of what constituted a non-functional requirement. For instance, some interviewees perceived performance to mean server up-time, whereas others perceived it as caching ratios.

Three of the practices we identified related to how automation impacted software developer decision making. While they are listed in our paper as strategies that represent how developers deal with non-functional requirements, I will reinterpret them in this section from the perspective of how automation plays a role in developer judgment.

5.3.1 Measure and Monitor the Non-functional Requirement

This first practice captures the monitoring activity our participants used to ensure their non-functional requirements were being considered in the development process. Organizations would lay out metrics to operationalize particular non-functional requirements and integrate them into the automated tools attached to their software projects (namely the pipeline). Changes or new features would flow into the pipeline and the metrics would indicate whether a change had a positive or negative impact on a non-functional re-

quirement. It was up to the development team to decide on the best strategy to handle a metric change, if it occurred.

Naturally, each organization prioritized some non-functional requirements over others and had their own interpretations of them (as seen in challenge 3). For instance, Gamma prioritized performance over other non-functional requirements and tracked it primarily via a "*[caching ratio] that we have from [redacted] our caching provider goes [on a dashboard] because that's usually a pretty good indication that something has gone wrong*" - P14.

Similarly, Alpha also prioritized performance as a non-functional requirement due to the data-centric nature of their projects. Alpha measured performance via response time, where they "*have a certain amount of monitoring set up [...] You're also defining which alarms are set. Therefore, if requests [drop] below [redacted] milliseconds [...], then that would be codified in the alarm*" - P5. The quantitative metric here is vital for Alpha to discern whether fixes are necessary.

Beta, being a customer-facing platform, prioritized usability as a non-functional requirement. They tracked usability by counting the number of actions a customer would have to perform in order to complete a transaction: "*I can tell you that 90% of our customers have less than [redacted] items [...]. They'll [say] we know that each [order] requires [redacted] page loads*" - P7. For Beta, the number of actions required is considered a "good enough" indicator of how usability should be measured.

The above metrics (and many more) are integrated into the pipeline and either continuously monitored the status of the project's production version, or are checked as part of either automated or manual testing. Different organizations (in this case Alpha and Gamma) had different metrics for the same concept (performance) which indicate different aspects of their applications.

5.3.2 Let Someone Else Manage the Non-functional Requirement

Another popular approach across the three organizations was delegating a non-functional requirement to an external provider via the tool they were using. All three organizations use cloud providers in some form to make their applications or API endpoints available to their customers. This behaviour re-framed some non-functional requirements from being the responsibility of the application developer to being the responsibility of a platform upon

which an application runs.

For instance, Alpha uses Amazon web services as a platform upon which to host their content distribution platform. To them, availability is a fairly straightforward endeavour, "*[your web application is] immediately spread across however many availability zones and nodes as you want*" - P5. . Delegating non-functional requirements removes the burden of having to deal with its technical implications and reduces its management to filing a support ticket if it should require fixing, "*cloud providers are awesome. I love being able to just file a support ticket*" - P12.

Another form of delegation is delegating non-functional requirements to a tool instead of a third-party service provider. A tool (e.g., Docker) is expected to provide some non-functional aspects just by virtue of using it, "*I believe [security]'s all codified in the like Docker and Kubernetes world*" - P6. Similarly to delegating a non-functional responsibility to a third-party provider, the tool is expected to in part (if not fully) handle some aspects of the application.

5.3.3 Write a Customized Tool to Check the Non-functional Requirement

When delegating a non-functional requirement is not a viable option, a team typically develops their own in-house tool to handle non-functional aspects of an application. These custom tools often appear in the form of scripts, manifests, or otherwise heavily customized tooling.

For instance, both Beta and Gamma codified their definition and operationalization of usability in source code, which the pipeline then runs against new changes, "*this is how we define usability and make sure that it's there. If you want to change our usability parameters or whatever we change it in source code and then we can test it and verify that it still meets our needs*" - P10. An example of heavily customizing an existing tool can be found in Alpha, where they run automated security checks whenever a new S3 bucket is created, "*so there's a lambda [function] that runs when you make a bucket, it triggers and goes 'you did not encrypt', it turns on encryption, tells you, 'you are an idiot'. Security non-functional requirement!*" - P4.

5.4 Discussion

In this section, I discuss the significance of the three practices we identified in Section 5.3 from a developer-automation interaction perspective. I also relate them to this thesis' overarching research goal and illustrate how they played a role in creating the theory presented in Chapter 7.

5.4.1 Practical Significance

There are three practices and two challenges discussed in the previous section that relate specifically to how developers make decisions with the assistance of automation. Two of these practices focus on how developers configure or make use of automation to manage non-functional requirements (let someone else manage the non-functional requirement and write your own tool to manage the non-functional requirement), and one relates to how they perceive the results from the automation (measure and monitor the non-functional requirement). The first challenge is that of the difficulty of automating some non-functional requirements, while the second is the lack of a shared understanding regarding non-functional requirements.

Letting a third-party tool manage a non-functional requirement means that the responsibility is delegated to an external party as opposed to the developers handling the non-functional requirement internally. However, they surrender control over the non-functional requirement to that external party, effectively trusting that the provider (or tool) will guarantee the successful application of the requirement. Placing this much trust in these external providers and tools could be considered *automation misuse* [76] or *complacency* [65] because it involves trusting a tool or service to provide a function with minimal interference from the developer. While the cases we investigated—and generally speaking, most cases—may not be as severe as the previous statement makes the situation out to be, developers are still expected to give up a certain modicum of control in exchange for an expectation that a provider or tool will "*handle*" some aspect of their application. Similarly, using a heavily customized tool, even if it was constructed internally, may lead to misuse in that developers may grow to implicitly trust it or encourage new hires to place an abnormally high importance on its output [77].

On the opposite end of the spectrum, checking non-functional require-

ments manually can be considered *automation disuse* [76]. In all three organizations, developers were expected to test for usability and user experience by manually using the application and ensuring it conforms to the product's overall design and other previously set criteria. The first reason for manually testing for usability was that automating usability testing using tools such as Selenium is considered too expensive of a trade-off: the effort required to automate usability testing is more than the effort required for manual testing. The second reason was because tools that test for usability could not automate our interviewees' interface design testing. For instance, despite Beta having automated usability tests using Selenium that simulate user interaction with the different pages in their application, developers were still expected to test the interface manually, and those tests were later repeated by a quality assurance team implying that automation was not trusted to capture all the aspects of usability testing.

How the different organizations measured and monitored non-functional requirements indicated how they perceived them. The organizations' perceptions of non-functional requirements drove how checks for these requirements were implemented, which was later integrated into the automated pipeline. As we discussed in the previous section, there was a case where Gamma and Alpha both measured performance yet used different metrics to express it. Gamma used a page caching ratio as a metric, which indicated how many times a cached version of a page was used (a high caching ratio meant the caching strategy was suitable and the application was not wasting resources). On the other hand, Alpha used response time to indicate performance (a low response time indicates smaller pages or more efficient networking and/or caching strategies). While both worked for each of their respective organizations' use cases, it illustrates that some aspects of software applications and services cannot be generalized across software projects without taking into account the use case driving the measurement, and consequently, the developer's perception of that particular aspect.

5.4.2 Implications for my Dissertation

In the previous section, I illustrated that the relationship between developer and non-functional requirements is nuanced, dependent on context, and does not lend itself easily to generalizability. For instance, different perceptions of non-functional requirements led to different implementations which could potentially impact the practices and tools organizations used. I also discussed

how developers can experience both automation misuse through reliance on external services and disuse due to difficulties when automating certain application aspects, most notably usability which may influence their decision making regarding aspects of their application. Misuse and disuse depend on the practices developers choose to employ regarding the different aspects of their application, and these application aspects are typically dependent on how developers, product managers, and other team members perceive and prioritize non-functional aspects of their project.

For future studies in this dissertation, it was necessary to elevate realism at the cost of generalizability because it was apparent that perceptions of application aspects drive how automation is configured to handle them. Different contexts could mean different perceptions of, for instance, non-functional requirements, which would mean different automation configuration and strategies, and eventually different ways developers could be influenced by these unique automation tools. Further studies would need to account for these differences in context and explore the different factors that cause them. However, this study lays the foundation for the relationship between developers and automation (discussed later in Chapter 7) and illustrates the impact of different perspectives across organizations.

5.5 Limitations and Threats to Validity

In this section, I discuss the limitations of this study and how we attempted to mitigate them. I use the total quality framework by Roller and Lavrakas [106] to discuss credibility, analyzability, transparency, and usefulness.

5.5.1 Credibility

Credibility is related to how accurate and complete the data collected is. It encompasses two aspects: scope and data gathering.

Scope

The scope of our study was limited to organizations that implemented a continuous software development methodology. Because we selected organizations that were willing to participate in the study and showed active interest in the topic, our sample may suffer from sampling bias. However,

two of our authors had spent a significant time collaborating with these organizations and even went as far as to conduct a preliminary study to determine whether or not the organizations implemented continuous practices to reduce that risk.

Data Gathering

With respect to the interviewees, they were representative of their organizations in terms of role, gender, and experience. Our analysis did not indicate that role, gender, or experience had a significant impact on the themes we uncovered, however, this may be an interesting objective for a future study.

With respect to the interviews, we made sure two authors were present at all interviews to mitigate the researcher bias that typically occurs when a single researcher is responsible for the data collection process. We used our list of prepared questions attached in Appendix A.

To enhance our construct validity, we made sure to control for participant perspectives on continuous practices and non-functional requirements. We began each interview by asking the interviewee about these concepts, and guiding them to the literature-approved definitions using examples to guarantee that all interviewees had a similar level of understanding regarding the concepts we were studying.

To mitigate risks to our data analysis in terms of coding and thematic analysis, we followed best practices in the literature [114]. While our usage of multiple codes per segment limited our ability to use inter-rater agreement metrics that would account for chance agreement, we conducted extensive agreement sessions, both for coding and thematic analysis, with the last two authors on the paper serving as independent reviewers of the analysis.

Finally, our analysis may be susceptible to researcher-participant interactions because we conducted the interviews in person.

5.5.2 Analyzability

Analyzability pertains to the accuracy of the analysis we conducted and the interpretations drawn from it. It consists of two aspects: processing and verification.

Processing

We recorded the interviews with the interviewees' permission. The audio was later used to transcribe the interviews using a computer-aided transcription service. Where the transcription was not clear, we made sure to check the transcripts against the original audio.

We also used an open coding approach to mitigate any potential bias the coders may bring. We performed regular peer debriefings and double-checked any deviating codes as a way of verifying our analysis, and guaranteeing to the best of our ability that our results were neutral and consistent.

Verification

We triangulated by interviewing a cross-section of organization members to ensure that the information we gathered was not biased by a single perspective. The organization members we interviewed belonged to different teams within the same organization. Furthermore, we conducted member checking in two ways as mentioned previously: we sent our participants a survey containing the major themes that our analysis uncovered to make sure the themes resonated with them, then we sent them a copy of our paper to make sure we were not misinterpreting their responses in the practices we were discussing.

5.5.3 Transparency

We used thick descriptions and quotes to augment our analysis wherever possible. We also made our research artifacts and scripts available via Zenodo [115]. The package includes our codebook as well as our interview questions. However, we were unable to include the interview transcripts due to non-disclosure agreements we signed with the three organizations.

5.5.4 Usefulness

Usefulness is an indicator of how actionable the results from a study are, and their ability to transfer to other contexts (external validity). With this study, we aimed to bring together the domains of continuous software engineering and non-functional requirements. We identify several areas that researchers can investigate further, especially considering the current trends towards rapid, automated software development lifecycles. While we recognize that

non-functional requirements cannot be treated as a single entity because each of them is a highly complex and nuanced topic in its own right, this study lays the groundwork for further in-depth investigation of individual non-functional requirements. For practitioners, this study helps illustrate *how* a non-functional requirement can be integrated in the modern, automated software development lifecycle as well as the pitfalls associated with that integration.

6 Continuous Practices: Context Versus Best Practices

There can be no justice so long as laws are absolute.

Cpt. Jean-Luc Picard
Star Trek: The Next Generation

This chapter is based on a study I led in collaboration with Colin Werner, Ze Shi Li, Derek Lowlind, Neil Ernst, and Margaret-Anne Storey [6]. Throughout this chapter, I recount how we conducted an in-depth exploration of continuous practices in three different organizations through a multiple case study. I discuss how we established that context is extremely important in understanding the nuances of how automation impacts development workflow and the developers themselves. This study was published in Transactions on Software Engineering in 2021 [6].

6.1 Motivation

This study aims to answer both the first and second overarching research questions we established earlier in Chapter 3, namely:

RQ1: How does the presence of an automated pipeline impact the software development process projects use in a continuous context?
RQ2: How does the presence of an automated pipeline impact developer decision making in a continuous context?

We conducted a multiple case study with the three software development organizations we studied in the previous study and refined the above questions

to the following three research questions:

RQ1: Which continuous practices do these organizations implement, and how do they implement them?

RQ2: What benefits do these organizations attribute to the use of these continuous practices?

RQ3: What challenges do they experience when implementing these continuous practices?

We intentionally focused on the individual practices because it would allow us to capture how developers use automation, all the while keeping in mind the organizational context within which these practices were implemented.

In the previous chapter, we observed how automated non-functional requirement testing can have an impact on developer decision making through the different strategies developers used to deal with non-functional requirements. As mentioned earlier, automation is one of ten continuous practices outlined by Fowler and Foemmel [9] and later elaborated on by Humble and Farley [26]. These practices include:

1. maintain a single source repository,
2. automate the build,
3. make your build self-testing,
4. everyone commits to the mainline every day
5. every commit should build the mainline on an integration machine,
6. keep the build fast,
7. test in a clone of the production environment,
8. make it easy for anyone to get the latest executable,
9. ensure that system state and changes are visible, and
10. automate deployment.

Automation is the most frequently discussed practice in the literature (as shown in Chapter 2), and while other research has attempted to explore other continuous practices [68, 60, 12], they are typically explored from a technical aspect or using data-centric strategies [116]. Seldom are continuous practices investigated while considering all their socio-technical aspects, such as developer perception (human) or process and context impact (environment). To capture the social, technical, and environmental aspects of continuous software engineering, we conducted a multiple case study of continuous practices within three organizations. We discovered that each organization had their own version of what a continuous practice was and how it should be implemented. As a result of this study, we identified the factors impacting the different implementations of continuous practices and illustrate how they

can be used as the building blocks of this dissertation's main contribution: a socio-technical theory of continuous practices (see Chapter 7).

6.2 Study Design

The goal of this study was to conduct an in-depth investigation of continuous practices, the role automation played in the development workflow, and its impact on developers in terms of workflow and productivity. To achieve that objective, we structured our study as a multiple case study, and relied on mixed methods when it came to data collection and analysis [92]. From what we learned in our previous study (Chapter 5) [8], we wanted to capture the context within which continuous practices were being implemented. We also chose to use Fowler's ten practices [9] as a basis for this investigation. This section discusses how we recruited our participants, how we collected data, and finally how we conducted our analysis.

6.2.1 Recruitment

To facilitate conducting the case studies with the inclusion of context, we approached the organizations from our previous study (Chapter 5). We decided to involve these organizations because we were able to visit in person, and could be exposed to the context within which they operated as well as establish trust so they would feel comfortable sharing their data with us and having us conduct interviews. The organizations also considered our study of value to them as it would help them reflect on how they were implementing continuous practices and why these practices were employed in the first place. Finally, and most importantly, we had worked with these organizations before and were familiar with their contexts.

Organization A operates using data processing as its primary business model, and frequently collects, processes, and makes large amounts of data available for its customers. Organization B is an online content provider who also incorporates advertisement management in their online platform. Organization C is an online e-commerce platform that specializes in electronic bookings.

All three organizations utilize a software-as-a-service model where their platforms are made available to their customers online as API endpoints or cloud applications [117]. They use automation in the form of automated build

and deployment tools as advocated by continuous practices. Organizations A and B self-identify as being continuous (i.e., implementing continuous practices), while C self-identifies as "*becoming continuous*". More information on organizations A, B, and C can be found in Table 6.1.

Table 6.1: The characteristics of organizations A, B, and C [6].

	Organization A	Organization B	Organization C
Domain	Data Processing	Online Content Provider	Online Bookings
Team(s) Interviewed	4	1	3
Average Team Size	3-4	8-10	6-12
Participants Interviewed	5	5	8
Employee Distribution	Co-located	Distributed	Co-located
Reported Continuous Status	Continuous	Continuous	Moving towards Continuous

6.2.2 Development Activity Log Mining

All three participating organizations were kind enough to allow us to access their development activity logs. We also managed to access build and deployment logs for organization B. Our data spanned the following chronological windows:

- **For organization A:** From January 2019 to November 2019.
- **For organization B:** From January 2017 to June 2019.
- **For organization C:** From October 2017 to October 2019.

We collected this data as a form of triangulation to verify that the different organizations implemented continuous practices.

However, the data we extracted had its limitations. Using the development activity logs, we could only successfully verify four out of the ten practices listed by Fowler and Foemmel [9] (practices 2, 4, 5, and 6), even then with some caveats.

- **Automate the build:** The fact that build logs existed confirmed that the organization included some form of automated build process in their workflow.
- **Everyone commits to the mainline every day:** Even though we had access to commit activity data, it was possible that some commits were lost due to squashing or cherry-picking from different branches.
- **Every commit should build the mainline on an integration machine:** Since each commit had a build status indicator, we were able to establish whether every commit resulted in a build.

- **Keep the build fast:** Using the build logs, we were able to calculate build durations.

Although we had access to development activity data, the data did not explain *why* certain practices were implemented, and if they were implemented, why they were implemented in that fashion. We were also not able to ascertain the status of the remaining practices from the log data or whether the practices in use produced the benefits claimed in Table 2.1 according to Chapter 2. Furthermore, the data gave no indication as to whether the practice trends we were seeing were considered challenges by the developers. Therefore, we used interviews to fill in the blanks.

6.2.3 Interviews

Within the organizations, we managed to interview a cross-section of members belonging to different roles (developers, managers, teams leads, and directors). We followed an opportunistic recruitment strategy to find interviewees. Our contacts within the organizations would announce that we were conducting a study that focused on continuous practices and that we were looking for participants. In total, we had a pool of 18 interviewees across all three organizations once the recruitment phase was complete. The pool consisted of the following roles:

- 1 senior developer,
- 1 data science team lead,
- 1 front-end web developer,
- 1 project lead,
- 3 DevOps engineers,
- 2 full-stack developers,
- 2 product owners,
- 1 chief marketing officer,
- 1 director of technical support,
- 1 chief technology officer,
- 1 account manager,
- 1 chief operating officer,
- 1 junior developer, and
- 1 director of sales.

Having a wide cross-section of organizational roles helped ensure that the information our interviewees were sharing with us was not biased by a single perspective, though it is possible that the automation technology may bias

Table 6.2: Interviewee mapping to organization and role [6].

Participant	Role	Organization
P1	Developer	A
P2	Developer	A
P3	Developer	A
P4	Developer	A
P5	Manager	A
P6	Developer	B
P7	Developer	B
P8	Developer	B
P9	Manager	B
P10	Manager	B
P11	Manager	C
P12	Developer	C
P13	Manager	C
P14	Manager	C
P15	Manager	C
P16	Manager	C
P17	Developer	C
P18	Manager	C

participants on the same team. Furthermore, it was a way of triangulating the data across different sources.

To preserve the participants' anonymity, we reclassified their roles based on how technical they were. Highly technical roles were classified as developers, whereas the rest of the interviewees were classified as managers similarly to the previous study (Chapter 5). An overview of our interviewees can be seen in Table 6.2.

We conducted semi-structured interviews with our participants based on the ten core continuous practices discussed in the previous section. For each participant, we started by determining how familiar they were with continuous practices, and then slowly moved on to the practices themselves. The questions were structured to be open-ended, and we encouraged participants to elaborate on the impact of continuous practices as well as the reasons behind choosing to implement or disregard a practice.

Two interviewers were present for each interview to alleviate any bias a single interviewer may introduce. We made sure to ask interviewees to repeat unclear segments in their responses for further clarification. We also occasionally asked them to repeat what they were saying and sometimes

rephrased their answers back to them to ensure we had captured their intended meaning. Our full list of interview questions is available in Appendix B as well as our reproduction package [118]. We recorded the interviews with the interviewees' permission and used a voice-to-text service to transcribe them later. Once the interviews were transcribed, a human revised them and compared them to the audio to guarantee consistency.

6.2.4 Thematic Analysis of Interview Data

Similarly to Heikkilä et al. [119], we followed an inductive coding approach and allowed codes to emerge organically from the transcripts. At the beginning, we selected four transcripts to produce our master codebook. Each transcript was coded by two coders: I personally coded all 18 of the transcripts. We followed the constant comparison method, whereby we would assign a code to each passage within a transcript. If none of the codes in our codebook reflected the meaning conveyed in the passage, we added new codes as necessary. After a transcript had been coded, the two coders conducted an agreement session where they discussed and resolved their disagreements. Any new codes were added to the codebook and duplicate codes were merged.

Once transcript coding was complete, we conducted thematic analysis sessions where we revisited our notes and the transcripts, and combined codes to form higher-level themes. We discussed the themes and our interpretations of them during these sessions. Once we had merged duplicate themes, we built a final list of themes which would be used as the basis for our member-checking survey.

6.2.5 Member Checking

To ensure our interpretation of the data matched what our interviewees had discussed with us, we used our themes to build a member-checking survey. We used a five-point Likert scale so survey respondents could indicate the extent of their agreement with the themes, and we asked the participants to focus on the projects with which they were most familiar, similarly to the interviews. We sent the survey to our participant organizations for feedback and received a total of 8 responses from our original interviewee pool of 18 (44% response rate). Some of the themes in the survey did not receive broad agreement from our respondents and consequently were removed from the

themes we discuss in the following sections. Our survey is also included in the reproduction package.

Using the results from the interviews, thematic analysis, and the survey, we built what we referred to as "*case profiles*". Case profiles are documents containing all the context we observed within the different organizations, their status regarding individual continuous practices, challenges they were facing, and recommendations grounded in literature on how to overcome these challenges. We sent each organization their corresponding case profile for two reasons: for feedback and as a way to share our knowledge with them on their current status with respect to continuous practices. The case profiles are also included in the reproduction package. Finally, we sent each organization a copy of our paper to collect more feedback as well as ensure we were not misrepresenting them.

6.3 Results

From our analysis, we identified three factors that impacted *how* developers implemented continuous practices within their organization. The first factor is **practice perception**, which refers to a developer's understanding and interpretation of the practice. For instance, a developer may interpret "make it easy for anyone to get the latest executable" to mean that an executable of their project should be available for others to use. Another interpretation for "make it easy for anyone to get the latest executable" we came across was that a person should be able to recreate the infrastructure required and successfully deploy the application with ease.

The second factor is **project context**, which refers to the project nature, requirements, and characteristics that may influence how a practice is being implemented. For instance, data-centric projects involve significant testing to ensure data integrity, processing large amounts of data, and sometimes fetching data from external sources. When all of these tasks are integrated into the automated pipeline, "keeping the build fast" may mean somewhere in the vicinity of 30 minutes or an hour, compared to the expected 10 minute rule of thumb that typically applies to application-centric projects [9, 26].

The last factor is **tool constraints**. Automation can do a lot of things, but it cannot do everything. For instance, the directive "make the build self-testing" may mean to include tests for an application that runs within the build. However, there are some tests that still require human involvement,

like usability. While there are usability test suites being used in practice as discussed in Chapter 5, a human is still needed to evaluate the user experience and judge whether changes have an impact on it. Table 6.3 lists the ten practices we investigated. We illustrate how each organization implemented a practice, what their rationale was for implementing it, and the challenges they faced during its implementation. We also include whether the differences in practice implementation were due to practice perception, project context, or tool constraints. In the next sections, I detail the more interesting differences in practice implementation that we encountered.

6.3.1 Maintain a Single Source Repository

The claim behind this practice is that it centralizes dependencies, facilitates builds, and enhances code visibility. However, we found that each organization tends to implement it a different way, primarily due to their different perceptions of the practice.

Organization A prioritizes reducing merge conflicts and separates project components across different repositories based on how frequently they need to be simultaneously changed or updated: "*Because all the three pieces just interact with one another, they're not sort of set up really to like produce something for another project*" - P3. Organization B structures its platform components in a micro-service like architecture, resulting in the components being fetched at build time via a dependency management system. Organization B did not consider using a single source repository mainly because "*no one's been able to show us the benefit of mono repo or I have not been able to see the benefit of a mono repo*" - P9. Organization C keeps its main project in a single repository because it is what its business model is based on. However, over time, new in-house components were separated from the project into their own repositories along with older tooling components because "*the biggest reason for that file storage, like git LFS wasn't mature enough at the time*" - P12. While the git LFS limitations have improved over time, organization C still has not made the change to a single source repository.

The differences in practice implementation here are due to *how developers perceive the practice*. In organization A's case, the benefit of minimizing merge conflicts outweighs the claimed benefits from using a single source repository. In organization B's case, the claimed benefits of a single source repository have not been adequately investigated, and therefore are not a

strong enough motivation to implement this practice. Finally, in organization C's case, they actually moved in the opposite direction (from a single source repository to separate repositories) initially to avoid tooling constraints, but there is no motivation to implement the practice because how their projects are structured right now is considered good enough for them.

6.3.2 Make the Build Self-testing

All three organizations include tests in some capacity in their automated pipelines. However, *how* they include tests differs across organizations. For instance, organization A only includes lightweight tests on pull requests, with the lengthier tests being performed by reviewers when the change is ready to merge because "*I don't want long tests on the developer's side*" - P1. Conversely, for the data-centric projects in organization A, the benefit of extensive testing outweighs the benefit of a faster build.

Organization B follows a similar approach to organization A but for different reasons. For organization B, running tests requires infrastructure resources which, in turn, require a financial investment; They prioritize investing in their production environment instead of the development side of the process. Furthermore, they discarded flaky tests because "*we found that our end-to-end testing and smoke tests were much less useful because they had lots of false positives because they involved the real world*" - P6. Therefore, it made more sense to invest in the real-world version of the application (i.e., the production version) as opposed to the developer-facing version.

Organization C has the most comprehensive test process across the three organizations because they prioritize quality over build duration. They run their full test suite on pull requests and the mainline. Then they run a batch of manual tests to establish usability, before handing off a feature to quality assurance personnel for further testing. Finally, a product manager is expected to test the features that have traversed this pipeline, "*there will be automation and manual sanity checks after it's ready to go*" - P13.

While all organizations report doing some form of usability testing, there is not much investment in automated usability tests because "*it would be too much work. There wouldn't be enough value for how much money I have to spend to get that testing to automate all that selenium. All that would be too cost prohibitive effectively, you know with how much websites change.*" - P1.

The differences in practice implementation here are due to three factors:

tool limitation, practice perception, and project context. Tool limitation manifests itself with automated usability tests not being feasible, while different practice perceptions are clear with how organizations consider the trade-off for implementing integration tests (the cost of investing in integration tests outweighs their perceived benefits). Finally, differences in project context show when data-centric tests require communication with external services.

6.3.3 Keep the Build Fast

The rule of thumb when it comes to this practice is to keep builds under 10 minutes long [9, 2] to facilitate rapid feedback. While all three organizations attempt to minimize their build durations, what constitutes an acceptable amount of time varies across organizations, and sometimes within the same organization. These variations are mainly due to project context.

Build durations in organization A vary from one minute to a few hours depending on which project the build is attached to and how complex it is. Application-centric projects typically have short build durations to reduce developer context switching: developers tend to switch to other tasks instead of waiting for a build to finish. However, data-centric projects in organization A have significantly longer build durations due to the extensive data processing involved. However, long build durations can prove problematic because "*if it ran for 40 minutes and then it failed and then you got to go in and figure out why it failed then re-push*" - P2.

In organization B, builds typically last for 10 minutes or less to reduce developer context switching (similarly to organization A) and to reduce developer blocking. The short build duration formed a positive impression on developers in that they "*don't really think about it or feel that it's holding me up*" - P8. The shorter build durations may be a result of their micro-service-like architecture where each component has its own independent builds, and they do not need to be rebuilt or tested again in the mainline.

Organization C's builds last between 10 and 20 minutes for two reasons: comprehensive testing is prioritized over build duration, and infrastructure constraints. Due to the limited infrastructure resources they have access to, they are forced to run their build components in an inefficient manner which involves "*some [test] serialization inside of each child as well*" - P12. The longer build durations in organization C also create bottlenecks "*if like a whole bunch of people are already kicking off their builds. Like we only have*

a limit of like five workers at a time [...] so if a whole bunch of people push right before I pushed my build is going to be at the back of the line" - P17.

The differences in practice implementation here are primarily due to project context. Some of organization A's projects require extensive data processing which impacts build duration. In comparison, because organization B and C's projects are application-centric, their builds do not involve data-heavy operations. In organization C's case, tool limitations do play a role in the form of how much infrastructure resources they can allocate to builds.

6.3.4 Automate Deployment

All three organizations have a manually triggered deployment process. The main differences are in who has deployment privileges to production and when deployments occur. For organizations A and C, deployment to production is typically performed by a senior team member "*just so not anybody can just do it whenever they want as a team grows*" - P1. For organization B, the developer responsible for the change is in charge of deploying it to production and "*babysit it all the way through because when deploying you need to know the context, you know what to look for when it's actually out and deployed. And the person who knows it best is the person who wrote it. So as much as possible, we like to have that person press the button*" - P7. This creates a culture of ownership where developers are considered to *own* the change they made until it safely lands in production.

Regarding deployment frequency, deployments for organizations A and B are typically more frequent in that they deploy to production several times a day depending on how frequently a developer merges their work to the mainline, while organization C follows a stricter deployment schedule and deploys once a workday excluding Fridays. Because of the extensive testing that happens in organization C, they deploy once a day, and deployments require additional preparation to include release notes and more testing. While these deployments occur on a daily basis, organization C developers avoid deploying close to weekends because "*if we deploy things on Fridays, it's when we're winding down for the week. And if we encounter any unexpected behaviour, especially maybe our European customers encounter unexpected behaviour, that means they're reaching out to us at 2 in the morning on Friday night. I don't have a team in place to be able to handle those requests*" - P11.

The differences in who is able to deploy are a result of practice perception, with some organizations valuing security over a culture of change ownership. The differences in deployment schedule are also due to practice perception because organization C values being able to handle customer issues in business hours over the possibility of breaking the build or pushing a breaking change to production before a period of extended inactivity.

6.4 Discussion

In this section, I discuss the significance of the factors we identified that impact the implementation of continuous practices. I also relate these factors to the overarching research question and illustrate how they fit into our theory that I will develop in Chapter 7.

6.4.1 Study Contributions

This study has two major contributions. First, we conducted a literature review that helped us map claimed benefits and challenges to the individual continuous practices (see Table 2.1). The second contribution is the identification of the factors that impact the differences in continuous practice implementation across three organizations (see Table 6.1).

Mapping claimed benefits and challenges to individual practices

Up until this study, most literature ascribed the various benefits and challenges to the paradigm that is "continuous practices" without delving into why these phenomena were occurring in a continuous context or whether non-automation-related continuous practices were necessary for the successful implementation of the continuous paradigm. For instance, Vasilescu et al. [10] indicate that continuous integration is correlated with higher software quality and developer productivity. However, based on their selection criteria, continuous integration only refers to the second continuous practice "automate the build" as they investigated open source projects that use Travis CI. Even in the case of human-centric phenomena, such as the studies of Gupta et al. [22] and Souza and Silva [21], the project selection criteria implies the only continuous practice considered is automation and is referred to as continuous integration. Not considering the other continuous practices

risks and not including non-technical practices (such as commit patterns, communication practices, etc.) reduces the observation of the continuous integration paradigm from socio-technical to purely technical.

Our literature review (featured in Section 2.3 and summarized earlier in Table 2.1) served two purposes: to establish our understanding of the state of the art with respect to the benefits and challenges related to continuous practices as preparation for our in-depth study in this chapter, and to attribute the various claimed benefits and challenges to the individual continuous practices in order to view the continuous paradigm through a socio-technical lens. Mapping the claimed benefits and challenges to the individual practices allowed us to establish a baseline which we could use to identify when the organizations deviated from what the literature considered a typical implementation of a continuous practice and whether that deviation was intentional. Furthermore, this baseline allowed us to understand what the different organizations expected from each of the practices, and the unique challenges they faced that corresponded to individual practices as opposed to the *continuous paradigm*.

Factors that impact continuous practice implementation

We observed the impact practice perception, project context, and tool limitations had on implementing the different continuous practices. *Practice perception* comprised how developers understood, interpreted, and prioritized a continuous practice. *Project context* reflected the impact of the project's requirements and nature on the implementation of a continuous practice. Finally, *tool limitations* illustrated how the constraints and limitations within the automation itself affected how a continuous practice was implemented.

Identifying these factors has major implications for both researchers and practitioners. On the research side, the differing practice implementations meant that there was no "correct" way of implementing a continuous practice. Out of the three organizations we studied, two claimed to be continuous and were satisfied with how they were implementing continuous practices, and the third claimed to be transitioning to a fully continuous paradigm but viewed their approach so far as satisfactory. Pointing out that the practices the organizations were implementing were an incorrect version of the original practice would not be a good enough reason for them to change what they were doing, as it would not align with the benefits they wished to maximize by implementing a practice. Consequently, research that aims at identifying

incorrect implementations of continuous practices [60] or automation configurations [66] based on predefined anti-patterns [59] must make an effort to relate these anti-patterns to the organization's context and use the factors we identified to determine whether an anti-pattern is indeed an undesired practice.

On the practitioner side, the factors resulting in different implementations of the same practice meant that care should be taken when interpreting a practice. Practitioners should also consider the practice as a way to augment their projects and organizations instead of implementing a practice simply because it is considered a "best practice". For instance, small and frequent commits are a recommended practice according to Fowler [9]. However, small and frequent commits also result in disjointed software changes from the tester's perspective, which forces them to throttle these changes until a feature is fully implemented and ready to test. This throttling is a situation we encountered in organization C.

6.4.2 Implications for My Dissertation

Practice perception, *project context*, and *tool limitations* revealed relationships between the different entities that interact within the continuous software engineering space. Automation does indeed influence developers (e.g., build durations vs. context switching or blocking) and can be influenced (or configured) based on developer perceptions and priorities. In combination with the insights from Chapter 5, we had a better understanding of the dual nature of the automation-developer relationship.

The practices a team adopts may condition developers into behaving in a certain way that is considered efficient for the project. The practices, as part of the development process, offered an in-depth investigation of how developers interpret these practices, how they choose to implement them, and why (process-developer relationship).

Automation is generally configured to support the process developers operate within, and their perception of its benefits is heavily influenced by contextual project factors such as a project's nature, requirements, or infrastructure constraints. Thus, we also gained insights into how and more importantly *why* automation has an impact on the practices (automation-process relationship). These relationships form the basis for our socio-technical theory of continuous practices that we develop and present next in Chapter 7.

6.5 Limitations and Threats to Validity

In this section, I discuss the limitations of this study and how we attempted to mitigate them. I use the total quality framework by Roller and Lavrakas [106] to break this section down in terms of credibility, analyzability, transparency, and usefulness.

6.5.1 Credibility

Credibility is related to how accurate and complete the data collected is. It encompasses two aspects: scope and data gathering.

Scope

Our study's scope was limited to three organizations that provide software-as-a-service solutions in our immediate geographic area. While we did employ an opportunistic interviewee recruitment strategy within the organizations which may bias our results in favor of interviewees interested in our study's topic, we attempted to mitigate this sampling bias by including several interviewees from different roles across the organizations to ensure we were obtaining information from multiple perspectives.

Data Gathering

To enhance our construct validity, we followed an approach similar to the literature [12, 120] which operationalized continuous practices in relation to project contexts. In our case, however, this was done in relation to organizational contexts. To ensure the practices we investigated were grounded in a realistic software engineering context, we used the practices listed by Fowler and Foemmel [9], upon which Humble and Farley further elaborated [26].

To reduce the impact of researcher bias introduced when a single researcher gathers/analyzes/interprets the majority of data, we did the following:

- **Interviews:** For each of our interviews, there were always two researchers present. We used a list of prepared questions (in Appendix B) while asking for elaboration where necessary. We also made sure to ask participants to repeat, explain, or rephrase unclear segments of their responses as a form of member checking.

- **Transcript coding:** We made sure that two researchers coded each transcript, with the primary author coding all transcripts. We used the same approach as Heikklä et al. [119] whereby two coders would come together to conduct an "agreement session" once a transcript was coded. In these agreement sessions, we discussed the differences in our coding schemes regarding segments we disagreed about and eventually agreed on a singular code that reflected our interpretation of the data.
- **Thematic analysis:** Similarly to how we coded the transcripts, our thematic analysis was conducted in a large agreement session. The themes we generated based on our codes were later included in a survey that we sent back to our interviewees to ensure that the themes resonated with them.

Afterwards, we sent copies of our case reports to the corresponding organizations to ensure we had not misunderstood the contexts within which they were operating. Finally, we sent the organizations a copy of the study to ensure we did not misrepresent or misinterpret the collected data.

6.5.2 Analyzability

Analyzability pertains to the accuracy of the analysis we conducted and the interpretations drawn from it. It consists of two aspects: processing and verification.

Processing

We recorded the interviews with the interviewees' permission and transcribed them using a voice-to-text transcription service. The transcripts were verified by a human who listened to the actual interviews and compared them against the transcripts to minimize errors.

Verification

To ensure the information we were collecting was not biased by a single perspective, we triangulated by recruiting participants from a cross-section of organizational roles. We included the themes that emerged from our thematic analysis process in a member-checking survey that we sent back to our interviewees. Only 8 respondents of our interviewee pool of 18 responded (response rate: 44%). In addition, once we had created the organization case

profiles, we sent those back to the corresponding organizations for feedback to make sure we properly understood the context and were not misinterpreting interviewee responses. Finally, we sent the paper itself to our participant organizations for the same purpose. The responses to our survey, case profiles, and paper were positive, and we adjusted our findings based on the feedback we received during this process (see Section 6.2.5).

Where possible, we augmented our qualitative analysis by mining team development activity logs within the different organizations. However, these logs were only useful for verifying some continuous practices (2, 4, 5, and 6), not all of them. These logs are also included in our reproduction package.

6.5.3 Transparency

We attempted to include as many rich details and quotes as possible to enhance the transparency of our analysis. We also provide a reproduction package that contains the following:

- our codebook,
- the themes we found,
- the member-checking survey,
- the case reports we generated from the transcripts,
- the anonymized, sanitized version of the development activity logs, and
- our analysis of the log data.

6.5.4 Usefulness

Usefulness is an indicator of how actionable the results from a study are, and their ability to transfer to other contexts (external validity). Our results were focused on the organizations we studied, their workflows, and widely accepted continuous integration practices. However, the main point we made in this study was that context is extremely important. Context must be considered when investigating continuous practices, thus organizations that are similar to the ones we investigated make the best candidates for our results to be transferable. Furthermore, our approach to investigating continuous practices may be useful for organizations who are contemplating adopting continuous practices.

Table 6.3: Comparing CI practices at organizations A, B, C. We list, per CI practice, how that practice is implemented and rationalized, what trade-offs are perceived, and why its implementation differs (italics). Organization codes in parentheses indicate data from the organization supports the finding.

How a Practice is Implemented (RQ1)	Rationale/Benefit (RQ2)	Challenges/Trade-Offs (RQ3)
Maintain a single source repository		*Differences due to practice perception*
• Logical project separation across repositories (A). • Microservice-like architecture (B). • Inconsistent grouping of dependencies and main project (C).	• Minimizes merge conflicts (A). • Mono-Repo rationale not obvious at the time (B). • Mono-Repo tool support not available at the time (C).	• Boundaries not always clear (A). • Dependency management is complex (B). • Workflow duplication (B).
Automate the build		*Differences due to project context*
• Via Jenkins (A, C). • Via AWS CodePipeline and AWS CodeBuild (B).	• Consistency (A, B) • Reproducibility (A, B) • Swift development (A). • Delegates repeatable tasks to build (A, C).	• Build can be a bottleneck (B).
Make the build self-testing		*Differences due to tool constraints, practice perception, and project context*
• Unit tests in PR builds (A, B, C). • Regression tests in mainline builds (A, B, C). • Integration tests in PR builds. Local, PR, mainline, QA, then product manager tests (C).	• Reliable bug detection (A). • Ensures consistency (B). • Minimizes breaking changes (C).	• Fast tests due to limited coverage (B). • Limited by infrastructure (B). • UI difficult to test. Cost-Benefit of adding automated UI tests not clear (A). • Longer builds (C). • Decreased velocity (C).
Everyone commits to the mainline every day		*Differences due to practice perception*
• Frequent commits on most projects (A). • Daily commits are norm (B). • Feature branches with daily commits encouraged (C).	• Reduces merge conflicts (A, B, C). • Rapid user feedback (B).	• A developer works in isolation until merge (C).
Every commit should build the mainline on an integration machine		*Differences due to practice perception*
• Builds on PR and mainline levels (A, B, C).	• Ensures test execution (A). • Faster feedback (B). • Minimizes regression bugs (C).	• Bottleneck due to PR review (A). • Frequent PRs swamp reviewers (A). • Build infrastructure is bottleneck (C).
Keep the build fast		*Differences due to project context and practice perception*
• One minute to a few hours (A). • Ten minutes or less (B). • Ten to twenty minutes (C).	• Less developer context switching (A, B). • Reduces developer blocking (B).	• Reliance on external services in build is time-consuming (A). • Infrastructure runs tests inefficiently (C). • Decreases perceived velocity (C).
Test in a clone of the production environment		*Differences due to practice perception*
• Environment flags in the Dockerfile (A, B, C).	• Prod and dev environments in sync, consistent tests (A). • Ensures consistent tests across builds (B). • Application execution consistency (C).	• Scalability issues due to resource restrictions on dev environments (A). • More maintenance (B). • Complete clones hard due to non-transferable aspects (B, C).
Make it easy for anyone to get the latest executable		*Differences due to practice perception*
• Docker facilitates project spin-up (A, B, C).	• Reduced build complexity (A, C). • Easier onboarding (B).	• Reducing build complexity requires effort (B).
Ensure that system state and changes are visible		*No difference*
• Dedicated Slack channel (A, B, C).	• Keeps everyone in the loop (A, B, C).	• Notification fatigue from frequent builds (A). • Requires discipline and collides with responsibilities (B, C).
Automate deployment		*Differences due to practice perception*
• Via manual trigger. Not all can trigger for specific projects (A). • Via manual trigger. Everyone but newest members can trigger (B). • Via manual trigger once a day, except on weekends. Not everyone deploys (C).	• Minimizes deployment effort (A). • Maintains security (A, C). • Consistency and repeatability (B). • Everyone deploying creates ownership culture (B). • Minimizes deploying breaking changes over period of non-activity (C).	• No testing in deployment builds to reduce build time (B). • Requires preparation (C).

Part III

Synthesis: ADEPT

7 The ADEPT Theory

There is a way out of every box, a solution to every puzzle; it's just a matter of finding it.

Cpt. Jean-Luc Picard
Star Trek: The Next Generation

In this chapter, I discuss how we used the three studies we conducted to build an explanatory theory for the phenomena observed. I used the dimensions of **A**utomation, **D**ocumentation, **E**nvironment, **P**rocess, and **T**eam Member to interpret human-centric phenomena we observed in these studies, which resulted in the ADEPT theory. The ADEPT theory combines the constructs and relationships we observed in our case studies to form both an interpretative as well as an exploratory instrument. Furthermore, ADEPT can interpret existing phenomena and frame future human-centric automation research in software engineering.

7.1 Motivation

Software engineering, and by extension continuous software engineering, has always been a human-driven activity. It is software developers who decide what issues to tackle, who come up with creative solutions to these issues, who decide what constitutes "acceptable code", and whether or not to accept the build results [16, 17]. No matter what standards, practices, and tools are used, they cannot replace the creative and human-centric aspects of the software development process because it is ultimately human decision-making that drives this process forward [18].

7.1.1 How Does Automation Impact the Software Development Process?

There is an abundance of literature on how automation impacts the software development process, as shown in Chapter 2. We know that projects that use automation experience an increase in their levels of software quality and development velocity [10, 64]. We also know that when projects use automation to augment their development workflows, that increase in development velocity plateaus afterwards with no apparent reason [12]. Furthermore, we know that while automation itself does increase development velocity, it is not the sole factor [6]. Human and environmental factors such as developer perceptions of continuous practices and their consequent implementation of these practices have given rise to the assumption that achieving the best workflow utility from automation comes as a result of the implementation of continuous practices that—ostensibly—draw forth the full capabilities of the tool. In response, researchers have reformulated their assumptions and started investigating *continuous practice* [60] and *automation tool* [66] anti-patterns. The underlying assumption driving this shift in approach is likely the claim that there is a correct way of implementing continuous practices as well as incorrect ways [59, 121] if an organization wishes to achieve their touted benefits.

However, as discussed in Chapters 4 and 6, we know that a lot of the decisions organizations make when implementing continuous practices depend heavily on the *context* within which they are operating and the *perceived continuous benefit* they want to maximize. The humans involved (namely developers, testers, program managers, and other influencing stakeholders) drive the benefits derived from the implementation of continuous practices–and in turn, automation–which influences how these practices are implemented. Thus, if studies aim to investigate automation without considering human factors (or other contextual factors), they miss constructs which could influence their results or render their recommendations too abstract to apply in a specific context.

7.1.2 How Do Developers Interact with and React to Automation?

Unlike the large amount of literature focused on automation and the development process in a continuous context, the literature on how humans interact

with continuous automation is sparse, as discussed in Chapter 2. For instance, we know there is a relationship between developer mood and broken builds [21] in the sense that broken builds correlate with negative commit messages, which in turn produce builds that are more likely to fail. We also know that adopting continuous automation–in this case, Travis CI–may have a negative correlation with a project's ability to attract and retain developers [22]. Projects that use Travis CI may fall victim to its pain points [79], which may drive contributors away. Furthermore, automation-specific phenomena such as misuse have been found to occur in a software development context where developers place too much trust in an automation's results and in return feel a false sense of confidence [77].

However, much like studies that focus on how automation impacts the process, these previous human-centric studies also aim towards generalizability and tend to abstract away possible confounding contextual factors. From my studies in Chapters 5 and 6, we know that there are several factors developers consider when using automation. These factors are primarily based on the perceived benefit they want to maximize and the context within which automation is used. For instance, automation used for a data-centric project will differ vastly in how it is configured and how developers define its *benefit* from an automation that is meant for a user-centric project [6]. Thus, similarly to how automation's impact on the development process cannot be considered without contextual factors, we cannot consider automation's impact on developers in a vacuum.

7.1.3 Why Is a Theory Necessary?

I use the definition of a theory as discussed by Stol and Fitzgerald [122] and reinforced by Varpio et al. [123]. A theory is a set of propositions that connect a group of constructs based on the relationships between them. Constructs can be operationalized in conjunction with the propositions to generate testable hypotheses.

There are four reasons why we need a theory to represent the insights we have gathered so far. The first and possibly the most important reason: we needed a way to interpret the phenomena we were seeing in practice [8, 6]. For instance, when testers throttled what to them seemed like frequent, unrelated changes in order to test a fully implemented feature, or when developers perceived the automation to be a bottleneck because of inadequate infrastructure resources.

The second reason for needing a theory is that most theories discussed in software engineering tend to be borrowed from other domains and as such do not incorporate the nuances a software engineering context requires when using a non-software engineering theory in a software engineering context. Software engineering is a socio-technical activity and theories that attempt to address propositions within it should address the three core components of socio-technical systems as laid out by Hall and Rapanotti [5]: human, technology, and environment. When Ralph et al. [124] showcase several theories from other domains and how they can be transitioned to a software engineering domain, we find that most of these theories, being human-centric or social in nature, do not capture *all* three components. For instance, Actor-Network theory addresses the human component, and can be made to address the technological one as well, but it is not capable of addressing the environment component as specific to software development [124].

The third reason we need a new theory is that when we did come across theories that addressed all three components, the focus was not on the human-automation relationship. Most software engineering theories attempt to interpret software engineering as an activity, from defining the problem to implementing the solution. However, we needed a theory that would focus explicitly on the interaction between human, process, and automation in the context of software development. For instance, the Tarpit theory [125] exhaustively represents the different entities involved in carrying out a software project and identifies the relationships between them with the purpose of representing how software engineering as an activity occurs. Similarly, Hall and Rapanotti's design theory [5] represents the software engineering process as a problem-solving activity. Complexity theory combines the three concepts, but the focus is more on how a software project can be modelled as a system of complex, interconnected components as interpreted by Ralph et al. [124].

Finally, the fourth reason we need a theory is that we should not move from empirical generalization directly to suggesting tools or guidelines, which Stol and Fitzgerald consider a "*shortcut*" [122], instead one should aim to move from empirical generalization to formal theory formation, test hypotheses based on the resulting theory, and create best practices from the hypothesis test results. We hope to build a theory that can be used to frame future research in this area, interpret existing research, and generate meaningful and testable hypotheses.

This chapter synthesizes the findings from the studies presented in Chapters 4, 5, and 6 to constitute the primary contribution of this dissertation. Our first study highlighted how automation was featured in documentation [7]. Our second study showcased how developers treated non-functional requirements that were facilitated by automation [8]. Finally, our third study investigated the various individual continuous practices in depth as they relate to automation, and how they impacted development workflow [6]. My synthesis is also informed by the previous literature discussed in Chapter 2. It combines the answers to both overarching research questions, and uses them to build a theory that can be used to interpret existing research as well as frame future research in the continuous-automation area.

RQ1: How does the presence of an automated pipeline impact the software development process projects use in a continuous context?
RQ2: How does the presence of an automated pipeline impact developer decision-making in a continuous context?

The theory presented in this chapter is a more detailed version of the accepted paper at ICSE NIER 2021 [23].

7.2 Theory Structure

Over the course of this dissertation, we have conducted several studies that investigated the nature of how automation interacts with both development workflow, and developers themselves within the context of continuous practices (Chapters 4, 5, and 6). We built the constructs and propositions contained within this theory from these studies [7, 8, 6].

We named our theory **ADEPT**, with each initial representing the different interacting components we observed throughout our investigation. **A**utomation, **D**ocumentation, **E**nvironment, **P**rocess, and **T**eam Member all combine to give a socio-technical representation of how automation interactions with both developers and the processes within which they operate. The automation represents the technical aspect of continuous software engineering, the project's requirements and infrastructure represent the environmental aspect within which the project is being developed, and the team member and process represent the human aspect in terms of individuals and protocols governing their interaction according to Hall and Rapanotti [5].

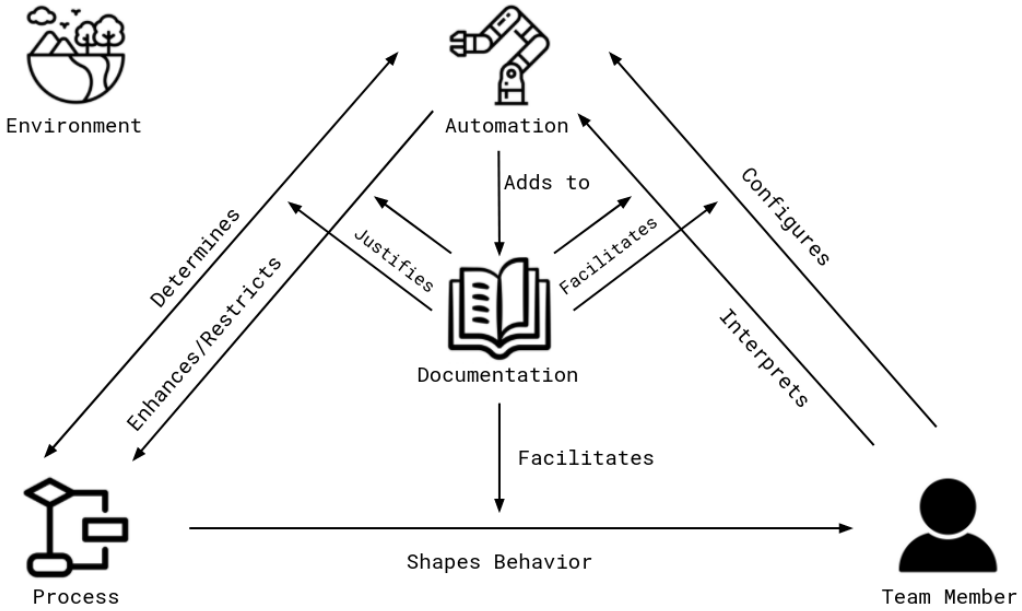

Figure 7.1: Visual representation of the ADEPT theory: icons represent constructs, and edges represent propositions

ADEPT is a *meso-level theory* because it attempts to explain the relationship between automation and developer behaviour while still keeping the process in consideration [83]. ADEPT is also explanatory because it highlights the relationships we observed between its different constructs, which may not have been fully explored in the literature. A visual representation of ADEPT can be found in Figure 7.1.

7.2.1 Constructs

The main constructs in ADEPT represent the various entities we observed when investigating how developers, automation, and processes interacted with each other over the course of the studies we conducted:

Automation

This construct refers to any type of automated tool that is triggered when a team member pushes a commit to a software project. It is the abstract form

of the automation described by Humble and Farley [26] as well as Forsgren et al. [121], which refers to an automated build system, more commonly referred to as a pipeline. An example of automation is Travis CI, which was studied by Beller et al. [126]. This construct can include jobs that trigger once a commit is pushed to a repository, or when a pull request is created or merged. It can also be expanded to include tools that run within generic build tools, such as dependency management tools, build tools, testing tools, or even bots.

Documentation

This construct represents explicitly externalized information available to developers regarding the project they are working on; specifically documentation on the tools used to scaffold a project, which are typically in the form of an official artifact (or artifacts, depending on how thoroughly a project is documented). It includes artifacts such as a README file or contribution guidelines that communicate technical and process knowledge [7, 95]. It can also include artifacts of a finer grain that document the various changes happening throughout a project's development lifecycle, such as commit messages, pull request descriptions, issues, etc. Finally, it can also include artifacts that communicate knowledge about the build process–typically stored in tacit form in a developer's head–such as build scripts (also known as infrastructure-as-code) [127].

Environment

This construct encompasses two environmental aspects we observed that influenced the relationships between the other constructs. The first is infrastructure constraints. Infrastructure constraints can limit an automation's operating resources, which can impose restrictions on the benefits it can supply to the development process (e.g., efficiency, velocity, etc.).

The second aspect is project requirements or domains. Project requirements/domains represent the nature of the projects being developed. Different types of projects will have different expectations from automation, process, and other constructs. For instance, a data-centric project with the goal of data processing will have different expectations from its attached automation, and its developers will perceive the automation based on the project's needs (e.g., data collection from external sources, data integrity tests, etc.).

On the other hand, product-centric projects with the goal of producing a product an end user can utilize will have different expectations in the sense that the automation's primary purpose is to provide rapid feedback, and generally facilitate packaging and deployment. Similarly, a project's non-functional requirements can drive automation functionality as discussed in Chapter 5.

Process

This construct represents the steps and activities a feature goes through from its inception, to development, to testing, until it lands in production. It is intended as an abstract representation of the software development workflow regardless of the paradigm or methodology a team follows. However, with the widespread adoption of the pull request model [81] in both open source and industry and the fact that this theory is based on our previous studies that involved organizations that used the pull request model, we assume the process construct is similar in its structure and workflow to the pull request model. A process can include a pipeline that automates some aspects of its steps, but not all process steps are automated.

Team Member

The core of the ADEPT theory, this construct captures the different types of humans that interact with automation in one form or another. Software developers, DevOps engineers, testers, reviewers, quality assurance personnel, and even product managers fall within this construct. While product managers will only interact with automation in a limited manner, we found they are also affected by the impact automation has on development workflow [6]. We included this construct based on studies of the development process [16, 17, 105, 26, 121], studies of the commonly used pull request model [81, 10, 12], and our own investigation [6, 8].

7.2.2 Propositions

The propositions presented in this section link the constructs discussed above to each other, and represent phenomena we observed in our previous studies [7, 8, 6] as well as phenomena discussed in the literature.

Team member configures automation

Software developers or DevOps engineers are typically responsible for configuring the automation attached to a software project as well as its supporting infrastructure. A configuration's complexity is a direct result of the way developers and DevOps engineers choose to configure builds and other relevant automation. In this proposition, we can capture possible instances of automation *abuse* where automation is configured in such a way that it actually decreases the efficiency it is meant to provide as a benefit [76]. This proposition also captures configuration issues and smells [66]. Finally, this proposition captures how differences in configuration we observed in our previous studies regarding non-functional requirements and tests [8, 6].

Automation is interpreted by team member

This proposition captures the effect automation may have on the team members it interacts with regarding their decision making process when reviewing code or interpreting build/tool results. Here we may observe cases of automation *misuse* (the over-reliance on the results produced by automation such that they are almost always unquestionably accepted) as well as *disuse* (the non-reliance on automation results because of lack of trust) as discussed in Chapter 5 [8]. For instance, the observations in Souza and Silva's work [21] regarding build impact on developer sentiment are encapsulated by this proposition. It also includes overconfidence due to developers putting too much trust in automation results [77].

Process shapes team member behaviour

The process, typically defined by core team members (or team leads), determines the set of activities team members are *expected* to follow in order to produce software. Typically, this process is ingrained in team culture and may be documented as part of the tacit process knowledge. In this sense, the process guides team members and instructs them to behave in a predetermined way [7]. For instance, developers and reviewers may be expected to check code contributions against a checklist of acceptance criteria. Another example is reviewers resorting to manual usability testing because the automation cannot perform the task to the reviewers' satisfaction [6]. This proposition captures developer behaviour as dictated (or shaped) by the process within which they operate.

Process determines automation (or vice-versa)

In a perfect world, it is generally expected that the structure of the development process plays a role in determining the type of automation attached to the project. For instance, projects that follow the pull request model may prioritize automation that triggers on pull requests and compares them to predefined pull request criteria, whereas projects that prefer trunk-based development may prioritize automation that triggers on commits and runs the relevant tests. However, it is also possible that the opposite may happen, and a process is adapted or constructed around a team's desire to use a particular tool or automation, which may be an indicator of automation abuse [76]. This proposition captures why specific tools are chosen by developers for a particular project and why automation is configured the way it is. Contextual factors also come into play here because infrastructure constraints and project requirements influence the type of tools a project uses and how automation is configured.

Automation enhances/restricts a process

This proposition captures the impact automation has on the development process. While automation is typically added to a project to enhance some aspect of it (quality, onboarding, etc.) [9, 1, 10], it is possible that automation results in negative effects as well. For instance, badly configured automation can result in problems such as bottlenecks in build queues due to complex builds that take too much time [59, 6].

Automation adds to documentation

One phenomenon we observed in our study [6] is automation being used as a form of documentation. Developers and DevOps engineers would direct newcomers to project build scripts to familiarize themselves with the project's build process as a way of communicating that tacit knowledge. This proposition captures this phenomenon and allows for different types of artifacts to be included, such as configuration files (infrastructure-as-code) and other similar artifacts.

Documentation facilitates a team member's interpretation of automation

Depending on the automation configuration and the tools being used, it is possible that the output requires documentation to facilitate interpretation. For instance, Travis CI builds consist of large log dumps that require a developer to read through them to find where a problem occurred. Two types of documentation fall under this proposition: tool documentation which can be used to properly configure a tool and orient new tool users, and infrastructure-as-code documentation where developers use configuration files and build scripts as a way to communicate tacit build process knowledge. This proposition captures how developers use documentation (either tool documentation or tool configuration) to facilitate their understanding of automation results as we observed in our study of continuous practices in Chapter 6 [6].

Documentation facilitates the process shaping a team member's behaviour

This proposition captures the function of README files and contribution guidelines in open source projects, as well as other forms of documentation in non-open source projects that communicate process knowledge to developers [7, 95, 6]. These artifacts communicate to team members the activities and steps they are required to follow in order to be a functional member of the development team, and are listed as a mitigating factor to barriers newcomers face in open source projects [105].

Documentation justifies how automation enhances/restricts a process

Documentation can, to an extent, provide developers with an indicator of *why* a development process was structured the way it was. Consequently, it can also shed light on why a tool or automation was configured the way it was in relation to the process. This proposition can explore why the process functions the way it does, why tool/automation selection and configuration choices were made, and ultimately the automation/tool's perceived role and effects on a process.

7.3 Theory Validity

In this section, I describe how we established that ADEPT meets the different criteria for a theory as laid out by Sjøberg et al. [128]. We use existing studies to demonstrate ADEPT's explanatory power and utility, and discuss the remaining four criteria separately in their own subsections.

7.3.1 Explanatory Power and Utility

We use ADEPT's constructs and propositions to determine whether it meets the requirements for explanatory power and utility. We use two examples for workflow related phenomena, and two for human centric phenomena.

Vasilescu et al. [10] find that having automation in the form of Travis CI attached to open source projects is typically correlated with a positive impact on both software quality and developer productivity. They find that the projects they study tend to have fewer bugs overall and experience a higher pull request merge rate than projects without automation. The proposition *automation enhances/restricts process* captures these findings, with both findings being examples of when automation *enhances* a development process. However, what is not clear is the impact adopting Travis CI has had on developers, both directly when they interact with it, and indirectly with respect to behavioural changes they need to make in order to accommodate how Travis CI operates. Productivity is a function of human behaviour, with the human being its core concept [51]. Considering the increases in project quality and developer productivity through ADEPT we observe that there are other factors that, in the best of cases, open up new avenues for investigation, and in the worst of cases confound study results.

Another process-centric example is that of Ståhl et al. [103], which prescribes a method to establish traceability between different project artifacts and strengthen documentation. While stronger documentation plays a role in facilitating the application of CI practices according to ADEPT's propositions (*automation enhances/restricts process* and *documentation justifies how automation enhances/restricts a process*), it is not clear how these practices have impacted human behaviour or developer perception of the automation. Is it still a tool meant to facilitate development and increase efficiency, or has it evolved into a documentation tracking mechanism? Both these examples focus on the technology and environment aspects of socio-technical systems,

with the process being an environmental aspect, but they do not adequately consider the role human developers play.

Similarly, human-centric studies also tend to focus on only a subset of socio-technical concepts. The study conducted by Pinto et al. [77] observes developers experiencing a false sense of confidence when they trust tests too much. In the human-automation domain, placing too much trust in automation is commonly referred to as misuse or complacency [76, 65], and is captured in the proposition *automation is interpreted by team member*. However, there are other relationships that ADEPT highlights which may impact this phenomenon. For instance, does the false confidence occur because of how the automation was configured (*team member configures automation*), or because the process dictates a certain behaviour when interacting with automation (*process shapes team member behaviour*)?

Another example of a human-automation study is that of Souza and Silva [21] when they observe negative commit message sentiment following failing builds. The phenomenon is captured by another of ADEPT's propositions: *automation is interpreted by team member*. However, like previous phenomena discussed in this section, it does not exist in a vacuum. Is the negative sentiment solely a result of broken builds, or is the build configuration also culpable if the build results are not actionable, thereby transforming the bug fixing process into an exercise in trial and error (*team member configures automation*)? Could it be that the process mandates a "do not break the build" culture that leads to frustration and consequently negative builds when builds stay broken (*process shapes team member behavior*)? These examples consider both the technology and human aspects of socio-technical systems, but do not consider the environment's part in the phenomenon, which limits the possible reasons for a phenomenon's occurrence and introduces confounding variables.

7.3.2 Testability

Testability or falsifiability represents the degree to which a theory's propositions can be disproved using empirical evidence. The constructs we use in our theory are based on the different entities that interact with each other in the software development process and are well established in literature [14, 81]. They are also entities we observed interacting with each other in the empirical studies we conducted [7, 8, 6]. However, the propositions we introduce are primarily based on phenomena we observed empirically within a specific

context. For instance, we observed a team that had to overhaul their development process due to switching to a different form of automation (in that case, Kubernetes). While the proposition "*automation determines process*" may hold for this case, other teams may experience a smoother transition or have a process flexible enough to accommodate a change in automation. Context, in terms of future studies, has a direct impact on falsifiability in this case.

7.3.3 Empirical Power

Empirical power is an indicator of how well a theory explains the reasons *why* a phenomenon occurs. Two aspects govern empirical power, namely analogy and explanatory breadth. Regarding analogy, we built our theory using existing empirical studies to form our constructs and propositions. Our constructs are based on the well established pull request model [81] and how automation is integrated into it [126], and our propositions are based on previous empirical studies in the literature [16, 17] as well as our own [7, 8, 6]. We illustrate explanatory breadth by using ADEPT to frame existing empirical studies and reason as to why their observed phenomena take place, as seen in Section 7.3.1. Furthermore, the next chapter of this dissertation investigates the extent of ADEPT's maturity by applying it to continuous software engineering studies published in the three most prestigious conferences in software engineering (ICSE, ESEC/FSE, ASE) in the past five years.

7.3.4 Parsimony

Parsimony is an indicator of how minimalist a theory is with the expectation that less is better. In this sense, a parsimonious theory contains only the necessary amount of constructs and propositions for it to serve its purpose. To make ADEPT parsimonious, we chose not to include the project's source code as a construct as it would increase the theory's complexity due to the larger number of dimensions. Source code also varies vastly between software projects, and including it would have forced us to weigh down ADEPT with a large number of assumptions, which would make it harder to meet its next criterion: generality.

7.3.5 Generality

Generality reflects the breadth of a theory's scope and how independent it is from its setting. ADEPT is still a work in progress and will likely grow in the future. However, the constructs and propositions we used to build it are rooted in both empirical studies we conducted as well as previous literature. Furthermore, in Section 7.3.1, we demonstrated its utility in both open source and industry settings. To clarify, because ADEPT was based primarily on studies conducted in specific industry settings, we do not claim it is transferable in its entirety to other industry contexts, but we hope its constructs and propositions are general enough to frame future CI research moving forward.

8 Investigating ADEPT's Utility: An Exploration of the Literature

There's theory and then there's application. They don't always jibe.

Cmdr. Geordi La Forge
Star Trek: The Next Generation

This chapter is based on an exploratory study in collaboration with Margaret-Anne Storey, Enrique Larios Vargas, and Alessandra Maciel Paz Milani. Throughout this chapter I recount how we conducted a literature mapping study that examined software engineering literature for the past five years. I discuss how we used ADEPT [23] to frame previous research in continuous software engineering to determine ADEPT's utility and illustrate how it can be used to represent propositions in contexts other than the ones within which ADEPT was built.

8.1 Motivation

Continuous software engineering is a socio-technical endeavour. It is the combination of social and human factors along with technological factors within a larger environmental context that facilitates the rapid development, assessment, and production of software. Human developers write and evaluate code, and make decisions on whether to merge code changes to the codebase. These human developers operate within a set of practices (the continuous software engineering paradigm) as a form of human-environmental context interaction. They also use automation to facilitate, and in some ways augment, their development workflow, and make decisions based on the re-

sults of these automated tools (human-technological interaction). Finally, the automated tools they use are constrained by the project in question. A project's characteristics impose tool limitations that impact the extent to which quality is checked within that particular project, and a project's infrastructure can limit the effectiveness and efficiency of these tools (technological-environmental interaction). While tool limitations are one factor that may impact quality, how a project defines quality is the primary factor that impacts how quality checks are implemented, which may drive tool adoption. A team member's interpretation of "acceptable quality" is also a factor. Studying continuous software engineering phenomena therefore involves considering social, technological, and environmental aspects together as inextricably interconnected components of a larger software-producing process.

In Chapter 7, we introduced the ADEPT theory that aims to represent these interconnected components and how they relate to each other. However, ADEPT was derived from the propositions in the particular organizational contexts within which we conducted our previous studies. To test whether ADEPT can be used for different contexts in the continuous software engineering field, we conducted a literature mapping study to interpret existing phenomena using ADEPT's constructs and propositions. Furthermore, based on our experience from previous studies that used artifact-centric research strategies [7, 6] and were unable to capture the full socio-technical nature of continuous software engineering, we aim to use ADEPT as a lens to explore how recent research has explored continuous software engineering, and the research strategies used to investigate ADEPT's socio-technical propositions. To that end, we use ADEPT to answer the following questions:

RQ1: What are the most investigated socio-technical aspects of continuous software engineering?

RQ2: What research strategies are used to investigate propositions in continuous software engineering?

Determining the extent of ADEPT's utility and the maturity of its components would allow us to use its propositions to frame socio-technical propositions in continuous software engineering literature. Furthermore, it would allow us to identify possible socio-technical propositions that may have been overlooked by the literature so far, thus charting a research roadmap.

In this chapter, I discuss how we conducted a literature mapping study to collect continuous software engineering research literature that included possible socio-technical phenomena. I elaborate on the analysis we conducted and the results obtained. Finally, I illustrate how we used these results to

determine the utility of the ADEPT theory, and how ADEPT can be used to interpret continuous software engineering phenomena in different contexts.

8.2 Study Design

To explore the literature on continuous software engineering via the socio-technical lens ADEPT provides, we followed a literature mapping approach similar to the one laid out by Petersen et al. [129]. Our study focused on publications from the three most prestigious conferences in software engineering over a period of five years. In this section, I discuss how we sampled the literature and coded the different studies in terms of research methodology and the ADEPT constructs used.

8.2.1 Conducting the Search

As this study was exploratory, we selected the three most prestigious conferences in the software engineering community to examine their publications:

- International Conference on Software Engineering (ICSE)
- European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)
- Automated Software Engineering (ASE)

These conferences each produce between 90-120 papers on a yearly basis. For pragmatic reasons, we limited our sample to papers from these conferences for a period of time going back 5 years to cover 2016, 2017, 2018, 2019, and 2020.

8.2.2 Paper Screening

We examined the conference proceedings and included papers based on whether they met any of the following inclusion criteria:

- The paper addressed continuous software engineering practices directly as units of analysis, or the practices were a context within which the investigation was occurring:
  - **As a unit of analysis:** The paper investigated the continuous practices themselves. In this case, the paper seeks to investigate a claim or proposition directly pertinent to a continuous practice (or multiple practices). For instance, the work of Zampetti et al. [60]

regarding investigating anti-patterns in continuous software engineering practices treats the practices themselves (or rather the anti-patterns) as units of analysis.
  - **As a context:** The paper investigated a proposition that occurs within a setting that uses continuous practices. For instance, the work of Souza and Silva [21] about analyzing commit message sentiment in relation to failing builds involves the use of developer sentiment as a unit of analysis and their observations occur in a context where the continuous practice of build automation is used. However, the study must establish that its context is indeed continuous by controlling for the development practices used and checking that at least one of the ten continuous practices exists in the investigation's context, not merely claim it is so.
- The paper featured the build automation tool as either the unit of analysis or as context for a different unit of analysis:
  - **As a unit of analysis:** The paper investigated the build automation tool itself as the main construct. For instance, the work of Widder et al. [79] about conceptually reproducing and investigating Travis CI pain points features the tool (Travis CI) as a unit of analysis.
  - **As a context:** The paper investigated or proposed a tool or automation that runs within a build. For instance, the work of Beller et al. [130] that investigates the reasons for test failures within Travis CI builds uses the automated build tool Travis CI as a context within which they investigate tests (more specifically, test failures) as a unit of analysis.
- The paper was included if it investigated an actual proposition and produced research results as opposed to merely a plan of future research (excluding papers from doctoral symposiums).

Thus, for each paper in each conference's proceedings (5 proceedings per conference, one for each year), I applied the inclusion criteria to determine whether a paper presented a continuous software engineering phenomena or not. To prevent researcher bias, I also divided the proceedings between two other researchers: researcher A focused on the FSE and ASE proceedings, and researcher B focused on the ICSE proceedings. Then we conducted agreement sessions where I discussed whether a paper met the inclusion criteria with the corresponding researcher. Applying these criteria to the 15 proceedings yielded a sample of 46 papers out of the approximate total of

2039 distributed across the three conferences as seen in Figure 8.2.

8.2.3 Data Extraction and Mapping

To map the extent to which the papers addressed the socio-technical aspects of continuous software engineering, I coded every paper using the ADEPT theory's constructs and propositions shown in Figure 8.1.

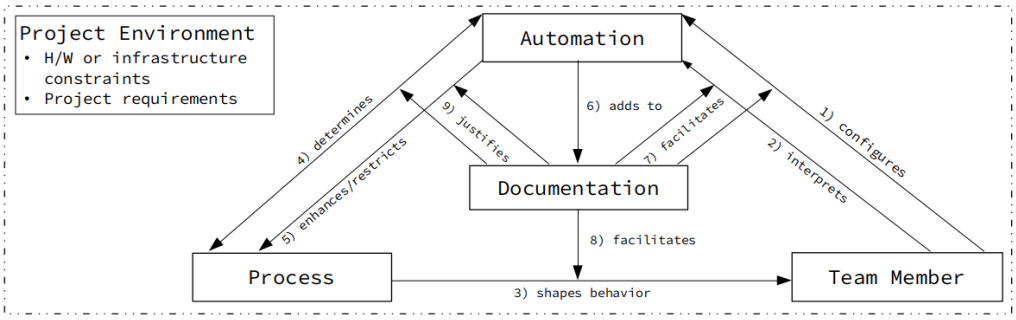

Figure 8.1: The ADEPT theory constructs and propositions.

To mitigate researcher bias, I triangulated with researchers A (coded the FSE and ASE papers) and B (coded the ICSE papers). We used the following interpretation when determining what components a paper was addressing with respect to the ADEPT theory based on the original ADEPT publication [23]:

- **Constructs**
  - *Automation:* The study focuses on the build tool as a unit of a analysis or in some other investigative capacity. The automation may be featured in the analysis either as its main focus or contextually. For instance, the work of Beller et al. [130] investigates reasons for test failures by inspecting tests that occur within Travis CI builds. Automation (Travis CI) is a contextual factor for the study.
  - *Documentation:* The study focuses on documentation as a unit of analysis or as part of the context studied. For instance, in our study on GitHub project contribution guidelines [7], we investigated their contents and the extent to which projects adhere to them using the contribution guidelines as a unit of analysis.

  - *Environment:* The study accounts for environmental factors when conducting their analysis. These can include infrastructure and/or project properties having an impact on builds and other similar approaches. For instance, our study on the factors that influence continuous practice implementation [6] investigates the impact external environmental factors such as infrastructure resources may have on build duration.
  - *Process:* The study produces insights about the process, treats the process as a unit of analysis, or includes it as a contextual factor. For instance, the work of Zhao et al. [12] where they investigate the impact of Travis CI on commit patterns and pull request merge rates features development practices as a unit of analysis.
  - *Team Member:* The study uses the team member as a unit of analysis, contextual factor, or attempts to produce insights about humans that are actionable for humans. The conceptual replication of the Travis CI pain points paper [79] is an example of a human being used as a contextual factor while considering Travis CI itself as a unit of analysis. In that paper, Widder et al. investigate pain points reported in the literature about automated build tools, and verify them using a survey of developers who had stopped using Travis CI.

- **Propositions**[1]
  1. *Team member configures automation:* Configuring a build tool (or a tool that lives within a build tool) is the focus of the study. For instance, the work of Gallaba et al. [66] investigates Travis CI configuration anti-patterns.
  2. *Team member interprets automation:* Human interpretation of an automated tool is the focus of the study. This can include pain points or other human-centric phenomena. For instance, the work of Widder et al. [79] investigates the pain points developers perceive when using Travis CI. Merely claiming the study is beneficial to humans does not warrant this proposition being assigned to a paper in our mapping.
  3. *Process shapes team member behaviour:* The study focuses on how developers behave in a continuous practices context with respect

[1]Numbers in the list correspond to numbers in Figure 8.1.

to the process. For instance, the work done by Hilton et al. [131] explores the different trade-offs developers make when they adopt continuous integration.

4. *Process determines automation:* The study investigates why automation was introduced to a project and how that relates to the development process the project follows. For example, the work done by Lamba et al. [132] investigates how and why developers adopt automated build tools in open source projects. Alternatively, a study coded with this label can investigate how a development process changes due to the introduction of automation.
5. *Automation enhances/restricts process:* The study focuses on the relationship between automation and the development process. For example, "does automation speed up pull request review times?" or "does automation create a bottleneck in the process?" For instance, the work done by Vasilescu et al. [10] explores how using Travis CI correlates with faster development throughput and higher code quality, and the work done by Zhao et al. [12] explores the changes in commit and pull request patterns after adopting Travis CI.
6. *Automation adds to documentation:* The study looks at how automation is used to transfer knowledge. For instance, our study explored the factors that impact how continuous practices are implemented [6], and found that developers directed newcomers to build scripts and configuration files to familiarize them with the project infrastructure and build process.
7. *Documentation facilitates how:*
   - *Team member configures automation:* The study investigates how documentation is used as a factor in how team members configure automation. For instance, Henkel et al. [133] explore using curated DockerFiles as *Golden Rules* to guide developers when writing their own DockerFiles.
   - *Team member interprets automation:* The study investigates how documentation is used as a factor in how a team member perceives an automated tool and interprets its results. For instance, our previous study on the factors that impact continuous practice implementation [6] also found that team members refer to the build scripts and configuration files to understand how a build works, and subsequently interpret the

results.

8. *Documentation facilitates how the process shapes team member behaviour:* The study investigates how documentation factors into how developers implement continuous development practices. For instance, our study of GitHub contribution guidelines [7] revealed that the contribution guidelines did not cover certain aspects of the development process, such as reopening issues and pull requests.
9. *Documentation justifies why:*
   - *Automation enhances/restricts process:* The study investigates the role of documentation in justifying either the positive or negative impact an automated tool has on the development process. For instance, a study can investigate how build scripts (in their role as documentation) can be used to detect build execution bottlenecks.
   - *Process determines automation (or vice versa):* The study investigates the role of documentation in justifying why a certain automated tool was incorporated into a development process. Conversely, a study can also investigate the role of documentation in justifying why a development process was structured around the desire to use and accommodate a particular automated tool. For instance, a study can use build scripts as units of observation to infer whether the process was built around the tool, or if the tool was incorporated into the process. The build scripts can be used as one of several data sources in this case to understand why development practices were implemented in a particular manner.

With respect to research methods, we used the Who, What, How framework developed by Storey et al. [108] to classify the papers in our sample in terms of research strategy. Similarly to how we coded the papers in terms of ADEPT constructs and propositions, we followed the same process for the research strategies using the following interpretation:

- **Empirical Strategies**:
  - *Field Strategy:* A field strategy indicates that the researcher involves themselves in the realistic software engineering setting within which their investigation takes place. For instance, a field study or a field experiment means that a researcher, as part of their investigation, spent time in the actual study or experiment setting,

which serves to collect information about context. Using a field strategy in a socio-technical setting means the researcher directly observed developers in their realistic context to enrich their study observations with that context.
  - *Respondent Strategy:* A respondent strategy involves asking subjects to respond to inquiries as a form of data collection. Inquiries may take the form of interviews, questionnaires, focus groups, etc. Using a respondent strategy in a socio-technical setting means the researcher considers developers a unit of observation (and/or a unit of analysis depending on the study), and may interact with them to collect data and test hypotheses.
  - *Lab Strategy:* Using lab strategies indicates that a study tested particular hypotheses in a highly controlled setting and aimed to maximize control of human subjects over realism. For instance, an eye-tracking study that investigates which portions of an automated build tool's log developers tend to focus on would involve creating a semi-realistic environment where developers would come in as subjects and go through a set of tasks while wearing an eye tracker. Using a lab strategy in a socio-technical setting means the researcher engages in a direct observation of developers and may interact with them to collect data.
  - *Data Strategy:* A data strategy indicates that the researcher used trace-data in their data collection process to gather information about the proposition they are studying. For instance, a study that investigates the impact of adopting automation on development process throughput (e.g., faster pull request merges) by examining the development activity logs of projects that use automation and comparing them against projects that don't use a data strategy. Using a data strategy in a socio-technical study typically implies an indirect study of developers via the artifacts they generate.
- **Non-Empirical Strategies**:
  - *Formal Theory:* A formal theory strategy indicates that a study was more conceptual in nature, and may not have focused on including empirical data so much as constructing and proposing an abstract model, framework, or theory. For instance, Fitzgerald and Stol's work on the continuous-star framework [1] combines both a meta strategy as well as a formal theory strategy to produce

a framework that depicts the various continuous activities in a modern software engineering setting.
- *Meta:* A meta strategy typically indicates a study used the literature as its primary source of data and employed methods such as systematic literature reviews or literature mapping. The study depicted in this chapter is an example of a study that uses a meta strategy.

After we coded a subset of papers (usually 10 papers per round), the two researchers responsible for coding it (researcher A or B, and myself) would conduct an agreement session. We compared, justified, and debated why different elements of the ADEPT theory and research strategies were assigned to a paper. The coding process we used was based on the deductive coding approach discussed by Linneberg and Korsgaard [134]. In this case, we combined an existing theory (ADEPT) and elements of an existing framework (Who, What, How) as a lens through which we interpreted previous research and utilized the theory's constructs and propositions as well as the framework's concepts as preexisting code categories. A full list of the papers in our sample, how we coded them, and other relevant material is included in our reproduction package as well as in Appendix C.

8.3 Results

Following our analysis described in the previous section, we were able to use ADEPT to frame existing continuous software engineering studies. Throughout this section, I discuss what socio-technical aspects of continuous software engineering these studies addressed by answering the questions we posed in Section 8.1:

RQ1: What are the most investigated socio-technical aspects of continuous software engineering?

RQ2: What research strategies are used to investigate propositions in continuous software engineering?

Using our sample of papers from three conferences (ICSE, FSE, ASE), we answer the first research question by demonstrating the most investigated socio-technical aspects of continuous software engineering in the first subsection. In the second subsection, we answer the second research question by discussing the most frequent research strategies used to investigate propositions in continuous software engineering.

8.3.1 RQ1: What Are the Most Investigated Socio-technical Aspects of Continuous Software Engineering?

Through our screening process, we discovered that studies in continuous software engineering have been steadily increasing over the past five years, as shown in Figure 8.2. Of particular note is the moderate increase in papers between 2017 and 2018, which coincides with when the Mining Software Repositories (MSR) challenge with the TravisTorrent dataset was released.

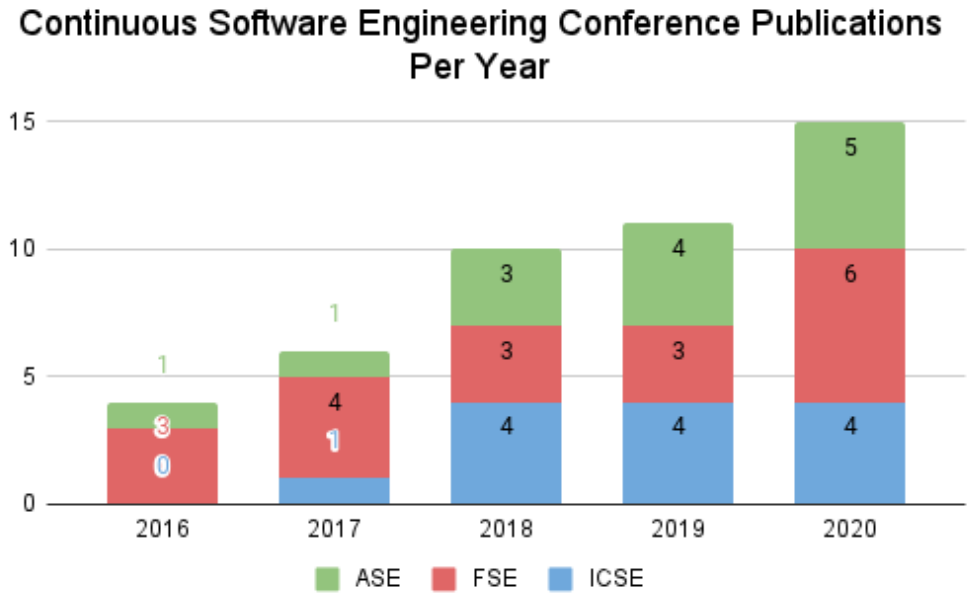

Figure 8.2: The distribution of continuous software engineering conference papers in our sample per publication year after applying the inclusion criteria.

As a result of our analysis, we mapped papers to the different elements in the ADEPT theory based on the constructs and propositions involved in their investigation. A count of these elements is shown in Figure 8.3. The counts in the figure represent how many papers investigated an individual socio-technical element of ADEPT, with the numbers in the boxes indicating how many times a construct was featured in a paper, and the numbers in the circles indicating how many times a proposition was featured in a paper. It is worth noting that the same paper may address multiple constructs and

propositions. A list of which papers correspond to the different ADEPT elements is shown in Table 8.1. The codes contained in the table (F09, I02, A10, etc.) correspond to the individual papers listed in Table C.1 and are summarized by ADEPT component.

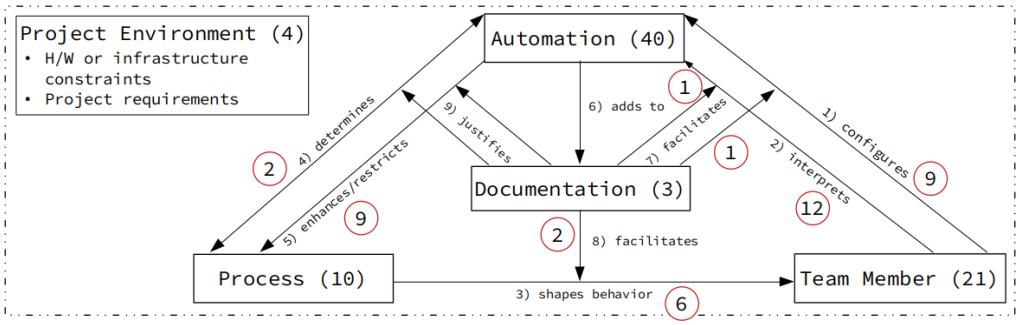

Figure 8.3: The number of times a paper included an ADEPT construct (in boxes) and/or proposition (in circles) in our paper sample.

In our sample of 46 papers, 40 papers (86.96%) investigated or proposed an automated solution meant to enhance some aspect of the development process. For instance, F09 identifies two distinct image building platforms (one via DockerHub and one self-contained in continuous integration tools) and compares their differences. F13 compares different project gating heuristics for their impact on development workflow when it comes to bug prediction.

We also found that, while multiple papers addressed a combination of constructs and propositions, 17 out of the 40 papers (36.96%) featured automation as a construct for investigation with no other constructs or propositions investigated. These papers focused on optimizing or describing an aspect of the automation itself (e.g., decreasing build time or resources) or proposing solutions to purely technical problems whereby the solution would be integrated into the automation. For instance, F04 investigates broken builds and creates a taxonomy of the different types of build failures.

In total, 21 papers out of 46 (45.65%) included or centered on the human in their analysis. Papers that studied human developers primarily focused on the developers' relationship with automation, be it configuration (9 papers - 19.57%) or interpretation (12 papers - 26.09%). Most configuration papers addressed detecting tool configuration smells or included a caveat that a developer would be responsible for *configuring* the solution they proposed. For

instance, F01 proposes CD-Linter as a tool to automatically detect GitLab CI configuration smells and F09 identifies two distinct image building platforms (one via DockerHub and one self-contained in continuous integration tools) and compares their differences to illustrate how configuration can result in two different build workflows. Regarding interpretation, most papers that proposed solutions or optimizations claimed the primary beneficiary was the developer and aimed to reduce developer blocking due to build duration or developer perception of automation. For example, F11 finds that developers tend to ignore flaky test results. However, the developers' tendency to ignore the flaky test results correlates with a high number of crash reports being submitted by software product users.

In our sample, 10 papers (21.74%) addressed the development process either as a unit of analysis or as a contextual factor as part of their analysis. The most common process-related proposition was "Automation Enhances/Restricts Process" which was featured in 9 papers (19.57%), followed by "Process Shapes Team Member Behavior" which was featured in 6 papers (13.04%). Most papers that investigated the automation's impact on the process focused on high-level process metrics to indicate changes in issue closure rates or commit rates which reflected the increased speed brought about by automation. For instance, F16 reports on automating the Facebook mobile app deployment process to an extent where the team is able to deploy multiple times a day, and how that impacts developers. Papers that studied how the process directed team members to perform tasks focused on the developers' perceptions of adopting a continuous paradigm, but mostly did not include details on what the paradigm's component practices were. For instance, A10 investigated the impact of adopting Travis CI (practice 2: automate the build) on software project development activity throughput as measured by commits and pull requests but did not examine other continuous practices.

Few papers discussed documentation (3 papers - 6.52%) or took into account a project's environment (4 papers - 8.7%). Papers that included documentation focused on build scripts and configuration files in their analysis that could possibly inform developers on how build processes ran within the automation, but were not presented as such. For example, I04 used Dockerfiles written by experts to infer a set of "Golden Rules" that indicated the best practices of writing Dockerfiles and how developers could use these image configuration files as a form of documentation. Regarding project environment, papers would only mention it when taking into account how the

solution they were proposing was configured, and whether the configuration took into account project or infrastructure characteristics.

There were also 5 papers (10.86%) that did not involve automation as a unit of analysis or contextual factor (F08, F14, F16, A09, A12). These papers primarily focus on continuous practices (including automation) but do not delve into the automation's technical aspects or characteristics. An example that was mentioned previously is F16, which investigates the impact automation has on the development process for the Facebook mobile app and how it facilitates multiple deployments a day. Similarly, A09 investigates whether projects implement continuous code quality practices and standards. While they collect their data from the automation attached to software projects on GitHub (unit of observation), the study's contribution focuses on the practices as a unit of analysis.

Table 8.1: A list of papers in our sample grouped by ADEPT theory elements they feature in their investigation.

ADEPT Element	**Papers**
ADEPT Constructs	
Automation	F01, F02, F03, F04, F05, F06, F07, F09, F10, F11, F12, F13, F15, F17, F18, A01, A02, A03, A04, A05, A06, A07, A08, A10, A11, I01, I02, I03, I04, I05, I06, I07, I08, I09, I10, I11, I12, I13, F19, A13, A14
Documentation	I04, I06, I12
Environment	I02, I08, I10, I12
Process	F08, F13, F14, F16, A09, A10, A11, I06, F19, A12
Team Member	F01, F02, F07, F08, F09, F11, F12, F16, A01, A02, A03, A05, A07, A08, A09, A10, A11, I01, I06, I10, I12
ADEPT Propositions	
1. Team member configures automation	F01, F02, F06, F09, F10, A02, A05, I01, F19
2. Team member interprets automation	F07, F09, F11, F12, A01, A03, A05, A07, A08, I06, I10, I12
3. Process shapes team member behaviour	F08, F16, A09, A10, A11, F19
4. Process determines automation	A11, A12
5. Automation enhances/restricts process	F09, F10, F13, F16, A10, A11, I01, I05, I06
6. Automation adds to documentation	No papers.
7.1 Documentation facilitates how team member configures automation	I12
7.2 Documentation facilitates how team member interprets automation	I12
8. Documentation facilitates how process shapes team member behaviour	I04, I06
9.1 Documentation justifies why automation enhances/restricts process	No papers.
9.2 Documentation justifies why process determines automation (or vice versa)	No papers.

8.3.2 RQ2: What Research Strategies Are Used to Investigate Propositions in Continuous Software Engineering?

Through our analysis, we were also able to answer our second research question about research strategies by mapping the papers in our sample to the research strategies they used in their investigation. An overview of the research methods used in our sample is shown in Figure 8.4. A list of which papers correspond to each research strategy is shown in Table 8.2.

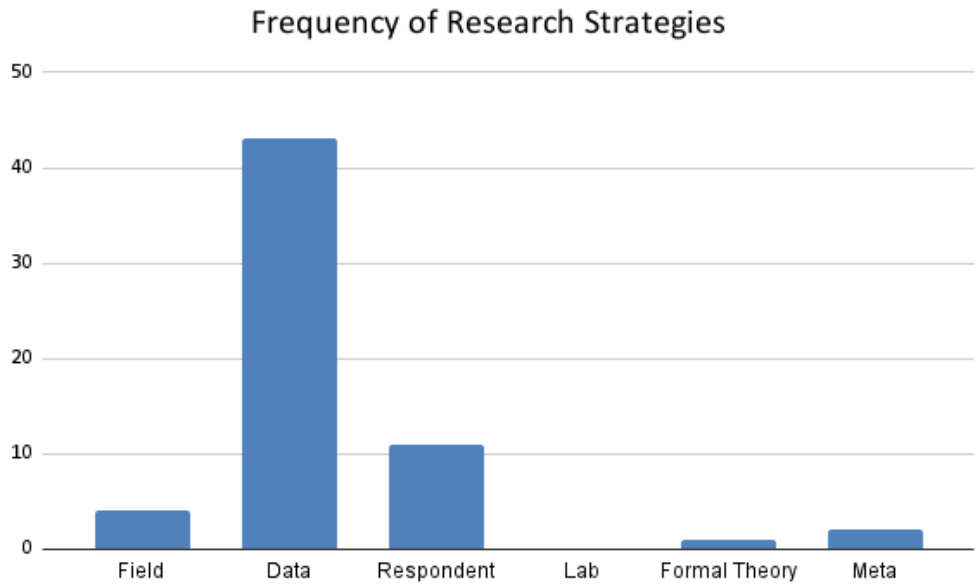

Figure 8.4: The occurrences of Who, What, How framework research strategies in our paper sample.

The most common research strategy employed in investigating continuous software engineering phenomena is "Data" (43 papers - 93.48%) followed by "Respondent Strategies" (11 papers - 23.91%). Data strategies indicate using artifact-centric methods (such as mining activity logs or building machine learning frameworks) to analyze a phenomenon or propose a contribution. For example, I02 proposes SmartBuildSkip, an approach that uses historical build data to predict whether a sequence of future builds will fail and then

Table 8.2: A list of papers in our sample grouped by research strategy.

Research Strategy	Papers
Field Strategy	F10, F16, A05, I13.
Data Strategy	F01, F03, F04, F05, F06, F07, F08, F09, F11, F13, F14, F15, F16, F17, F18, A01, A02, A03, A04, A05, A06, A07, A08, A09, A10, A11, I01, I02, I03, I04, I05, I06, I07, I08, I09, I10, I11, I12, I13, F19, A12, A13, A14.
Respondent Strategy	F01, F02, F07, F08, F09, F12, A01, A07, A10, A11, I06.
Lab Strategy	No papers.
Formal Theory	F02.
Meta	F02, F07.

skip them, thus saving build system resources. SmartBuildSkip was evaluated on a testing dataset to determine its effectiveness. As an example of a paper that solely relies on a respondent strategy, F12 investigates the various barriers and needs that developers indicate they expect an automated build tool to help them with by conducting semi-structured interviews and surveys.

Data strategies were most frequently employed in combination with respondent strategies in 9 papers (19.56%) out of our sample of 46 (F01, F07, F08, F09, A01, A07, A10, A11, I06). Researchers would mainly investigate a development activity log or build a data-centric solution then perform a qualitative study (typically in the form of a survey) to test whether their proposed solution met its set acceptance criteria. For instance, I06 uses a survey to validate what developers consider "smells" in their automation configuration, build a data-centric predictive tool (CI-Odor) to automate smell detection, then use another survey to evaluate CI-Odor from a developer perspective.

With respect to papers that address specific ADEPT constructs, we found data strategies were the most common research strategy across all constructs, as shown in Figure 8.5. Respondent strategies were used mostly with the automation (10 papers - 21.74%) and team member (11 papers - 23.91%) constructs, and was minimally featured with the process (4 papers - 8.70%) and documentation (1 paper - 2.17%) constructs. The remaining strategies were seldom used for studying ADEPT constructs.

Regarding ADEPT propositions, we found that data strategies were the most common across all propositions, as shown in Figure 8.6. However, respondent and field strategies were used in papers that investigated propositions involving social (team member) and environmental (process) constructs. Meta strategies were mainly used for propositions that involved social (team

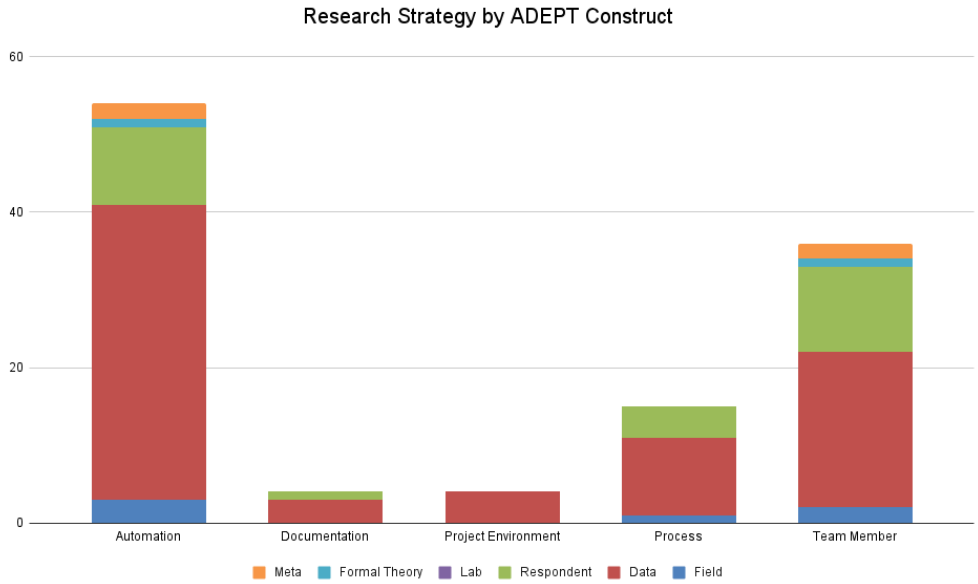

Figure 8.5: The occurrences of research strategies per ADEPT construct in our paper sample.

member) constructs.

8.3.3 Summary

Our results indicated that the most investigated socio-technical construct was automation, which indicates that continuous software engineering research tends to focus mostly on investigating tools used to facilitate software development, or propose tools that facilitate an aspect of software development. With respect to the propositions, the relationship between automation and the team member (both interpretation and configuration) was featured substantially, with automation's impact on the process being the second most frequent. Combining the focus on automation with data-centric research strategies being the predominant data collection methods (mostly artifact-centric), we can conclude that most continuous software engineering literature tends to focus on the technical aspect of its socio-technical nature with less attention paid to the human and social aspects.

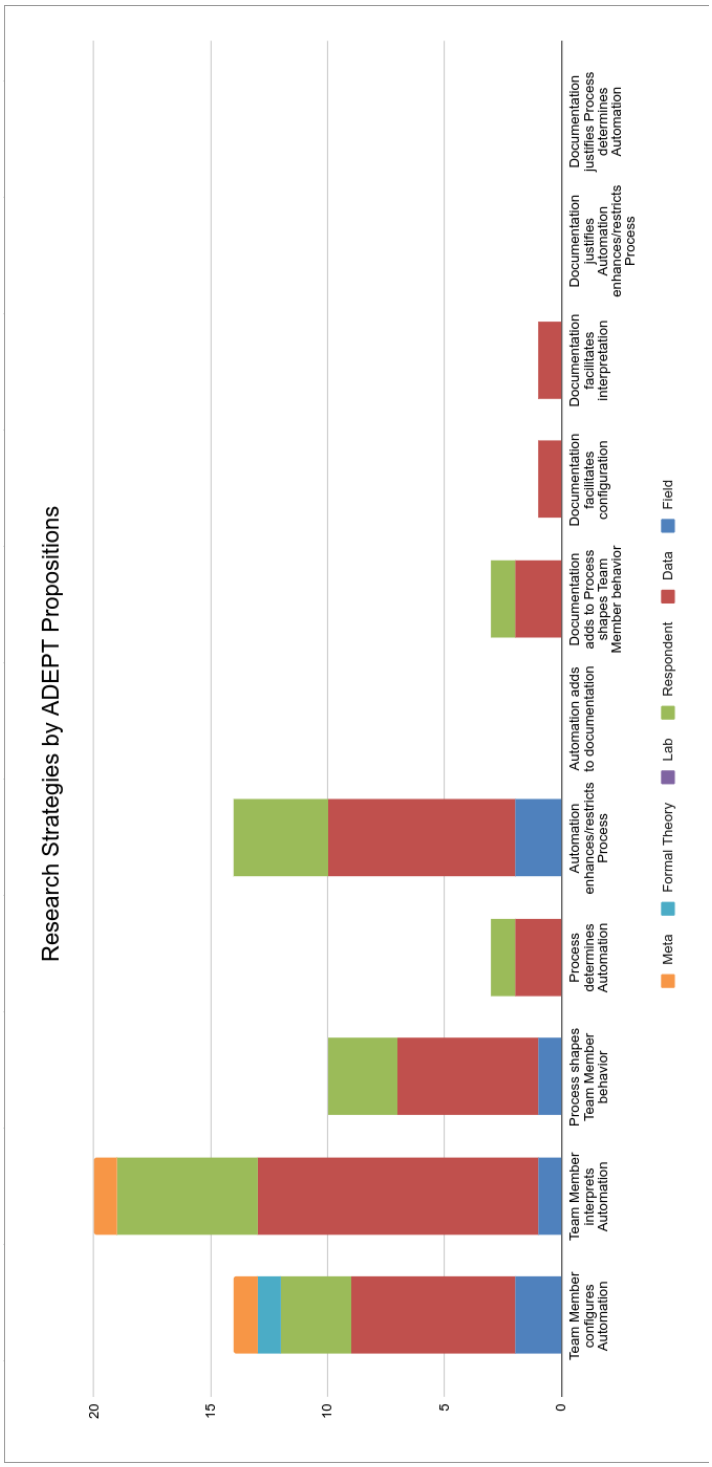

Figure 8.6: The occurrences of research strategies per ADEPT proposition in our paper sample.

8.4 Discussion

In this section, I discuss the significance of this study's results and establish how they indicate the utility of the ADEPT theory. I also show these results' implications for the overarching research questions in my dissertation.

8.4.1 Study Contributions

Using the results we identified in the previous section, there are two major contributions this study makes: ADEPT can be used to interpret existing phenomena in continuous software engineering, and most studies in continuous software engineering (at least in our sample) do not tend to consider the full socio-technical nature of continuous software engineering.

ADEPT theory utility

By using the ADEPT theory constructs and propositions as a lens, we were able to interpret the phenomena investigated in continuous software engineering research over the past five years (as limited by our sample). The major concern regarding ADEPT's utility was that it was constructed based on studies conducted in specific industrial contexts, and that it would not generalize beyond these specific contexts. However, we found that ADEPT indeed transferred to other industrial and even open source contexts. For instance, the work by Zhao et al. [12] was conducted with open source projects in mind, because they mined open source projects on GitHub that used Travis CI as an automation. They explored the impact adopting automation had on other development practices with respect to how automation increased commit and pull request processing rates. ADEPT was able to capture the fact that this study focused on the impact of automation on the process using the proposition "automation enhances/restricts process" and its component constructs. This study is one among several studies conducted in an open source context that we were able to map using ADEPT.

Using ADEPT as a lens can also have its disadvantages. ADEPT's constructs and propositions are derived from our observations of specific continuous industrial contexts, and even though they were able to represent the papers in our sample, it is quite possible that they do not capture newly emerging propositions. Applying ADEPT in a non-continuous context may

have risks as well, chief among them the different interactions between process, automation, and team member in more standardized software development contexts such as embedded systems. Finally, the major risk using ADEPT creates is its propensity to generate bias. It is quite possible that, by using ADEPT as a lens, a researcher may only focus on the propositions currently contained within ADEPT and use that as a silver bullet to interpret a large array of incompatible or new phenomena. In its current form, ADEPT should be used as a guiding socio-technical lens through which to consider continuous software engineering propositions in future studies, but it is by no means perfect or all-encompassing. However, ADEPT can be improved to accommodate new propositions.

Overlooking the socio-technical nature of continuous software engineering

In addition to mapping continuous software engineering phenomena and research strategies, our analysis also highlighted how several aspects of the socio-technical nature of continuous software engineering were not included. Most of the studies we investigated focused on the technical aspect of the continuous software engineering process, namely automation. These studies rarely included or considered the full aspects of socio-technical phenomena, such as the technical (automation), the human (team member), and the environment (process and project environment). More often than not, a study would include one or two aspects, but rarely all three, possibly risking not accounting for the impact these missing aspects could exert. Furthermore, most studies tended to use data strategies to conduct their investigation, which we know from previous work [7] did not capture the full socio-technical nature of continuous software engineering. For instance, using the same example by Zhao et al. [12], humans are not considered as a possible factor in how the automation increased process speed. Neither, for that matter, is the individual projects' environment in terms of their characteristics (i.e., what kind of project it is, what benefits the team members wished to maximize by adopting Travis CI, etc.).

8.4.2 Implications for My Dissertation

In addition to establishing ADEPT's utility and mapping recent continuous software engineering research, our analysis also highlighted research areas

that were infrequently explored. In the next chapter, I discuss these *gaps* in the continuous software engineering literature, what propositions they may hold, and suggest strategies on how to explore them. I also provide an introspective analysis of the studies conducted in this dissertation as a reflection on my journey, and critique them using ADEPT to identify possible avenues for growth as well as opportunities for future work.

8.5 Limitations and Threats to Validity

In this section, I discuss the limitations of this study and how we attempted to mitigate them. I use the total quality framework by Roller and Lavrakas [106] to consider credibility, analyzability, transparency, and usefulness.

8.5.1 Credibility

Credibility is related to how accurate and complete the collected data is. It encompasses two aspects: scope and data gathering.

Scope

The scope of this study is limited to the venues we chose during our publication venue selection phase. These venues are considered the most prominent software engineering conferences in our domain, usually encompassing a broad array of topics. Our sampling window for these conferences covered 5 years from 2016 to 2020. Within these conferences, our study focused on conference papers that discussed automation within the context of continuous practices because that was the context within which ADEPT was created. However, it may be possible to involve other types of papers (not explicitly in a continuous context) that address automation. We also did not include journals and other publication venues which could potentially contain continuous software engineering literature due to limited time for our analysis. We suggest consideration of other venues as future work.

Data Gathering

With respect to our constructs, we use the ADEPT theory [23] to frame existing research in continuous software engineering and to capture the research strategies used by previous studies. We mitigated our researcher bias

by having two researchers sample each conference window. Each researcher identified papers in isolation before coming together to compare their samples in an agreement session. Throughout a session, we debated whether a paper should be included based on our inclusion criteria specified in Section 8.2.2. Papers both researchers agreed on were included, which resulted in the inclusion of 46 papers out of an approximate total of 2039.

8.5.2 Analyzability

Analyzability pertains to the accuracy of the analysis we conducted and how accurate the interpretations we drew from it are. It consists of two aspects: processing and verification.

Processing

As mentioned previously, we coded 46 papers using ADEPT's constructs and propositions. We used an online spreadsheet service (Google Sheets) to facilitate coding and collaboration between multiple researchers. All data processing was conducted manually.

Verification

To mitigate the researcher bias that typically occurs when a single researcher codes the majority of the data, each paper was screened and coded by at least two researchers, with the first author coding all the papers. Once a set of papers was coded, we conducted extensive agreement sessions where we debated which labels applied and why, eventually settling on what both researchers agreed. When two researchers disagreed on a label assigned to a paper, they both referred to the paper to provide evidence for their labelling choice. After determining which label was more dominant upon referring to the paper, that label was used to signify agreement. For instance, a paper that claimed to study continuous practices (automation being a continuous practice), but focused on optimizing the automation was assigned the automation label and not the process label, because the paper's overall dominant theme was automation optimization.

8.5.3 Transparency

We attempted to provide rich details and quotes from papers wherever possible. We also provide a reproduction package [135] that includes the following:

- our coded paper set, and
- our observations.

8.5.4 Usefulness

Usefulness is an indicator of how actionable the results from a study are, and their ability to transfer to other contexts (external validity). Our study explored how socio-technical phenomena are investigated in continuous software engineering using the ADEPT theory as a lens. Our approach can be used to extend this study (and similar mapping studies) further to cover a more comprehensive set of publication venues. Furthermore, our results indicate that ADEPT can be used to frame socio-technical research in continuous software engineering. However, our study's results are dependent on ADEPT since we rely on its constructs and propositions to interpret the data we collected. ADEPT was constructed with a continuous software engineering context in mind to interpret and explore socio-technical phenomena. While it is possible that our process (and consequently ADEPT) can be used in different contexts, we cannot claim that we can do so without modification.

Part IV

Discussion and Conclusion

9 Discussion and Insights

You can use logic to justify almost anything. That's its power. And its flaw.

Cpt. Kathryn Janeway
Star Trek: Voyager

In this chapter, I synthesize the insights from our previous studies to develop a socio-technical roadmap of continuous software engineering research. I discuss how continuous software engineering is a socio-technical endeavour, and use our findings in Chapter 8 to illustrate possible future research directions that consider continuous software engineering's socio-technical nature.

9.1 The Socio-technical Nature of Continuous Software Engineering

A socio-technical system is one that combines both social and technical components that interact together to produce an output [136]. The human provides inputs to a technical system, which operates on them and produces some output. This general definition of a socio-technical system was meant to interpret the relationship a human had with the machine they were operating. Baxter and Sommerville [137] indicated that such systems, particularly IT systems (software), also operate within an organizational context. The organizational context facilitated the identification and inclusion of factors that normally were excluded when considering these systems, such as external stakeholders and business processes.

Recently, Hall and Rapanotti [5] proposed interpreting the software engineering process from a socio-technical perspective when they proposed their

design theory of software engineering which considered software engineering as a problem-solving activity occurring within a socio-technical context. More importantly, they identified various components in their interpretation of the three ellipse model of requirements-centric socio-technical systems, as shown in Figure 9.1. The socio-technical system involves the human and the system interacting within the context of their environment.

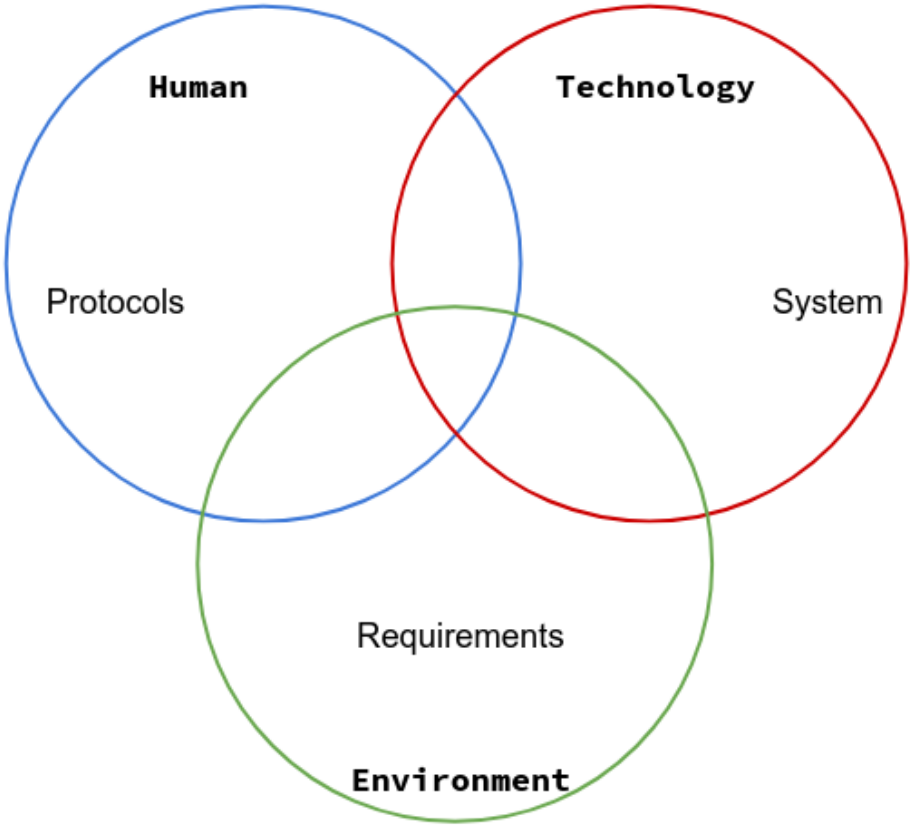

Figure 9.1: The three ellipse model of socio-technical software engineering as adapted from Hall and Rapanotti [5].

Continuous software engineering is a paradigm that involves several components working together as a whole to achieve certain benefits (rapid feedback, higher quality, etc.) [1]. The human aspect of these components includes developers, product managers, and other development team members

who can make decisions regarding the software changes being made, or otherwise act as decision "gates" for software changes. The technical aspect of these components includes the tools developers use to facilitate aspects of their development workflow (most commonly, automated build tools) and the software being developed. The environmental components include external constructs that may influence or limit how developers and their tools interact and work together throughout the development workflow. For instance, the process dictates a set of practices developers are expected to follow, and governs the role of automation in the workflow. The project's characteristics (project type, its requirements, etc.) influence what benefits are expected from automation, as well as the development workflow. For example, a data-centric project that focuses on collecting, processing, and sharing data via an API will prioritize data integrity via comprehensive testing as opposed to faster feedback via faster builds. Finally, the project infrastructure resources limit automation's capacity to achieve its expected benefits.

9.2 Applying ADEPT to our Previous Studies: A Self-Critique

In this section, I discuss our previous studies using ADEPT as a lens. I use ADEPT's constructs and propositions to determine the extent to which the studies were conducted from a socio-technical perspective. I also explore possible future work that could result from our studies based on ADEPT's interpretation.

9.2.1 Contribution Guidelines in GitHub

This study aimed to explore the role of continuous integration tools (automation) in a typical open source project's development workflow (process). We explored project contribution guidelines as a proxy for process documentation and mined GitHub development activity logs to produce process maps to better compare documentation to reality. We identified what information was being communicated in the contribution guidelines and found that continuous integration tools were only ever mentioned as a form of test infrastructure. Finally, we uncovered some differences between the documented contribution process and the actual process as depicted in the logs.

From ADEPT's perspective, and from a pragmatist's point of view, this study touched on some socio-technical aspects, namely the human and the environment as represented by developers and their contribution process respectively. However, it did not take into account the automation as a construct, which may have had an impact on the finer details of the contribution process. For instance, considering the automation running whenever a pull request was submitted might have given more insights into the review process, and allowed us to compare it to the documented acceptance criteria to better explore the role of contribution process documentation. Furthermore, this study did not consider the different projects' contexts as factors in the analysis, which could have offered insights into why the actual contribution process deviated from its documented counterpart. Finally, while constituting a learning experience in its own right, solely using data-centric methods to exploring contribution guidelines and mine development activity logs severely hindered my ability to capture project contexts since most of that information is tacit (as I would learn later when conducting the two following studies).

9.2.2 Non-functional Requirements in a Continuous Context

Through this study, we explored how developers treated non-functional requirements in a continuous context. We highlighted the different strategies they used to either monitor, implement, or delegate non-functional requirements, and provided insights into a possible manifestation of misuse when offloading non-functional requirements to third parties. We also identified several challenges developers faced when dealing with non-functional requirements, chief among them the fact that some non-functional requirements did not lend themselves easily to automation.

From ADEPT's perspective, and due to the study being conducted as a multiple case study, we were able to focus more on the different socio-technical aspects of the three organizations' development processes, and how non-functional requirements and automation complemented each other in some cases and clashed in others. We explored the process, automation (to an extent), and team member (only in terms of their perception of non-functional requirements) all the while considering the three vastly different contexts within each organization. We used field strategies to conduct this

study and relied on interviews to gather data, which gave us a much richer source of information compared to the previous study. The interviews allowed us to better explore the socio-technical aspects at play in our investigation and understand their impact on our research goal.

9.2.3 In-depth Investigation of Continuous Practices

Through this study, we conducted an in-depth investigation of Fowler's ten continuous practices [9] as a multiple case study with the organizations we had worked with in the non-functional requirements study. We had the added benefit of already being familiar with their contexts from this previous study, and we had cultivated a trust that allowed us to obtain and analyze development activity logs in addition to conducting interviews. We identified three main factors that influenced the implementation of continuous practices across the three organizations, and observed the role of documentation in their development processes, including automation-as-documentation.

From ADEPT's perspective, we captured the different socio-technical aspects of the organizations' continuous development processes. We explored how developers perceived and dealt with automation, and how that automation was integrated into the development process. We uncovered how the automated tools the developers were using impacted how they implemented continuous development practices, and how the practices themselves influenced how they configured automation. Finally, we learned how the environment (in terms of infrastructure and project characteristics) had an impact on the automation's efficiency, and how developers prioritized the different aspects of automation. We combined both data strategies (mining activity logs) and field strategies (using interviews for data collection) to gather data, and while the development activity logs did not offer much in terms of insights, they did allow us to triangulate our results using an alternative data source. This is the main study upon which ADEPT is built.

9.2.4 Reflection on Our Overall Research Methodology

In the beginning, our purpose was to explore this new socio-technical perspective of continuous software engineering. Starting with a literature review, we followed a primarily exploratory approach [83] which meant we conducted individual studies to explore the scope of the problem in our GitHub study in Chapter 4, then moved on to a more targeted approach guided by our

research goals in Chapters 5 and 6. Once we had explored our research goals to a reasonable extent, we used our findings to synthesize a tool or framing device (ADEPT) that could be used by others to inform research in continuous software engineering. The benefit this approach offers is that our contribution is grounded in empirical observations much like the results of grounded theory approaches despite us using Fowler's ten practices as a guiding framework. From a pragmatist's perspective, we were able to capture and investigate phenomena as they were occurring in their realistic context, and did not introduce an additional layer of bias that would typically be introduced by top-down approaches. The major downside of this approach is that it requires both significant time, effort, and resources. Each study had to be conducted more or less from scratch, and care had to be taken not to allow bias from our previous work to interfere with new studies.

An alternative methodology would have been to follow a top-down approach. We would have started by a literature review (similarly to the first step in our actual methodology) and then used preexisting constructs in the literature to frame our research questions and plan our studies. This approach would have been more confirmatory in nature, much like a sequential-explanatory strategy, with us aiming to test and confirm hypotheses based on the results of the literature review and then exploring why we saw particular results. The benefit here would have been the consumption of less time and resources because we would have had a clear target. However, given the direction taken by the literature when it comes to exploring continuous software engineering, it is very unlikely the main goal of this dissertation would have been related to the socio-technical aspects of continuous software engineering. The more likely direction would have been developer-automation interaction with a heavy focus on data- and artifact-centric research methods, which meant we would not have been able to account for contextual factors. Furthermore, we would not have been able to build an explanatory theory like ADEPT, instead we would have adopted a preexisting theory from the literature. Continuous software engineering literature does not have theories of its own yet, so that theory would either have come from the parent domain of software engineering, or from a different domain altogether considering automation's relationship with developers. Finally, the preconceptions we would have had from relying on the literature more in this approach would have introduced an additional layer of bias that would have reduced our results' realism. This is because our understanding and impressions of the phenomena we observed would have been focused only on what had been

discussed in the literature before or might have only slightly deviated from the literature.

9.3 Socio-technical Research Directions in Continuous Software Engineering

Based on our findings in Chapter 8, we identified several socio-technical propositions in ADEPT that the literature had not deeply explored because of a prevalence of tool-focused studies. These propositions comprised the process and documentation constructs and their relevant propositions:

- automation adds to documentation,
- process determines automation (or automation determines process),
- documentation facilitates how the process shapes team member behaviour,
- documentation facilitates how team members configure automation,
- documentation facilitates how team members interpret automation,
- documentation justifies why the automation enhances/restricts the process, and
- documentation justifies why the process determines the automation (or vice versa).

These constructs and propositions offer possibly rich areas of future research when it comes to socio-technical phenomena in software engineering. While these propositions are single, isolated edges within the ADEPT theory, they should not be investigated as such. In fact, in order to capture the full socio-technical nature of continuous software engineering, they should be investigated while considering how the remaining constructs and propositions may impact them. In the following subsections, I detail what each research direction could entail in terms of possible research questions.

9.3.1 Automation Adds to Documentation

In our study in Chapter 6, we found that developers used build scripts and automated tool configuration files to familiarize themselves with a software project's build process, and to communicate information about that process. From a human perspective, one can explore how the different types of automation-as-documentation can serve different purposes. How can these

scripts and configuration files be used to gain an understanding of the system? How can they be used to communicate information between developers, especially in the case of newcomers? They can also be investigated from a technical perspective. For instance, Terraform infrastructure configuration scripts are notoriously hard to read and exploring ways to enhance their readability may lead to interesting results.

9.3.2 Process Determines Automation (Or Vice Versa)

The relationship between the choice of automation and why the development process is structured the way it is remains underexplored. We conducted a cursory investigation of this proposition when we identified the factors that resulted in different implementations of the same practice [6], but there may be many more factors our cases did not exhibit. For instance, do different development approaches (e.g., test-driven development) influence the types of automated tools developers choose to incorporate into their development workflow? Do different types of automated tools require developers to adapt their process around them in order to maximize the expected benefit? Currently, generic automated build systems can accommodate any type of tool within their environment. Does using a tool that targets a particular software aspect (e.g., SonarQube, CodeCov, etc.) that also operates independently from an automated build tool require process changes or force developers to change how they implement continuous development practices?

9.3.3 Documentation Facilitates How the Process Shapes Team Member Behaviour

Documentation is generally underexplored in software engineering research. Our previous study on continuous practices (Chapter 6) found that automation-as-documentation can help a team member understand how a software build and deployment process is structured. It is possible that this knowledge impacts how they interpret the different continuous practices (or other development practices) they are expected to follow. In this proposition, exploring the role documentation plays in informing human behaviour in a development process may offer many possibilities. For instance, does natural language documentation work better than infrastructure-as-code documentation when it comes to communicating the build process to a new developer? How can we enhance infrastructure-as-code (e.g., DockerFiles or Terraform

scripts) to serve as infrastructure documentation? How closely do developers follow the build process prescribed via build scripts? Do developers have their own preferred variations of a prescribed build process?

9.3.4 Documentation Facilitates a Team Member's Interaction with Automation

There are two forms of documentation that can influence how a team member can interact with an automated tool: tool documentation and automation-as-documentation. Tool documentation refers to tool-specific documentation that communicates the tool's basic functionality and features. Tool documentation can be official on a tool's website or unofficial via question answering sites such as StackOverflow. Automation-as-documentation involves using build scripts and configuration files to gain an understanding of how a tool works. In both our GitHub study on contribution guidelines (Chapter 4) and our study on continuous practices (Chapter 6), we observed that documentation (in the form of official docs or automation-as-code) communicated information about how an automated build tool was configured and what functionality it provided. What is not clear is how the information documented in either form can impact how a developer configures automation or interprets automation results. Both concepts can be used to generate interesting research questions.

Documentation facilitates how team members configure automation

This proposition focuses on how team members define the functions and settings an automated tool uses to perform its tasks. Automation configuration can generally fall under one of two broad categories: infrastructure and tasks. Infrastructure configuration involves determining what scaffolding is required to run a build, such as operating system, allocated resources, networking settings, dependency management, etc. Task configuration involves defining and linking tasks, and specifying what a valid task's output should be. For instance, what role does tool documentation play in facilitating (or hindering) automation configuration? Is automation-as-documentation better at communicating information than other forms of documentation (e.g., project README files or contribution guidelines)? Do developers use infrastructure/task configuration templates that help them in setting up new

project configurations? What do these templates look like?

Documentation facilitates how team members interpret automation

This proposition delves more into the human-computer interaction nature of how developers view automation, and whether or not documentation (either standard or as automation-as-documentation) can impact a developer's perception of automation. Developers make decisions in several stages of the development workflow, most of which are assisted by automation. Exploring this research direction could lead to interdisciplinary research that borrows from the psychological and sociological domains. For instance, do any of the automation-related phenomena (misuse/complacency or disuse) occur in relation to good/poor documentation? Would the introduction of documentation mitigate developer frustration about failing builds as noted by Souza and Silva [21]? Does the way documentation is written (natural language, technical form, etc.) influence how a developer interprets the results automation produces? If so, how? Do developers consider the readability of automation configuration scripts before adopting an automated tool?

9.3.5 Documentation Justifies the Automation's Relationship with the Process

According to ADEPT, there are two ways automation can influence or be influenced by the development process. The first involves examining how automation influences the process in terms of its efficiency (enhances/restricts). Influencing efficiency is a uni-directional relationship that flows from automation to the process in the sense that automation can either facilitate (positive effects such as faster feedback, higher productivity, etc.) or hinder (negative effects such as automation abuse manifested as slow feedback, complex build processes, etc.) the development process. The second involves examining the impact of adopting an automated tool on the development process, or how a development process is built around the desire to use a particular tool (similarly to our GitHub study in Chapter 4). The impact of adopting a particular tool can be examined from a purely process-based perspective where the adoption can be correlated to an increase in pull request merges, or it can be examined from a socio-technical perspective where developers have to

adapt to a new development process (or new changes to existing practices). Documentation can be a factor in both.

Documentation justifies why the automation enhances/restricts the process

This proposition represents whether developers justify the trade-offs they make regarding tool choices and development workflow convenience. It encompasses both positive and negative effects resulting from developers choosing one aspect of the trade-off over the other. From our previous study exploring GitHub contribution guidelines [7], we know that automated tool choices are rarely justified in open source projects. Is documentation ever used to justify process changes in relation to adopting or introducing automation in different contexts? If developers do not document their justification for a particular automation configuration, how do they prevent rollbacks that lead to previously undesired changes? Would justifying tool choices and their impacts on the process facilitate exploring the decrease in developer attraction and retention that Gupta et al. [22] observed when adopting Travis CI?

Documentation justifies why the process determines the automation (or vice versa)

This proposition represents the possible role documentation can play in integrating an automation into a development process, or adapting a development workflow around a newly introduced tool. Rather than studying the disruptive impact that a newly introduced automated tool brings to a project, the proposition instead focuses on how the tool and project's documentation conflict with or compliment each other. For instance, does integrating a new automated tool become easier/harder with or without the presence of process documentation? How does automation-as-documentation evolve or change when the automated tool is replaced to accommodate development process changes? Do developers use automation-as-documentation to build their development process around the tool they want to use? Does it provide a suitable justification that new developers can use to understand how the development process functions?

10 Conclusion

Improve a mechanical device and you may double productivity. But improve man, you gain a thousandfold.

Khan Noonien Singh
Star Trek: The Original Series

Throughout this dissertation, I established the fact that continuous software engineering is a socio-technical endeavour that requires human, technical, and environmental aspects all working together to be effective. However, upon examining the continuous software engineering literature, I found the focus mostly centered on the technical aspect while overlooking the other two aspects. To that end, we designed and conducted several studies to determine how automation, the most popular of continuous practices, impacted software development workflow and developer decision making.

The first study involved examining open source projects on GitHub to determine the impact automation had on the development workflow and its role within the GitHub contribution process. We found that automation was primarily used as a vessel for test execution, with little to no indication that other continuous practices were being applied around it. From there, we proceeded with the assumption that artifact-centric research methods might not give us the full picture regarding the role of automation in continuous software engineering, and we resolved to adopt a more comprehensive research approach.

Our second study focused on how developers handled non-functional requirements in a continuous context. This would allow us to observe developer decision making in an area where requirements were not easy to define, allowing for more creative decision making. We found that developers tended to

delegate non-functional requirements either to third-party providers or tools, which indicated they tended to surrender control and trust an external entity (typically the automation). However, that was not true for all cases because developers would also build their own tools from scratch when a commercial tool was not flexible enough to accommodate their use case. In this study, we established the relationship between developers and automation, and what automation-related phenomena (misuse/disuse) could transfer to a software engineering context.

Our third study adopted a more exhaustive approach where we conducted an in-depth investigation of continuous practices, automation included, across three organizations. We were able to conduct a study that also considered the context within which continuous practices were being applied. Thus, we were able to identify the factors that resulted in different implementations of the same practice across different organizations. This study would also serve as the basis for the main contribution of this dissertation, ADEPT, our socio-technical theory of continuous integration.

We built ADEPT from insights we collected from the previous three studies. We needed to capture the different interacting entities, how they related to each other, and more importantly how they impacted by the context surrounding them. ADEPT would allow us to make sense of the phenomena we had observed in our studies as well as provide a theory to guide future research in this area. To establish its utility, we embarked on our fourth exploratory study, one that used ADEPT as a lens to explore and map past literature. We found that ADEPT could be used to interpret previous continuous software engineering studies from a socio-technical perspective, and it could also be used to generate possible areas of future research.

As future work, generating and testing hypotheses from the propositions contained in ADEPT will prove interesting because we now have a reasonable starting point with which we can explore socio-technical propositions in continuous software engineering. It might also prove interesting to capture the codebase as a construct and integrate it into ADEPT to see how it interacts with automation, developers, and the remaining constructs. Furthermore, it may be possible to use ADEPT for framing research about different forms of automation, not just build tools and automated pipelines. One example that comes to mind is chatbots, especially considering their recent popularity in software engineering research. Software engineering is still a human-driven socio-technical endeavour, and while ADEPT offers a socio-technical representation of software engineering, it is still limited by its continuous con-

text and the human-centric phenomena we observed. In closing, the most beneficial course of future research would be to apply ADEPT to different socio-technical phenomena that have not been considered in its construction in order to further improve and augment it, and in turn improve software engineering.

Appendix

A Chapter 5 Supplementary Material

This appendix lists the various additional material relevant to our study on how non-functional requirements are treated within a continuous context. It includes the following:

- a list of our interview questions,
- and the ethics certificate pertaining to this study.

A.1 Interview Questions

Because our interviews were semi-structured in nature, we started out with a set of prepared questions, asking follow-ups when necessary. The prepared questions included:

1. Are you familiar with the term DevOps? Are you familiar with the term continuous (integration, delivery, deployment)?
    - If you are familiar with these terms, how do you define them?
2. Does your organization practice any continuous practices/DevOps?
    - If yes, what are they?
3. How do you define non-functional requirements?
    - If the interviewee answers no, we can first provide an example of a FR to help an interview conceptualize. If the interviewee is still confused, we can provide an example of a quality attribute of a system (i.e. performance requirements) or activities that the organization conducts to ensure a specific quality. Those activities may be using infrastructure as code to improve maintainability.
4. How do you define a non-functional requirement?
5. Which non-functional requirements are important to your organization?
6. How do you manage non-functional requirements?
7. How do you document a non-functional requirement?
8. Are there any non-functional requirements in particular in your source control?
9. How do you test or ensure a non-functional requirement is satisfied?

10. What happens when a non-functional requirement fails? Is there a feedback loop from continuous development?
11. Does a non-functional requirement require additional resources (additional approval, testing, ...etc.) when developing it?
12. How do you ensure everyone is aware of a particular non-functional requirement?

University of Victoria

Office of Research Services | Human Research Ethics Board

[redacted]

Certificate of Approval

PRINCIPAL INVESTIGATOR	**Margaret-Anne Storey** (Supervisor)	**ETHICS PROTOCOL NUMBER** Expedited review - delegated	**19-0213**
PRINCIPAL APPLICANT	**Omar Elazhary** **PhD student**	ORIGINAL APPROVAL DATE	26-Nov-2019
UVIC DEPARTMENT	**Computer Science**	APPROVED ON	26-Nov-2019
		APPROVAL EXPIRY DATE	25-Nov-2020

PROJECT TITLE **A Study of the Interplay between Software Developers and Software Development Automation**

RESEARCH TEAM MEMBERS
Neil Ernst - Co-investigator, Assistant Professor

DECLARED PROJECT FUNDING
NSERC Discovery, University of Victoria

DOCUMENTS INCLUDED IN THIS APPROVAL
consent_form_explicit_interviews_modified.pdf - 21-Nov-2019
Survey Questions_v1.pdf - 25-Oct-2019
consent_form_implied_survey.pdf - 15-Oct-2019
Interview Questions_v1.pdf - 15-Oct-2019
cse_workflow.png - 15-Oct-2019
Parasuraman_Performance consequences of automation-induced complacency.pdf - 15-Oct-2019
Parasuraman_Complacency and bias in human use of automation_An attentional integration.pdf - 15-Oct-2019
Fowler_Continuous integration.pdf - 15-Oct-2019

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a “Request for Modification” form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a “Request for Renewal” form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a “Notice of Project Completion” form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropiate standards of ethcis as outlines by the University of Victoria Research Regulations Involving Human Participants.

[redacted]

Dr. Rachael Scarth
Associate VP Research Operations

Certificate Issued On: 26-Nov-2019

B Chapter 6 Supplementary Material

This appendix lists the various additional material relevant to our study on continuous practices. It includes the following material:

- a list of our interview questions,
- and the ethics certificate this study falls under.

B.1 Interview Questions

Because our interviews were semi-structured in nature, we started out with a set of prepared questions, asking follow-ups when necessary. The prepared questions included:

1. What is your position at ORG?
2. Are you familiar with the concept of continuous practices (integration, deployment)?
3. How are you applying continuous practices at your organization?
4. What impact do you perceive continuous practices have had at your organization?

For the project you are most familiar with:

5. How is your source code organized (repository structure)? Why/Why not?
6. Do you have automated builds? What is the extent of your automation? Why/Why not?
7. Are tests included in your builds? Why/Why not?
8. What types of tests are included in the build? Why/Why not?
9. How often do you commit to the mainline? Why?
10. Describe how a feature you've developed gets merged to the mainline.
11. Does every commit to the mainline result in a build? Why/Why not?
12. How long would you estimate is your build? Why?
13. How similar is your development environment to your production environment? Why?
14. How much effort is required for anyone to run your project on their machine? Why?

15. Does everyone know what is going on (in terms of building, feature release, test status, ...etc.)? Why/Why not?
    (a) Does everyone have visibility into what builds are running, their status, and previous builds? Why/Why not?
    (b) Does everyone have visibility into what features are included in every build? Why/Why not?
16. Is deployment automation? What is the extent of your deployment automation? Why/Why not?
    (a) How long does deployment take? Why?
    (b) How many people can deploy (if manual)? Why?
    (c) How often do you deploy? Why?
    (d) How often does a deployment fail?
    (e) What are the common causes for deployment failure?
    (f) How long does it take to repair (or rollback) a deployment?
    (g) How often do you have to repair (or rollback) a failing deployment? Why?

University of Victoria

Office of Research Services | Human Research Ethics Board

[redacted]

Certificate of Approval

PRINCIPAL INVESTIGATOR	**Margaret-Anne Storey** (Supervisor)	**ETHICS PROTOCOL NUMBER** Expedited review - delegated	**19-0213**
PRINCIPAL APPLICANT	**Omar Elazhary** **PhD student**	ORIGINAL APPROVAL DATE	26-Nov-2019
UVIC DEPARTMENT	**Computer Science**	APPROVED ON	26-Nov-2019
		APPROVAL EXPIRY DATE	25-Nov-2020

PROJECT TITLE **A Study of the Interplay between Software Developers and Software Development Automation**

RESEARCH TEAM MEMBERS
Neil Ernst - Co-investigator, Assistant Professor

DECLARED PROJECT FUNDING
NSERC Discovery, University of Victoria

DOCUMENTS INCLUDED IN THIS APPROVAL
consent_form_explicit_interviews_modified.pdf - 21-Nov-2019
Survey Questions_v1.pdf - 25-Oct-2019
consent_form_implied_survey.pdf - 15-Oct-2019
Interview Questions_v1.pdf - 15-Oct-2019
cse_workflow.png - 15-Oct-2019
Parasuraman_Performance consequences of automation-induced complacency.pdf - 15-Oct-2019
Parasuraman_Complacency and bias in human use of automation_An attentional integration.pdf - 15-Oct-2019
Fowler_Continuous integration.pdf - 15-Oct-2019

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a “Request for Modification” form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a “Request for Renewal” form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a “Notice of Project Completion” form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropiate standards of ethcis as outlines by the University of Victoria Research Regulations Involving Human Participants.

[redacted]

Dr. Rachael Scarth
Associate VP Research Operations

Certificate Issued On: 26-Nov-2019

C Chapter 8 Supplementary Material

This appendix includes supplementary material relevant to our exploratory study of socio-technical aspects of continuous software engineering discussed in Chapter 8. In this appendix we include the following:

1. a table of the papers in our sample along with their identifiers in Table C.1,
2. and the ethics application this study falls under.

University of Victoria

Office of Research Services | Human Research Ethics Board

[redacted]

Certificate of Approval - Annual Renewal

PRINCIPAL INVESTIGATOR	**Margaret-Anne Storey** (Supervisor)	**ETHICS PROTOCOL NUMBER** Expedited review - delegated	**19-0213**
PRINCIPAL APPLICANT	**Omar Elazhary** **PhD student**	ORIGINAL APPROVAL DATE	26-Nov-2019
UVIC DEPARTMENT	**Computer Science COSI**	APPROVED ON	23-Nov-2020
		APPROVAL EXPIRY DATE	25-Nov-2021

PROJECT TITLE **A Study of the Interplay between Software Developers and Software Development Automation**

RESEARCH TEAM MEMBERS
Neil Ernst - Co-investigator, Assistant Professor

DECLARED PROJECT FUNDING
NSERC Discovery, University of Victoria

DOCUMENTS INCLUDED IN THIS APPROVAL
Fowler_Continuous integration.pdf - 15-Oct-2019
Parasuraman_Complacency and bias in human use of automation_An attentional integration.pdf - 15-Oct-2019
Parasuraman_Performance consequences of automation-induced complacency.pdf - 15-Oct-2019
cse_workflow.png - 15-Oct-2019
Interview Questions_v1.pdf - 15-Oct-2019
consent_form_implied_survey.pdf - 15-Oct-2019
Survey Questions_v1.pdf - 25-Oct-2019
consent_form_explicit_interviews_modified.pdf - 21-Nov-2019

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a “Request for Modification” form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a “Request for Renewal” form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a “Notice of Project Completion” form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropiate standards of ethcis as outlines by the University of Victoria Research Regulations Involving Human Participants.

[redacted]

Dr. Rachael Scarth
Associate VP Research Operations

Certificate Issued On: 23-Nov-2020

Table C.1: A list of the 46 papers included in our sample along with relevant information (identifier, outlet, year)

Paper Title	Paper Identifier	Publication Outlet	Year
BuildFast: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration	A01	ASE	2020
PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects	A02	ASE	2020
Team Discussions and Dynamics during DevOps Tool Adoptions in OSS Projects	A03	ASE	2020
JITBot: An Explainable Just-in-Time Defect Prediction Bot	A04	ASE	2020
Continuous compliance	A05	ASE	2020
Automated Trainability Evaluation for Smart Software Functions	A06	ASE	2019
RefBot: Intelligent Software Refactoring Bot	A07	ASE	2019
Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI	A08	ASE	2018
Continuous Code Quality: Are We (Really) Doing That?	A09	ASE	2018
The impact of continuous integration on other software development practices: A large-scale empirical study	A10	ASE	2017
Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects	A11	ASE	2016
The Impact of Structure on Software Merging: Semistructured Versus Structured Merge	A12	ASE	2019
Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing	A13	ASE	2019
Scalable incremental building with dynamic task dependencies	A14	ASE	2018
Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab	F01	ESEC/FSE	2020
Dimensions of Software Configuration: On the Configuration Context in Modern Software Development	F02	ESEC/FSE	2020
Heard It through the Gitvine: An Empirical Study of Tool Diffusion across the npm Ecosystem	F03	ESEC/FSE	2020
Understanding Build Issue Resolution in Practice: Symptoms and Fix Patterns	F04	ESEC/FSE	2020
A Comprehensive Study on Challenges in Deploying Deep Learning Based Software	F05	ESEC/FSE	2020
A Large-Scale Empirical Study of Compiler Errors in Continuous Integration	F06	ESEC/FSE	2019
A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI	F07	ESEC/FSE	2019
Releasing Fast and Slow: An Exploratory Case Study at ING	F08	ESEC/FSE	2019
One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows	F09	ESEC/FSE	2018
Building Lean Continuous Integration and Delivery Pipelines by Applying DevOps Principles: A Case Study at Varidesk	F10	ESEC/FSE	2018
The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds	F11	ESEC/FSE	2018
Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility	F12	ESEC/FSE	2017
Screening Heuristics for Project Gating Systems	F13	ESEC/FSE	2017
Strong Agile Metrics: Mining Log Data to Determine Predictive Power of Software Metrics for Continuous Delivery Teams	F14	ESEC/FSE	2017
Measuring the cost of regression testing in practice: a study of Java projects using continuous integration	F15	ESEC/FSE	2017
Continuous Deployment of Mobile Software at Facebook (Showcase)	F16	ESEC/FSE	2016
Build System with Lazy Retrieval for Java Projects	F17	ESEC/FSE	2016
Learning for Test Prioritization: An Industrial Case Study	F18	ESEC/FSE	2016
Dads: dynamic slicing continuously-running distributed programs with budget constraints	F19	ESEC/FSE	2020
Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration	I01	ICSE	2020
A cost-efficient approach to building in continuous integration	I02	ICSE	2020
Practical fault detection in puppet programs	I03	ICSE	2020
Learning from, understanding, and supporting DevOps artifacts for docker	I04	ICSE	2020
FastLane: Test Minimization for Rapidly Deployed Large-Scale Online Services	I05	ICSE	2019
Automated Reporting of Anti-Patterns and Decay in Continuous Integration	I06	ICSE	2019
BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes	I07	ICSE	2019
DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets	I08	ICSE	2019
DeFlaker: automatically detecting flaky tests	I09	ICSE	2018
Automated localization for unreproducible builds	I10	ICSE	2018
Redefining prioritization: continuous prioritization for continuous integration	I11	ICSE	2018
HireBuild: an automatic approach to history-driven repair of build scripts	I12	ICSE	2018
What causes my test alarm?: automatic cause analysis for test alarms in system and integration testing	I13	ICSE	2017

D Noun Project Attributions

Below are attributions to the original authors of images and figures used throughout this document.

- API by faisalovers from the Noun Project
- Document by ibrandify from the Noun Project
- push by Marko Fuček from the Noun Project
- Mining by art shop from the Noun Project
- users by alvianwijaya from the Noun Project
- Lens by Alexandr Cherkinsky from the Noun Project
- interview by Gan Khoon Lay from the Noun Project
- Survey by unlimicon from the Noun Project
- comment by arjuazka from the Noun Project
- slack by Danil Polshin from the Noun Project

Bibliography

[1] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[2] "Continuous integration certification," https://martinfowler.com/bliki/ContinuousIntegrationCertification.html, accessed: 2018-10-11.

[3] M. Beller, "Toward an empirical theory of feedback-driven development," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 503–505.

[4] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[5] J. G. Hall and L. Rapanotti, "A design theory for software engineering," *Information and Software Technology*, vol. 87, pp. 46–61, 2017.

[6] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, 2021.

[7] O. Elazhary, M.-A. Storey, N. Ernst, and A. Zaidman, "Do as i do, not as i say: Do contribution guidelines match the github contribution process?" in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 286–290.

[8] C. Werner, Z. S. Li, D. Lowlind, O. Elazhary, N. Ernst, and D. Damian, "Continuously managing nfrs: Opportunities and challenges in practice," *IEEE Transactions on Software Engineering*, 2021.

[9] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf*, vol. 122, p. 14, 2006.

[10] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 805–816.

[11] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, and M. Stumm, "Continuous deployment of mobile software at facebook (showcase)," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 12–23.

[12] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: a large-scale empirical study," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 60–71.

[13] M. Fowler and M. Foemmel, "Continuous integration (original version)," *Available from Martin Fowler, http://www. martinfowler. com/) Retrieved February*, vol. 9, p. 2007, 2000.

[14] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM, 2014, pp. 1–9.

[15] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[16] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: the integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 358–368.

[17] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 285–296.

[18] M.-A. Storey, T. Zimmermann, C. Bird, J. Czerwonka, B. Murphy, and E. Kalliamvakou, "Towards a theory of software developer job sat-

isfaction and perceived productivity," *IEEE Transactions on Software Engineering*, 2019.

[19] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in github," in *Proceedings of the 36th international conference on Software engineering*. ACM, 2014, pp. 356–366.

[20] M. Petre, J. Buckley, L. Church, M.-A. Storey, and T. Zimmermann, "Behavioral science of software engineering," *IEEE Software*, vol. 37, no. 6, pp. 21–25, 2020.

[21] R. Souza and B. Silva, "Sentiment analysis of travis ci builds," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 459–462.

[22] Y. Gupta, Y. Khan, K. Gallaba, and S. McIntosh, "The impact of the adoption of continuous integration on developer attraction and retention," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 491–494.

[23] O. Elazhary, M.-A. Storey, N. A. Ernst, and E. Paradis, "Adept: A socio-technical theory of continuous integration," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2021, pp. 26–30.

[24] G. Booch, *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., 1990.

[25] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[26] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Addison-Wesley Boston, 2011.

[27] L. Lehtola, M. Kauppinen, J. Vähäniitty, and M. Komssi, "Linking business and requirements engineering: is solution planning a missing activity in software product companies?" *Requirements engineering*, vol. 14, no. 2, pp. 113–128, 2009.

[28] G. Lohan, "A brief history of budgeting: Reflections on beyond budgeting, its link to performance management and its appropriateness for software development," in *Lean Enterprise Software and Systems*. Springer, 2013, pp. 81–105.

[29] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Continuous verification of large embedded software using smt-based bounded model checking," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*. IEEE, 2010, pp. 160–169.

[30] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 2003, pp. 281–292.

[31] B. Fitzgerald, K.-J. Stol, R. O'Sullivan, and D. O'Brien, "Scaling agile methods to regulated environments: An industry case study," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 863–872.

[32] M. Merkow and L. Raghavan, "An ecosystem for continuously secure application software," *CrossTalk, March/April*, 2011.

[33] C. Del Rosso, "Continuous evolution through software architecture evaluation: a case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 5, pp. 351–383, 2006.

[34] L. Gebauer, M. Söllner, and J. Leimeister, "Towards understanding the formation of continuous it use," 2013.

[35] T. Zhou, "An empirical examination of continuance intention of mobile payment services," *Decision support systems*, vol. 54, no. 2, pp. 1085–1091, 2013.

[36] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst, "Continuous monitoring of software services: Design and application of the kieker framework," 2009.

[37] X. Chen, P. Sorenson, and J. Willson, "Continuous spa: Continuous assessing and monitoring software process," in *Services, 2007 IEEE Congress on*. IEEE, 2007, pp. 153–158.

[38] H. Olsson Holmström, H. Alahyari, and J. Bosch, "Climbing the" stairway to heaven" a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012, pp. 392–399.

[39] F. Fagerholm, A. S. Guinea, H. Mäenpää, and J. Münch, "Building blocks for continuous experimentation," in *Proceedings of the 1st international workshop on rapid continuous software engineering*. ACM, 2014, pp. 26–35.

[40] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Relationship of devops to agile, lean and continuous deployment," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2016, pp. 399–415.

[41] M. C. Paulk, "Extreme programming from a cmm perspective," *IEEE software*, vol. 18, no. 6, pp. 19–26, 2001.

[42] L. Lindstrom and R. Jeffries, "Extreme programming and agile software development methodologies," *Information systems management*, vol. 21, no. 3, pp. 41–52, 2004.

[43] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," *Information and Software Technology*, vol. 82, pp. 55–79, 2017.

[44] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini, "Canaryadvisor: a statistical-based tool for canary testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 418–422.

[45] V. Pulkkinen, "Continuous deployment of software," in *Proc. of the Seminar*, vol. 58312107, 2013, pp. 46–52.

[46] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the 14th working conference on mining software repositories*, 2017.

[47] M. Meyer, "Continuous integration and its tools," *IEEE software*, vol. 31, no. 3, pp. 14–16, 2014.

[48] S. Kim, S. Park, J. Yun, and Y. Lee, "Automated continuous integration of component-based software: An industrial experience," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 423–426.

[49] T. Sundberg, "Continuous integration–how do you know that your application still works?" in *International Conference on Agile Processes and Extreme Programming in Software Engineering*. Springer, 2009, pp. 224–225.

[50] A. Miller, "A hundred days of continuous integration," in *Agile, 2008. AGILE'08. Conference*. IEEE, 2008, pp. 289–293.

[51] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, and T. Fritz, "The work life of developers: Activities, switches and perceived productivity," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1178–1193, 2017.

[52] M. Brandtner, E. Giger, and H. Gall, "Supporting continuous integration by mashing-up software quality information," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 184–193.

[53] M.-A. Storey, A. Zagalsky, L. Singer, D. German *et al.*, "How social and communication channels shape and challenge a participatory culture in software development," *IEEE Transactions on Software Engineering*, no. 1, pp. 1–1, 2017.

[54] M. Brandtner, E. Giger, and H. Gall, "Sqa-mashup: A mashup framework for continuous integration," *Information and Software Technology*, vol. 65, pp. 97–113, 2015.

[55] C. E. Brandt, A. Panichella, A. Zaidman, and M. Beller, "Logchunks: A data set for build log analysis," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 583–587.

[56] K. Berg, "4 common problems with continuous integration and deployment and how to avoid them," https://tinyurl.com/ybp9n9cn, 2019, [Online; accessed 14-July-2020].

[57] M. Heller, "Continuous integration: The answer to life, the universe, and everything?" https://tinyurl.com/y8rhxnt8, 2015, [Online; accessed 14-July-2020].

[58] Y. Bugayenko, "Continuous integration is dead," https://tinyurl.com/ybsuvhxr, 2014, [Online; accessed 14-July-2020].

[59] P. Duvall, "Continuous delivery patterns and antipatterns in the software lifecycle," https://tinyurl.com/yb2o78m6, 2011, [Online; accessed 07-July-2020.

[60] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1095–1135, 2020.

[61] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. K. Smith, C. Winter, and E. Murphy-Hill, "Advantages and disadvantages of a monolithic repository: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 225–234.

[62] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 102–112.

[63] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at google," in *2012 Third International Workshop on Managing Technical Debt (MTD)*. IEEE, 2012, pp. 1–6.

[64] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 426–437.

[65] R. Parasuraman and D. H. Manzey, "Complacency and bias in human use of automation: An attentional integration," *Human factors*, vol. 52, no. 3, pp. 381–410, 2010.

[66] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci," *IEEE Transactions on Software Engineering*, 2018.

[67] D. Ståhl and J. Bosch, "Experienced benefits of continuous integration in industry software product development: A case study," in *The 12th IASTED International Conference on Software Engineering*, 2013, pp. 736–743.

[68] D. Ståhl, A. Martini, and T. Mårtensson, "Big bangs and small pops: on critical cyclomatic complexity and developer integration behavior," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 81–90.

[69] M. R. Islam and M. F. Zibran, "Insights into continuous integration build failures," in *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 2017, pp. 467–470.

[70] L. Hukkanen *et al.*, "Adopting continuous integration-a case study," 2015.

[71] M. Schluse and J. Rossmann, "From simulation to experimentable digital twins: Simulation-based development and operation of complex technical systems," in *2016 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, 2016, pp. 1–6.

[72] A. Barbie, W. Hasselbring, and N. Pech, "Continuous integration testing of embedded software with digital twin prototypes," 2021.

[73] T. B. Sheridan and W. L. Verplank, "Human and computer control of undersea teleoperators," MASSACHUSETTS INST OF TECH CAMBRIDGE MAN-MACHINE SYSTEMS LAB, Tech. Rep., 1978.

[74] L. Save, B. Feuerberg, and E. Avia, "Designing human-automation interaction: a new level of automation taxonomy," *Proc. Human Factors of Systems and Technology*, vol. 2012, 2012.

[75] R. Parasuraman, T. B. Sheridan, and C. D. Wickens, "A model for types and levels of human interaction with automation," *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and Humans*, vol. 30, no. 3, pp. 286–297, 2000.

[76] R. Parasuraman and V. Riley, "Humans and automation: Use, misuse, disuse, abuse," *Human factors*, vol. 39, no. 2, pp. 230–253, 1997.

[77] G. Pinto, M. Rebouças, and F. Castor, "Inadequate testing, time pressure, and (over) confidence: a tale of continuous integration users," in *Proceedings of the 10th International Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE Press, 2017, pp. 74–77.

[78] M. Zolfagharinia, B. Adams, and Y.-G. Guéhénuc, "Do not trust build results at face value-an empirical study of 30 million cpan builds," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 312–322.

[79] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "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*, 2019, pp. 647–658.

[80] P. Rigby, D. German, and M.-A. Storey, "Open source software peer review practices," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 541–550.

[81] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.

[82] "Github octoverse," https://tinyurl.com/x7ap99u3, accessed: 2021-09-22.

[83] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[84] K. Petersen and C. Gencel, "Worldviews, research methods, and their relationship to validity in empirical software engineering research," in *2013 joint conference of the 23rd international workshop on software measurement and the 8th international conference on software process and product measurement*. IEEE, 2013, pp. 81–89.

[85] V. Kaushik and C. A. Walsh, "Pragmatism as a research paradigm and its implications for social work research," *Social Sciences*, vol. 8, no. 9, p. 255, 2019.

[86] R. B. Johnson, A. J. Onwuegbuzie, and L. A. Turner, "Toward a definition of mixed methods research," *Journal of mixed methods research*, vol. 1, no. 2, pp. 112–133, 2007.

[87] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *Software maintenance and evolution (icsme), 2014 ieee international conference on*. IEEE, 2014, pp. 401–405.

[88] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran, "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*. ACM, 2018, pp. 8–14.

[89] S. R. Terrell, "Mixed-methods research methodologies," *The qualitative report*, vol. 17, no. 1, pp. 254–280, 2012.

[90] M. Sandelowski, "Combining qualitative and quantitative sampling, data collection, and analysis techniques in mixed-method studies," *Research in nursing & health*, vol. 23, no. 3, pp. 246–255, 2000.

[91] J. Greene and C. McClintock, "Triangulation in evaluation: Design and analysis issues," *Evaluation review*, vol. 9, no. 5, pp. 523–545, 1985.

[92] R. K. Yin, *Case study research and applications: Design and methods*. Sage publications, 2017.

[93] J. W. Creswell, *A concise introduction to mixed methods research.* Sage Publications, 2014.

[94] D. L. Driscoll, A. Appiah-Yeboah, P. Salib, and D. J. Rupert, "Merging qualitative and quantitative data in mixed methods research: How to and why not," 2007.

[95] G. A. A. Prana, C. Treude, F. Thung, T. Atapattu, and D. Lo, "Categorizing the content of github readme files," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1296–1327, 2019.

[96] G. Gousios, "The ghtorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[97] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[98] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[99] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 334–344.

[100] "Github 2018 octoverse report," https://tinyurl.com/52fuujx2, accessed: 2021-07-09.

[101] O. Elazhary, M.-A. Storey, N. Ernst, and A. Zaidman, "Do as i do, not as i say: Do contribution guidelines match the github contribution process?" Oct 2020. [Online]. Available: https://tinyurl.com/537h9cyd

[102] D. Ståhl, K. Hallén, and J. Bosch, "Continuous integration and delivery traceability in industry: Needs and practices," in *Software Engineering*

and Advanced Applications (SEAA), 2016 42th Euromicro Conference on. IEEE, 2016, pp. 68–72.

[103] ——, "Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework," *Empirical Software Engineering*, vol. 22, no. 3, pp. 967–995, 2017.

[104] "Setting guidelines for repository contributors," https://tinyurl.com/3pcj4pf6, accessed: 2019-06-10.

[105] I. Steinmacher, M. A. G. Silva, M. A. Gerosa, and D. F. Redmiles, "A systematic literature review on the barriers faced by newcomers to open source software projects," *IST*, vol. 59, pp. 67–85, 2015.

[106] M. R. Roller and P. J. Lavrakas, *Applied qualitative research design: A total quality framework approach*. Guilford Publications, 2015.

[107] J. E. McGrath, "Methodology matters: Doing research in the behavioral and social sciences," in *Readings in Human–Computer Interaction*. Elsevier, 1995, pp. 152–169.

[108] M.-A. Storey, N. A. Ernst, C. Williams, and E. Kalliamvakou, "The who, what, how of software engineering research: a socio-technical framework," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4097–4129, 2020.

[109] M. Glinz, "On non-functional requirements," in *15th IEEE international requirements engineering conference (RE 2007)*. IEEE, 2007, pp. 21–26.

[110] L. Chung and J. C. S. do Prado Leite, "On non-functional requirements in software engineering," in *Conceptual modeling: Foundations and applications*. Springer, 2009, pp. 363–379.

[111] D. Ameller, C. Ayala, J. Cabot, and X. Franch, "Non-functional requirements in architectural decision making," *IEEE software*, vol. 30, no. 2, pp. 61–67, 2012.

[112] C. R. Camacho, S. Marczak, and D. S. Cruzes, "Agile team members perceptions on non-functional testing: influencing factors from an empirical study," in *2016 11th international conference on availability, reliability and security (ARES)*. IEEE, 2016, pp. 582–589.

[113] A. Jarzębowicz and P. Weichbroth, "A systematic literature review on implementing non-functional requirements in agile software development: Issues and facilitating practices," in *International Conference on Lean and Agile Software Development*. Springer, 2021, pp. 91–110.

[114] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 international symposium on empirical software engineering and measurement*. IEEE, 2011, pp. 275–284.

[115] C. Werner, Z. S. Li, D. Lowlind, O. Elazhary, N. Ernst, and D. Damian, "Continuous non-functional requirements: Practices, opportunities, and trade-offs for small, agile organizations," Aug 2019.

[116] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella, "A tale of ci build failures: An open source and a financial organization perspective," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 183–193.

[117] K. Costello, "Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019," https://tinyurl.com/yalom8fc, 2019, [Online; accessed 07-July-2020].

[118] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," Jul 2020.

[119] V. T. Heikkilä, M. Paasivaara, C. Lasssenius, D. Damian, and C. Engblom, "Managing the requirements flow from strategy to release in large-scale agile development: a case study at ericsson," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2892–2936, 2017.

[120] T. Mårtensson, P. Hammarström, and J. Bosch, "Continuous integration is not about build systems," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2017, pp. 1–9.

[121] N. Forsgren, J. Humble, and G. Kim, "Accelerate: The science of lean software and devops," *Portland, OR: ITRevolution*, 2018.

[122] K.-J. Stol and B. Fitzgerald, "Theory-oriented software engineering," *Science of Computer Programming*, vol. 101, pp. 79–98, 2015.

[123] L. Varpio, E. Paradis, S. Uijtdehaage, and M. Young, "The distinctions between theory, theoretical framework, and conceptual framework," *Academic Medicine*, vol. 95, no. 7, pp. 989–994, 2020.

[124] P. Ralph, M. Chiasson, and H. Kelley, "Social theory for software engineering research," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, 2016, pp. 1–11.

[125] P. Johnson and M. Ekstedt, "The tarpit–a general theory of software engineering," *Information and Software Technology*, vol. 70, pp. 181–203, 2016.

[126] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 447–450.

[127] A. Bartusevics and L. Novickis, "Models for implementation of software configuration management," *Procedia Computer Science*, vol. 43, pp. 3–10, 2015.

[128] D. I. Sjøberg, T. Dybå, B. C. Anda, and J. E. Hannay, "Building theories in software engineering," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 312–336.

[129] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12*, 2008, pp. 1–10.

[130] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[131] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in

Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 197–207.

[132] H. Lamba, A. Trockman, D. Armanios, C. Kästner, H. Miller, and B. Vasilescu, "Heard it through the gitvine: an empirical study of tool diffusion across the npm ecosystem," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 505–517.

[133] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "Learning from, understanding, and supporting devops artifacts for docker," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 38–49.

[134] M. S. Linneberg and S. Korsgaard, "Coding qualitative data: A synthesis guiding the novice," *Qualitative research journal*, 2019.

[135] O. Elazhary, E. L. Vargas, A. M. P. Milani, and M.-A. Storey, "Investigating ADEPT's utility: An exploration of the literature," Sep 2021.

[136] E. L. Trist, *The evolution of socio-technical systems*. Ontario Quality of Working Life Centre Toronto, 1981, vol. 2.

[137] G. Baxter and I. Sommerville, "Socio-technical systems: From design methods to systems engineering," *Interacting with computers*, vol. 23, no. 1, pp. 4–17, 2011.