AN ABSTRACT OF THE DISSERTATION OF

Michael Hilton for the degree of Doctor of Philosophy in Computer Science presented on April 17, 2017.

Title: Understanding Software Development and Testing Practices

Abstract approved: ______________________________

Daniel Dig

A bad software development process leads to wasted effort and inferior products. In order to improve a software process, it must be first understood. In this work I focus on understanding software processes.

The first process we seek to understand is Continuous Integration (CI). CI systems automate the compilation, building, and testing of software. Despite CI rising as a big success story in automated software engineering, it has received almost no attention from the research community. For example, how widely is CI used in practice, and what are some costs and benefits associated with CI? Without answering such questions, developers, tool builders, and researchers make decisions based on folklore instead of data.

We use three complementary methods to study the usage of CI in open-source projects. To understand *which* CI systems developers use, we analyzed 34,544 open-source projects from GitHub. To understand *how* developers use CI, we analyzed 1,529,291 builds from the most commonly used CI system. To understand *why* projects use or do not use CI, we surveyed 442 developers. With this data, we answered several key questions related to the usage, costs, and benefits of CI. Among our results, we show evidence that supports the claim that CI helps projects release more often, that CI is widely adopted by the most popular projects, as well as finding that the overall percentage of projects using CI continues to grow, making it important and timely to focus more research on CI.

Furthermore, we present a qualitative study of the barriers and needs developers face when using CI. In this paper, we conduct 16 semi-structured interviews with developers from different industries and development scales. We triangulate our findings by running two surveys. The *Focused Survey* samples 51 developers at a single company. The *Broad Survey* samples a population of 523 developers from all over the world. We identify trade-offs developers face when using and implementing CI. Developers face trade-offs between speed and certainty (*Assurance*), between better access and information security (*Security*), and between more configuration options and better ease of use (*Flexibility*). We present implications of these trade-offs for developers, tool builders, and researchers.

Additionally, we seek to use code and test changes to understand conformance to the Test Driven Development (TDD) process. We designed and implemented TDDViz, a tool that supports developers in better understanding how they conform to TDD. TDDViz supports this understanding by providing novel visualizations of developers' TDD process. To enable TDDViz's visualizations, we developed a novel automatic inferencer that identifies the phases that make up the TDD process solely based on code and test changes.

We evaluate TDDViz using two complementary methods: a controlled experiment with 35 participants to evaluate the visualization, and a case study with 2601 TDD Sessions to evaluate the inference algorithm. The controlled experiment shows that, in comparison to existing visualizations, participants performed significantly better when using TDDViz to answer questions about code evolution. In addition, the case study shows that the inferencing algorithm in TDDViz infers TDD phases with an accuracy (F-measure) of 87%.

Understanding Software Development and Testing Practices

by

Michael Hilton

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented April 17, 2017
Commencement June 2017

ACKNOWLEDGEMENTS

I would like to thank Danny Dig, my advisor, for all the feedback and guidance without whom, this work would not be possible. Many thanks also go to my committee, Margaret Burnett, Carlos Jensen, Ron Metoyer, and Todd Palmer for their encouragement and advice.

I also would like to thank Darko Marinov, for encouraging me to investigate Continuous Integration, and then for all his guidance along the way. This work also benefited from the lessons I learned while working on other projects with Tien N Nguyen, Jonathan Aldrich, Matthew Hammer, and Cyrus Omar.

Thanks to Andy Begel, Matt Hammer, Cyrus Omar, Anita Sarma, Sean McGregor, Souti Chattopadhyay, Mihai Codoban, Kendall Bailey, Amin Alipour, Denis Bogdanas, Mihai Codoban, Alex Gyori, Kory Kraft, Nicholas Lu, Shane McKee, Nicholas Nelson, Semih Okur, August Shi, and Sruti Srinivasa Ragavan, for feedback on earlier versions of these works. Thanks to Lutz Prechelt for his helpful and very thorough review. Thanks to CloudBees for sharing the list of Projects using CloudBees, and Travis for fixing a bug in their API to enable us to collect all relevant build history. Thanks to Brian Marick, Martin Fowler, and Joel Spolsky for promoting the CI survey. Thanks to Matthew Kocher for his help with conducting a survey at Pivotal Labs, and for his valuable insights. Thanks to Jon Jagger for making the TDD data available.

This work was partially funded through the NSF CCF-1421503, CCF-1439957, and CCF-1553741 grants.

Thanks to my fellow students Nick Nelson, Mihai Codoban, Sruti Srinivasa Ragavan, Mohammad Amin Alipour, David Piorkowski and Beatrice Moissinac for the encouragement, advice, laughs, ideas, and inspiration.

Finally, and most importantly, I want to thank my family, who were so supportive of me and without whom I would have never been able to accomplish this. Thanks to my lovely wife Ann, for her constant encouragement and the knowledge that she would be with me no matter what. Thanks to my kids Elena, David, James, and Andrew, for giving me motivation to work hard during the day, and a reason to go home at night. Your unconditional love and support has been my constant companion for these last 4 years.

CONTRIBUTION OF AUTHORS

Dr. Marinov provided inspiration for Chapters 2 and 3. He also assisted with the writing. Timothy Tunnell wrote code for data collection and processing for Chapters 2 and 3. Kai Huang helped find cloudbees projects in Chapter 2. Nicolas Nelson helped with the interview design of Chapter 3, as well as performing qualitative coding on the interview transcripts. He also generated the bar charts. Dr Metoyer provided assistance with the development of the visualization elements. Nicholas Nelson, Hugh McDonald, Sean McDonald all helped develop code to implement the inference algorithm as a part of their summer REU.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

My research is motivated by the decade I spent as a software developer. In my experience, having a good process is crucial for a developer to be successful. However, improving one's process is very difficult. The first step in improving one's process is understanding. There are many ways for a developer to evaluate the product of their work (e.g., the code they write), however, there are not many ways to evaluate the process of their work (e.g., *how* they write code). It is only through understanding developers' processes that we can develop tools that will be most effective at addressing the core needs of developers.

I believe that it is important for developers to better understand their own development, but also for researchers to better understand how software is created, so that they can contribute to process improvement.

My research focuses on increasing understanding of software development processes, specifically focusing on how developers use the process of Continuous Integration(**ASE** [59]). One of the enabling technologies for CI is unit tests. One process that developers often use to write unit tests is Test Driven Development (TDD). My research also helps developers better understand their own development process when using TDD (**XP** [58]).

1.0.1 Continuous Integration

Continuous Integration (CI) is a process that involves merging, building, and testing code continuously, or at least very frequently. It has been widely adopted by the software development industry. Travis CI [17], a popular CI service, reports that over 300,000 projects are using Travis. The State of Agile industry survey [87], with 3,880 participants, found 50% of respondents use CI. The State of DevOps report [91] finds CI to be one of the indicators of "high performing IT organizations". Google Trends [11] shows a steady increase of interest in CI: searches for "Continuous Integration" increased 350% in the last decade.

Despite the growth of CI, the only published research paper related to CI usage [112] is a preliminary study, conducted on 246 projects, which compares several quality metrics of projects that use or do not use CI. However, the study does not present any detailed information on *how* projects use CI. In fact, despite some folkloric evidence about the use of CI, there is no systematic study about the use of CI as a practice.

Not only do we lack basic knowledge about the extent to which open-source projects are adopting CI, but also we have no answers to many important questions related to CI. What are the costs of CI? Does CI deliver on the promised benefits, such as releasing more often, or helping make changes (e.g., to merge pull requests) faster? Do developers maximize the usage of CI? Despite the widespread popularity of CI, we have very little quantitative evidence on its benefits. This lack of knowledge can lead to poor decision making and missed opportunities. Developers who choose not to use CI can be missing out on the benefits of CI. Developers who do choose to use CI might not be using it to its fullest potential. Without knowledge of how CI is being used, tool builders can be misallocating resources instead of having data about where automation and improvements are most needed by their users. By not studying CI, researchers have a blind spot which prevents them from providing solutions to the hard problems that practitioners face.

In this paper we use three complementary methods to study the usage of CI in open-source projects. To understand the extent to which CI has been adopted by developers, and *which* CI systems developers use, we analyzed 34,544 open-source projects from GitHub. To understand *how* developers use CI, we analyzed 1,529,291 builds from Travis CI, the most commonly used CI service for GitHub projects (Section 2.5.1). To understand *why* projects use or do not use CI, we surveyed 442 developers.

With this data, we answer several research questions that we grouped into three themes:

Theme 1: *Usage of CI*
RQ: *What percentage of open-source projects use CI?*
RQ: *What is the breakdown of usage of different CI services?*
RQ: *Do certain types of projects use CI more than others?*
RQ: *When did open-source projects adopt CI?*
RQ: *Do developers plan on continuing to use CI?*
We found that CI is widely used, and the number of projects which are adopting CI is growing. We also found that the most popular projects are most likely to use CI.

Theme 2: *Costs of CI*

RQ: *Why do open-source projects choose* not *to use CI?*
RQ: *How often do projects evolve their CI configuration?*
RQ: *What are some common reasons projects evolve their CI configuration?*
RQ: *How long do CI builds take on average?*
We found that the most common reason why developers are not using CI is lack of familiarity with CI. We also found that the average project makes only 12 changes to their CI configuration file and that many such changes can be automated.

Theme 3: *Benefits of CI*
RQ: *Why do open-source projects choose to use CI?*
RQ: *Do projects with CI release more often?*
RQ: *Do projects which use CI accept more pull requests?*
RQ: *Do pull requests with CI builds get accepted faster (in terms of calendar time)?*
RQ: *Do CI builds fail less on master than on other non-master branches?*
We first surveyed developers about the perceived benefits of CI, then we empirically evaluated these claims. We found that projects that use CI release twice as often as those that do not use CI. We also found that projects with CI accept pull requests faster than projects without CI.

However, we still did not know what barriers and needs developers face when using CI, or what trade-offs developers must make to solve these problems.

To fill in the gaps in knowledge about developers and their use of CI, we ask the following questions: What needs do developers have that are unmet by their current CI system(s)? What problems have developers experienced when configuring and using CI systems(s)? How do developers feel about using CI? Without answers to these questions, *developers* can potentially find CI more obstructive than helpful, *tool builders* will implement unneeded features, and *researchers* will not be aware of areas of CI usage that require further examination and solutions that can further enable practitioners.

To answer these questions, we employ complementary established research methodologies. Our primary methodology is interviews with 16 software developers from 14 different companies of all sizes. To triangulate our findings, we deployed two surveys. The *Focused Survey* samples 51 developers at Pivotal[1]. The *Broad Survey* samples 523 participants,

[1]https://pivotal.io/

of which 95% are from industry, and 70% have 7 or more years of software development experience. The interviews provide the content for the surveys, and the *Focused Survey* provides depth while the *Broad Survey* provides breadth. Analyzing all this data, we answer four research questions:

RQ: *What barriers do developers face when using CI*

RQ: *What unmet needs do developers have with CI tools*

RQ: *Why do developers use CI*

RQ: *What benefits do developers experience when using CI*

Based on our findings, we identify three trade-offs developers face when using CI. Other researchers [90, 93, 117] have identified similar trade-offs in different domains. We name these trade-offs *Assurance*, *Security*, and *Flexibility*.

Assurance describes the trade-off between increasing the added value that extra testing provides, and the extra cost of performing that testing. Rothermel et al. [93] identify this trade-off as a motivation for test prioritization.

Security describes the trade-off between increased security measures, and the ability to access and modify the CI system as needed. Post and Kagan [90] found a third of knowledge workers report security restrictions hinder their ability to perform their jobs. We observe this issue also applies to CI users.

Flexibility describes the trade-off that occurs when developers want systems that are both powerful and highly configurable, yet at the same time, they want those systems to be simple and easy to use. Xu et al. [117] identify the costs of over-configurable systems and found that these systems severely hinder usability. We also observe the tension from this trade-off among developers using CI.

In the context of these three trade-offs, we present implications for three audiences: *Developers*, *Tool Builders*, and *Researchers*. For example, *Developers* face difficult choices about how much testing is enough, and how to choose the right tests to run. *Tool Builders* should create UIs for CI users to configure their CI systems, but these UIs should serialize configurations out to text files so that they can be kept in version control. *Researchers* have much to bring to the CI community, such as helping with fault localization and test parallelization when using CI, and examining the security challenges developers face when using CI.

1.0.2 Software Development Process Understanding Through Visualization

A bad software development process leads to wasted effort and inferior products. Unless we understand how developers are following a process, we cannot improve it.

We use Test Driven Development (TDD) as a case study on how software changes can illuminate the development process. To help developers achieve a better understanding of their process, we examined seminal research [26, 72, 121] that found questions software developers ask. From this research, we focused on three question areas. We felt that the answers to these could provide developers with a better understanding of their process. We choose three questions from the literature to focus on, and they spanned three areas: *identification*, *comprehension*, and *comparability.*

RQ: *"Can we detect strategies, such as test-driven development?" (Identification)* [121]
RQ: *"Why was this code changed or inserted?" (Comprehension)* [72]
RQ: *"How much time went into testing vs. into development?" (Comparability)* [26]

To answer these questions, we use code and test changes to understand conformance to a process. In this paper, we present TDDViz, our tool which provides visualizations that support developers' understanding of how they conform to the TDD process. Our visual design is meant to answer RQ1-3 so that we ensure that our visualizations support developers in answering important questions about *identification*, *comprehension*, and *comparability* of code.

In order to enable these visualizations, we designed a novel algorithm to infer TDD phases. Given a sequence of code edits and test runs, TDDViz uses this algorithm to automatically detect changes that follow the TDD process. Moreover, the inferencer also associates specific code changes with specific parts of the TDD process. The inferencer is crucial for giving developers higher-level information that they need to improve their process.

One fundamental challenge for the inferencer is that during the TDD practice, not all code is developed according to the textbook definition of TDD. Even experienced TDD developers often selectively apply TDD during code development, and only on some parts of their code. This introduces lots of noise for any tool that checks conformance to proceses. To ensure that our inference algorithm can correctly handle noisy data, we add a fourth Research Question.

RQ: *"Can an algorithm infer TDD phases accurately?" (Accuracy)*

To answer this question, in this paper we use a corpus of data from cyber-dojo[2] , a website that allows developers to practice and improve their TDD by coding solutions to various programming problems. Each time a user run tests, the code is committed to a git repository. Each of these commits becomes a fine-grained commit. Our corpus contains a total of 41766 fine-grained snapshots from 2601 programming sessions, each of which is an attempt to solve one of 30 different programming tasks.

To evaluate TDDViz, we performed a controlled experiment with 35 student participants already familiar with TDD. Our independent variable was using TDDViz or existing visualizations to answer questions about the TDD Process.

1.0.3 Contributions

This dissertation makes the following contributions:

- **Research Questions:** We designed 14 novel research questions about CI. We are the first to provide in-depth answers to questions about the usage, costs, and benefits of CI.
- **Data Analysis:** We collected and analyzed CI usage data from 34,544 open-source projects. Then we analyzed in-depth all CI data from a subset of 620 projects and their 1,529,291 builds, 1,503,092 commits, and 653,404 pull requests. Moreover, we surveyed 442 open-source developers about why they chose to use or not use CI.
- **Implications:** We provide practical implications of our findings from the perspective of three audiences: researchers, developers, and tool builders. Researchers should pay attention to CI because it is not a passing fad. For developers we list several situations where CI provides the most value.
- **Interviews:**We conduct exploratory semi-structured interviews with 16 developers, then triangulate these findings with a *Focused Survey* of 51 developers at Pivotal and a *Broad Survey* of 523 developers from all over the world.
- **Motivations:**We provide an empirically justified set of developers' motivations for using CI.

[2]www.cyberdojo.org

- **Gaps:**We expose gaps between developers' needs and existing tooling for CI.
- **Process Conformance:** We propose a novel usage of software changes to infer conformance to a process. Instead of analyzing metrics taken at various points in time, we analyze deltas (i.e., the changes in code and tests) to understand conformance to TDD.
- **TDD Visualization Design and Analysis:** We present a visualization designed specifically for understanding conformance to TDD. Our visualizations show the presence or absence of TDD and allow progressive disclosure of TDD activities.
- **TDD Phase Inference Algorithm:** We present the first algorithm to infer the activities in the TDD process solely based on snapshots taken when tests are run.
- **Implementation and Empirical Evaluation:** We implement the visualization and inference algorithm in TDDViz, and empirically evaluate it using two complementary methods. First, we conduct a controlled experiment with 35 participants, in order to answer **RQ1-3**. Second, we evaluate the accuracy of our inferencer using a corpus of 2601 TDD sessions from cyber-dojo, in order to answer **RQ4.** Our inferencer achieves an accuracy of 87%. Together, both of these show that TDDViz is effective.

Chapter 2: Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects

Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, Danny Dig

Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)

2.1 Abstract

Continuous integration (CI) systems automate the compilation, building, and testing of software. Despite CI rising as a big success story in automated software engineering, it has received almost no attention from the research community. For example, how widely is CI used in practice, and what are some costs and benefits associated with CI? Without answering such questions, developers, tool builders, and researchers make decisions based on folklore instead of data.

In this paper, we use three complementary methods to study the usage of CI in open-source projects. To understand *which* CI systems developers use, we analyzed 34,544 open-source projects from GitHub. To understand *how* developers use CI, we analyzed 1,529,291 builds from the most commonly used CI system. To understand *why* projects use or do not use CI, we surveyed 442 developers. With this data, we answered several key questions related to the usage, costs, and benefits of CI. Among our results, we show evidence that supports the claim that CI helps projects release more often, that CI is widely adopted by the most popular projects, as well as finding that the overall percentage of projects using CI continues to grow, making it important and timely to focus more research on CI.

2.2 Introduction

Continuous Integration (CI) is emerging as one of the biggest success stories in automated software engineering. CI systems automate the compilation, building, testing and deployment of software. For example, such automation has been reported [22] to help Flickr deploy to production more than 10 times per day. Others [62] claim that by adopting CI and a more agile planning process, a product group at HP reduced development costs by 78%.

These success stories have led to CI growing in interest and popularity. Travis CI [17], a popular CI service, reports that over 300,000 projects are using Travis. The State of Agile industry survey [87], with 3,880 participants, found 50% of respondents use CI. The State of DevOps report [91] finds CI to be one of the indicators of "high performing IT organizations". Google Trends [11] shows a steady increase of interest in CI: searches for “Continuous Integration” increased 350% in the last decade.

Despite the growth of CI, the only published research paper related to CI usage [112] is a preliminary study, conducted on 246 projects, which compares several quality metrics of projects that use or do not use CI. However, the study does not present any detailed information on *how* projects use CI. In fact, despite some folkloric evidence about the use of CI, there is no systematic study about CI systems.

Not only do we lack basic knowledge about the extent to which open-source projects are adopting CI, but also we have no answers to many important questions related to CI. What are the costs of CI? Does CI deliver on the promised benefits, such as releasing more often, or helping make changes (e.g., to merge pull requests) faster? Do developers maximize the usage of CI? Despite the widespread popularity of CI, we have very little quantitative evidence on its benefits. This lack of knowledge can lead to poor decision making and missed opportunities. Developers who choose not to use CI can be missing out on the benefits of CI. Developers who do choose to use CI might not be using it to its fullest potential. Without knowledge of how CI is being used, tool builders can be misallocating resources instead of having data about where automation and improvements are most needed by their users. By not studying CI, researchers have a blind spot which prevents them from providing solutions to the hard problems that practitioners face.

In this paper we use three complementary methods to study the usage of CI in open-source projects. To understand the extent to which CI has been adopted by developers, and *which* CI systems developers use, we analyzed 34,544 open-source projects from GitHub. To understand *how* developers use CI, we analyzed 1,529,291 builds from Travis CI, the most commonly used CI service for GitHub projects (Section 2.5.1). To understand *why* projects use or do not use CI, we surveyed 442 developers.

With this data, we answer several research questions that we grouped into three themes:

Theme 1: *Usage of CI*
RQ: *What percentage of open-source projects use CI?*
RQ: *What is the breakdown of usage of different CI services?*
RQ: *Do certain types of projects use CI more than others?*
RQ: *When did open-source projects adopt CI?*
RQ: *Do developers plan on continuing to use CI?*
We found that CI is widely used, and the number of projects which are adopting CI is growing. We also found that the most popular projects are most likely to use CI.

Theme 2: *Costs of CI*
RQ: *Why do open-source projects choose* not *to use CI?*
RQ: *How often do projects evolve their CI configuration?*
RQ: *What are some common reasons projects evolve their CI configuration?*
RQ: *How long do CI builds take on average?*
We found that the most common reason why developers are not using CI is lack of familiarity with CI. We also found that the average project makes only 12 changes to their CI configuration file and that many such changes can be automated.

Theme 3: *Benefits of CI*
RQ: *Why do open-source projects choose to use CI?*
RQ: *Do projects with CI release more often?*
RQ: *Do projects which use CI accept more pull requests?*
RQ: *Do pull requests with CI builds get accepted faster (in terms of calendar time)?*
RQ: *Do CI builds fail less on master than on other non-master branches?*
We first surveyed developers about the perceived benefits of CI, then we empirically evaluated these claims. We found that projects that use CI release twice as often as those that do not use CI. We also found that projects with CI accept pull requests faster than

projects without CI.

This paper makes the following contributions:

1. **Research Questions:** We designed 14 novel research questions. We are the first to provide in-depth answers to questions about the usage, costs, and benefits of CI.
2. **Data Analysis:** We collected and analyzed CI usage data from 34,544 open-source projects. Then we analyzed in-depth all CI data from a subset of 620 projects and their 1,529,291 builds, 1,503,092 commits, and 653,404 pull requests. Moreover, we surveyed 442 open-source developers about why they chose to use or not use CI.
3. **Implications:** We provide practical implications of our findings from the perspective of three audiences: researchers, developers, and tool builders. Researchers should pay attention to CI because it is not a passing fad. For developers we list several situations where CI provides the most value. Moreover, we discovered several opportunities where automation can be helpful for tool builders.

More details about our data sets and results are available at
`http://cope.eecs.oregonstate.edu/CISurvey`

2.3 Overview of CI

2.3.1 History and Definition of CI

The idea of Continuous Integration (CI) was first introduced in 1991 by Grady Booch [31], in the context of object-oriented design: "At regular intervals, the process of *continuous integration* yields executable releases that grow in functionality at every release..." This idea was then adopted as one of the core practices of Extreme Programming (XP) [25].

However, the idea began to gain acceptance after a blog post by Martin Fowler [51] in 2000. The motivating idea of CI is that the more often a project can integrate, the better off it is. The key to making this possible, according to Fowler, is automation. Automating the build process should include retrieving the sources, compiling, linking, and running automated tests. The system should then give a "yes" or "no" indicator of whether the build was successful. This automated build process can be triggered either manually or automatically by other actions from the developers, such as checking in new code into version control.

These ideas were implemented by Fowler in CruiseControl [9], the first CI system, which was released in 2001. Today there are over 40 different CI systems, and some of the most well-known ones include Jenkins [12] (previously called Hudson), Travis CI [17], and Microsoft Team Foundation Server (TFS) [15]. Early CI systems usually ran locally, and this is still widely done for Jenkins and TFS. However, CI as a service has become more and more popular, e.g., Travis CI is only available as a service, and even Jenkins is offered as a service via the CloudBees platform [6].

2.3.2 Example Usage of CI

We now present an example of CI that comes from our data. The pull request we are using can be found here: `https://github.com/RestKit/RestKit/pull/2370`. A developer named "Adlai-Holler" created pull request #2370 named "Avoid Flushing In-Memory Managed Object Cache while Accessing" to work around an issue titled "Duplicate objects created if inserting relationship mapping using RKInMemoryManagedObjectCache" for the project *RestKit* [13]. The developer made two commits and then created a pull request, which triggered a Travis CI build. The build failed, because of failing unit tests. A RestKit project member,"segiddins", then commented on the pull request, and asked Adlai-Holler to look into the test failures. Adlai-Holler then committed two new changes to the same pull request. Each of these commits triggered a new CI build. The first build failed, but the second was successful. Once the CI build passed, the RestKit team member commented "seems fine" and merged the pull request.

2.4 Methodology

To understand the extent to which CI is used and *which* CI systems developers use, we analyzed 34,544 open-source projects from GitHub with our *breadth corpus.* To understand *how* developers use CI, we analyzed 1,529,291 builds on the most popular CI system in our *depth corpus.* To understand *why* projects use or do not use CI, we surveyed 442 developers.

2.4.1 Breadth Corpus

The breadth corpus has a large number of projects, and information about what CI services each project uses. We use the breadth corpus to answer broad questions about the usage of CI in open-source projects. We collected the data for this corpus primarily via the GitHub API. We first sorted GitHub projects by their popularity, using the star rating (whereby users can mark, or "star", some projects that they like, and hence each project can accumulate stars). We started our inspection from the top of the list, first by manually looking at the top 50 projects. We collected all publicly available information about how these projects use CI. We then used what we learned from this manual inspection to write a script to programmatically classify which CI service (if any) a project uses. The four CI services that we were able to readily identify manually and later by our script are (sorted in the order of their usage): Travis CI [17], CircleCI [5], AppVeyor [2], and Werker [18]. All of these services provide public API's which we queried to determine if a project is using that service.

Moreover, we wanted to ensure that we had collected as complete data as possible. When we examined the data by hand, we found that several projects were using CloudBees [6], a CI service powered by the Jenkins CI. However, given a list of GitHub projects, there is no reliable way to programmatically identify from the GitHub API which projects use CloudBees. (In contrast, Travis CI uses the same organization and project names as GitHub, making it easy to check correspondence between Travis CI and GitHub projects.) We contacted CloudBees, and they sent us a list of open-source projects that have CloudBees build set up. We then wrote a script to parse that list, inspect the build information, and search for the corresponding GitHub repository (or repositories) for each build on CloudBees. We then used this data to identify the projects from our breadth corpus that use CloudBees. This yielded 1,018 unique GitHub repositories/projects. To check whether these projects refer to CloudBees, we searched for (case insensitive) "CloudBees" in the README files of these projects and found that only 256 of them contain "CloudBees". In other words, had we not contacted CloudBees directly, using only the information available on GitHub, we would have missed a large number of projects that use CloudBees.

Overall, the breadth corpus consists of 34,544 projects. For each project, we collected the following information: project name and owner, the CI system(s) that the project uses (if any), popularity (as measured by the number of stars), and primary programming language (as determined by GitHub).

2.4.2 Depth Corpus

The depth corpus has fewer projects, but for each project we collect all the information that is publicly available. For this subset of projects, we collected additional data to gain a deeper understanding of the usage, costs, and benefits of CI. Analyzing our breadth corpus, as discussed in Section 2.5.1, we learned that Travis CI is by far the most commonly used CI service among open-source projects. Therefore, we targeted projects using Travis CI for our depth corpus. First, we collected the top 1,000 projects from GitHub ordered by their popularity. Of those 1,000 projects, we identified 620 projects that use Travis CI, 37 use AppVeyor, 166 use CircleCI, and 3 use Werker. We used the Travis CI API[1] to collect the entire build history for each project in our depth corpus, for a total of 1,529,291 builds. Using GHTorrent [56], we collected the full history of pull requests for each project, for a total of 653,404 pull requests. Additionally, we cloned every project in our corpus, to access the entire commit history and source code.

2.4.3 Survey

Even after collecting our diverse breadth and depth corpora, we were still left with questions that we could not answer from the online data alone. These questions were about *why* developers chose to use or not use CI. We designed a survey to help us answer a number of such "why" questions, as well as to provide us another data source to better understand CI usage. We deployed our survey by sending it to all the email addresses publicly listed as belonging to the organizations of all the top 1,000 GitHub projects (again rated by the popularity). In total, we sent 4,508 emails.

Our survey consisted of two flows, each with three questions. The first question in both flows asked if the participant used CI or not. Depending on the answer they gave to this question, the second question asked the reasons why they use or do not use CI. These questions were multiple-choice, multiple-selection questions where the users were asked to

[1]We are grateful to the Travis CI developers for promptly resolving a bug report that we submitted; prior to them resolving this bug report, one could not query the full build history of all projects.

Table 2.1: Breadth corpus projects CI usage

Project Uses CI?	Percentage	Number of Projects
Yes	40.27%	13,910
No	59.73%	20,634

select all the reasons that they agreed with. To populate the choices, we collected some common reasons for using or not using CI, as mentioned in websites [1, 7], blogs [3, 8, 19], and Stack Overflow [14]. Optionally, the survey participants could also write their own reason(s) that we did not already list. The third question asked if the participant plans on using CI for future projects.

To incentivize participation, we raffled off a 50 USD gift card among the survey respondents. 442 (9.8% response rate) participants responded to our survey. Of those responses, 407 (92.1%) indicated that they do use CI, and 35 (7.9%) indicated that they do not use CI.

2.5 Results

In this section, we present the results to our research questions. Section 2.5.1 presents the results about the usage of CI. Section 2.5.2 discusses the costs of CI. Finally Section 2.5.3 presents the benefits of CI. Rather than presenting implications after each research question, we draw from several research questions to triangulate implications that we present in Section 2.6.

2.5.1 Usage of CI

To determine the extent to which CI is used, we study what percentage of projects actively use CI, and we also ask developers if they plan to use CI in the future. Furthermore, we study whether the project popularity and programming language correlate with the usage of CI.

RQ: *What percentage of open-source projects use CI?*

We found that 40% of all the projects in our breadth corpus use CI. Table 2.1 shows the breakdown of the usage. Thus, CI is indeed used widely and warrants further investigation. Additionally, we know that our scripts do not find all CI usage (e.g., projects that run privately hosted CI systems, as discussed further in Section 2.7.2). We can reliably detect the use of (public) CI services only if their API makes it possible to query the CI service based on knowing the GitHub organization and project name. Therefore, the results we present are a lower bound on the total number of projects that use CI.

Table 2.2: CI usage by Service. The top row shows percent of all CI projects using that service, the second row shows the total number of projects for each service. Percents add up to more than 100 due to some projects using multiple CI services.

Usage by CI Service				
Travis	CircleCI	AppVeyor	CloudBees	Werker
90.1%	19.1%	3.5%	1.6%	0.4%
12528	2657	484	223	59

RQ: *What is the breakdown of usage of different CI services?*

Next we investigate which CI services are the most widely used in our breadth corpus. Table 2.2 shows that Travis CI is by far the most widely used CI service. Because of this result, we feel confident that our further analysis can focus on the projects that use Travis CI as a CI service, and that analyzing such projects gives representative results for usage of CI services in open-source projects.

We also found that some projects use more than one CI service. In our breadth corpus, of all the projects that use CI, 14% use more than one CI. We think this is an interesting result which deserves future attention.

RQ: *Do certain types of projects use CI more than others?*

To better understand which projects use CI, we look for characteristics of projects that are more likely to use CI.

CI usage by project popularity: We want to determine whether more popular projects are more likely to use CI. Our intuition is that if CI leads to better outcomes, then we would expect to see higher usage of CI among the most popular projects (or, alternatively, that projects using CI get better and thus are more popular). Figure 2.1 shows that the most popular projects (as measured by the number of stars) are also the most likely to use CI (Kendall's τ, $p < 0.00001$). We group the projects from our breadth

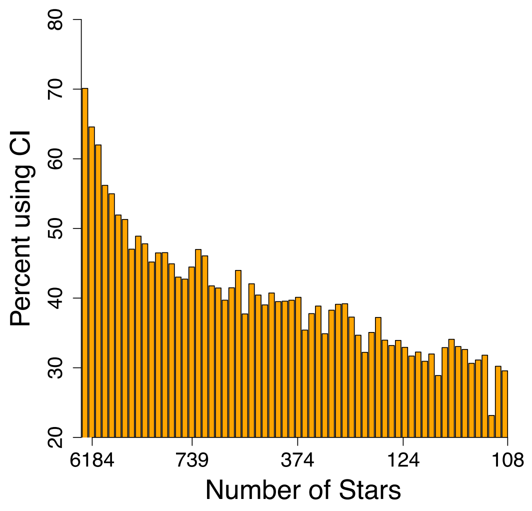


Figure 2.1: CI usage of projects in GitHub. Projects are sorted by popularity (number of stars).

corpus into 64 even groups, ordered by number of stars. We then calculate the percent of projects in each group that are using CI. Each group has around 540 projects. In the most popular (starred) group, 70% of projects use CI. As the projects become less popular, the percentage of projects using CI declines to 23%.

Observation

Popular projects are more likely to use CI.

CI usage by language: We now examine CI usage by programming language. Are there certain languages for which the projects written primarily in such languages use CI more than others? Table 2.3 shows projects sorted by the percentage of projects that use CI for each language, from our breadth corpus. The data shows that in fact there are certain languages that use CI more than others. Notice that the usage of CI does not perfectly correlate with the number of projects using that language (as measured by the number of projects using a language, with its rank by percentage, Kendall's τ, $p > 0.68$). In other words, some of the languages that use CI the most are both popular languages like Ruby and emerging languages like Scala. Similarly, among projects that use CI less, we notice both popular languages such as Objective-C and Java, as well as less popular

Table 2.3: CI usage by programing language. For each language, the columns tabulate: the number of projects from our corpus that predominantly use that language, how many of these projects use CI, the percentage of projects that use CI.

Language	Total Projects	# Using CI	Percent CI
Scala	329	221	67.17
Ruby	2721	1758	64.61
Go	1159	702	60.57
PHP	1806	982	54.37
CoffeeScript	343	176	51.31
Clojure	323	152	47.06
Python	3113	1438	46.19
Emacs Lisp	150	67	44.67
JavaScript	8495	3692	43.46
Other	1710	714	41.75
C++	1233	483	39.17
Swift	723	273	37.76
Java	3371	1188	35.24
C	1321	440	33.31
C#	652	188	28.83
Perl	140	38	27.14
Shell	709	185	26.09
HTML	948	241	25.42
CSS	937	194	20.70
Objective-C	2745	561	20.44
VimL	314	59	18.79

languages such as VimL.

However, we did observe that many of the languages that have the highest CI usage are also dynamically-typed languages (e.g., Ruby, PHP, CoffeeScript, Clojure, Python, and JavaScript). One possible explanation may be that in the absence of a static type system which can catch errors early on, these languages use CI to provide extra safety.

Observation

We observe a wide range of projects that use CI. The popularity of the language does not correlate with the probability that a project uses CI.

RQ: *When did open-source projects adopt CI?*

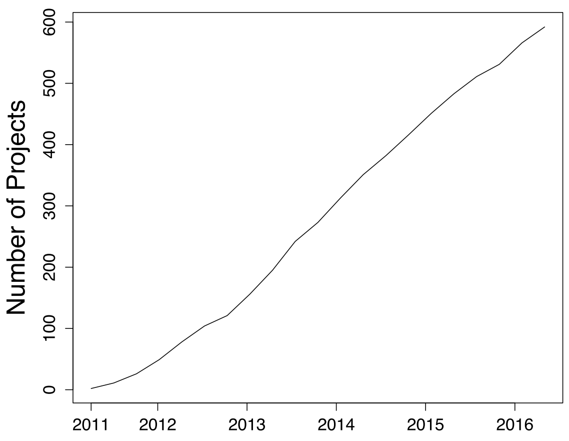


Figure 2.2: Number of projects using CI over time. Data is tabulated by quarter (3 months) per year.

We next study when projects began to adopt CI. Figure 2.2 shows the number of projects using CI over time. We answer this question with our depth corpus, because the breadth corpus does not have the date of the first build, which we use to determine when CI was introduced to the project. Notice that we are collecting data from Travis CI, which was founded in 2011 [10]. Figure 2.2 shows that CI has experienced a steady growth over the last 5 years.

We also analyze the age of each project when developers first introduced CI, and we found that the median time was around 1 year. Based on this data, we conjecture that while many developers introduce CI early in a project's development lifetime, it is not always seen as something that provides a large amount of value during the very initial development of a project.

Observation

The median time for CI adoption is one year.

RQ: *Do developers plan on continuing to use CI?*

Is CI a passing "fad" in which developers will lose interest, or will it be a lasting practice? While only time will tell what the true answer is, to get some sense of what the future could hold, we asked developers in our survey if they plan to use CI for their next project. We asked them how likely they were to use CI on their next project, using a 5-point Likert scale ranging from definitely will use to definitely will not use. Figure 2.3 shows that developers feel very strongly that they will be using CI for their next project. The top two options, 'Definitely' and 'Most Likely', account for 94% of all our survey respondents, and the average of all the answers was 4.54. While this seems like a pretty resounding endorsement for the continued use of CI, we decided to dig a little deeper. Even among respondents who are not currently using CI, 53% said that they would 'Definitely' or 'Most Likely' use CI for their next project.

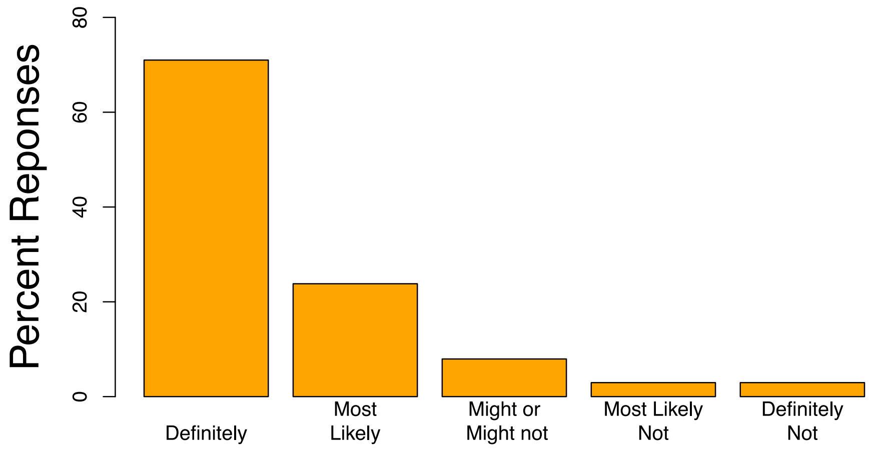


Figure 2.3: Answers to "Will you use CI for your next project?"

Observation

While CI is widely used in practice nowadays, we predict that in the future, CI adoption rates will increase even further.

2.5.2 Costs of CI

To better understand the costs of CI, we analyze both the survey (where we asked developers why they believe CI is too costly to be worth using) and the data from our depth corpus. We estimate the cost to developers for writing and maintaining the configuration for their CI service. Specifically, we measure how often the developers make changes to their configuration files and study why they make those changes to the configuration files. We also analyze the cost in terms of the time to run CI builds. Note that the time that the builds take to return a result could be unproductive time if the developers do not know how to proceed without knowing that result.

RQ: *Why do open-source projects choose* not *to use CI?*

Table 2.4: Reasons developers gave for not using CI

Reason	Percent
The developers on my project are not familiar enough with CI	47.00
Our project doesn't have automated tests	44.12
Our project doesn't commit often enough for CI to be worth it	35.29
Our project doesn't currently use CI, but we would like to in the future	26.47
CI systems have too high maintenance costs (e.g., time, effort, etc.)	20.59
CI takes too long to set up	17.65
CI doesn't bring value because our project already does enough testing	5.88

One way to evaluate the costs of CI is to ask developers why they do *not* use CI. In our survey, we asked respondents whether they chose to use or not use CI, and if they indicated that they did not, then we asked them to tell us *why* they do not use CI.

Table 2.4 shows the percentage of the respondents who selected particular reasons for not using CI. As mentioned before, we built the list of possible reasons by collecting information from various popular internet sources. Interestingly, the primary cost that respondents identified was not a technical cost; instead, the reason for not using CI was that *"The developers on my project are not familiar enough with CI."* We do not know if the developers are not familiar enough with the CI tools themselves (e.g., Travis CI), or if they are unfamiliar with all the work it will take to add CI to their project, including perhaps fully automating the build. To completely answer this question, more research is needed.

The second most selected reason was that the project does not have automated tests. This speaks to a real cost for CI, in that much of its value comes from automated tests, and some projects find that developing good automated test suites is a substantial cost. Even in the cases where developers had automated tests, some questioned the use of CI (in particular and regression testing in general); one respondent (P74) even said *"In 4 years our tests have yet to catch a single bug."*

Observation

The main reason why open-source projects choose to not use CI is that the developers are not familiar enough with CI.

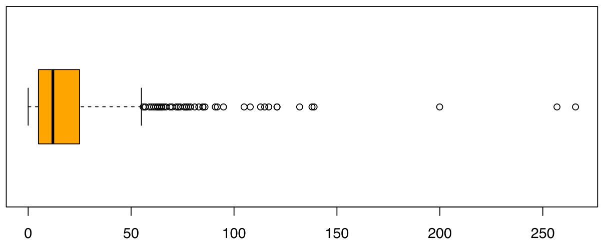


Figure 2.4: Number of changes to CI configs, median number of changes is 12

RQ: *How often do projects evolve their CI configuration?* We ask this question to identify how often developers evolve their CI configurations. Is it a "write-once-and-forget-it" situation, or is it something that evolves constantly? The Travis CI service is configured via a YAML [20] file, named .travis.yml, in the project's root directory. YAML is a human-friendly data serialization standard. To determine how often a project has changed its configuration, we analyzed the history of every .travis.yml file and counted how many times it has changed. We calculate the number of changes from the commits in our depth corpus. Figure 2.4 shows the number of changes/commits to the .travis.yml file over the life of the project. We observe that the median of number of changes to a project's CI configuration is 12 times, but one of the projects changed the CI configuration 266 times. This leads us to conclude that many projects setup CI once and then have minimal involvement (25% of projects have 5 or less changes to their CI configuration), but some projects do find themselves changing their CI setup quite often.

Observation

Some projects change their configurations relatively often, so it is worthwhile to study what these changes are.

RQ: *What are some common reasons projects evolve their CI configuration?*

To better understand the changes to the CI configuration files, we analyzed all the changes that were made to the .travis.yml files in our depth corpus. Because YAML is a structured language, we can parse the file and determine which part of the configuration was changed. Table 2.5 shows the distribution of all the changes. The most common changes were to the *build matrix*, which in Travis specifies a combination of *runtime*, *environment*, and *exclusions/inclusions*. For example, a build matrix for a project in Ruby could specify the runtimes *rvm 2.2*, *rvm 1.9*, and *jruby*, the build environment *rails2* and

Table 2.5: Reasons for CI config changes

Config Area	Total Edits	Percentage
Build Matrix	9718	14.70
Before Install	8549	12.93
Build Script	8328	12.59
Build Language Config	7222	10.92
Build Env	6900	10.43
Before Build Script	6387	9.66
Install	4357	6.59
Whitespace	3226	4.88
Build platform Config	3058	4.62
Notifications	2069	3.13
Comments	2004	3.03
Git Configuration	1275	1.93
Deploy Targets	1079	1.63
After Build Success	1025	1.55
After Build Script	602	0.91
Before Deploy	133	0.20
After Deploy	79	0.12
Custom Scripting	40	0.06
After Build Failure	39	0.06
After Install	14	0.02
Before Install	10	0.02
Mysql	5	0.01
After Build Success	3	0.00
Allow Failures	2	0.00

rails3, and the exclusions/inclusions, e.g., *exclude: jruby with rails2*. All combinations will be built except those excluded, so in this example there would be 5 different builds. Other common changes included the dependent libraries to install before building the project (what .travis.yml calls *before install*) and changes to the build script themselves. Also, many other changes were due to the version changes of dependencies.

Observation

Many CI configuration changes are driven by dependency changes and could be potentially automated.

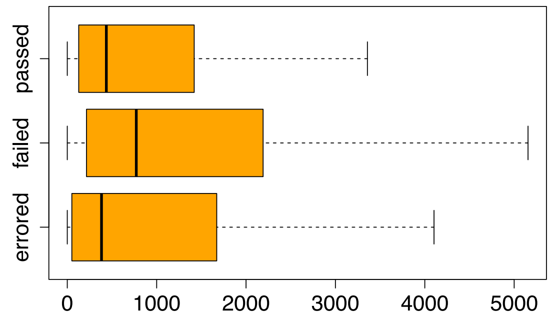


Figure 2.5: Build time distribution by result, in seconds

RQ: *How long do CI builds take on average?*

Another cost of using CI is the time to build the application and run all the tests. This cost represents both a cost of energy[2] for the computing power to run these builds, but also developers may have to wait to see if their build passes before they merge in the changes, so having longer build times means more wasted developer time.

The average build time is just under 500 seconds. To compute the average build times, we first remove all the canceled (incomplete, manually stopped) build results, and only consider the time for errored, failed, and passed (completed builds). Errored builds are those that occur before the build begins (e.g., when a dependency cannot be downloaded), and failed builds are those that the build is not completed succesfully (e.g., a unit test fail). To further understand the data, we look at each outcome independently. Interestingly, we find that passing builds run faster than either errored or failed builds. The difference between errored and failed is significant (Wilcoxon, $p < 0.0001$), as is the difference between passed and errored (Wilcoxon, $p < 0.0001$) and the difference between passed and failed (Wilcoxon, $p < 0.0001$).

We find this result surprising as our intuition is that passing builds should take longer, because if an error state is encountered early on, the process can abort and return earlier. Perhaps it is the case that many of the faster running pass builds are not generating a

[2]This cost should not be underestimated; our personal correspondence with a Google manager in charge of their CI system TAP reveals that TAP costs millions of dollars just for the computation (not counting the cost of developers who maintain or use TAP).

meaningful result, and should not have been run. However, more investigation is needed to determine what the exact reasons for this are.

2.5.3 Benefits of CI

We first summarize the most commonly touted benefits of CI, as reported by the survey participants. We then analyze empirically whether these benefits are quantifiable in our depth corpus. Thus, we confirm or refute previously held beliefs about the benefits of CI.
RQ: *Why do open-source projects choose to use CI?*

Table 2.6: Reasons for using CI, as reported by survey participants

Reason	Percent
CI makes us less worried about breaking our builds	87.71
CI helps us catch bugs earlier	79.61
CI allows running our tests in the cloud, freeing up our personal machines	54.55
CI helps us deploy more often	53.32
CI makes integration easier	53.07
CI runs our tests in a real-world staging environment	46.00
CI lets us spend less time debugging	33.66

Having found that CI is widely used in open-source projects (RQ1), and that CI is most widely used among the most popular projects on GitHub (RQ3), we want to understand why developers choose to use CI. However, *why* a project uses CI cannot be determined from a code repository. Thus, we answer this question using our survey data.

Table 2.6 shows the percentage of the respondents who selected particular reasons for using CI. As mentioned before, we build this list of reasons by collecting information from various popular internet sources. The two most popular reasons were "CI makes us less worried about breaking our builds" and "CI helps us catch bugs earlier". One respondent (P371) added: *"Acts like a watchdog. You may not run tests, or be careful with merges, but the CI will. :)"*

Martin Fowler [7] is quoted as saying "Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove." However, in our survey, very few projects felt that CI actually helped them during the debugging process.

Observation

Projects use CI because it helps them catch bugs early and makes them less worried about breaking the build. However, CI is not widely perceived as helpful with debugging.

RQ: *Do projects with CI release more often?*
One of the more common claims about CI is that it helps projects release more often, e.g., CloudBees motto is "Deliver Software Faster" [6]. Over 50% of the respondents from our survey claimed it was a reason why they use CI. We analyze our data to see if we can indeed find evidence that would support this claim.

Table 2.7: Release rate of projects

Uses Travis	Versions Released per Month
Yes	.54
No	.24

We found that projects that use CI do indeed release more often than either (1) the same projects before they used CI or (2) the projects that do not use CI. In order to compare across projects and periods, we calculated the release rate as the number of releases per month. Projects that use CI average .54 releases per month, while projects that do not use CI average .24 releases per month. That is more than double the release rate, and the difference is statistically significant (Wilcoxon, $p < 0.00001$). To identify the effect of CI, we also compared, for projects that use CI, the release rate both before and after the first CI build. We found that projects that eventually added CI used to release at a rate of .34 releases per month, well below the .54 rate at which they release now with CI. This difference is statistically significant (Wilcoxon, $p < 0.00001$).

Observation

Projects that use CI release more than twice as often as those that do not use CI.

RQ: *Do projects which use CI accept more pull requests?*
For a project that uses a CI service such as Travis CI, when the CI server builds a pull request, it annotates the pull request on GitHub with a visual cue such as a green check mark or a red 'X' that shows whether the pull request was able to build successfully

Table 2.8: Comparison of pull requests merged for pull requests that had or did not have CI information

CI Usage	% Pull Requests Merged
Using CI	23
Not Using CI	28

on the CI server. Our intuition is that this extra information can help developers better decide whether or not to merge a pull request into their code. To determine if this extra information indeed makes a difference, we compared the pull request acceptance rates between pull requests that have this CI information and pull requests that do not have it, from the depth corpus. Note that projects can exclude some branches from their repository to not run on the CI server, so just because a project uses CI on some branch, there is no guarantee that every pull request contains the CI build status information.

Table 2.8 shows the results for this question. We found that pull requests without CI information were 5pp more likely to be merged than pull requests with CI information. Our intuition of this result is that those 5pp of pull requests have problems which are identified by the CI. By not merging these pull requests, developers can avoid breaking the build. This difference is statistically significant (Fisher's Exact Test: $p < 0.00001$). This also fits with our survey result that developers say that using CI makes them less worried about breaking the build. One respondent (P219) added that CI *"Prevents contributors from releasing breaking builds"*. By not merging in potential problem pull requests, developers can avoid breaking their builds.

Observation

CI build status can help developers avoid breaking the build by not merging problematic pull requests into their projects.

RQ: *Do pull requests with CI builds get accepted faster (in terms of calendar time)?*
Once a pull request is submitted, the code is not merged until the pull request is accepted. The sooner a pull request is accepted, the sooner the code is merged into the project. In the previous question, we saw that projects using CI accept fewer (i.e., reject or ignore more) pull requests than projects not using CI. In this question, we consider only accepted pull requests, and ask whether there is a difference in the time it takes for projects to accept pull requests with and without CI. One reason developers gave for using CI is

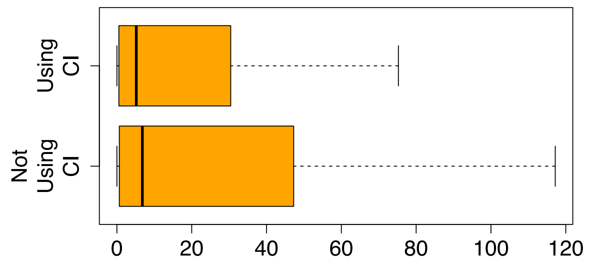


Figure 2.6: Distribution of time to accept pull requests, in hours

that it makes integration easier. One respondent (P183) added *"To be more confident when merging PRs"*. If integration is easier, does it then translate into pull requests being integrated faster?

Figure 2.6 shows the distributions of the time to accept pull requests, with and without CI. To compute these results, we select, from our depth corpus, all the pull requests that were accepted, both with and without build information from the CI server. The mean time with CI is 81 hours, but the median is only 5.2 hours. Similarly, the mean time without CI is 140 hours, but the median is 6.8 hours. Comparing the median time to accept the pull requests, we find that the median pull request is merged 1.6 hours faster than pull requests without CI information. This difference is statistically significant (Wilcoxon, $p < .0000001$).

Observation

CI build status can make integrating pull requests faster. When using CI, the median pull request is accepted 1.6 hours sooner.

Table 2.9: Percentage of builds that succeed by pull request target

Pull Request Target	Percent Passed Builds
Master	72.03
Other	65.36

RQ: *Do CI builds fail less on master than on other non-master branches?* The most popular reason that participants gave for using CI was that it helps avoid breaking the build. Thus, we analyze this claim in the depth corpus. Does the data show a difference in the way developers use CI with the master branch vs. with the other branches? Is there any difference between how many builds fail on master vs. on the other branches? Perhaps developers take more care when writing a pull request for master than for another branch.

Table 2.9 shows the percentage of builds that pass in pull requests to the master branch, compared to all other branches. We found that pull requests are indeed more likely to pass when they are on master.

Observation

CI builds on the master branch pass more often than on the other branches.

2.6 Implications

We offer practical implications of our findings for researchers, developers, and tool builders.

Researchers

RQ1, RQ3, RQ4, RQ5: CI is not a "fad" but is here to stay. Because CI is widely used and more projects are adopting it, and has not yet received much attention from the research community, it is time for researchers to study its use and improve it, e.g., automate more tasks (such as setting up CI). We believe that researchers can contribute many improvements to the CI process once they understand the current state-of-the-practice in CI.

RQ2: Similarly with how GitHub has become the main gateway for researchers who study software, we believe Travis CI can become the main gateway for researchers who study CI. Travis offers a wealth of CI data, accessible via public API. Therefore, researchers can maximize their impact by studying a single system.

RQ7, RQ8: We found evidence of frequent evolution of CI configuration files (similar evolution was found for Makefiles [21]), so researchers can focus on providing support for safe automation of changes in configuration files, e.g., via safe refactoring tools.

RQ8: We confirmed that continuously running CI takes a non-trivial amount of time (and resources), so the testing research community should investigate methods for faster build and test, similar to the ongoing efforts on TAP at Google [4, 45, 47, 108, 109], Tools for Software Engineers (TSE) at Microsoft [16], or regression testing [55, 119].

RQ6, Table 2.4 : The most common reason why developers do not use CI is unfamiliarity with CI, so there is tremendous opportunity for providing educational resources. We call upon university educators to enrich their software engineering curriculum to cover the basic concepts and tooling for CI.

Developers

RQ3, Table 2.3: The data shows that CI is more widely embraced by the projects that use dynamically typed languages (e.g., 64% of 2721 Ruby projects use CI, compared with only 20% of 2745 Objective-C projects that use CI). To mitigate the lack of a static type system, developers that use dynamically typed languages should use CI to run tests and help catch errors early on.

RQ13: Our analysis of the depth corpus shows that the presence of CI makes it easier to accept contributions in open-source projects, and this was also indicated by several survey respondents, e.g., *"CI gives external contributors confidence that they are not breaking the project" (P310).* Considering other research [71] that reports a lack of diversity in open-source projects, attracting new contributors is desirable. Thus, projects that aim to diversify their pool of contributors should consider using CI.

RQ7, RQ9: Because the average times for a single CI build is fairly short, and CI configurations are maintainable, it appears that the benefits of CI outweigh the costs. Thus, developers should use CI for their projects.

RQ3, RQ11, RQ12, RQ14: The use of CI correlates with positive outcomes, and CI has been adopted by the most successful projects on GitHub, so developers should consider CI as a best practice and should use it as widely as possible.

Tool Builders

RQ6: CI helps catching bugs, but not locating them. The CI build logs often bury an important error message among hundreds of lines of raw output. Thus, tool builders that want to improve CI can focus on new ways to integrate fault-localization techniques into CI.

RQ1, RQ7, RQ8: Despite wide adoption, there are many projects that have yet to use CI. Tool builders could parse build files [122], and then generate configuration files necessary for CI. By automating this process, tool builders can lower the entry barrier for developers who are unfamiliar with CI.

2.7 Threats to Validity

2.7.1 Construct

Are we asking the right questions? We are interested in assessing the usage of CI in open-source projects. To do this we have focused on *what*, *how*, and *why* questions. We think that these questions have high potential to provide unique insight and value for different stakeholders: developers, tool builders, and researchers.

2.7.2 Internal

Is there something inherent to how we collect and analyze CI usage data that could skew the accuracy of our results?

Once a CI server is configured, it will continue to run until it is turned off. This could result in projects configuring a CI server, and then not taking into account the results as they continue to do development. However, we think this is unlikely because Travis CI and GitHub have such close integration. It would be difficult to ignore the presence of CI when there are visual cues all throughout GitHub when a project is using CI.

Some CI services are run in a way such that they cannot be detected from the information that is publicly available in the GitHub repository. This means that we could have missed some projects. However, this would mean that we are underestimating the extent to which CI has been adopted.

Despite a 9.8% response rate to our survey, still over 90% of our targeted population did not respond. We had no control over who responded to our survey, so it may suffer from self-selection bias. We think this is likely because 92% of our survey participants reported using CI, much higher than the percentage of projects we observed using CI in the data. In order to mitigate this, we made the survey short and provided a raffle as incentive to participate, to get the most responses as possible.

2.7.3 External

Are our results generalizable for general CI usage? While we analyzed a large number of open-source repositories, we cannot guarantee that these results will be the same for proprietary (closed-source) software. In fact, we consider it very likely that closed-source projects would be unwilling to send their source over the internet to a CI service, so our intuition is that they would be much more likely to use a local CI solution. Further work should be done to investigate the usage of CI in closed-source projects.

Because we focused on Travis CI, it could be that other CI services are used differently. As we showed in RQ2, Travis CI was the overwhelming favorite CI service to use, so by focusing on that we think our results are representative.

Additionally, we only selected projects from GitHub. Perhaps open-source projects that have custom hosting also would be more likely to have custom CI solutions. More work is needed to determine if these results generalize.

2.8 Related Work

We group our related work into three different areas: (i) *CI usage*, (ii) *CI technology*, and (iii) *related technology*.

CI Usage The closest work to ours is by Vasilescu et al. [112] who present two main findings. They find that projects that use CI are more effective at merging requests from core members, and the projects that use CI find significantly more bugs. However, the paper explicitly states that it is a preliminary study on only 246 GitHub projects, and treats CI usage as simply a boolean value. In contrast, this paper examines 34,544 projects, 1,529,291 builds, and 442 survey responses to provide detailed answers to 14 research questions about CI usage, costs, and benefits.

A tech report from Beller et al. [28] performs an analysis of CI builds on GitHub, specifically focusing on Java and Ruby languages. They answer several research questions about tests, including "How many tests are executed per build?", "How often do tests fail?", and "Does integration in different environments lead to different test results?". These questions however, do not serve to comprehensively support or refute the productivity claims of CI.

Two other papers [74, 80] have analyzed a couple of case studies of CI usage. These are just two case studies total, unlike this paper that analyzes a broad and diverse corpus.

Leppänen et al. [76] interviewed developers from 15 software companies about what they perceived as the benefits of CI. They found one of the perceived benefits to be more frequent releases. One of their participants said CI reduced release time from six months to two weeks. Our results confirm that projects that use CI release twice as fast as projects that do not use CI.

Beller et al. [27] find that developers report testing three times more often than they actually do test. This over-reporting shows that CI is needed to ensure tests are actually run. This confirms what one of our respondents (P287) said: *"It forces contributors to run the tests (which they might not otherwise do)"*. Kochhar et al. [68] found that larger Java open-source projects had lower test coverage rates, also suggesting that CI can be beneficial.

CI technology Some researchers have proposed approaches to improve CI servers by having servers communicate dependency information [43], generating tests during CI [39], or selecting tests based on code churn [67]. Also researchers [32] have found that integrating build information from various sources can help developers. In our survey, we found that developers do not think that CI helps them locate bugs; this problem has been also pointed out by others [33].

One of the features of CI systems is that they report the build status so that it is clear to everyone. Downs et al. [44] developed a hardware based system with devices shaped like rabbits which light up with different colors depending on the build status. These devices keep developers informed about the status of the build.

Related Technology A foundational technology for CI is build systems. Some ways researchers have tried to improve their performance has been by incremental building [48] as well as optimizing dependency retrieval [37].

Performing actions continuously can also bring extra value, so researchers have proposed several activities such as continuous test generation [118], continuous testing (continuously running regression tests in the background) [94], continuous compliance [50], and continuous data testing [82].

2.9 Conclusions

CI has been rising as a big success story in automated software engineering. In this paper we study the usage, the growth, and the future prospects of CI using data from three complementary sources: (i) 34,544 open-source projects from GitHub, (ii) 1,529,291 builds from the most commonly used CI system, and (iii) 442 survey respondents. Using this rich data, we investigated 14 research questions.

Our results show there are good reasons for the rise of CI. Compared to projects that do not use CI, projects that use CI: (i) release twice as often, (ii) accept pull requests faster, and (iii) have developers who are less worried about breaking the build. Therefore, it should come as no surprise that 70% of the most popular projects on GitHub heavily use CI.

The trends that we discover point to an expected growth of CI. In the future, CI will have an even greater influence than it has today. We hope that this paper provides a call to action for the research community to engage with this important field of automated software engineering.

Chapter 3: Trade-offs in Continuous Integration (CI): Assurance, Security, and Flexibility

Trade-offs in Continuous Integration (CI): Assurance, Security, and Flexibility

Michael Hilton,Nicholas Nelson,Timothy Tunnell,Darko Marinov,Danny Dig

Under Review at: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESE/FSE 17)

3.1 Abstract

Continuous Integration (CI) systems automate the compilation, building, and testing of software. Despite CI being one of the most widely used processes in software engineering, we do not know what motivates developers to use CI, and what barriers and unmet needs they face. Without such knowledge developers make easily avoidable errors, tool builders invest in the wrong direction, and researchers miss many opportunities for improving the practice of software engineering.

We present a qualitative study of the barriers and needs developers face when using CI. In this paper, we conduct 16 semi-structured interviews with developers from different industries and development scales. We triangulate our findings by running two surveys. The *Focused Survey* samples 51 developers at a single company. The *Broad Survey* samples a population of 523 developers from all over the world. We identify trade-offs developers face when using and implementing CI. Developers face trade-offs between speed and certainty (*Assurance*), between better access and information security (*Security*), and between more configuration options and better ease of use (*Flexibility*). We present implications of these trade-offs for developers, tool builders, and researchers.

3.2 Introduction

Continuous integration (CI) systems automate the compilation, building, and testing of software. For example, the “State of DevOps” report [92], a survey of over 4,600 technical professionals from around the world, finds CI to be an indicator of “high performing IT organizations”.

CI usage is widespread throughout the software development industry. The “State of Agile” industry survey [88], with 3,880 participants, found half of the respondents use CI. Researchers also reported [60] that 40% of the 34,000 most popular open-source projects on GitHub use CI, and the most popular projects are more likely to use CI (70% of the top 500 projects).

Despite the widespread adoption of CI, there are still many unanswered questions about CI. In one study, Vasilescu et al. [113] show that CI correlates with positive quality outcomes. Hilton et al. [60] examine the usage of CI among open-source projects on GitHub, and show that projects that use CI release more frequently than projects that do not. However, these studies do not present what barriers and needs developers face when using CI, or what trade-offs developers must make to solve these problems.

To fill in the gaps in knowledge about developers and their use of CI, we ask the following questions: What needs do developers have that are unmet by their current CI system(s)? What problems have developers experienced when configuring and using CI systems(s)? How do developers feel about using CI? Without answers to these questions, *developers* can potentially find CI more obstructive than helpful, *tool builders* will implement unneeded features, and *researchers* will not be aware of areas of CI usage that require further examination and solutions that can further enable practitioners.

To answer these questions, we employ complementary established research methodologies. Our primary methodology is interviews with 16 software developers from 14 different companies of all sizes. To triangulate our findings, we deployed two surveys. The *Focused Survey* samples 51 developers at Pivotal[1]. The *Broad Survey* samples 523 participants, of which 95% are from industry, and 70% have 7 or more years of software development experience. The interviews provide the content for the surveys, and the *Focused Survey* provides depth while the *Broad Survey* provides breadth. Analyzing all this data, we answer four research questions:

[1]https://pivotal.io/

RQ1: *What barriers do developers face when using CI* (see §3.5.1)
RQ2: *What unmet needs do developers have with CI tools* (see §3.5.2)
RQ3: *Why do developers use CI* (see §3.5.3)
RQ4: *What benefits do developers experience when using CI* (see §3.5.4)

Based on our findings, we identify three trade-offs developers face when using CI. Other researchers [90, 93, 117] have identified similar trade-offs in different domains. We name these trade-offs *Assurance*, *Security*, and *Flexibility*.

Assurance describes the trade-off between increasing the added value that extra testing provides, and the extra cost of performing that testing. Rothermel et al. [93] identify this trade-off as a motivation for test prioritization.

Security describes the trade-off between increased security measures, and the ability to access and modify the CI system as needed. Post and Kagan [90] found a third of knowledge workers report security restrictions hinder their ability to perform their jobs. We observe this issue also applies to CI users.

Flexibility describes the trade-off that occurs when developers want systems that are both powerful and highly configurable, yet at the same time, they want those systems to be simple and easy to use. Xu et al. [117] identify the costs of over-configurable systems and found that these systems severely hinder usability. We also observe the tension from this trade-off among developers using CI.

In the context of these three trade-offs, we present implications for three audiences: *Developers*, *Tool Builders*, and *Researchers*. For example, *Developers* face difficult choices about how much testing is enough, and how to choose the right tests to run. *Tool Builders* should create UIs for CI users to configure their CI systems, but these UIs should serialize configurations out to text files so that they can be kept in version control. *Researchers* have much to bring to the CI community, such as helping with fault localization and test parallelization when using CI, and examining the security challenges developers face when using CI.

This paper makes the following contributions:

1. We conduct exploratory semi-structured interviews with 16 developers, then triangulate these findings with a *Focused Survey* of 51 developers at Pivotal and a *Broad Survey* of 523 developers from all over the world.

2. We provide an empirically justified set of developers' motivations for using CI.

3. We expose gaps between developers' needs and existing tooling for CI.
4. We present actionable implications that developers, tool builders, and researchers can build on.

The interview script, code set, survey questions, and responses can be found on *this study's website*[2].

3.3 Background

The core premise of Continuous Integration (CI) is that the more often a project integrates, the better off it is. CI systems are responsible for retrieving code, collecting all dependencies, compiling the code, and running automated tests. The system should output "pass" or "fail" to indicate whether the CI process was successful.

We asked our interview participants to describe their CI pipeline. While not all pipelines are the same, they generally share some common elements.

Changesets are a group of changes that a developer makes to the code. They may be a single commit, or a group of commits, but they should be a complete change, so that after the changeset is applied, it should not break the program.

When a CI system or service observes a change made by developers, this *triggers* a CI event. How and when the CI is triggered is based on how the CI is configured. One common way to trigger CI is when a commit is pushed to a repository.

For the CI to test the code without concern for previous data or external systems, it is important for this to happen in a *cleanroom* environment. The automated build script should be able to start with a clean environment, and build the product from scratch before executing tests. Many developers use containers (e.g., Docker[3]) to implement cleanroom builds.

An important step in the CI pipeline is verifying that the changeset was integrated correctly into the application. One common method is a regression test suite, including unit tests and integration tests. The CI system can also perform other analysis, such as linting, or evaluating test coverage.

[2]Link omitted for anonymous submission.
[3]docker.io

The last step is to deploy the built and verified artifact. We found some developers consider deploying to be part of CI, and others consider *Continuous Deployment* (CD) to be a separate process.

3.4 Methodology

Inspired by established guidelines [73, 84, 89, 100, 107], the primary methodologies we employ in this work are interviews with software developers and surveys of developers to triangulate our findings.

Interviews are a qualitative method and are effective at discovering the knowledge and experiences of the participants. However, they often have a limited sample size [100]. Surveys are a quantitative technique that summarizes information over a larger sample size and thus provides broader results. Together, they provide a much clearer picture than either can provide alone.

We first use interviews to elicit developers experiences and expectations when working with CI, and we build a taxonomy of barriers, unmet needs, motivations and experiences. We build a survey populating the answers to each question with the results of the interviews. We deploy this survey at Pivotal, a software and services company, who also develops a continuous integration solution, Concourse[4]. To gain an even broader understanding, we also deploy the survey via social media to reach as many developers as possible.

The interview script, code set, questionnaires, and the survey responses can be found on this study's website.

3.4.1 Interviews

We used semi-structured interviews "which include a mixture of open-ended and specific questions, designed to elicit not only the information foreseen, but also unexpected types of information" [100]. We developed our interview script by performing iterative pilots.

We initially recruited participants from previous research, and then used snowball sampling to reach more developers. We interviewed 16 developers from 14 different companies, including large software companies, CI service companies, small development companies, a telecommunications company, and software consultants. Our participants had over seven years of development experience on average. We assigned each participant

[4]concourse.ci

Table 3.1: Interview Participants.

Subject	Exp.	Domain	Org. Size
S1	8 yrs.	Content Platform Provider	Small
S2	20 yrs.	Content Platform Provider	Small
S3	4 yrs.	Developer Tools	Large
S4	10 yrs.	Framework Development	Large
S5	10 yrs.	Content Management	Large
S6	10 yrs.	Computer Security Startup	Small
S7	5 yrs.	Framework Development	Small
S8	5 yrs.	Media Platform	Medium
S9	6 yrs.	Language Development	Medium
S10	9 yrs.	CI Platform Development	Medium
S11	6 yrs.	Software Development Consulting	Medium
S12	10 yrs.	CI Platform Development	Small
S13	12 yrs.	Telecommunications	Large
S14	5 yrs.	Software Development Consulting	Medium
S15	2 yrs.	Infrastructure Management	Medium
S16	8 yrs.	Cloud Software Development	Medium

a subject number (Table 3.1). They all used CI, and a variety of CI tools, including Jenkins[5], TravisCI[6], Concourse[7], Wercker[8], CruiseControl.NET[9], CircleCI[10], TeamCity[11], XCode Bots[12], Buildbot[13], appVeyor[14], as well as some proprietary CI solutions. Each interview lasted between 30 and 60 minutes, and the participants were offered a US$50 Amazon gift card.

The interviews were based on the research questions presented in Section 3.2. The following is an example of some of the questions we asked in the interview:

- How would you define Continuous Integration (CI)?

[5]jenkins.io
[6]travis-ci.org
[7]concourse.ci
[8]wercker.com
[9]cruisecontrolnet.org
[10]circleci.com
[11]www.jetbrains.com/teamcity
[12]developer.apple.com/xcode
[13]buildbot.net
[14]appveyor.com

- Tell me about the last time you used CI.
- What tasks prompt you to interact with your current CI tools?
- Comparing projects that do use CI with those that don't, what differences have you observed?
- What, if anything, would you like to change about your current CI solution?

We coded the interviews using established guidelines from the literature [95] and followed the guidance from Campbell et al. [34] on specific issues related to coding semi-structured interview data, such as segmentation, codebook evolution, and coder agreement.

The first author segmented the transcript from each interview by units of meaning [34]. The first two authors then collaborated on coding the segmented interviews, using the negotiated agreement technique to achieve agreement [34]. Negotiated agreement is a technique where both researchers code a single transcript and discuss their disagreements in an effort to reconcile them before continuing on. We coded the first eight interviews together using this negotiated agreement technique. Because agreement is negotiated along the way, there is no inter-rater agreement number. After the eighth interview, the first and second author independently coded the remaining interviews. Our final codebook contained 25 codes divided into 4 groups: demographics, systems/tools, process, and human CI interaction. The full codeset is available on our companion site.

3.4.2 Survey

We created a survey with 21 questions to quantify the findings from our semi-structured interviews. The questions for the survey were created to answer our research questions, focusing on what *benefits*, *barriers* and *unmet needs* developers have when using CI.

The survey consisted of multiple choice questions, with a final open-ended text field to allow participants to share any additional information about CI. The answers for these multiple choice questions were populated from the answers given by interview participants. We ensured completeness by including an "other" field where appropriate. We prevented the order of answers from biasing our participants by using a survey tool which randomized the order of answers for the survey participants.

Focused Population We deployed our survey to a focused population of developers at Pivotal. Pivotal embraces agile development, and also sponsors the development of Concourse CI. We sent our survey via email to 294 developers at Pivotal, and we collected 51 responses for a response rate of 17.3%. All of the Pivotal developers reported using CI.
Broad Population We believe there are many voices among software developers, and we wanted to hear from as many of them as possible. We chose our sampling method for the *Broad Survey* to reach as many developers as possible. We recruited participants by advertising our survey on social media[15]. As with all survey approaches, we were forced to accept some trade-offs [30]. When recruiting participants online, we can reach larger numbers of respondents, but in doing so, results suffer self-selection bias. To maximize participation, we followed guidelines from the literature [101], including keeping the survey as short as possible, and raffling a gift certificate (specifically, a US$50 Amazon gift card) to survey participants.

We collected 523 complete responses, and a total of 691 survey responses, from over 30 countries. Over 50% of our participants had over 10 years of software development experience, and over 80% had 4 or more years experience.

3.5 Analysis of results

3.5.1 Barriers

In this section, we answer *What barriers do developers face when using CI* (**RQ1**).

We collected a list of barriers that our interview participants reported experiencing when using CI. We asked our survey participants to select up to three problems that they had experienced. If they had experienced more than three, we asked them to choose the three most common.
B1 Troubleshooting a CI build failure. When a CI build fails, some participants begin the process of identifying why the build failed. Sometimes, this can be fairly straightforward. However, for some build fails on the CI server, where the developer does not have the same access as they have when debugging locally, troubleshooting the failure can be quite challenging. S4 described one such situation:

If I get lucky, I can spot the cause of the problem right from the results from

[15]Facebook, Twitter, and reddit/r/SampleSize

Table 3.2: Barriers developers encounter when using CI.

Barriers	Broad	Focused
B1 Troubleshooting a CI build failure	266 (50%)	64%
B2 Overly long build times	203 (38%)	50%
B3 Automating the build process	176 (34%)	26%
B4 Lack of support for the desired workflow	163 (31%)	42%
B5 Setting up a CI server or service	145 (27%)	29%
B6 Maintaining a CI server or service	145 (27%)	40%
B7 Lack of tool integration	136 (26%)	12%
B8 Security and access controls	110 (21%)	14%

the Jenkins reports, and if not, then it becomes more complicated.

One way tool makers have tried to help developers is via better logging and storing test artifacts to make it easier to examine failures. One participant described how they use Sauce Labs[16], a service for automated testing of web pages, in conjunction with their CI. When a test fails on Sauce Labs, there is a recording of it that the developers can then watch to determine exactly how their test failed. Another participant described how Wercker[17] saves a container from each CI run, so one can download the container and run the code in the container to debug a failed test.

B2 Overly long build times. Because CI must verify that the current changeset is integrated correctly, it must build the code and run automated testing. This is a blocking step for developers, because they do not want to accept the changeset until they can be certain that it will not break the build. If this blocking step becomes too long, it reduces developers' productivity. Many interview participants reported that their build times slowly grow over time. S10 said:

> *Absolutely [our build times grow over time]. Worst case scenario it creeps with added dependencies, and added sloppy tests, and too much I/O. That's the worst case scenario for me, when it is a slow creep.*

Other participants told us they had seen build times increase because of bugs in

[16]saucelabs.com
[17]wercker.com

their build tools, problems with caching, dependency issues during the build process, and adding different styles of tests (e.g., acceptance tests) to the CI builds.

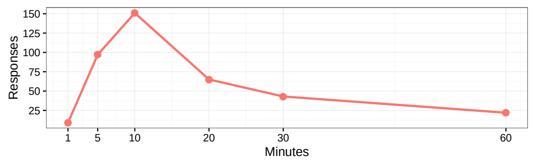


Figure 3.1: Maximum acceptable build time (minutes)

To dig a little deeper, we examined in-depth what developers meant by overly long build times. S9 said:

> *My favorite way of thinking about build time is basically, you have tea time, lunch time, or bedtime. Your builds should run in like, 5ish minutes, however long it takes to go get a cup of coffee, or in 40 minutes to 1.5 hours, however long it takes to go get lunch, or in 8ish hours, however long it takes to go and come back the next day.*

Martin Fowler [52] suggests most projects should try to follow the XP guideline of a 10-minute build. When we asked our Broad Survey participants what the maximum acceptable time for a CI build to take, the answer was also 10 minutes, as shown in Figure 3.1.

Many of our interview participants reported having spent time and effort reducing the build time for their CI process. S15 said:

> *[When the build takes too long to run], we start to evaluate the tests, and what do we need to do to speed up the environment to run through more tests in the given amount of time. ... Mostly I feel that CI isn't very useful if it takes too long to get the feedback.*

When we asked our survey participants, 96% of Focused Survey participants and 78% of Broad Survey participants said they had actively worked to reduce their build times. This shows long build times are a common barrier faced by developers using CI.

B3 Automating the build process. CI systems automate the manual process that developers previously followed when building and testing their code. The migration of these manual processes to automated builds requires that developers commit time and resources before the benefits of CI can be realized.

B4 Lack of support for the desired workflow. Interview participants told us that CI tools are often designed with a specific workflow in mind. When using a tool to implement a CI process, it can be difficult to use if one is trying to use a different workflow than the one for which the tool was designed. For example, when asked how easy it is to use CI tools, S2 said:

> *Umm, I guess it really depends on how well you adopt their workflow. For me that's been the most obvious thing. ... As soon as you want to adopt a slightly different branching strategy or whatever else, it's a complete nightmare.*

B5 Maintaining a CI server or service. This barrier is similar to *N1 Easier configuration of CI servers or services*, see discussion in section 3.5.2.

B6 Setting up a CI server or service. For our interview participants, setting up a CI server was not a concern when writing open-source code, as they can easily use one of several CI services available for free to open-source projects. We found that large commercial projects, while very complex, often have the resources to hire dedicated personnel to manage their CI pipeline. However, developers on small proprietary projects do not have the resources to afford CI as a service, nor do they have the hardware and expertise needed to setup CI locally. S9, who develops an app available on the Apple App Store, said:

> *[Setup] took too much time. All these tools are oriented to server setups, so I think it's very natural if you are running them on a server, but it's not so natural if you are running them on your personal computer. ...this makes a lot of friction if you want to set [CI] up on your laptop.*

Additionally, in the comments section of our survey, we received several comments on this issue, for example:

> *[We need] CI for small scale individual developers! We need better options IMO.*

While some of these concerns can be addressed by tool builders creating tools targeted for smaller scale developers, more research is needed to determine how project size impacts the usage of CI.

B7 Lack of tool integration. This barrier is similar to *N2 Better tool integration*, see discussion in section 3.5.2.

B8 Security and access controls. Because CI pipelines have access to the entire source code of a given project, security and access controls are vitally important. For CI pipelines that exist entirely inside of a company firewall, this may not be as much of a concern, but for projects using CI as a service, this can be a major issue. For developers working on company driven open-source projects, this can also be a concern. S9 said:

> *depending on your project, you may have an open-source project, but secrets living on or near your CI system.*

Configuring the security and access controls is vital to protecting those secrets. S16, who uses CI as a service, described how their project uses a secure environment variable (SEV) to authenticate a browser-based testing service with their CI. Maintaining the security of SEVs is a significant concern to their project.

Observation

Developers encounter increased complexity, increased time costs, and new security concerns when working with CI. Many of these issues are side-effects of implementing new CI features such as more configurability, more rigorous testing, and greater access to the development pipeline.

3.5.2 Needs

In this section, we answer *What unmet needs do developers have with CI tools* (**RQ2**)

In addition to describing problems they encounter when using CI, our interview participants also described gaps where CI was not meeting their needs.

N1 Easier configuration of CI servers or services. While many CI tools offer a great deal of flexibility in how they can be used, this flexibility can require a large amount of configuration even for a simple workflow. From our interviews, we find that developers for large software companies rely on the CI engineers to ensure that the configuration is correct, and to help instantiate new configurations. Open-source developers often use CI as a service, which allows for a much simpler configuration. However, for developers trying to configure their own CI server, this can be a substantial hurdle. S8, who was running his own CI server, said:

Table 3.3: Developer needs unmet by CI

Needs	Broad	Focused
N1 Easier configuration of CI servers or services	267 (52%)	32%
N2 Better tool integration	198 (38%)	17%
N3 Better container/virtualization support	191 (37%)	27%
N4 Debugging assistance	153 (30%)	30%
N5 User interfaces for modifying CI configurations	147 (29%)	20%
N6 Better notifications from CI servers or services	111 (22%)	25%
N7 Better security and access controls	85 (16%)	32%

> *The configuration and setup is costly, in time and effort, and yeah, there is a learning curve, on how to setup Jenkins, and setup the permissions, and the signing of certificates, and all these things. At first, when I didn't know all these tools, I would have to sort them out, and at the start, you just don't know...*

N2 Better tool integration. Our interview participants told us that they would like their CI system to better integrate with other tools. For Example, S3 remarked:

> *It would also be cool if the CI ran more analysis on the code, rather than just the tests. Stuff like Lint, FindBugs, or it could run bug detection tools. There are probably CIs that already do that, but ours doesn't.*

Additionally, in our survey responses, participants added in the "other" field both technical problems, such as poor interoperability between node.js and Jenkins, as well as non-technical problems, such as "The server team will not install a CI tool for us".

N3 Better container/virtualization support. One core concept in CI is that each build should be done in a clean environment, i.e., it should not depend on the environment containing the output from any previous builds. Participants told us that this was very difficult to achieve before software-based container platforms; e.g. Docker. However, there are still times when the build fails, and in doing so, breaks the CI server. S15 explained:

> *...there will be [CI] failures, where we have to go through and manually clean up the environment.*

S3 had experienced the same issues and had resorted to building Docker containers inside other Docker containers to ensure that everything was cleaned up properly.

N4 Debugging assistance. When asked about how they debug test failures detected by their CI, most of our participants told us that they get the output logs and start their search there. These output logs can be quite large in size though, with hundreds of thousands of lines of output, from thousands of tests. This can create quite a challenge when trying to find a specific failure. S7 suggested that they would like their CI server to diff the output from the previous run and hide all the output which remained unchanged. S15, who worked for a large company, had developed an in-house tool to do exactly this, to help developers find errors faster by filtering the output to only show changes from the previous CI run.

N5 User interfaces for modifying CI configurations. Many participants described administering their CI tools via configuration scripts. However, participants expressed a desire to make these configuration files editable via a user interface, which they felt would be easier. S3 said:

> *Most of the stuff we are configuring could go in a UI. ... We are not modifying heavy logic. We just go in a script and modify some values. ... So all of the tedious stuff you modify by hand could go into a UI.*

Additionally, multiple participants also added "Bad UI" as a free-form answer to the question about problems experienced with CI. Developers want to be able to edit their configuration files via user interfaces, but they also want to be able to commit these configurations to their repository. Our interview participants told us they want to commit the configurations, because then when they fork a repository, the CI configurations are included with the new fork as well.

N6 Better notifications from CI servers or services. Almost all participants had the ability to setup notifications from their CI server, but very few found them to be useful. When asked about notifications from his CI, S7 said that he will routinely receive up to 20 emails from a single pull request, which he will immediately delete. Other participants did in fact find the notifications useful, though, including S10 who reads through them every morning, to refresh his memory of where he left off the day before.

N7 Better security and access controls. This need is similar to *B8 Security and access controls*, see discussion in section 3.5.1.

Observation

Developers want CI to be both a highly-configurable platform, and simple to setup and maintain. This creates tension because adding configurability increases complexity, whereas simplification necessarily seeks to reduce complexity.

3.5.3 Motivations

In this section we answer *Why do developers use CI* (**RQ3**)

Table 3.4: Developers' motivation for using CI

Motivation	Broad	Focused
M1 CI helps us catch errors earlier	375 (75%)	86%
M2 CI makes us less worried about breaking our builds	359 (72%)	82%
M3 CI provides a common build environment	349 (70%)	78%
M4 CI helps us deploy more often	339 (68%)	75%
M5 CI allows faster iterations	284 (57%)	76%
M6 CI makes integration easier	283 (57%)	75%
M7 CI can enforce a specific workflow	198 (40%)	51%
M8 CI allows testing across multiple platforms	143 (29%)	73%

M1 CI helps catch bugs earlier. Preventing the deployment of broken code is a major concern for developers. Finding and fixing bugs in production can be an expensive and stressful endeavor. Kerzazi and Adams [65] reported that 50% of all post-release failures were because of bugs. We would expect that preventing bugs from going into production is a major concern for developers. Indeed, many interview participants said that one of the biggest benefits of CI was that it identifies bugs early on, keeping them out of the production code. For example, S3 said:

> *[CI] does have a pretty big impact on [catching bugs]. It allows us to find issues even before they get into our main repo, ... rather than letting bugs go unnoticed, for months, and letting users catch them.*

M2 Less worry about breaking the build. Kerzazi et al. [66] reported that for one project, up to 2,300 man-hours were lost over a six month period due to broken builds. Not surprisingly, this was a common theme among interview participants. For instance, S3 discussed how often this happened before CI:

> *...and since we didn't have CI it was a nightmare. We usually tried to synchronize our changes, ... [but] our build used to break two or three times a day.*

S2 talked about the repercussions of breaking the build:

> *[When the build breaks], you gotta wait for whoever broke it to fix it. Sometimes they don't know how, sometimes they left for the day, sometimes they have gone on vacation for a week. There were a lot of points at which all of us, a whole chunk of the dev team was no longer able to be productive.*

M3 Providing a common build environment. One challenge developers face is ensuring that the environment contains all dependencies needed to build the software. By starting the CI process with a clean environment, fetching all the dependencies, and then building the code each time, developers can be assured that they can always build their code. Several developers told us that in their team if the code doesn't build on the CI server, then the build is considered broken, regardless of how it behaves on an individual developer's machine. For example, S5 said:

> *...If it doesn't work here (on the CI), it doesn't matter if it works on your machine.*

M4 CI helps projects deploy more often. Previous work [60] has found open-source projects that use CI deploy twice as often as projects that do not use CI. In our interviews, developers told us that they feel that CI helped them deploy more often. Additionally, developers told us that CI enabled them to have shorter development cycles than they otherwise would have, even if they did not deploy often for business reasons. For example, S14 said:

> *[Every two weeks] we merge into master, and consider that releasable. We don't often release every sprint, because our customer doesn't want to. Since we are services company, not a products company, its up to our customer to decide if they want to release, but we ensure every two weeks our code is releasable if the customer chooses to do so.*

M5 CI allows faster iterations. Participants told us that running CI for every change allows them to quickly identify when the current change set will break the build, or will cause problems in some other location(s) of the codebase. Having this immediate feedback enables much faster development cycles. This speed allows developers to make large changes quickly, without introducing a large amount of bugs into the codebase. S15 stated:

> *We were able to run through up to 10 or 15 cycles a day, running through different tests, to find where we were, what solutions needed to be where. Without being able to do that, without that speed, and that feedback, there is no way we could have accomplished releasing the software in the time frame required with the quality we wanted.*

M6 CI makes integration easier. Initially, CI was presented as a way to avoid painful integrations [52]. However, while developers do think CI makes integration easier, it is not the primary reason that motivates developers to use CI. For many developers, they see their VCS as the solution to difficult integrations, not the CI.

M7 Enforcing a specific workflow. Prior to CI, there was no common way for tools to enforce a specific workflow (e.g., ensuring all tests are run before accepting changes).

This is especially a concern for distributed teams, where it is harder to overcome tooling gaps through informal communication channels. However, with CI, not only are all the tests run on every changeset, but everyone knows what the results are. Everyone on the team is aware when a code breaks the tests or the builds, without having to download the code, and verifying the test results on their own machine. This can help find bugs faster and increase team awareness, both of which are important parts of code review [24]. S16 told us that he was pretty sure that before they added CI to their project, contributors were not running the tests routinely.

M8 Test across all platforms. CI allows a system to be tested on all major platforms (Windows, Linux, and OS X), without each environment being setup locally by each developer. For example, S16 stated:

> *We are testing across more platforms now, it is not just OS X and Linux, which is mostly what developers on projects run. That has been useful.*

Nevertheless, one survey participant responded to our open-ended question at the end of the survey:

Simplifying CI across platforms could be easier. We currently want to test for OS X, Linux and Windows and need to have 3 CI services.

While this is a benefit already realized for some participants, others see this as an area in which substantial improvements could be made to CI to provide additional support.

Observation

Developers use CI to guarantee quality, consistency, and viability across different environments. However, adding and maintaining automated tests causes these benefits to come at the expense of increased time and effort.

3.5.4 Experiences

In this section, we answer *What benefits do developers experience when using CI* (**RQ4**)

Devanbu et al. [41] found that developers have strongly held beliefs, often based on personal experience more than research results, and that practitioner beliefs should be given due attention. In this section we present developers' beliefs about using CI. Our results show developers are very positive about the use of CI.

E1 Developers believe projects with CI give more value to automated tests. Several participants told us that before using CI, although developers would write unit tests, they often would not be run, and developers did not feel that writing tests were worth the effort. S11 related:

Several situations I have been in, there is no CI, but there is a test suite, and there is a vague expectation that someone is running this test sometimes. And if you are the poor schmuck that actually cares about tests, and you are trying to run them, and you can't get anything to pass, and you don't know why, and you are hunting around like "does anyone else actually do this?"

However, due to the introduction of CI, developers were able to see their tests being run for every change set, and the whole team becomes aware when the tests catch an error that otherwise would have made it into the product. S16 summarized this feeling:

[CI] increases the value of tests, and makes us more likely to write tests, to always have that check in there. [Without CI, developers] are not always going

to run the tests locally, or you might not have the time to, if it is a larger suite.

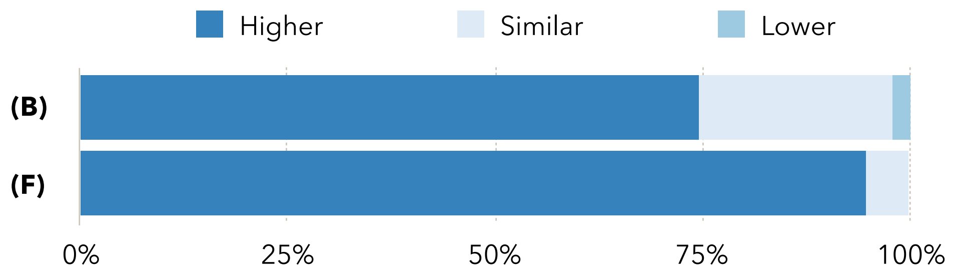


Figure 3.2: Do projects with CI have (higher/similar/lower) test quality? Data Sources: (B)road Survey, (F)ocused Survey.

E2 Developers believe projects with CI have higher quality tests. Interview participants told us that because projects that use CI run their automated tests more often, and the results are visible to the entire team, this motivates developers to write higher quality tests.

Several participants claimed that using CI resulted in higher test coverage, which they equate with higher quality tests. For example, S8 said:

> *... We jumped the coverage from a single digit to 50% of the code base in one year.*

To confirm this, we asked the same question of survey participants. Figure 3.2 shows that the survey participants overwhelmingly agree that projects with CI have higher quality tests.

E3 Developers believe projects that use CI have higher code quality. Developers believe that using CI leads to higher code quality. By writing a good automated test suite, and running it after every change, developers can quickly identify when they make a change that does not behave as anticipated, or breaks some other part of the code. S10 said:

> *CI for me is a very intimate part of my development process. ... I lean on it for confidence in all areas. Essentially, if I don't have some way of measuring my test coverage, my confidence is low. ... If I don't have at least one end-to-end*

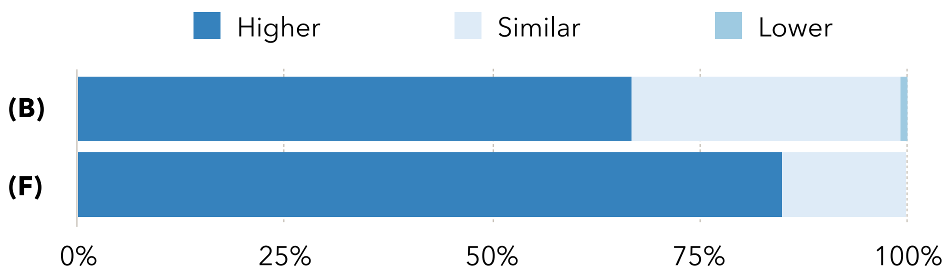


Figure 3.3: Do projects with CI have (higher/similar/lower) code quality? Data Sources: (B)road Survey, (F)ocused Survey.

> *test, to make sure it runs as humans expect it to run, my confidence is low. So I lean pretty heavily on CI, I use it all day everyday.*

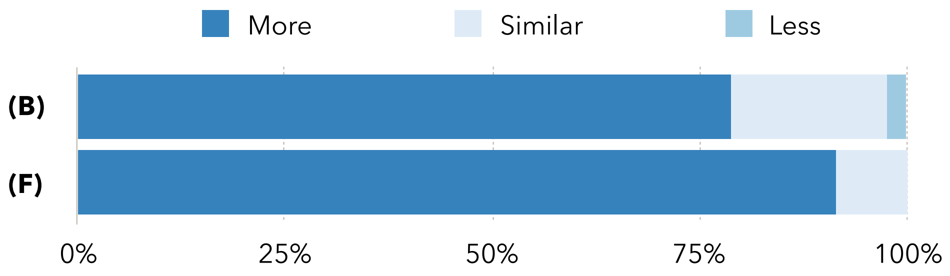


Figure 3.4: Are developers on projects with CI (more/similar/less) productive? Data Sources: (B)road Survey, (F)ocused Survey.

E4 Developers believe projects with CI are more productive.
According to our participants, CI allows developers to focus more on being productive, and to let the CI take care of boring, repetitive steps, which can be handled by automation. S2 said:

> *It just gets so unwieldy, and trying to keep track of all those bits and pieces that are moving around, ... [CI makes it] easier it is for them to just focus on what they need to do.*

Another reason interview participants gave for being more productive with CI was that CI allows for faster iterations, which helps developers be more productive. S16 said:

I think [CI] has made it easier to iterate more quickly, because you can have more trust that you are not breaking all the things.

Observation

Some perceived benefits of CI are also benefits from automated testing, test suit maintenance, and adherence to project standards. However, developers attribute these benefits to CI.

3.6 Discussion

In this section we discuss the trade-offs developers face when using CI, the implications of those trade-offs, and the differences between our two surveys.

3.6.1 CI Trade-offs

As with any technology, developers who use CI should be aware of the trade-offs that arise when using that technology. In this section we will look into three trade-offs that developers should be aware of when using CI: *Assurance*, *Security*, and *Flexibility*.

Assurance (Speed vs Certainty): Developers must consider the trade-off between speed and certainty. One of the benefits of CI is that it improves validation of the code (see M1, M2, and M9).

However, the certainty that code is correct comes at a price. Building and running all these additional tests causes the CI to slow down, which developers also considered a problem (see B2, M10). Ensuring that their code is correctly tested, but keeping build times manageable is a trade-off developers must be aware of. Rothermel et al. [93] also identify this trade-off in terms of running tests as a motivation for test prioritization.

Security (Access vs Information Security): Information security should be considered by all developers. Developers are concerned about security when using CI (see B8, N7). This is important because a CI pipeline should protect the integrity of the code passing through the pipeline, protect any sensitive information needed during the build and test process (e.g., credentials to a database), as well as protect the machines that are running the CI system.

However, limiting access to the CI pipeline conflicts developers' need for better access (see B1, N4). During our interviews, developers reported that troubleshooting CI build

fails was often difficult because they did not have the same access to code running on a CI system, as they did when running it locally on their own machine. Providing more access may make debugging easier, but poses challenges when trying to ensure the integrity of the CI pipeline. Post and Kagan [90] examine this trade-off for knowledge workers, and found security restrictions hinder a third of workers from being able to perform their jobs.
Flexibility (Configuration vs Simplicity): Another trade-off that developers face is between the flexibility and power of highly configurable CI systems, and the ease of use that comes from simplicity. Developers wish to have more flexibility in configuring and using their CI systems (see B4, B7, N2, and N3). More flexibility increase the power of a CI system, while at the same time also increasing its complexity.

However, the rising complexity of CI systems is also a concern for developers (see B5, B6, N1, and N5). Developers needs for more flexibility directly opposes the desire for more simplicity. Xu et al. [117] examine over-configurable systems and also found that these systems severely hinder usability.

3.6.2 Implications

Each of these three trade-offs leads to direct implications for *Developers*, *Tool Builders*, and *Researchers*.

Assurance (Speed vs Certainty)

Developers should be careful to only write tests that add value to the project. Tests that do not provide value still consume resources every CI build, and slow down the build process. As more tests are written over time, build times will trend upward. Teams should schedule time for developers to maintain their test suites, where they can perform tasks such as removing unneeded tests, improving the test suite by filling in gaps in coverage, or increasing the quality of tests.

Developers face difficult choices about the extent to which each project should be tested, and to what extent they are willing to slow down the build process to achieve that level of testing. Some projects can accept speed reductions because of large, rigorous tests. However, for other projects, it may be better to keep the test run times faster, by only executing some of the tests. While this can be done manually, developers should consider using established test selection/minimization approaches [29, 35, 54, 61, 120].

Tool Builders can support developers by creating tools that allow developers to easily run subsets of their testing suites [120]. Helping developers perform better test selection can trade some certainty for speed gains.

Researchers should investigate the trade-offs between speed and certainty. Are there specific thresholds where the build duration matters more than others? Our results suggest that developers find it important to keep build times under 10 minutes. Researchers should find ways to give the best possible feedback to developers within 10 minutes.

Another avenue for researchers is to build upon previous work [46] using test selection and test prioritization to make the CI process more cost effective.

Security (Access vs Information Security)

Developers should be cognizant of the security concerns that extra access to the CI pipeline introduces. This is especially a concern for developers inside companies where some or all of their code is open source. One interview participant told us that they navigate the dichotomy between security and openness by maintaining both an internal CI server that operates behind their company firewall, and using Travis CI externally. They cannot expose their internal CI due to confidentiality requirements, but they use external CI to be taken seriously and maintain a positive relationship with the developer community at large.

Tool Builders should provide developers with the ability to have more access to the build pipeline, without compromising the security of the system. One way of accomplishing this is to provide fine-grained account management with different levels of access; restricting less trusted accounts to view-only access of build results, and allowing trusted accounts to have full access to build results and management features in the CI system.

Researchers should explore the security challenges that arise when using CI. Although CI aims at automating and simplifying the testing and validation process, the increased infrastructure provides additional attack vectors that can be exploited. The security implications of CI require more thorough examination by security researchers in particular.

Researchers should also examine the feasibility of creating systems that allow developers to safely expose their CI systems without compromising their security.

Flexibility (Configuration vs Simplicity)

Developers should recognize that custom development processes bring complexity, increase maintenance costs, installation costs, etc. They should consider adopting convention over configuration if they want to reduce the complexity of their CI system. Developers should strive to keep their processes as simple as possible, to avoid adding unneeded complexity.

Developers should consider the long-term costs of highly complex custom CI systems. If the CI becomes overly complex, and the administration is not shared among a team, there is a vulnerability to the overall long-term viability if the maintainer leaves the project. Developers should also consider the long-term maintenance costs when considering adding complexity to their CI pipeline.

Tool Builders must contend with developers that want expanded UIs for managing the CI pipeline, as well as having the underlying configurations be captured by version-control systems. Tool Builders should create tools that allow for UI changes to configurations, but also output those configurations in simple text files that can be easily included in version control.

Researchers should collect empirical evidence that helps developers, who wish to reduce complexity by prioritizing convention over configuration, to establish those conventions based on evidence, not on arbitrary decisions. Researchers should develop a series of empirically justified "best practices" for CI processes.

Also, developers who use CI believe strongly that CI improves test quality, and that CI makes them more productive. Researchers should evaluate whether these claims are indeed true.

3.6.3 Focused (Pivotal) vs Broad Survey Results

We deployed a Focused Survey at a single company (Pivotal), and a Broad Survey to a large population of developers using social media. After performing both surveys, we discussed the findings with a manager at Pivotal, and these discussions allowed us to develop a deeper understanding of the results.

Flaky Tests The survey deployed at Pivotal contained 4 additional questions requested by Pivotal. One question asked developers to report the number of CI builds failing each week due to true test failures. Another question asked developers the number of CI builds failing due to non-deterministic (flaky) tests [78]. Figure 3.5 shows the reported number of CI build failures because of flaky tests, as well as failures due to true test failures. There was no significant difference between the two distributions (Pearson's Chi-squared test, p-value = 0.48), suggesting that developers experienced similar numbers of flaky and true CI failures per week. However, for the largest category, >10 fails a week, there were

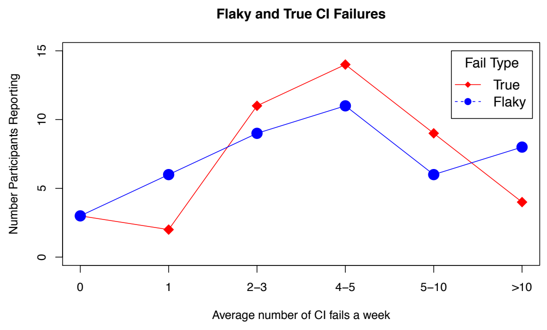


Figure 3.5: Flaky vs True Test failures reported by Pivotal Developers (N=42)

twice as many flaky failures as true failures.

When we discussed our findings with the manager at Pivotal, he indicated this was the most surprising finding. He related that at Pivotal, they had a culture of trying to remove flakiness from tests whenever possible. That claim was supported by our survey response, where 97.67% of Pivotal participants reported that when they encounter a flaky test, they fix it. Nevertheless, CI failures were just as likely to be caused by flaky tests as by true failures.

Build Times Focused Survey respondents indicated that their CI build times typically take "greater than 60 minutes". This is in contrast with the "5-10 minutes" average response from respondents in the Broad Survey. This difference can also be observed in the acceptable build time question, in which Focused Survey respondents selected "varies by project" most often compared to the Broad Survey respondents that selected "10 minutes" as the most commonly acceptable build time.

Pivotal management promotes the use of CI, and the accompanying automation that comes with it, for as many aspects of their software development as possible. According to the manager at Pivotal, the difference in responses for build times and acceptable build times can be explained by the belief that adhering to Test-Driven Development (TDD) results in significantly more unit tests, but for Pivotal, the extra testing is worth the longer BI build times. The manager also suggested that the addition of multiple target platforms in the CI builds will also necessarily increase build times. Therefore, at Pivotal, while they seek to reduce those times whenever possible, they accept longer than average build times when necessary.

Maintenance Costs Focused Survey respondents reported experiencing "troubleshooting a CI build failure", "overly long CI build times", and "maintaining a CI server or service" more often than the Broad Survey respondents. When asked about this difference, the manager at Pivotal indicated that they actively promote a culture of process ownership within their development teams, so the developers are responsible for maintaining and configuring the CI services that they use. They also said said that the CI that they use is more powerful and more complex than other CI solutions, resulting in a more complicated setup, but provides more control over the build process.

3.7 Related Work

Continuous Integration Studies Vasilescu et al. [113] performed a preliminary quantitative study of quality outcomes for open-source projects using CI. Hilton et al. [60] presented a quantitative study of the costs, benefits, and usage of CI in open-source software. They do not examine barriers or needs when using CI, nor do they address the trade-offs developers must contend with.

In contrast to these studies, we develop a deep understanding of the the barriers and unmet needs of developers through both interviews and surveys. We also discover three trade-offs users face when using CI.

Other researchers have studied ways to improve CI. Vos et al. [114] run CI tests even after deployment, to verify the production code. Staples et al. [103] describe Continuous Validation as a potential next step after CI/CD. Muslu et al. [85] ran the tests continuously in the IDE, even more often than running them in CI would entail. Elbaum et al. [46] examined the use of regression test selection techniques to increase the cost-effectiveness in CI processes.

Other work related to CI and automated testing includes generating acceptance tests from unit tests [64], black box test prioritization [57], ordering of failed unit tests [53], generating automated tests at runtime [23], and prioritizing acceptance tests [102].

Continuous Delivery Continuous Delivery (CD), the automated deployment of software, is enabled by the use of CI. Olsson et al. [86] performed a case study of four companies which examines barriers associated with transitioning to continuous delivery, where continuous integration is a step along the transition roadmap. They found that automating processes, migrating and integrating old tools, interdependencies between development teams, slow development cycles, lack of transparency, and quality issues in builds/systems were direct barriers to the transition to CI. This confirms our results that *automating the build process* (B3), *lack of support for desired workflow* (B4), and *lack of tool integration* (B7) are barriers that developers experience when using CI.

Leppänen et al. [77] conducted semi-structured interviews with 15 developers to learn more about CD. Their paper does not have any quantitative analysis and does not claim to provide generalized findings. Others have studied CD and MySQL schemas [40], CD at Facebook [97], and the tension between release speed and software quality when doing CD [98].

Developer Studies In this paper, we perform a study of developers to learn about their barriers, unmet needs, motivations, and experiences. Many other researchers have also studied developers, e.g., to learn how DevOps handles security [111], developers' debugging needs [75], how developers examine code history [38], and what barriers newcomers face in open-source projects [104].

Automated Testing Previous work has examined the intertwined nature of CI and automated testing. Campos et al. [36] introduced the concept of Continuous Test Generation (CTG), which includes automated unit test generation during continuous integration. Stolberg [105] and Sumrell [106] both provide experience reports of the effects of automating tests during transitions to CI. Others [42] have used test coverage to develop coverage based test-selection techniques. Santos and Hindle [96] used Travis CI build status as proxy for code quality.

3.8 Threats to Validity

Replicability *Can others replicate our results?* Qualitative studies in general are very difficult to replicate. We address this threat by conducting an interview, a focused survey at a single company, and a large-scale survey of a broad range of developers. The interview script, code set, survey questions, and raw data can be found on our website. We cannot publish the transcripts because the interview participants were told that the transcripts would not be released.

Construct *Are we asking the right questions?* To answer our research questions, we used semi-structured interviews [100], which explore themes while letting participants bring up new ideas throughout the process. By allowing participants to have the freedom to bring up topics, we avoid biasing the interviews with our preconceived ideas of CI.

Internal *Did we skew the accuracy of our results with how we collected and analyzed information?* Interviews and surveys can be affected by bias and inaccurate responses. These could be intentional or unintentional. To mitigate these concerns, we followed established guidelines in the literature [89, 99, 101] for designing and deploying our survey. We ran iterative pilots for both studies and the surveys, we kept the surveys as short as possible.

External *Do our results generalize?* By interviewing selected developers, it is not possible to understand the entire developer population. To mitigate this, we attempted to recruit as diverse a population as possible, including 14 different companies, and a wide variety of company size and domain. We then validate our responses using a Focused Survey with 51 responses, and a Broad Survey with 523 responses from over 30 countries. Because we recruited participants for the survey by advertising online, our results may be affected by self-selection bias.

3.9 Conclusions

Software teams use CI for many activities, including to catch errors, make integration easier, and deploy more often. Our paper presented an in-depth qualitative study using 16 interviews and two surveys of how developers perceive CI. On the positive, developers experience being more productive when using CI. Despite the many benefits of CI, developers still encounter a wide variety of problems when using CI. We hope that this paper motivates researchers to tackle the hard problems that developers face with CI. CI is here to stay as a development practice, and we need continuous improvement ("CI" of a different kind) of CI to realize its full potential.

Chapter 4: TDDViz:
Using Software Changes to Understand Conformance to Test Driven Development

TDDViz:
Using Software Changes to Understand Conformance to Test Driven Development

Michael Hilton,Nicholas Nelson,Hugh McDonald,Sean McDonald,Ron Metoyer,Danny Dig

Agile Processes, in Software Engineering, and Extreme Programming: 17th International Conference (XP 2016)

4.1 Abstract

A bad software development process leads to wasted effort and inferior products. In order to improve a software process, it must be first understood. Our unique approach in this paper uses code and test changes to understand conformance to the Test Driven Development (TDD) process.

We designed and implemented TDDViz, a tool that supports developers in better understanding how they conform to TDD. TDDViz supports this understanding by providing novel visualizations of developers' TDD process. To enable TDDViz's visualizations, we developed a novel automatic inferencer that identifies the phases that make up the TDD process solely based on code and test changes.

We evaluate TDDViz using two complementary methods: a controlled experiment with 35 participants to evaluate the visualization, and a case study with 2601 TDD Sessions to evaluate the inference algorithm. The controlled experiment shows that, in comparison to existing visualizations, participants performed significantly better when using TDDViz to answer questions about code evolution. In addition, the case study shows that the inferencing algorithm in TDDViz infers TDD phases with an accuracy (F-measure) of 87%.

4.2 Introduction

A bad software development process leads to wasted effort and inferior products. Unless we understand how developers are following a process, we cannot improve it.

In this paper we use Test Driven Development (TDD) as a case study on how software changes can illuminate the development process. To help developers achieve a better understanding of their process, we examined seminal research [26, 72, 121] that found questions software developers ask. From this research, we focused on three question areas. We felt that the answers to these could provide developers with a better understanding of their process. We choose three questions from the literature to focus on, and they spanned three areas: *identification*, *comprehension*, and *comparability*.

RQ1: *"Can we detect strategies, such as test-driven development?" (Identification)* [121]
RQ2: *"Why was this code changed or inserted?" (Comprehension)* [72]
RQ3: *"How much time went into testing vs. into development?" (Comparability)* [26]

To answer these questions, we use code and test changes to understand conformance to a process. In this paper, we present TDDViz, our tool which provides visualizations that support developers' understanding of how they conform to the TDD process. Our visual design is meant to answer RQ1-3 so that we ensure that our visualizations support developers in answering important questions about *identification*, *comprehension*, and *comparability* of code.

In order to enable these visualizations, we designed a novel algorithm to infer TDD phases. Given a sequence of code edits and test runs, TDDViz uses this algorithm to automatically detect changes that follow the TDD process. Moreover, the inferencer also associates specific code changes with specific parts of the TDD process. The inferencer is crucial for giving developers higher-level information that they need to improve their process.

One fundamental challenge for the inferencer is that during the TDD practice, not all code is developed according to the textbook definition of TDD. Even experienced TDD developers often selectively apply TDD during code development, and only on some parts of their code. This introduces lots of noise for any tool that checks conformance to proceses. To ensure that our inference algorithm can correctly handle noisy data, we add a fourth Research Question.

RQ4: *"Can an algorithm infer TDD phases accurately?" (Accuracy)*

To answer this question, in this paper we use a corpus of data from cyber-dojo[1] , a website that allows developers to practice and improve their TDD by coding solutions to various programming problems. Each time a user run tests, the code is committed to a git repository. Each of these commits becomes a fine-grained commit. Our corpus contains a total of 41766 fine-grained snapshots from 2601 programming sessions, each of which is an attempt to solve one of 30 different programming tasks.

To evaluate TDDViz, we performed a controlled experiment with 35 student participants already familiar with TDD. Our independent variable was using TDDViz or existing visualizations to answer questions about the TDD Process.

This paper makes the following contributions:

Process Conformance: We propose a novel usage of software changes to infer conformance to a process. Instead of analyzing metrics taken at various points in time, we analyze deltas (i.e., the changes in code and tests) to understand conformance to TDD.

TDD Visualization Design and Analysis: We present a visualization designed specifically for understanding conformance to TDD. Our visualizations show the presence or absence of TDD and allow progressive disclosure of TDD activities.

TDD Phase Inference Algorithm: We present the first algorithm to infer the activities in the TDD process solely based on snapshots taken when tests are run.

Implementation and Empirical Evaluation: We implement the visualization and inference algorithm in TDDViz, and empirically evaluate it using two complementary methods. First, we conduct a controlled experiment with 35 participants, in order to answer **RQ1-3**. Second, we evaluate the accuracy of our inferencer using a corpus of 2601 TDD sessions from cyber-dojo, in order to answer **RQ4.** Our inferencer achieves an accuracy of 87%. Together, both of these show that TDDViz is effective.

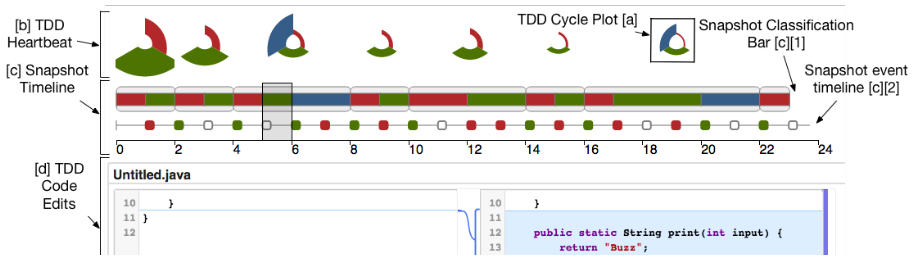
4.3 Visualization

4.3.1 Visualization Elements

4.3.1.1 TDD Cycle Plot

We represent a TDD cycle using a single glyph as shown in Figure 4.1 [a]. This repre-

[1]www.cyberdojo.org



```
    }
}
```

```
    }

    public static String print(int input) {
        return "Buzz";
```

Figure 4.1: Interactive visualization of a TDD session. The user can choose any arbitrary selection they wish. This example shows a session that conforms to TDD. Sizes in the TDD Heartbeat plot represent time spent in each phase. The different parts of the visualization have been labeled for clarity. [a] a TDD Cycle plot, [b] TDD Heartbeat, [c] Snapshot Timeline, [d] TDD Code Edits

sentation was inspired by hive plots [70] and encodes the nominal cycle data with a positional and color encoding (red=test, green=code, blue=refactor). The position of the segment redundantly encodes the TDD cycle phase (e.g. the red phase is always top right, the green phase is always at the bottom, and the blue phase is always top left). The time spent in a phase is a quantitative value encoded in the area [79, 83] of the cycle plot segment (i.e., the larger the area, the more time spent in that phase during that cycle). All subplots are on the same fixed scale. Taken together, a single cycle plot forms a glyph or specific 'shape' based on the characteristics of the phases, effectively using a 'shape' encoding for different types of TDD cycles. This design supports both *characterization* of entire cycles as well as *comparison* of a developer's time distribution in each phase of a cycle. We illustrate the shape patterns of various TDD cycles in the next section.

4.3.1.2 TDD Heartbeat

To support comparison of TDD cycles over time, we provide a small multiples view [110] that we call the TDD Heartbeat view. The TDD Heartbeat view consists of a series of TDD cycle plots, one for every cycle of that session (See Figure 4.1, [b]) We call this the TDD heartbeat because this view gives an overall picture of the *health* of the TDD process as it evolves over time. This particular view particularly supports the abstract tasks of *characterization* and *comparison*.

In particular, the user can compare entire cycles over time to see how they evolve, and she can characterize how her process is improving or degrading. For example, by looking at all the cycles that make up the TDD Heartbeat in Figure 4.1, the user sees that for every cycle in this kata, the developer spent relatively more time writing production code than writing tests. They can also observe that the relationship between the time spent in each phase was fairly consistent.

4.3.1.3 Snapshot Timeline

The snapshot timeline provides more information about the TDD process, specifically showing all the snapshots in the current session. An example snapshot timeline is shown in Figure 4.1 [c]. The snapshot timeline consists of two parts, the snapshot classification bar (F. 4.1 [c][1]) on the top, and the snapshot event timeline on the bottom (F. 4.1 [c][2]). In the snapshot event timeline, each snapshot is represented with a rounded square. The color represents the outcome of the tests at that snapshot event. Red signifies the tests were run, but at least one test failed. If all the tests passed, then it is colored green. If the code and tests do not compile, we represent this with an empty white rounded box. The distance between each snapshot is evenly distributed, since the time in that phase is encoded in the TDD Cycle Plot.

The snapshot classification bar shows the cycle boundaries, and inside each cycle the ribbon of red, green and blue signifies which snapshot events fall into which phases. For example, in Figure 4.1, snapshots 17-20 are all part of the same green phase. Snapshots 17-19 the developer is trying to get to a green, but they are not successful in making the tests pass until snapshot 20.

This view answers questions specifically dealing with how consistent coders followed the TDD process, what snapshots were written by coders using the TDD process, and which ones were not.

The snapshot timeline answers questions about identification. The timeline enables developers to identify which parts of the session conform to TDD and which do not.

This view also allows the user to interactively select snapshots that are used to populate the code edit area (described below). To select a series of snapshots, the user interactively drags and resizes the gray selection box. In Figure 4.1, snapshots 5 and 6 are selected.

The snapshot timeline also answers questions dealing with comprehension. By seeing how TDDViz catagorizes a snapshot, a user can determine why selected changes were made. For example, Figure 4.1 shows a selected snapshot which represents the changes between snapshots numbers 5 and 6. Since the selected changes are part of a green phase (as noted by the green area in the Snapshot Classification Bar), a user can determine that these were production changes to make a failing test pass. This can be confirmed by observing the code edits. This encoding supports the same questions as the cycle plot and heartbeat arrangement, but, it does so at a finer granularity, showing each individual test run.

4.3.1.4 TDD Code Edits

Figure 4.1, [d] shows an example of a code edit, which displays the changes to the code between two snapshots. To understand the TDD process, a coder must be able to look at the code that was written, and see how it evolved over time. By positioning the selection box on the timeline as described above, a user can view how all the code evolved over any two arbitrary snapshots. The code edit region contains an expandable and collapsable box for each file that was changed in the selected range of snapshots. Each box contains two code editors, one for the code at the selection's starting snapshot, and one for the code at the ending snapshot.

Whenever the user selects a new snapshot range, these boxes dynamically repopulate their content with the correct diffs. There are additional examples of our visualizations on our accompanying web page `http://cope.eecs.oregonstate.edu/visualization.html`.

4.4 TDD Phase Inferencer

In order to build the visualizations we have presented thus far, we needed a TDD phase inference algorithm which uses test and code changes to infer the TDD process. Instead of relying on static analysis tools, we present a novel approach where the algorithm analyzes the changes to the code. We designed our algorithm to take as input a series of snapshots. The algorithm then analyzes the code changes between each snapshot and uses that information to determine if the code was developed using TDD. If the algorithm infers the TDD process, then it determines which parts of the TDD process those changes belong to.

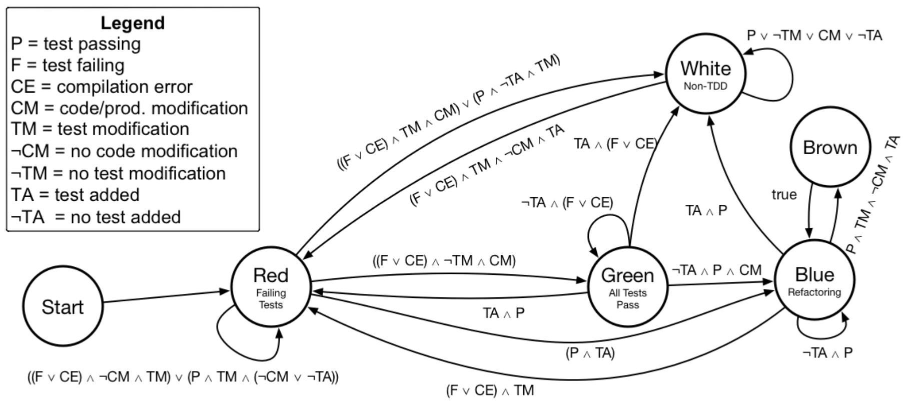


Figure 4.2: Pseudo-code of the TDD Phase Inference Algorithm.

4.4.1 Snapshots

We designed our algorithm to receive a series of snapshots as input. We define a *snapshot* as a copy of the code and tests at a given point in time. In addition to the contents of code and tests, the snapshot contains the results of running the tests at that point in time.

Our algorithm uses these snapshots to determine the developers' changes to the program. It then uses these changes to infer the TDD process. In this paper, we use a corpus of data where a snapshot was taken every time the code was compiled and the tests were run. It is important that the snapshots have this level of detail, because if they do not, we do not get a clear picture of the development process.

4.4.2 Abstract Syntax Tree

Since our inference algorithm must operate on the data that the snapshots contain, it is important to have a deeper understanding of code than just the textual contents. To this end, our inference algorithm constructs the Abstract Syntax Tree (AST) for each code and test snapshot in our data. This allows our inferencer to determine which edits belong to the production code and which edits belong to the test code. It also calculates the number of methods and assert statements at each snapshot. For the purposes of the algorithm, we consider code with asserts to be test codes, and code with no asserts to be production code. We consider each assert to be an individual test, even if it is in a method with other asserts. If a new assert is detected, we consider that to be a new test. All this information enables the algorithm to infer the phases of TDD. In our implementation of the algorithm in TDDViz, we use the Gumtree library [49] to create the ASTs.

4.4.3 Algorithm

We present the TDD phase inference algorithm using the state digram in Figure 4.2. Our algorithm encodes a finite-state machine (FSM), where the state nodes are phases, and the transitions are guided by predicates on the current snapshot.
We define each of the states as follows:
Red: This category indicates that the coder was writing test code in an attempt to make a failing test
Green: This category is when the coder is writing code in an attempt to make a failing test pass
Blue: This is when the coder has gotten the tests to pass, and is refactoring the code
White: This is when the code is written in a way that deviates from TDD
Brown: This is a special case, when the coder writes a new test and it passes on the first try, without altering the existing production code. It could be they were expecting it to fail, or perhaps they just wanted to provide extra tests for extra security.

The predicates take a snapshot and using the AST changes to the production and test code, as well as the result of the test runs, compute a boolean function. We compose several predicates to determine a transition to another state. For example: in order to transition from green to blue, the following conditions must hold true. All the current unit tests must pass, and the developer may not add any new tests.

The transition requires passing tests, because if not, the developer either remains in the green phase or has deviated from TDD. No new tests are allowed because the addition of a new test, while a valid TDD practice, would signify that the developer has skipped the optional blue phase and moved directly to the Red phase.

There are a few special cases in our algorithm. The algorithm's transition from Red to Blue is the case when a single snapshot comprised the entire Green phase, and therefore the algorithm has moved on to the blue phase. Another thing to note is that by definition, the brown phase only contains a single commit. Therefore, after the algorithm identifies a brown phase, it immediately moves back to the blue phase.

4.5 Evaluation

To evaluate the usefulness of TDDViz, we answer the following research questions:
RQ1. *Can programmers using* TDDViz *identify whether the code was developed in conformance with TDD? (Identification)*
RQ2. *Can programmers using* TDDViz *identify the reason why code was changed or inserted? (Comprehension)*
RQ3. *Can programmers using* TDDViz *determine how much time went into testing vs. development of production code? (Comparability)*
RQ4. *Can an algorithm infer TDD phases accurately? (Accuracy)*

In order to answer these research questions, we used two complementary empirical methods. We answer the first three questions with a controlled experiment with 35 participants, and the last question with a case study of 2601 TDD sessions. The experiment allows us to quantify the effectiveness of the visualization as used by programmers, while the case study gives more confidence that the proposed algorithm can handle a wide variety of TDD instances.

4.5.1 Controlled Experiment

Participants. Our participants were 35 students in a 3rd-year undergrad Software Engineering class who were in week 10 of a course in which they had used TDD for their class project.
Treatment. Our study consisted of two treatments. For the experimental treatment, we asked the participants to answer questions dealing with identification, comprehension, and comparability (RQ1–RQ3) by examining several coding sessions from cyber-dojo presented with our visualization. For the control treatment, we used the same questions applied to the same real-life code examples, but the code was visualized using the visualization[2] that is available on the cyber-dojo site. This visualization shows both the code, and the test results at each snapshot, but it does not present any information regarding the phases of TDD. We used this visualization for our control treatment because it is specifically

[2]http://cyber-dojo.org/dashboard/show/C5AE528CB0

designed to view the data in our corpus. Also, it is the only available visualization other than our own which shows both the code and the history of the test runs.

Experimental procedure. In order to isolate the effect of our visualization, both treatments had the same introduction, except for when describing the parts of the visualizations which are different across treatments. Both treatments received the exact same questions on the same real-life data in the same order. The only independent variable was which visualization was presented to each treatment. We randomly assigned the students into two groups, one group with 17 participants and the other group with 18 participants. We then flipped a coin to determine which group received which treatment. We gave both treatments back to back on the same day.

Tasks. The experiment consisted of three tasks. To evaluate *identification*, we asked "Is this entire session conforming to TDD?" To evaluate *comprehension* we asked "Why was this code changed?" To evaluate *comparability* we asked "Was more time spent writing tests or production code?" For each task we asked the same question on four different instances of TDD sessions. The students we accustomed to using clickers to answer questions, and so for each task they answered questions using questions with their clickers. Each question was a multiple choice question.

Measures. The independent variable in this study was the visualization used to answer the questions. The dependent measure was the answers to the questions. For each of the three tasks we showed the subjects four different instances and evaluated the total correct responses against the total incorrect responses. We then looked at each question and compared the control treatment versus the experimental treatment. We used Fisher's Exact Test to determine significance because we had non-parametric data.

4.5.2 Controlled Experiment Results

Table 4.1 tabulates the results for the three questions. We will now explain each result in more detail.

RQ1:*Identification.* When we asked the participants to identify TDD, we found that significantly more participants correctly identified TDD and non-TDD sessions using TDDViz than when using the default cyber-dojo visualization, as Table 4.1 shows (Fisher's Exact Test: $p<.0005$). This shows that our visualization does indeed aid in identifying

	Identification		Comprehension		Comparability	
Treatment	Correct	Not Correct	Correct	Not Correct	Correct	Not Correct
Control ($n = 18$)	37	34	24	48	23	49
Experimental ($n = 17$)	55	13	42	26	33	35

Table 4.1: Results from Controlled Experiment.

TDD.

RQ2:*Comprehension.* When we asked participants why a specific code change had been made, we found that significantly more participants correctly identified why the code was changed when using TDDViz than when using the default cyber-dojo visualization (see Table 4.1: Comprehension, Fisher's Exact Test: p<.0013). They were able to identify if the given code was changed or inserted to make a test pass, make a test fail or to refactor.

RQ3:*Comparability.* When we asked our participants to compare the amount of time that went into writing tests vs. the time that went into writing code, the experimental participants were able to outperform the control group but only by a small margin. The difference was only just approaching significance (Fisher's Exact Test: p<0.0578). Additionally, as Table 4.1: Comparability shows, there were slightly more incorrect answers then correct answers for the experimental group. To answer this question, users had to mentally quantify whether the chart contained more red than green overall. In the future we plan on improving the visualization by providing a representation that provides a clear, numerical answer to this question.

4.5.3 Case Study

We now answer our fourth research question, which measures the accuracy of the TDD phase inference part of TDDViz, using a corpus of 2601 TDD sessions.

Corpus Origin. We use a corpus of katas that comes from cyber-dojo, a site that allows developers to practice and improve TDD by coding solutions to various katas.

Evaluation Corpus. To build our corpus we used *all* the Java/JUnit sessions as our evaluation framework currently only supports Java. Adding other languages would be straightforward, but is left as future work. This gives us a corpus of 2601 total Java/JUnit sessions.

We are using this corpus to evaluate our inferencer as all the sessions were attempted by people who had no knowledge of our work.

Corpus Preparation. We developed a Ruby on Rails application that allowed us to work with this corpus in an efficient manner. The raw data that we used to build the corpus consists of a repository and session data. The git repository contains commits of the code each time the coder pressed the "Test" button. This provides a fine-grained series of snapshots that allow us to evaluate the process used to develop the code. The session data contains meta-data files that track things such as when the session occurred, and what was the result of each compile and test run.

The Gold Standard. In order to evaluate our phase inferencer, we created a Gold Standard. The first two authors manually labeled 2489 snapshots with the TDD phase to which they belong.

We then graded our inferencer by comparing its results against the Gold Standard. In order to not bias the selection process, we randomly selected the sessions for our Gold Standard. To ensure that we were labeling consistently, we first verified that we had reached an inter-rater agreement of at least 85% between both of the authors that labeled the Gold Standard on 52 sessions (32% of the sessions).

Once we were convinced that we had reached a consensus among the raters, we divided the rest of the Gold Standard sessions up and rated them individually. We labeled a total of 2489 snapshots in our Gold Standard out of a corpus of 41766 snapshots in the corpus, which is 6% of the data. We labeled each snapshot as previously defined in Section 4.4.3.

Inference Evaluation. After we manually labeled each snapshot, we ran our inference algorithm against the sessions that compose the Gold Standard. We then compare the results of the algorithm at each snapshot and compare it against the labels that were assigned by hand. We next describe how we use this comparison to calculate precision and recall.

Accuracy. We calculate the accuracy of our inferencer by using the traditional F-measure, which considers both precision and recall. We compute precision and recall by first identifying $True$ and $False\ Positives$. If the inferencer identifies a snapshot to have the same category that it has in the Gold Standard, we consider this a $True\ Positive$. If the inferencer considers a snapshot to be in a different category than the Gold Standard, we consider this case to be a $False\ Positive$. A $False\ Negative$ is where a snapshot that should have been classified as one of the TDD phases was classified by the inferencer as white (non-TDD).

Once we calculated these for each session in the Gold Standard, we calculate precision and recall using the standard formulas. Next we calculate accuracy using the traditional harmonic mean of precision and recall.

4.5.4 Case Study Results

Precision. The Gold Standard contains 2489 snapshots. Of those, 2028 were correctly identified by the inferencer. This lead to a precision of 81%. The diversity of our corpus leads to a wide variety of TDD implementations, and there are quite a few edge cases. While our algorithm handles many of them, there are still a few edge cases that our algorithm cannot recognize in its current incarnation. These are cases that are hard even for human experts to agree upon.

Recall. Our Gold Standard contains 1517 snapshots that belong to one of the TDD phases (i.e., non-white phases). Of those, our inferencer correctly classified 1440, leading to a recall of 95%. Of the remaining 5% missed cases, most of them arise because of difficulty identifying the template code the katas start with. This is an issue that can be easily solved in our future work.

RQ4:*Accuracy.* We calculate the accuracy using the F-measure. This gives us an accuracy of 87%. This shows that our inferencer is accurate and effective.

4.6 Related Work and Conclusions

Related Work Multiple projects [81, 116] detect the absence of TDD activities and give warnings when a developer deviates from TDD by identifying when a developer spends too much time writing writing code without tests. In contrast, TDDVIZ provides detailed analysis of the TDD phases, infers the presence or absence of TDD not based on time intervals between test runs, but on code and test changes. Thus, it is much more precise.

Several projects [69, 115] infer TDD phases from low-level IDE edits. They all build on top of HackyStat [63], a framework for data collection and analysis. Hackystat collects "low-level and voluminous" data, which it sends to a web service for lexical parsing, event stream grouping, and development process analysis. In contrast to these approaches, by using AST analysis, TDDViz infers the TDD process without the entire stream of low-level actions.

TDD Dashboard[3] is a service offered by Industrial Logic, to visualize the TDD process. It is based on recording test and refactoring events in a IDE, but does not infer and visualize the phases of each cycle, thus enabling developers to answer questions on identification, comprehension, and comparability.

Conclusions Without understanding there can be no improvement. In this paper we presented visualizations that enable developers to better understand the development process. To design these visualizations, we developed an inferencer that infers the TDD process with a novel use of code changes. We implemented the visualizations and the inferencer in a tool, TDDViz. We evaluated TDDViz using two complementary methods. We evaluated the visualization using 35 participants. We found that participants that used our visualization had significantly more correct answers when answering questions on identification, comprehension, and comparability of code. We evaluated the TDD phase inferencer and showed that it is accurate and effective, with 81% precision and 95% recall.

Acknowledgements We thank Mihai Codoban, Kendall Bailey and anonymous reviewers for their feedback on earlier versions of this paper. We thank Jon Jagger for making the TDD data available. We also thank Lutz Prechelt for his helpful and very thorough review. This work was partially funded through the NSF CCF-1439957 grant.

[3]https://ecoach.industriallogic.com/ dashboard?team=il

Chapter 5: Conclusions

5.1 Discussion

This work seeks to better understand software development and testing practices, specifically both Test Driven Development (TDD) and Continuous Integration (CI). In this section we synthesize our findings from the previous chapters, and we discuss which findings are pointing in the same direction and which findings are pointing in different directions.

Both TDD and CI are practices of the Extreme Programming (XP) methodology. During this research, when talking with a manager at Pivotal software, he told us that he views TDD and CI as being strongly linked. He stated that they know that their builds take longer to run because they practice TDD, which results in more tests, but that is a conscious choice they made as an organization.

We also found interesting relationships between our study of open source usage of CI and our study of tradeoffs in CI. The developers that we surveyed in our study of tradeoffs were mostly developers of proprietary code. Only 7% of our survey respondents developed "mostly open source" code. For the open source survey, it was only sent to developers who had contributed to open source projects. From these two populations, we found some interesting similarities and some differences.

When we examined build times of open source projects, we found the average build time was around 8 minutes. When we asked proprietary developers what their maximum acceptable build times was, the most common answer was 10 minutes. These findings seem to complement each other. However, during the interviews, many of the proprietary developers described longer build times. 75% of developers from our broad survey population told us that they had worked to reduce their build times. In the future, it would be important to study build times in a variety of contexts, and to study the relationship of build time with programmer productivity.

However, we also found some differences. The most common reason open source developers gave for not using CI was that their other team members were not familiar enough with it. However, for proprietary developers, the most common response was that they were not using CI but they hoped to use it in the future. Many of those developers also added in the free form text that they were unable to use CI because management did not approve. Perhaps education efforts into CI could help both open source developers to introduce team members who are not familiar with CI to the basic principles, as well as help proprietary developers convince their managers to adopt CI as a practice.

Another difference that we found was that for open source projects, Travis CI was the most common CI service. However, for the proprietary developers, Jenkins was the most common, with over 70% of developers saying they use Jenkins. This confirmed our intuition, that Travis is more popular with open source, and Jenkins is more popular with proprietary developers. Travis is free for open source projects, while Jenkins is an application that is often run on proprietary hardware inside a company's own infrastructure.

Further work is needed to further examine the differences between the use of CI by proprietary developers and developers of open source code.

5.2 Conclusions and Future Work

This work is motivated by an effort to better understand developer practices.

The first practice we consider is Continuous Integration. We first examine how CI is used in Open Source (Chapter 2). From this we learn that projects who use CI do indeed deploy more often than projects who do not. We also see that the most popular Open Source projects are more likely to use CI.

We next perform a qualitative study of CI to develop a better understanding of what needs and barriers developers face when using CI (Chapter 3). From these interviews we identify three areas where developers must deal with trade-offs when using CI. We name these trade-offs *Assurance*, *Security*, and *Flexibility*.

Finally, we examine the developer practice of Test Driven Development. To better help developers understand their own process, we focus on three areas: *identification*, *comprehension*, and *comparability*. We develop an algorithm for identifying the TDD process, as well as a visualization enable developers to comprehend and learn from their past efforts.

There is still much to be learned about the practice of Continuous Integration.

Without data from real developers, researchers cannot be sure how practical their results will be. While there is much that can be learned from studying code repositories, there is a lot of information that is not captured in a repository. For example, when working with repositories, there is no way to know which tests were run, and what the results of the tests were at that point in time. However, this data is available from CI services.

CI servers compile the code, run tests, and record the output. Many CI servers also collect metrics including test coverage. Since CI servers operate in conjunction with source control, they also contain the code that was run at that specific point in time. Moreover, many systems also include features such as issue tracking, where failures in compilations and tests can be traced to issues which can then be traced to code patches. This data offers many new opportunities for research that were previously unavailable. One initial line of work could be validating current test research, with new real-world data from a large number of projects. This could provide significant insight into how certain testing approaches work in various different situations.

Moreover, what information that the CI system doesn't provide, could be captured by adding minor instrumentation at the CI server. The CI already has access to the entire code, and is building and testing it. By injecting instrumentation into the CI server, one could study almost any aspect of the development process. For example, with some minor instrumentation of the IDE, the CI server can perform analysis on how developers change the code, how code completion was used to write that code, and much more.

It is our belief that improvement can only come through understanding. Only by better understanding developer's practices, can we improve on them, both for individual developers, and for the entire field of Software Engineering.

Bibliography

[1] 7 reasons why you should be using continuous integration. `https://about.gitlab.com/2015/02/03/7-reasons-why-you-should-be-using-ci/`. Accessed: 2016-04-24.

[2] AppVeyor. `https://www.appveyor.com/`. Accessed: 2016-04-26.

[3] The benefits of continuous integration. `https://blog.codeship.com/benefits-of-continuous-integration/`. Accessed: 2016-04-24.

[4] Build in the cloud. `http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html`.

[5] CircleCI. `https://circleci.com/`. Accessed: 2016-04-26.

[6] CloudBees. `http://cloudbees.com/`. Accessed: 2016-04-26.

[7] Continuous integration. `https://www.thoughtworks.com/continuous-integration`. Accessed: 2016-04-24.

[8] Continuous integration is dead. `http://www.yegor256.com/2014/10/08/continuous-integration-is-dead.html`. Accessed: 2016-04-24.

[9] CruiseControl. `http://cruisecontrol.sourceforge.net/`. Accessed: 2016-04-21.

[10] CrunchBase. `https://www.crunchbase.com/organization/travis-ci#/entity`. Accessed: 2016-04-24.

[11] Google Search Trends. `https://www.google.com/trends/`. Accessed: 2016-04-24.

[12] Jenkins. `https://jenkins.io/`. Accessed: 2016-04-21.

[13] Restkit. `https://github.com/RestKit/RestKit`. Accessed: 2016-04-29.

[14] Stackoverflow. `http://stackoverflow.com/questions/214695/what-are-some-arguments-against-using-continuous-integration`. Accessed: 2016-04-24.

[15] Team Foundation Server. `https://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx`. Accessed: 2016-04-21.

[16] Tools for software engineers. http://research.microsoft.com/en-us/projects/tse/. Accessed: 2016-04-24.

[17] Travis CI. https://travis-ci.org/. Accessed: 2016-04-21.

[18] Werker. http://wercker.com/. Accessed: 2016-04-26.

[19] Why don't we use continuous integration? https://blog.inf.ed.ac.uk/sapm/2014/02/14/why-dont-we-use-continuous-integration/. Accessed: 2016-04-24.

[20] Yaml: Yaml ain't markup language. http://yaml.org/. Accessed: 2016-04-24.

[21] Jafar M. Al-Kofahi, Hung Viet Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Detecting semantic changes in Makefile build code. In *ICSM*, 2012.

[22] John Allspaw and Paul Hammond. 10+ deploys per day: Dev and ops cooperation at Flickr. https://www.youtube.com/watch?v=LdOe18KhtT4. Accessed: 2016-04-21.

[23] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A Framework for Automated Testing of Javascript Web Applications. In *ICSE*, 2011.

[24] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *ICSE*, 2013.

[25] K. Beck. Embracing change with Extreme Programming. *Computer*, 32(10):70–77, 1999.

[26] Andrew Begel and Thomas Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *ICSE 2014*, June 2014.

[27] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *ICSE*, 2015.

[28] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An analysis of travis ci builds with github. Technical report, PeerJ Preprints, 2016.

[29] John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *TOSEM*, 2001.

[30] Michael H Birnbaum. Human research and data collection via the internet. *Annu. Rev. Psychol.*, 2004.

[31] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., 1991.

[32] Martin Brandtner, Emanuel Giger, and Harald C. Gall. Supporting continuous integration by mashing-up software quality information. In *CSMR-WCRE*, 2014.

[33] Martin Brandtner, Sebastian C. Müller, Philipp Leitner, and Harald C. Gall. SQA-Profiles: Rule-based activity profiles for continuous integration environments. In *SANER*, 2015.

[34] John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. Coding in-depth semistructured interviews problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research*, 42, 2013.

[35] J. Campos and R. Abreu. Leveraging a constraint solver for minimizing test suites. In *International Conference on Quality Software*, 2013.

[36] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *ASE*, 2014.

[37] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. Build system with lazy retrieval for Java projects. In *FSE*, 2016.

[38] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey. Software history under the lens. In *ICSME*, 2015.

[39] José Carlos Medeiros de Campos, Andrea Arcuri, Gordon Fraser, and Rui Filipe Lima Maranhão de Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *ASE*, 2014.

[40] Michael de Jong and Arie van Deursen. Continuous Deployment and Schema Evolution in SQL Databases. In *RELENG*, 2015.

[41] Prem Devanbu, Thomas Zimmermann, and Christian Bird. Belief & Evidence in Empirical Software Engineering. In *ICSE*, 2016.

[42] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization. *Software Testing, Verification and Reliability*, 25, 2015.

[43] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *ASE*, 2012.

[44] John Downs, Beryl Plimmer, and John G. Hosking. Ambient awareness of build status in collocated software teams. In *ICSE*, 2012.

[45] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, 2014.

[46] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, 2014.

[47] Jakob Engblom. Virtual to the (Near) End. In *DAC*, 2015.

[48] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In *OOPSLA*, 2015.

[49] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[50] B. Fitzgerald, K. J. Stol, R. O'Sullivan, and D. O'Brien. Scaling agile methods to regulated environments: An industry case study. In *ICSE*, 2013.

[51] Martin Fowler. Continuous Integration. `http://martinfowler.com/articles/originalContinuousIntegration.html`. Accessed: 2016-04-21.

[52] Martin Fowler. Continuous Integration, 2006.

[53] M. Galli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *ICSM*, 2004.

[54] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, 2015.

[55] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, 2016.

[56] Georgios Gousios. The GHTorrent dataset and tool suite. In *MSR*, 2013.

[57] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing White-box and Black-box Test Prioritization. In *ICSE*, 2016.

[58] Michael Hilton, Nicholas Nelson, Hugh McDonald, Sean McDonald, Ron Metoyer, and Danny Dig. TDDViz: Using Software Changes to Understand Conformance to Test Driven Development. pages 53–65. Proceedings of Agile Processes, in Software Engineering, and Extreme Programming: 17th International Conference (XP'16), 2016.

[59] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 426–437, 2016.

[60] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *ASE*, 2016.

[61] Sheng Huang, Jun Zhu, and Yuan Ni. Orts: A tool for optimized regression testing selection. In *OOPSLA*, 2009.

[62] Jez Humble. Evidence and case studies. `http://continuousdelivery.com/evidence-case-studies/`. Accessed: 2016-04-29.

[63] Philip M Johnson. Project hackystat: Accelerating adoption of empirically guided software development through non-disruptive, developer-centric, in-process data collection and analysis. *Department of Information and Computer Sciences, University of Hawaii*, 22, 2001.

[64] M. Jorde, S. Elbaum, and M. B. Dwyer. Increasing Test Granularity by Aggregating Unit Tests. In *ASE*, 2008.

[65] N. Kerzazi and B. Adams. Botched Releases. In *SANER*, 2016.

[66] N. Kerzazi, F. Khomh, and B. Adams. Why Do Automated Builds Break? In *ICSME*, 2014.

[67] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. Supporting continuous integration by code-churn based test selection. In *RCoSE*, 2015.

[68] Pavneet Singh Kochhar, Ferdian Thung, David Lo, and Julia L. Lawall. An empirical study on the adequacy of testing in open source projects. In *APSEC*, 2014.

[69] Hongbing Kou, Philip M. Johnson, and Hakan Erdogmus. Operational definition and automated inference of test-driven development with zorro. *Automated Software Engineering*, 17(1):57–85, 11 2009.

[70] Martin Krzywinski, Inanc Birol, Steven JM Jones, and Marco A. Marra. Hive plots, rational approach to visualizing networks. *Briefings in Bioinformatics*, page bbr069, December 2011.

[71] Victor Kuechler, Claire Gilbertson, and Carlos Jensen. Gender differences in early free and open source software joining process. In *IFIP*, 2012.

[72] Thomas D. LaToza and Brad A. Myers. Hard-to-answer questions about code. In *PLATEAU '10*, pages 8:1–8:6, 2010.

[73] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining Mental Models. In *ICSE*, 2006.

[74] E. Laukkanen, M. Paasivaara, and T. Arvonen. Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study. In *AGILE*, 2015.

[75] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. Debugging Revisited. In *ESEM*, 2013.

[76] M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistä. The highways and country roads to continuous deployment. *IEEE Software*, 2015.

[77] M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32, 2015.

[78] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[79] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG)*, 5(2):141, 1986.

[80] A. Miller. A Hundred Days of Continuous Integration. In *AGILE*, 2008.

[81] Oren Mishali, Yael Dubinsky, and Shmuel Katz. The TDD-guide training and guidance tool for test-driven development. In *Agile Processes in Software Engineering and Extreme Programming*, pages 63–72. Springer, 2008.

[82] Kivanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *FSE*, 2013.

[83] Tamara Munzner. *Visualization Analysis and Design.* CRC Press, 2014.

[84] Kıvanç Muşlu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. Transition from Centralized to Decentralized Version Control Systems. In *ICSE*, 2014.

[85] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Preventing Data Errors with Continuous Testing. In *ISSTA*, 2015.

[86] H. H. Olsson, H. Alahyari, and J. Bosch. Climbing the stairway to heaven – a mulitiple-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*, Sept 2012.

[87] Version One. 10th annual state of Agile development survey. `https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf`, 2016.

[88] Version One. 10th Annual State of Agile Development Survey. 2016.

[89] Shaun Phillips, Thomas Zimmermann, and Christian Bird. Understanding and Improving Software Build Teams. In *ICSE*, 2014.

[90] Gerald V. Post and Albert Kagan. Evaluating information security tradeoffs: Restricting access can interfere with user tasks. *Computers & Security*, May 2007.

[91] Puppet and DevOps Research and Assessments (DORA). 2016 state of DevOps Report. `https://puppet.com/system/files/2016-06/2016%20State%20of%20DevOps%20Report_0.pdf`, 2016.

[92] Puppet and DevOps Research and Assessments (DORA). 2016 State of DevOps Report. 2016.

[93] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, 1999.

[94] David Saff and Michael D. Ernst. Continuous testing in Eclipse. In *ICSE*, 2005.

[95] Johnny Saldaña. *The coding manual for qualitative researchers.* Sage, 2015.

[96] Eddie Antonio Santos and Abram Hindle. Judging a Commit by Its Cover. In *MSR*, 2016.

[97] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *ICSE*, 2016.

[98] G. Schermann, J. Cito, P. Leitner, and H. C. Gall. Towards quality gates in continuous delivery and deployment. In *ICPC*, 2016.

[99] Irving Seidman. *Interviewing as Qualitative Research.* Teachers College Press, 2006.

[100] Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors. *Guide to Advanced Empirical Software Engineering.* 2008.

[101] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving developer participation rates in surveys. In *CHASE*, 2013.

[102] Hema Srikanth, Mikaela Cashman, and Myra B. Cohen. Test case prioritization of build acceptance tests for an enterprise cloud application. *Journal of Systems and Software*, 119, 2016.

[103] Mark Staples, Liming Zhu, and John Grundy. Continuous Validation for Data Analytics Systems. In *ICSE*, 2016.

[104] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. Overcoming Open Source Project Entry Barriers with a Portal for Newcomers. In *ICSE*, 2016.

[105] Sean Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.*, pages 369–374. IEEE, 2009.

[106] Megan Sumrell. From waterfall to agile-how does a qa team transition. In *Proceedings of AGILE*, 2007.

[107] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How Do Software Engineers Understand Code Changes? In *FSE*, 2012.

[108] Testing at the speed and scale of Google, Jun 2011. `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`.

[109] Tools for continuous integration at Google scale, October 2011. `http://www.youtube.com/watch?v=b52aXZ2yi08`.

[110] Edward R Tufte and PR Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics press, 1983.

[111] Akond Ashfaque Ur Rahman and Laurie Williams. Security Practices in DevOps. In *HotSos*, 2016.

[112] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *FSE*, 2015.

[113] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *FSE*, 2015.

[114] T. Vos, P. Tonella, W. Prasetya, P. M. Kruse, A. Bagnato, M. Harman, and O. Shehory. FITTEST. In *CSMR-WCRE*, 2014.

[115] Yihong Wang and Hakan Erdogmus. The role of process measurement in test-driven development. In *XP/Agile Universe '04*, 2004.

[116] Christian Wege. *Automated support for process assessment in Test-Driven Development*. Dissertation, Universitat Tubingen, 2004.

[117] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing with Over-designed Configuration in System Software. In *ESEC/FSE*, 2015.

[118] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. Continuous test suite augmentation in software product lines. In *SPLC*, 2013.

[119] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[120] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *STVR*, 22(2):67–120, 2012.

[121] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[122] Shurui Zhou, Jafar M. Al-Kofahi, Tien N. Nguyen, Christian Kästner, and Sarah Nadi. Extracting configuration knowledge from build files with symbolic analysis. In *RELENG*, 2015.