Bachelor Thesis

October 10, 2018

Investigating Continuous Delivery Practices and their Effectiveness in Open Source Projects

Faruk Acibal

of Zürich, Switzerland (14-933-642)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

Investigating Continuous Delivery Practices and their Effectiveness in Open Source Projects

Faruk Acibal

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Faruk Acibal, faruk.acibal@uzh.ch
URL: https://github.com/acfaruk
Project period: 13.04.2018 - 13.10.2018

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I'd like to thank Dr. Sebastiano Panichella, who accompanied me with this thesis and showed me a whole lot about R and statistics. He also, together with Dr. Michael Hilton, reviewed my thesis a few times and gave me good pointers on how I could improve it. I'd also like to thank Prof. Dr. Harald C. Gall for supporting this thesis in the last months.

Abstract

Continuous Integration(CI) and Continuous Deployment(CD) is a heavily used tool in software development in open source as well as industrial environments. To understand the effectiveness and efficiency of this tool, we start off by defining a taxonomy of variables that directly or indirectly could influence the effectiveness/efficiency of CI/CD practices. By performing an extensive literature review, we extract around 77 variables from 42 sources and StackExchange posts. We then state possible theoretical effects between these variables.

We continue by performing an empirical study going in depth for the build failure rate as well as the build duration and how they are affected by other variables. We use the datasets provided by TravisTorrent as well as GHTorrent. Looking at over 1200 projects and more than 680'000 builds, we confirm older studies but also contribute new findings.

Our work should help identifying problematic CI/CD practices that could influence the CI/CD effectiveness. The taxonomy we defined, should help with many upcoming research questions regarding the efficient and effective use of CI/CD practices. With these results, CI/CD effectiveness could be heightened in industrial as well as open source environments by manually or even automatically inspecting these variables and warning the maintainers of projects if problematic instances of these variables are detected.

Zusammenfassung

Continous Integration(CI) und Continous Deployment(CD) sind sehr häufig benutzte Werkzeuge für die Software Entwicklung in Open Source als auch industriellen Umgebungen. Um die Effektivität als auch die Effizienz dieser Werkzeuge zu verstehen, fangen wir in unserer Arbeit damit an, eine Taxonomie von Variablen, welche auf direkter oder indirekter Weise diese Effektivität/-Effizienz beeinflussen könnte, zu erstellen. Durch eine extensive Literaturrecherche sammeln wir 77 Variablen aus 42 Quellen und StackExchange Fragen/Antworten. Danach legen wir mögliche Effekte zwischen den Variablen dar.

Im zweiten Teil führen wir eine empirische Studie aus, in welcher wir genauer in die Build Fehlerrate und die Build Dauer eingehen und uns genauer anschauen was für einen Effekt andere Variablen auf diese beiden haben. Mit über 1200 Projekten und mehr als 680'000 Builds bestätigen wir ältere Aussagen in der Literatur und stellen neue auf.

Unsere Arbeit sollte helfen, problematische CI/CD Praktiken, welche die Effektivität/Effizienz beinträchtigen, zu identifizieren. Die Taxonomie, welche wir definiert haben, sollte mit aufkommenden Fragen bezüglich effizienter und effektiver Nutzung von CI/CD helfen. Mit diesen Resultaten, könnte man in Open Source und industriellen Umgebungen die Effizienz und Effektivität von CI/CD Prozessen steigern indem man die Variablen manuell oder automatisiert im Blick behält und die Maintainer von Projekten auf problematische Instanzen hinweist.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper is structured into seven chapters. In chapter 1 we start off by defining the terms Continuous Integration and Continuous Deployment and their context, we state our motivation, research goals and questions and how we plan to accomplish our goals.

In chapter 2 we look at related work concerning CI/CD practices and tools. We look at the challenges that arise when adopting a CI/CD pipeline and also look further into tools and datasets that are available.

In chapter 3 we describe the approach taken when creating our taxonomy. We also show the final taxonomy in a network of variables and try to find ways to measure the various variables that we defined.

In chapter 4 we describe how we prepared the data for our analysis, how we performed it and the results we got.

In chapter 5 we discuss our findings and reference relevant work as well as look at further research that could be done.

We close with possible threads to validity in chapter 6 and then a short conclusion in chapter 7.

1.1 Context

Continuous Delivery (CD) is a software engineering practice which lets developers build and deliver incremental versions of a software system in a very short time period. The main goal being, that the software is releasable at any point in time[5]. CD is often paired with Continuous Integration (CI). Simply put, it allows many developers to integrate their changes quickly to the software project. By providing testing and static code-quality checking through automated processes, integration errors are found and dealt with quickly[48].

Moving to CD/CI caused industrial organizations to report huge gains in developer productivity, software quality, release frequency and customer satisfaction[5]. These advantages led to open source developers also adapting CD in their development processes. It is currently one of the most used practices in the field[5, 15].

CI does not come without challenges though. Especially introducing it in already established development environments makes it very hard, since CI is heavily based upon automation practices[5, 15]. A lot of research has already been done to look at these challenges. Looking at the barriers and needs of developers themselves[16]. Or the challenges that come up, when transitioning to CI[5, 31].

Even though CI has become a best practice of modern software development, there is almost no quantitative and qualitative research available for it[2, 3], particularly, a general applicable taxonomy of variables in the CI process and how they affect each other is not available. There is some research accessible on different kinds of variables affecting the CI/CD process. Most of the

research done looks mostly at the build failure rate of CD builds[2, 17, 20, 25, 29, 49, 51]. Other research might look at social attributes, like how many followers a developer might have [39], the time of day/week[1], the project size [12, 17, 39, 44, 52] or the project age [44, 46, 50, 52] and many more. Taking all these research papers into account, there seems to be a need for some sort of taxonomy for possible variables. A clear taxonomy on variables, that affect the CI/CD effectiveness could help researchers and developers by providing possible research directions and actions in real instanced CI/CD pipelines. This paper aims to create a taxonomy of variables, that can be studied qualitatively. We would like to find the variables that are the most important for effective CI/CD usage and how to maximize their effectiveness by finding and tweaking independent variables.

1.2 Motivation

The main goal for this thesis is to optimize the CI/CD processes. To achieve this goal it is required to have an overview of the different aspects and decisions one can make when instancing a CI/CD process. The next step would be to see the effects of these decisions and aspects on the effectiveness of the CI process. Basically finding out how different variables affect CI/CD. An example of this could be how the contribution size/churn (how much was changed in a single contribution) affects the Build Failure Rate[34]. Our goal is to find factors that directly influence the effectiveness of CI processes, how much they influence them, as well as defining how this effectiveness could be measured. (*e.g.*, reliability/efficiency etc.)

There are many aspects/decisions that affect the investigated CD and CI pipelines. For example, it was found that dynamic programming languages (e.g Ruby, Python) have more tests written and more build breakages compared to statically typed languages[2]. Another interesting finding was that commits that come from pull requests seem to have a smaller build failure rate than direct commits on the development branch[46]. A paper also looked at social attributes[39]. It indicates, that core team members have a better chance of not causing build failures. The author of a commit having a big number of followers on GitHub (a feature, where one can follow the activities of another developer), also seems to positively influence the chance of a successful build[39]. Recent research follows this trend of looking at different factors and how they affect various metrics. However, this loses sight of the big picture. Which factors are the most important? Which metrics define CI/CD effectiveness the best?

We want to shed some light on these possible factors and metrics. To achieve this, we performed an empirical investigation aimed at defining possible factors and metrics that influence and measure the effectiveness of CI/CD processes. We looked at recent research as well as StackOverflow posts to find as many as possible. After clustering and grouping them, the final product is a taxonomy of variables. Researchers can use it to find new possible research directions and developers can decide which factors and metrics are important for their CI instances. This would lead to a clear way of looking for priorities when trying to make a CI/CD process more effective.

A secondary goal of this thesis would be to strengthen the research that was already done about CI/CD. Replication of the findings in other papers is very important here. So where possible, we would mention the findings of other researchers and compare them with our own. For example, there seem to be conflicting findings about whether being a "casual" contributor has negative effects on Build Failure rate or not[34, 39]. While Soto et al. find that core contributors of a repository tend to have a bigger likelihood of generating successful builds [39], the results of Rebouças et al. however suggest that being a casual contributor does not necessarily result in a higher chance of failing builds.

1.3 Research questions

Generally speaking we want to look at effectiveness metrics of CI/CD pipelines and the factors that influence said metrics. When talking about effectiveness, we generally mean the duration and frequency in a CI/CD pipeline. How often can it be run? How fast do the developers get feedback from the pipeline? Does the pipeline help improving the overall software quality?

Out of practical and data reasons we will only look at open source software. Naturally, it is much easier to gather data for open source software. It should be mentioned however, that the build failure rate of CI/CD builds is influenced by whether a project is open source or industrial[49].

As mentioned before, there is currently no study that somehow defines and investigates all the possible variables/factors that could influence the effectiveness of a CI/CD pipeline. So we state our research question:

- *What factors impact effectiveness of Continuous Delivery and Integration practices?*

To answer this question we split it up:

- *RQ1: What (dependent) variables are used to measure effectiveness of Continuous Delivery and Integration practices?*
- *RQ2: Which (independent) variables/factors impact effectiveness of Continuous Delivery and Integration practices?*

So with our first research question we aim to find the actual variables and factors that *measure* how effective a CI/CD pipeline is. The second question then focuses on variables / factors that *influence* the ones from RQ1. We do this by looking at already available research as well as conducting our own empirical investigation. Answering these questions means having an effective taxonomy of variables and factors that influence the CI/CD process and their rankings. With this taxonomy one could prioritize or improve specific variable values for CI/CD decisions and practices, providing automated software solutions.

We try to answer these questions in various steps. In section 3.1.1, we gather different factors that could be considered dependent or influencing ones as reported in recent research papers. We also look through StackOverflow questions and answers to complete our variable list in section 3.1.2. Finally merging and grouping them according to the phases and concepts of CI/CD we try to find the impacting relationships between the found variables by looking at the literature again. At the same time, we also mention possible relations between the variables, that could be further investigated.

1.4 Main Findings

Apart from our newly created taxonomy on CI/CD effectiveness variables, we have also some results from our empirical study. Looking at the project level and numerical variables, according to our results, the amount of builds, the amount of jobs, the average (source) files modified, the average active team size, the amount of commits, the amount of star watchers, the average lines of production code as well as the average amount of asserts per 1000 lines of code affect the build duration. The build failure rate however seems only affected by the duration of builds in our results. Looking at non numeric variables at the project level, the choice of Java or Ruby in programming language affects both the build failure rate as well as the build duration. While, whether or not a project belongs to an organization and the "Cloud Readiness" of it seems to only

affect the build duration of said project. Looking at the build level, branches with the keyword "release" in them have larger build times than those that do not. Branches with the "feature" keyword seem to have a 8% difference in build failure rate to the branches without that keyword in it. A detailed overview of the findings can be seen in chapter 5.

Chapter 2

Related Work

2.1 Continuous Delivery and Integration Practices

Fowler and Foemmel show very early how Continuous Integration has many benefits and what the general practices are. They mention how it reduces the risk involved when integrating chances, because before it, how long an integration might take was always an uncertainty. It is always clear what works and what does not with CI. In their paper, they also mention how it reduces the amount of bugs and increases the deployment frequency[10].

In the context of a specific software project developed by Microsoft, Miller shows how Continuous Integration affected it. Gathering the causes for build breakage, estimating the overhead of using CI and then comparing it to an estimate of an alternative process resulted in the conclusion, that CI made their work 40% more cost effective, even though the code quality stayed the same[29].

Another case-study was made by Stolberg. His report explains how he implemented a CI pipeline in the context of introducing Agile Testing methods. Showing how he introduced the CI system by prototype and demonstration, but also mentioning that this is probably not the best way[45].

Kim et al. looked further into the problems that come with software projects that are composed of many components and their integrations. Calling it "integration hell" they try to solve it with an automated integration procedure, specifically made for these kind of projects that rely on many components. The result is a system called "Nightbird" which was applied with success on a project of their own, consisting of hundreds of packages[21].

Hilton et al. tried to find trade-offs that come up with using CI. The main trade-offs they listen are Speed/Certainty, Better Access/Information Security and lastly More Configuration/Greater ease of use. They also find that developers use CI to guarantee quality, consistency and viability across different environments. Even though it increases the time spend[16]. Hilton et al. even combined qualitative surveys and empirical data analysis. They find that CI is widely used and still growing in open source software. They also mention how the more popular repositories on GitHub are more likely to use CI. Their qualitative data also states, that the biggest reason for not using CI is lack of familiarity. Another interesting finding they have, is that the projects that have CI release twice as often as those without CI[15].

Vassallo et al. performed a study with 152 developers in a large financial organization. Confirming and also contradicting common beliefs with their results. For example a third of the respondents dedicate more than half of their time to testing activities[48]. Also led by Vassallo et al., they analyzed the differences between open source and industrial software, coming to the conclusion that failures in open source are mainly caused by unit testing and in industrial software mainly because of release preparation work[49].

The impacts of introducing CI to external factors were also studied. Gupta et al. looked for example on the effects on Developer attraction and retention in open source projects. Surprisingly they come to the conclusion, that introducing CI is accompanied by attraction and retention decreases. However they do not claim causality and clearly indicate that further research is necessary to get a clearer answer[14]. Rahman and Roy looked into the impact of CI on code reviews. Their findings suggest that passing builds actually encourage more participation in code reviews. Projects that have a higher frequency of builds are shown to have a steady level of reviewing in them[32].

Trying to model differences in industrial CI implementations, Ståhl and Bosch performed a literature review to find different implementations and interpretations of CI. Listing descriptive statements that are mentioned in the literature and how they are interpreted in all the research, they show that there was no consensus in most of these. Because of this, they propose a model to better understand CI processes[41]. Using this model, it was later also applied to industrial cases to further improve their understanding of CI implementations and maybe even create improvements for said implementations[43]. Also trying to improve the CI process, Gambi et al. looked at the CI processes running in the cloud. Creating preconfigured virtual machines and reusing them wherever possible to reduce setup activities[11].

Martensson et al. tried to create a model for impediments regarding CI and use it to enable more frequent integrations of software. The result was a model they called EMFIS. It allows a company to visualize which factors it needs to focus on to have more frequent integrations[26].

Beller et al. looked at open source software hosted on GitHub. They found, that the most important reason as to why builds fails was testing. Their work also concludes, that the programming language of the software project has a big influence on the number of written tests (*e.g.*, dynamic languages needing more vs. statically typed ones). Their findings also suggest that CI/CD systems provide a growth of more than 10% in the amount of failures caught. However, they also conclude that CI/CD is not a replacement for local tests[2]. Beller et al. also introduced TravisTorrent, a big data-set synthesized from TravisCI and GitHub[3]. Many researchers used this dataset to gather further insights into CI processes and implementations[1, 2, 12, 14, 17, 24, 34, 49, 51].

2.2 Challenges of Moving to Continuous Delivery Practices

There are various challenges to adopting CI/CD practices[4, 5, 6, 16, 22].

Generally speaking, challenges for adopting CI/CD systems can be categorized in two categories: Social Adoption challenges and Technical Adoption challenges[6]. Social challenges refer to challenges which are cultural, interpersonal or even psychological. While technical challenges are more about system/project problems that could arise when moving to CI/CD.

A general list of challenges, compiled from various sources can be seen in Figure 2.1. Olsson et al. performed a multiple-case study in which they tried to find barriers when transitioning to CD. Interviewing four software development companies, they gather key-barriers but also provide actions to combat these[31]. Claps et al. performed an explorative case study involving detailed interviews with developers in an organization that already adopted CD. They found and categorized 20 different challenges[6]. Another noteworthy contribution in this regard comes from Hilton et al., where they performed a qualitative study with the goal of finding barriers and needs of developers when applying CI. They come to the conclusion, that developers that use CI have various trade-offs that they need to evaluate. They name Assurance (Speed vs Certainty), Security (Better Access vs Information Security) and Flexibility (More Configuration vs Ease of Use) while also looking at implications for developers, tool builders and researchers[16].

While there are many challenges when adopting CI/CD, there are also some ideas on how to overcome some of these challenges. Chen for example provides six strategies to overcome the challenges associated with adopting CD. Some examples are starting with easy but important applications, creating/employing dedicated teams or selling CD as a "painkiller" to other employees[5]. In their paper, Claps et al. also mention possible mitigation strategies for each of their twenty identified challenges[6]. Olsson et al. also discuss the actions that the actual software development companies performed to mitigate their respective challenges[31].

Technical
Infrastructure
Domain Constraints
Deploying
Upgrades
Legacy Systems
CI Process
Product Quality
Testing
Dev. Environments
Plugins
Source Code Control
Changing DB Schemas
Assurance (Speed vs. Certainty)
Security (Access vs. Info. Security)
Flexibility (Config. vs. Ease of Use)

Social
Pressure
Processes (+Doc.)
Company wide effort
Shorten Cust. Feedback
Changing Team Roles
Product Marketing
Cust. Adoption
Team Coordination
Team Experience
Tech. Product Writing
Cust. Feature Discovery
Client Wishes
Resistance To Change
Dev. Trust/Confidence

Figure 2.1: Various Technical and Social Challenges when it comes to CI/CD. Gathered from various sources such as Chen [4, 5], Claps et al. [6], Hilton et al. [16], Leppanen et al. [22].

2.2.1 Social Adoption Challenges

Social challenges refer to cultural, interpersonal or even psychological issues. One example is the organization in the company[4].

Since releasing the software can touch many divisions of a company, one would have to first get the blessing of said divisions. Process challenges are also part of the problem[4]. If there are other processes in the company that interfere with the CI pipeline or stall it for example. Another big social challenge seems to be the resistance to change[22]. Since many people from different divisions have to work together to make it work. It is also possible that a specific client does not want faster release cycles or does not care about them[22].

Developer trust and confidence is another interesting challenge. If every change is going to be deployed, any lack of confidence in the software application is going to be strengthened and might cause pressure because of the continuous releases. The developers also risk their reputation with the clients, since broken deployments are usually not very welcomed by them[6, 22].

Process documentation can be another problem. If the documentation for the CD system is lacking, new programmers will have problems understanding the process. This is further strengthened by the lack of industry standards[6]. A more complete list of social challenges is available in Figure 2.1.

2.2.2 Technical Adoption Challenges

Naturally, technical challenges are also always a topic when it comes to complex CI systems[4].

There could be domain constraints. An example provided by Leppanen et al. mentions control systems for factories, where a software release might mean stopping the whole system for a day or so just to update. A setting like that can not apply instant updates because they need to be scheduled[22].

Legacy systems can also be a problem when adopting CI/CD. Since they do not have to be designed to be automatically tested. This means that potential problems would only arise when the systems are integrated and running[22]. Another problem arises with different environments like production and development environments. If the targeted environment is too different from the development one, many unseen problems could arise[22].

Naturally, the product quality is also something that needs to be good. Having constant deployments could allow for more bugs to slip to production compared to a more slow release schedule[6].

Most of the trade-offs by Hilton et al. can also be seen as technical challenges. Should a company focus more on speed or certainty, better access or information security and more configuration options or greater ease of use[16]? While ease of use could be seen as a social challenge, a point can also be made that it could be a technical challenge. A more complete list of technical challenges is available in Figure 2.1.

2.3 Available Tools and Datasets for Continuous Delivery

There are various tools available for implementing CI/CD as well as analyzing the data the usage of CI/CD generates. In a CI/CD system hosting has to be considered, automation of tests and processes as well as deployment. Besides the usual mainstream tools, there is also new research for improving CI/CD practices and processes.

2.3.1 Hosting

For self hosted solutions Jenkins is probably the most known.[1] Though getting meaningful data from different projects running their own Jenkins installations is pretty hard. For open source software there are a few hosting providers. There is for example TravisCI[2], AppVeyor[3] and CircleCI [4] They usually offer free services for open source projects. These services can be used to gather data of CI/CD pipelines of various projects. TravisCI for example has a public API, which can be used for gathering data about various public CI/CD pipelines. All these solutions can be tightly integrated with the GitHub ecosystem and configured to use the hooks from GitHub to trigger the builds once the repository gets a new commit or a new pull request.

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

2.3.2 Mainstream Tools

Also there are the general build, testing and automation tools for various programming languages which run on the CI/CD servers. These tools usually vary with different programming languages though. The most known build tools in the Java world for example would be Maven, Gradle and Ant. The usual testing suite would be JUnit. What Maven, Gradle and Ant do is basically defining different builds and targets and how to make and automate them. So there could be definitions for testing, dependency management, packaging, deployment etc. in these tools. These definitions are then performed on a server whenever a new commit or pull request is made. If they successfully complete, then the software is theoretically ready for release. A very known packaging and release method are for example Docker containers[5]. They allow their users to define the environments in which their software runs and easily make the environment reproducible allowing for more secure deployments.

Besides build tools there are also static code analyzers. These tools check the source code for possible bugs or style mistakes. Again there are various tools depending on the programming language here. Some examples are SonarQube, FxCop, JSLint and CheckStyle. Depending on how the pipeline is configured, it might ask the developers to fix all the warnings and errors that these tools output before marking a build as successful or allowing it to be deployed.

2.3.3 Datasets

In 2017 a big dataset called TravisTorrent was published[3]. This dataset is based on TravisCI data and was made to further increase the research done on CI/CD systems. It also uses parts of the GHTorrent dataset, which is more focused on the GitHub side of things. With it they released the different tools they used to create the dataset and analyze it. These tools are all available on GitHub.[6] They are:

- **TravisPoker:** Checks if a given repository is using TravisCI.
- **TravisHarvester:** Aggregates general statistics of TravisCI builds for a given repository.
- **Buildlog Analyzer:** Analyzes build logs of Ruby / Java builds.

Building upon this dataset, Gautam et al. cleaned and added new elements for their research. They augmented it with things like the amount of watchers or issues in a repository[12]. Madeyski and Kawalerowicz also build on top of this dataset. They extend it with file change level data to predict changes that could break builds in an CI environment[24]. Also augmenting the TravisTorrent dataset, Reboucas et al. included the committers' name and e-mail address and also cleaned it for their purposes as well[34]. Soto et al. also augmented it by creating a web scraper to again get the username as well as the related data from a GitHub user[39].

Using a completely different dataset from Google, Elbaum et al. look at how the testing phase can be made more cost-effective. They also release the dataset with a sample of 3.5 Million test suite execution results[9].

[5]https://www.docker.com/
[6]https://github.com/TestRoots/travistorrent-tools

2.3.4 Research Approaches and Prototypes

There is also a lot of research going on for new tools and methods that could be applied to CI/CD systems. Macho et al. mention how neglecting build maintenance is one of the most frequent reasons for build breakage. They try to combat this by creating a tool called **BuildMedic** which tries to automatically repair Maven builds that break because of dependency issues. Their results show how their tool could repair 54% of the builds that they looked at[23]. Also trying to fix errors automatically, Soto and Le Goues use a probabilistic model to localize and fix bugs in source code[38].

Trying to extend the CI process, Dösinger et al. try to create communicating CI servers to increase effectiveness of automated testing. By allowing different CI servers (for different dependencies of projects) to communicate and share their CI process results, they show how this increases the effectiveness of automated testing[8]. Also trying to enhance the CI process, de Campos et al. try to extend it with automated test generation. Using their new approach, they tested it on open source projects with remarkable results. Branch Coverage increased by 58%, while the time spent for test generation was reduced by 83%[7].

Elbaum et al. try to make CI more cost-effective by creating algorithms that decide the order and relevance of tests. They show their improvements with the dataset from Google mentioned in section 2.3.3[9].

Chapter 3

Approach

3.1 Taxonomy Definition Concerning the Variables Characterizing or Affecting the CD Processes and Practices

In this section we try to create a taxonomy of dependent and independent variables characterizing and/or affecting the effectiveness of the CI process. The general goal is to find the relevant factors or variables that are the most discussed in the literature, considered relevant among developers and associated to CI/CD efficiency and effectiveness.

This taxonomy is conceived by relying on two sources of information: **(i)** Recent and relevant related work concerning CD/CI processes and **(ii)** StackExchange/StackOverflow posts concerning CD/CI practices. The following chapters deal with these sources. By creating a separate Taxonomy in each step and combining it at the end we build our final taxonomy.

3.1.1 Recent and Relevant Related Work

Finding Recent and Relevant Research

The first step was to find recent and relevant work on CI/CD processes. For our research we used two sources to find relevant work: Google Scholar [1] and dblp, the computer science bibliography [2]. Google Scholar was used because of its power backed by the Google Search engine and dblp was added to have a second search engine independent from the first one.

The search terms used to find relevant papers were mainly: **"Continuous Integration"**, **"TravisCI"** and **"TravisTorrent"**. The "Continuous Integration" query in itself creates around 200 results in dblp and around 24 thousand in Google Scholar (Google Scholar required us to put Continuous Integration in between quotation marks to not allow search hits with just one word). We also used **Continuous Deployment** and **Continuous Delivery** for the search queries.

After using some of these results, additional keywords were added at the end of the used search terms to limit the results further. Some of these were for example: **"efficiency"**, **"empirical"** or **"build"**.

Papers were chosen by the basis of a simple question. "Does this paper discuss information about potential variables / factors that influence CI/CD practices?" In this way a total of around

[1] https://scholar.google.com/
[2] https://dblp.uni-trier.de

42 papers were selected. There were a few papers, that also looked at past research to find variables. An example is the EMFIS model, that was created from interviews and a literature review of 74 research papers and a few books[26]. Another big literature review was done to find the differences in industrial CI implementations[41]. These reviews helped in getting variables from older papers, so we could focus our search on newer, more recent papers. The final selection of papers and sources can be seen in appendix A.

Finding Variables

After gathering the papers, the next step was to distill a first list of variables/factors that can affect CI/CD practices. The general approach here was to go trough the papers manually and search for them and keeping track of the variables/factors found in a table. Any mention of a possible variable that somehow could affect the CI/CD effectiveness was recorded this way. This table contains the variable name, a little description about it and in which paper(s) it was mentioned.

A slight confusion might come up when seeing the developer and contributor variables. While the variables that mention contributors look more closely at the individual level during a contribution, the developer related variables are meant to be more of a general term, representing the developer team as a whole.

There is also a variable "Usage of CI" which in itself was necessary because there were many papers that looked at the benefits of using CI in the first place. Since we did not want to omit the many variables that came from these papers, we included this one as well. Generally speaking though, it is not in itself relevant for our research.

Variables directly referencing TravisTorrent columns *e.g.*, *tr_jobs* or *gh_pushed_at* were generally also merged with the other variables. The variables found can be seen in Table 3.1

Variable	Description	Mentioned In
Amount of Builds	Frequency of CI Builds	[1][44][28]
Amount of Contribution Types	How many PR commits, direct ones etc.	[52][44][50]
Amount of Contributors	How many people contributed to the project	[20][17][52][25][44][50][12][37]
Amount of Jobs	How many Jobs are executed in a build	[34][50][2]
Amount of PR's (accepted)	How many Pull Requests are made/accepted	[47][15][52][50]
Amount of Tests	How many tests the Project has.	[2][34][17][52][12][22]
Amount of Dependencies	How many dependencies does the project have	[21]
Amount of Hotfixes	How many fixes were quickly deployed	[37]
Build Failure Rate	Likelihood of a Build Failure	[51][2][46][20][34][17][33][39][12][37][15][16][40][36]
Build Failure Type	The reason for a Build Failure (Percentage Distribution)	[49][29]
Build History	Results of the latest builds	[33]
CI Server used	*e.g.,* Jenkins, TravisCI etc.	[44][15]
Code Owner	*e.g.,* Open Source, Industrial, Organization	[49]
Code/Software Quality	The Quality of the Code/Software (*e.g.,* Amount of Bugs)	[47][29][37][5][6][10][16][22][48]
Configuration Complexity	Complexitiy of the CI/CD system configuration	[50][16]
Contribution Complexity	Complexity of the change set	[33]
Contribution File Type	Files types that were changed	[33][12][15]
Contribution Size/Churn	Size of the change set	[34][33][52][39][44][12][37][20][24][17]
Contribution Type	PR / Direct Commit / Merge	[46][17][33]
Contribution Work Item	Feature/Bug etc.	[20][50]
Contributor: Amount of Followers	Follower count on GitHub	[39]
Contributor: Amount of Starred repositories	How many repositories the contributor has starred on GitHub	[39]
Contributor: Amount of Follows	How many people the contributor follows on GitHub	[39]
Contributor: Amount of Contributions	Amount of Contributions by that specific contributor	[39]
Contributor Casualty	How casual a contributor is (core member vs. casual contributor)	[34][39]
Contributor Commit Frequency	How often does the developer commit	[33]
Contributor Experience	Experience as Software Developer	[33][39][6]
Contributor Project Knowledge	How much they know about the whole system	[26]
Communication	Amount of communication between team members	[42][6]
Communication between CI servers	*e.g.,* Dependency projects communicate with main their status	[8]
Commit Message Entropy	"unusualness" of a commit message	[36]
Defect detection & localisation time	How long does it take until a defect is detected / localised	[8]
Developer Attraction / Retention	Amount of developers joining / staying in the project	[14]
Developer Productivity / Efficiency	Productivity / Efficiency of developers working on the project	[37][42][5][16][22][28]
Developer Motivation	Motivation of developers working on the project	[37][45][5]
Developer Education	Education level of developers about CI/CD	[37][15]
Developer Sentiment	Sentiments, emotions, mood and stress of a developer.	[40]
Development Time	Time spend by Humans	[9]
Duration of Builds	How long a build takes	[2][26][41][50][45][10][15][16]
Duration of Tests	How long the tests take	[2][22][28]
Environment	The Operating System, Runtimes etc.	[2][10]
Feedback Time / Cycle Time	The time untill the system or customers react(s) with meaningful output	[30][2][6][5][22][33][48]
Hardware	Amount / Power of Hardware used for the CI execution	[6]
Management Support	Whether or not CI/CD is supported by management	[37][6][22][31]
New Commits in Build	The new commits in the build since the last one	[17]
Notification Quality	The quality of notifications of CI/CD systems	[16]
PR Latency	Time for a PR to close	[52]
PR Work-flow	How the project handles Pull Requests	[33][39]
Preconfigured Testing VM's used	Using virtual machines with preconfigured testing environments	[11]
Programming Language	The Programming Language of the Project	[2][46][52][50][15]
Project Age / Maturity	Age & Maturity of the project	[46][52][44][50][37]
Project Architecture/Type	Loosely Coupled, Modular, Micro-service etc.	[26][41][44]
Project Organization	How the team is organized etc.	[26][41][4][5][6]
Project Popularity	How popular a project is (often measured with the amount of stars)	[15][16]
Project Size	Size of the project *e.g.,* commits or lines of code	[17][52][39][44][12]
Project Growth	Growth of the Project (Size)	[37][1]
Release Frequency	How often the project gets released	[15][21][5][6][16]
Stakeholder of Build	For whom the build was made (Which branch etc.) (*e.g.,* Dev/Business)	[20][17][15][22]
Team Localisation	Geo. distance between team members	[20]
Testing	Whether or not the build includes testing activities	[2][31]
Tests: Amount of Environments	Amount of environments in which the test are executed	[2]
Tests Failure Rate	How often the tests fail	[2]
Tests Coverage	How much % of the source-code is tested	[7][30] [28][48]
Tests Strategy (order/selection...)	*e.g.,* start with tests that fail often	[9][26][7][22][48]
Tests generation for classes	Whether or not Tests should be generated for certain classes	[7]
Tests: given time	How long specific classes need to be tested	[7]
Time & Date of Builds/Commits	When the commit or Build happened	[1][33][24][20]
Tools Used	*e.g.,* Maven, Gradle etc.	[17][26]
Tools integration	How easily new tools are integrated into a CI/CD pipeline	[16]
Usage of CI	Whether or not the project is using CI	[47][29][15][52][50][5][14][16][31]
Version Control System	*e.g.,* Git, Subversion etc.	[44]
Work Breakdown / Guidelines	Having Clear Guidelines how a commit should be etc.	[26][6]

Table 3.1: Variables Found in Recent and Relevant Related Work

3.1.2 StackExchange/StackOverflow Posts

Selecting Relevant Posts

We also looked at StackExchange for our variables. Using the search of the generalized StackExchange site, we were able to search through all variants of StackExchange like for example the Software Engineering one. Our goal was to find questions regarding the overall effectiveness of instanced CI/CD pipelines. By looking at the answers on such questions, we could gather new affecting independent factors and the questions themselves might also provide some affected dependent metrics.

The initial search term used here was "Continuous Integration". This resulted in 25'856 results which was not very helpful (14'009 if both words were put in between quotation marks). Questions like "MS Unit Continuous Integration with TFS?" or "Continuous Integration (CI) with Phabricator?" are listed when searching for this term. These questions were usually tailored to one specific technology or tool and also usually asked a specific question regarding them. We quickly realized that we had to further specialize our search terms. We tried again adding a few extra keywords to this search term: "efficiency", "build", "optimize" and "best practice". The "best practice" addition was the most effective one for finding relevant answers. It lowered the number of questions to 541 and had many relevant ones. The additional keywords and their results can be seen in Table 3.2

Added Term	Results	Useful
efficiency	36	No
build	7'664	No
optimize	113	No
best practice	541	Yes

Table 3.2: Search Terms Added to Main Search Query and Their Effect

Most of these keywords were used to find relevant questions. Finding them was more difficult than we expected, since most of the questions are related to a specific problem that people have and there are generally very few questions that ask for how they could generally improve their CI/CD implementations. All the results we found came from StackOverflow or the Software Engineering StackExchange variant. The questions were picked by identifying them as questions that address efficiency in CI/CD instances. The full list of questions we picked can be seen in Appendix B.

Finding Factors and Variables

The results of the StackExchange search were interesting because they showed a few variables that were not even considered in the previous research work but had actually big impact on real implemented CI/CD pipelines. A good example for this would be the Hardware and / or the usage of parallel computing for the builds. The hardware for example was only mentioned in one paper by Claps et al. where they mention it as a technical challenge for moving to CI. A good question we found was "Improving CI build time (.NET)"[3]. The author is part of a team which has a CI instance on the .NET stack. He asks for ways of improving the build time. The question received four answers with many elaborate ways to optimize a CI instance. Here the hardware is mentioned in two answers and parallelizing the build process was also mentioned twice in the

[3]https://stackoverflow.com/questions/8633313/improving-ci-build-time-net

answers and even as a possible considered solution in the question itself. Parallelizing the build process was not found in the earlier literature review.

Most of the StackExchange variables found used the general structure of the Project or the CI Pipeline as the variable to improve. Usually listing specific examples that could help in the implementations of the question authors. Naturally, filtering out general applicable variables was pretty difficult. An example for this would be the question "To Clean or not to Clean"[4] where the author asks whether it is considered "best practice" to always do a clean build in the CI pipeline. We called these variables "Job Structure" and "Project Architecture/Type" to make it more applicable to different ways this could be influenced. In our example including a clean before each build would be part of the Job Structure. Simply calling the specific change the variable here would be a waste for many more potential changes to how a build pipeline could be structured.

Naturally there were a few overlapping variables between the StackOverflow ones and the research as well. The amount of tests or the duration of builds to name a few. We tried to name the variables the same if we found variables that were already found in the previous step to help with the merging later on. The final variables can be seen in Table 3.3

Variable	Description
Amount of Build Targets	*e.g.*, Release / Debug
Amount of Dependencies	How many libraries etc. are used
Amount of Tests	How many tests the Project has.
Duration of Builds	How long do builds take?
Duration of Tests	How long do tests take?
Feedback Time	Time to get some response from the build
Hardware	The hardware that the CI Server runs on
Incremental Builds	Only compile what has changed
Job Structure	How the jobs in CI are ordered/organized
Project Architecture/Type	Loosely Coupled, Modular, Micro-service etc.
Test Coverage	How much of the code base is unit-tested?
Tests order/selection	*e.g.*, start with tests that fail often
Usage of static Code analysis	*e.g.*, checkstyle
Using Parallel Computing	Leveraging parallel computation of builds

Table 3.3: Variables Found in StackExchange/StackOverflow Posts

3.1.3 Merging And Grouping the Variables

Merging

The merging process in itself was not very complicated. Because we reused the variables found in the literature research merging both tables was as simple as combining one table with the other one and removing any duplicates. The StackExchange variables that were new were: Amount of Build Targets, Incremental Builds, Job Structure, Usage of static code analysis and Using Parallel Computing. These variables are all very practical and usually directly measurable from the environment or build. It is kind of surprising that some of these were not found in relevant research by us.

Grouping Variables in CI/CD Stages and Deciding Influencing Factors

The next step would be to decide whether a variable is a dependent metric or an independent factor. Our goal is to find variables that measure the efficiency of a CI/CD pipeline. Thus, we

[4]https://stackoverflow.com/questions/5812872/to-clean-or-not-to-clean

look at variables that are directly bound to the CI/CD pipeline duration and frequency. To do this, we look at the CI/CD pipeline that we defined (see Figure 3.1) and search for variables that directly map to the duration or efficiency of these stages.

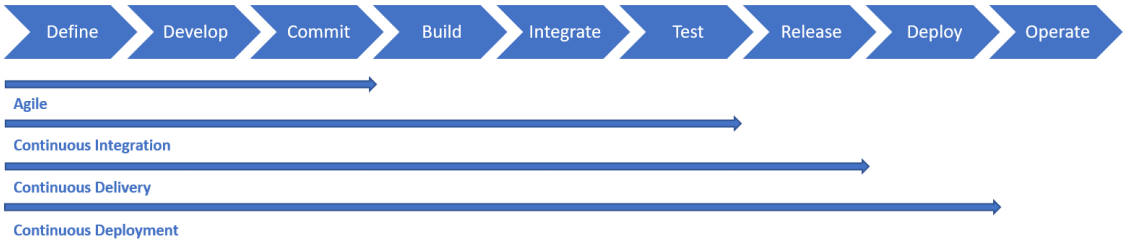

Figure 3.1: The CI pipeline stages. Note this is one possible way of defining a CI pipeline. We simply choose to use this one for our work.

The result of this grouping can be seen in Table 3.4. These variables are now the dependent ones that directly influence the CI/CD effectiveness.

The next step now is to find variables that could directly influence these. First, we went through the papers again and looked at the found influences of their research and second we made our own logical guesses for which variables might affect others. The resulting table contains references to the research if the influences are backed by it. The full table can be seen in Appendix C.

Stage	Variables	Explanation
Define	Developer Productivity	How efficiently can developers define what is needed for the project.
Develop	Developer Productivity, Development Time	How quickly and efficiently is the programming done?
Commit	Developer Productivity	How efficiently the developer writes the commit message etc.
Build	Build Failure Rate, Build Failure Type, Duration of Builds	How long do the builds take and how likely are they to fail?
Integrate	Build Failure Rate, Build Failure Type, Duration of Builds	The integration stage can be seen as an extended build stage.
Test	Test Failure Rate, Duration of Tests	How long do the tests take and how likely are they to fail?
Release	Release Frequency, Feedback Time	These variables define the efficiency of CI/CD more directly.
Deploy	Release Frequency, Feedback Time	These variables define the efficiency of CI/CD more directly.
Operate	-	-
Other	Project Size / Growth, Code/Software Quality	More general, Project Level Variables

Table 3.4: Grouping on CI stages

3.1.4 Visualizing the Taxonomy

Since the table in Appendix C is very big and full with duplicate variable names, we looked for an alternative way to visualize the result of our work. We decided to use the Graphviz[5] software suite to visualize our relationships as a directed Graph of nodes. We wrote a graphviz file which defines our variables as a graph where a connection is formed if a variable influences another one according to our taxonomy. In the resulting image, the dependent metrics are red square nodes where as all other variables are simply rectangular with different colors to more easily follow the lines. The connections that refer to actual research results are a little bit more bold and red. The final image can be seen in Figure 3.2. The nodes that differ because of the CI stage have their label start with [DEF], [COM] or [DEV] standing for, Define, Commit and the Development stage respectively.

[5]https://www.graphviz.org

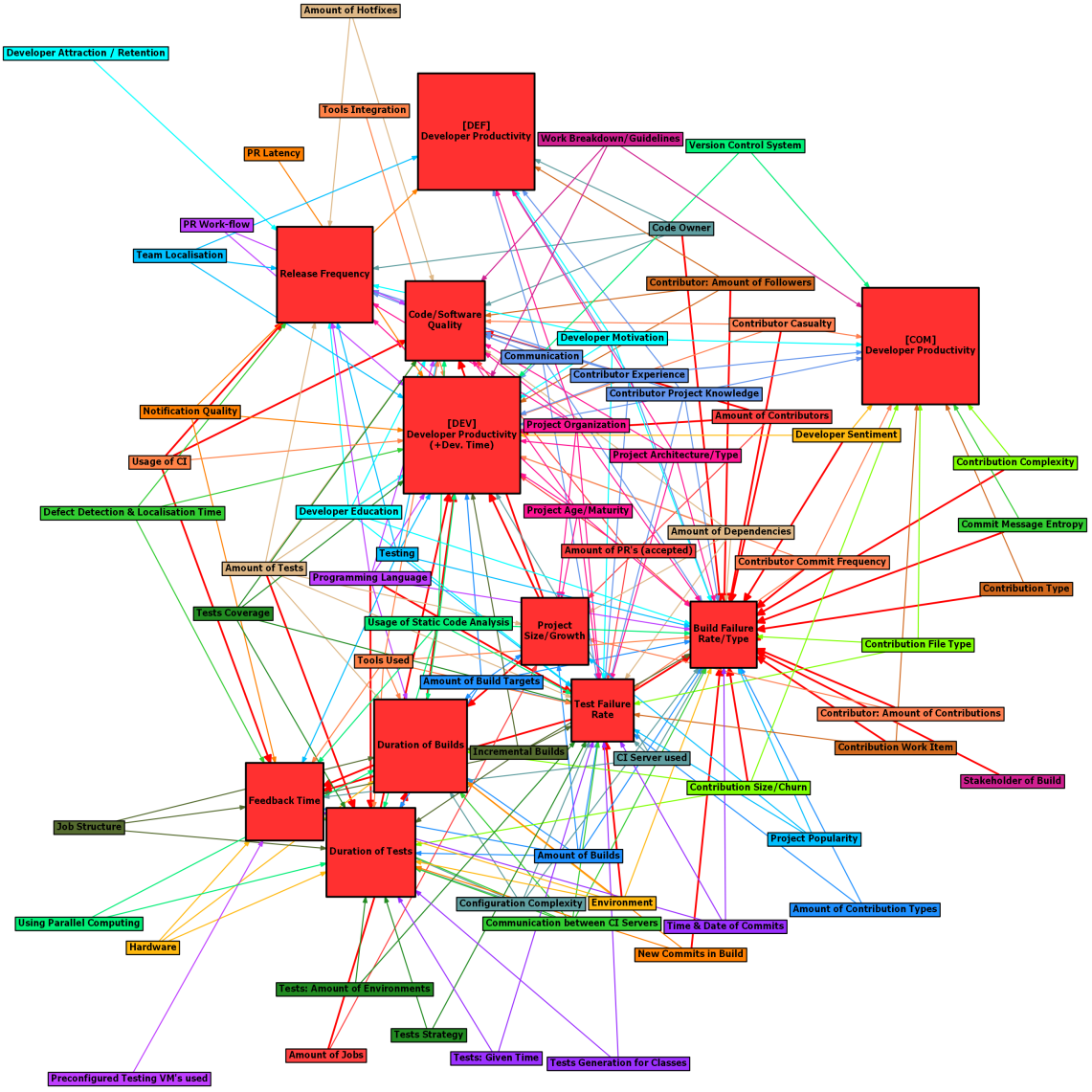

Figure 3.2: Final Taxonomy

3.2 Measuring the Variables

Before we could start with the analysis, it was necessary to define how we could measure the various variables that we found. For this, we went trough each and every variable and looked at the literature, the TravisTorrent dataset, the GHTorrent dataset [13], the GitHub API and the Travis API on how we could gather the required data. The result can be seen in Table 3.5.

Some of the variables are not really easy to measure. Some are only really available trough mining the repository or conducting interviews with the maintainers/company. Some of these were mined for the generation of the TravisTorrent Dataset though. After having a rough idea on how we could measure the variables, it was in our best interest to limit our variable set for the

sake of not creating too much work for little return value.

After creating a little script to scrape some simple data from GitHub, we quickly realized that the API limit that GitHub has, was very limiting. It only allowed 5000 requests per hour, which would cause a little bit of extra headache when gathering huge amounts of data. The solution was to use the dataset provided by GHTorrent instead [13]. The GHTorrent dataset is a really big ongoing project where API keys from many volunteers are used to gather the data. It also allowed us to get the data up until a specific date to better synchronize the data with the TravisTorrent one. This way our TravisTorrent data was up until 8.2.2017[6] and the GHTorrent set we choose had data up until 1.3.2017[7]. This allowed for very time equal data only separated by a month.

We finally decided to limit our variables to the ones we could somehow gather from these two datasets because of the mentioned problem with the GitHub API as well as the fact that most of the values we could get from the TravisCi API were already readily available in TravisTorrent. How we prepared and worked with the datasets is described in chapter 4.

[6] https://travistorrent.testroots.org/dumps/travistorrent_8_2_2017.sql.gz
[7] http://ghtorrent-downloads.ewi.tudelft.nl/mysql/mysql-2017-03-01.tar.gz

Variable	How to Measure
Amount of Builds	T: builds endpoint
Amount of Build Targets	T: builds endpoint: branch and/or tag
Amount of Contribution Types	TT: gh_is_pr or git_merged_with, GH: Commits or PR endpoint
Amount of Contributors	TT: gh_team_size, GH: Repository: Collaborators endpoint (They do not mean the same thing)
Amount of Jobs	T: Jobs endpoint, TT: tr_jobs
Amount of PR's (accepted)	GH: PR endpoint: state
Amount of Tests	TT: tr_log_num_tests_run, tr_test_cases_per_kloc
Amount of Dependencies	Configuration Files
Amount of Hotfixes	Quick successive builds? TT: gh_build_started_at or gh_pushed_at
Build Failure Rate	TT: tr_status, tr_log_status, T: build endpoint: state
Build Failure Type	TT: tr_status, tr_log_status, tr_log_bool_tests_failed, T: build endpoint: state
Build History	See Build Failure Rate and Type
CI Server used	Configuration Files
Code Owner	TT: git_branch, T: Branches endpoint, GH: Repository: branches endpoint
Code/Software Quality	GH: Issues endpoint
Configuration Complexity	Total count of lines in Configuration Files
Contribution Complexity	TT: All git_diff_* fields, gh_num_commits_in_push
Contribution File Type	Look at the Changes directly
Contribution Size/Churn	TT: All git_diff_* fields, gh_num_commits_in_push
Contribution Type	TT: gh_is_pr or git_merged_with, GH: Commits or PR endpoint
Contribution Work Item	Mineable through Bug/Issue Number in Commit / PR
Contributor: Amount of Followers	GH: User: Followers endpoint
Contributor: Amount of Starred repositories	GH: User endpoint: starred_url
Contributor: Amount of Follows	GH: User endpoint: following_url
Contributor: Amount of Contributions	GH: Commits endpoint with author parameter
Contributor Casualty	GH: Commits endpoint with author parameter
Contributor Commit Frequency	GH: Commits endpoint with author parameter
Contributor Experience	Amount of Followers, Public Data about the Author
Contributor Project Knowledge	Amount of Commits in Project, Public Data about the Author
Communication	Interviews
Communication between CI servers	Configuration Files
Commit Message Entropy	GH: Commits endpoint, T: commit endpoint,
Defect detection & localisation time	Time between successful / failed builds as approximation (see Build Failure Rate and Time/Date of Builds)
Developer Attraction / Retention	Active Contributors
Developer Productivity / Efficiency	Contributed Code/Feature in xyz time
Developer Motivation	Interviews / Commit Messages?

Variable	How to Measure
Developer Education	Interviews / Commit Messages? Public Data about the Author
Developer Sentiment	Observation of Authors
Development Time	Work Hours of Developers or Time between builds. (See Time/Date of Builds)
Duration of Builds	TT: tr_log_builddduration, tr_duration, T: build endpoint: duration
Duration of Tests	TT: tr_log_testduration
Environment	T: Environment Variables endpoint
Feedback Time / Cycle Time	See Build Duration
Hardware	Public Information about CI Server, or own servers
Incremental Builds	Mine from build logs. T: Log endpoint
Job Structure	T: Jobs Endpoint, TT: tr_jobs
Management Support	Interviews
New Commits in Build	TT: git_all_built_commits
Notification Quality	Interviews
PR Latency	GH: Pull Request endpoint: merged_at, closed_at and created_at
PR Work-flow	Contribution Guide File in Repository
Preconfigured Testing VM's used	Mining / Interview
Programming Language	GH: Repository: Languages endpoint, TT: gh_lang, tr_log_lan
Project Age / Maturity	GH: Repository endpoint: created_at
Project Architecture/Type	Interview or Code Inspection
Project Organization	Interview
Project Popularity	GH: Repository endpoint: stargazers_count
Project Size	TT: gh_sloc
Project Growth	TT: gh_sloc, maybe the gh_diff_* entries
Release Frequency	GH: Repository: Releases endpoint
Stakeholder of Build	TT: git_branch, T: Branch and Build endpoints
Team Localisation	Interviews
Testing	TT: tr_log_bool_tests_ran, tr_log_frameworks
Tests: Amount of Environments	Inspection
Tests Failure Rate	TT: tr_log_bool_tests_ran, tr_log_num_tests_*
Tests Coverage	Inspection
Tests Strategy (order/selection...)	Inspection / Interviews
Tests generation for classes	Inspection / Interviews
Tests: given time	Inspection / Interviews
Time & Date of Builds/Commits	TT: gh_build_started_at, gh_pushed_at, tr_duration, T: Build endpoint: started_at, finished_at, duration, GH: Repositories: Commit endpoint
Tools Used	TT: tr_log_frameworks, Inspection
Tools integration	Inspection / Interviews
Usage of CI	Configuration Files available
Usage of static code analysis	Configuration Files available
Using Parallel Computing	Does the CI server permit it? Is it not turned off?
Version Control System	Inspection / Interview
Work Breakdown / Guidelines	Inspection / Interview

Table 3.5: Ideas on how the various variables could be measured. TT = (TravisTorrent [3]) CI Server/Tool + Log inspection of builds), T = (Travis APIv3) CI Server/Tool, GH = (GitHub APIv3 or GHTorrent [13]) Version Control System

Chapter 4

Analysis

4.1 Preparing the Data for Analysis

We started of by creating a local Sql server on our machine. We imported both datasets, GHTorrent(1.3.2017) as well as TravisTorrent(8.2.2017) into our local server.

To prepare the data for analysis we decided on three levels of data to prepare. The first level would be a project level, the second a build level and the third the Travis job level. We separate the build and job level, because the TravisTorrent dataset only contained job entries, which lead to many duplicate rows for specific builds (*e.g.*, a build contains many jobs but once one of them fails, all of the jobs in the same build share the same build result in the dataset). Upon realizing this, we made a new table that aggregated these jobs to their specific builds and only had the data that was duplicated for each job in a build. This step prevented us from using duplicate data for our analysis. More about this can be read in section 5.1.1.

Since we only focused on the TravisTorrent data, we reduced the GHTorrent data to the Projects that were available in the TravisTorrent data. This was done with a few simple Sql queries where we simply created new tables where the project names or other relevant fields were members of the TravisTorrent dataset. The first Sql query that can be used to create the smaller project table is called `projects_sub.sql`. After reducing the GHTorrent set, the next step was to create the various levels of data we decided on. First the Project level. Using fields from the GHTorrent dataset we combined them with the fields from the TravisTorrent set that we aggregated differently. For example calculating the average build duration of all the builds in a project. This gave us a very broad table on the project level. This data level can be created by first running the `ghtorrent_projects.sql` which generates fields by only looking at the GHTorrent data, and then executing `combined_projects.sql` which then combines data from both datasets to create the whole projects table.

After this, the `travistorrent_builds.sql` and `travistorrent_jobs.sql` files can be executed for the other two data levels. The first script aggregates the job entries from TravisTorrent to reflect the build entries and the last one simply chooses the fields that are really uniquely tied to the jobs. These fields were mostly the log analyzer results for the different jobs that a build consisted of.

The result were three database tables with their specific data: Project, Build and Job. The exact process on preparing the dataset can be followed in the accompanying git repository of this thesis.

4.2 Creating Hypotheses

As mentioned earlier we split our data into three parts: Project level, Build level and Travis Job level. From these three parts we use the first two to create our hypothesis, that we want to prove/disprove. In the following, whenever a hypothesis does not come with a mention of a research paper, we can assume that our findings on that hypothesis are new and not yet researched.

4.2.1 Project Level

For the project level we focus on two dependent variables: The **build failure rate** and the **duration of builds**. The reason for this choice is reliability of our data. Both of these terms are very reliable and exact measures of the variables and not really measured differently because of tool or pipeline changes. We could look at the test failure rates since the jobs do have log analyzers that did analyze some of the builds. But these results are mined and do not have the same confidence of truth as the values coming directly from the APIs of GitHub and Travis. Also they would be not uniformly created because of the different pipelines used in all these projects. Future Research could look more into the other dependent variables by maybe focusing on one build tool. The failure rate is computed by looking at the builds table, counting how often the state was passed and finally dividing it by the amount of builds the project had. The duration of builds can be aggregated as an average value from all the builds that a project had. We look at the following variables and state our hypotheses:

Amount of Builds: This variable is easily extracted from the builds table. Our hypotheses would be that projects that have more builds would have a bigger build failure rate, simply because they feel more "experimental" with their builds. The duration of builds might also get smaller when a project has many builds. Simply because once a build might not take too long, one might be inclined to, again, do more "experimental" builds. This variable might be more interesting if corrected with the Project Size.

Duration of Builds: While this variable is a dependent one, it can act as an independent one for the build failure rate. The natural thing to think would be that the build failure rate increases with the duration of a build, simply because there is more time for things to go wrong.

Amount of Jobs: This is also an average that can be simply calculated from the respective field in the builds table. The natural thing to think would be that more jobs in a build would increase the build failure rate as well as the build duration, simply because there is more to do and therefore more that can go wrong.

Contribution Size/Churn: This variable can be calculated from various fields. Our assumption here is, that once the average contribution size/churn increases the build failure rate should also increase, simply because there are then more changes that could break a build. The build durations might also increase with a bigger average contribution size/churn. The reasoning for that is that, projects that have bigger changes in their builds should also have longer build times. Islam and Zibran find in their research, that the amount of source code churn influences statistically significantly the amount of failed builds [17].

Amount of Contributors: This variable can be aggregated as the maximal or the average of the active team size field on TravisTorrent. Our hypothesis would be, that with an increase in contributors, the build failure rate could increase, because of the struggles of integrating all the changes of various team members. It might also correlate with the build duration, as a big team might indicate a big project and therefore longer build times. But there should not be a causality there. According to Kerzazi et al. the amount of contributors on a branch has significant impact on the build failure rate [20]. Islam and Zibran however does not find any correlation between team size and build results [17].

Project Size: This can be the lines of production code, the amount of commits or the amount of stars watchers on a project. The amount of contributors could also be used as a measure here. Naturally a bigger project should have longer build duration times, since there would be more code to build, test and integrate. The build failure rate might increase as well, because of all the code that already exists and could break tests because of changes. Ståhl et al. look at how the size of software projects influences the continuity of continuous integration. The interviews they conducted make them believe, that it is plausible that the size does influence the continuity [44]. Islam and Zibran however finds that the size of projects does not correlate with the build results [17].

Test density: This variable can be found in the various fields of TravisTorrent that look at the tests or asserts. These fields are computed by the various analyzers of the TravisTorrent tools. We allow not as reliable data for our independent variables, but have to mention it here that this variable is indeed the result of an automatic extraction and not an exact value. If a specific project has a bigger density it should probably have also longer build times as well as a bigger build failure rate. It could also be, that projects with denser test suites indicate a bigger focus on quality and therefore less build breakage.

Contributor Casualty: This is based on a field called *gh_by_core_team_member*. Calculating the average per project gives us a number between 0 and 1 indicating how many people are core contributors. If the casualty is high, there might be more build breakage compared to a low casualty in contributors. Soto et al. found in their research that core team members have a higher chance to pass the build tests [39]. However, Rebouças et al. found in their work, that there is not a big enough difference between casual and non-casual contributors when it comes to build failures [34].

Programming Language: Based on the findings of Beller et al., dynamic languages like Ruby might cause a higher build breakage compared to a statically typed language like Java [2].

Code Owner: This can be seen from a field called *owner_company* from the GHTorrent set. It describes whether or not a project is from an organization or not. A project that does not belong to any organization might have bigger build failure rates. The work of Vassallo et al. is here worthy of mention. While it does not look into the build failure rate, it looks at the distribution of build failure types in an industrial organization and open source software [49].

Cloud Readiness: This is a new variable that we want to look into. It could be considered to be part of the Project Architecture / Type of a project. We measured this simply by searching through GitHub with a simple script. The search was done like this: The full project name in between parenthesis to not allow parts of a project name to be a search hit and then the keyword "cloud" or "microarchitecture" after it. If one of the searches had any results at all, we considered the project cloud ready. Luckily, Github considers the Dockerfile syntax as its own language, so that way we could include all the projects that had the "Dockerfile" language listed, as a cloud ready project. This increased our number of cloud ready projects from around 40 to around 100. Our hypothesis for cloud ready projects is that they might have better build failure rates compared to other projects and shorter build times because of the Microarchitecture style many of these projects might follow.

4.2.2 Build Level

The build level also focuses, for the same reasons as above on both the **build failure rate** and the **duration of builds**. This time however we look at the specific builds, meaning looking at the actual build status and the actual duration of that particular build. No averages this time.

Stakeholder of Build: This one can be measured by looking at the branch the build came from. Depending on the branch organization, builds on the master should have less build failures than temporal branches. If there are release branches, those would have probably the least amount

of build failures. According to Kerzazi et al. the stakeholder role is one of the most important variables that influence build breakage [20]. Also, Hilton et al. find that builds on the master branch are more likely to pass than on other branches, aligning with our own hypothesis [15].

4.3 Addressing the Hypotheses

Here, we look at the two data levels of Projects and Builds we have to address our hypotheses. First, we focus on the Project level in section 4.3.1 where we first look at numeric variables like the amount of builds, the amount of jobs or the lines of code and see how they affect the dependent variables of build failure rate and build duration. After the numeric variables we look at the remaining variables like *e.g.*, the programming language of a project and how those affect the failure rate and duration of builds. In section 4.3.2 we look at the build level. We also start with numeric ones and conclude with non-numeric ones as well there.

It is important to note, that we only look at the results here. Further discussion about possible explanations of these results and possible directions for future research is available in chapter 5.

4.3.1 Project Level

Influence of Numeric Variables on Build Failure Rate and Build Duration

We start our analysis with the project level and the numeric variables. We wrote a R script, that loops through each numeric variable to create all the relevant plots and statistics that we need for our two dependent variables. With a combination of the correlation coefficient, box plots for the various quantile ranges and calculation of the Wilcox test as well as Cliff's Delta we get a good understanding of the correlation between two variables. The correlation coefficient gives us a general idea on how much these variables are correlated. Then looking at the box plots of the distributions, we can visualize possible effects that are maybe only there after a certain value or in specific ranges. The Wilcox test and the effect size, then tell us whether or not our observations from the box plot are statistically backed or not. We can also first look at those values and then see if we can see their effects in the box plots respectively. The R script does the following steps for each combination of Dependent (Build Failure Rate/Build Duration) and numerical independent variable:

1. **Calculate the correlation coefficient between the dependent and independent value with the Kendall method[19].** This method was used because our values were not normal distributed and therefore the Pearson method was not recommended. The correlation values can be seen in table 4.1.

2. **Calculate the quantile ranges of the independent variables.** We calculate the quantile values of 0, 25, 50, 75 and 100% and then create sets of dependent-values within those quantiles. (All values between the 0 and 25 quantiles, 25 and 50 etc.) This splits our dataset in four equal sized parts. The quantiles can also be referenced in table 4.1.

3. **Create box-plots of these ranges for the distribution of the dependent variable.** These created plots show the distribution of the dependent variable on the different quantile ranges of the independent variable. If the boxes get higher for example, it could point to an increase of the dependent variable when the independent one is increased. The following box plots are all calculated this way until described differently.

4. **For each pair of quantile ranges calculate the following values:**

(a) **Wilcox test:** This is done with the `wilcox.test` function in the `stats` package. Here the two sets of independent variable values are compared. The resulting p-value is an indication on whether the two sets have a statistically significant variation. Because we automated the process we always calculate the p-value for the hypothesis that the first distribution is "less" than the second one. When the p-value is below 0.05, we can conclude that the sets are not identical and that indeed, the first one is "less" than the second one. If the value is however bigger than 0.95, then it would be the same as a p-value below 0.05 when calculating for "greater". This means that once the value is above 0.95, we can deduce that the difference of distributions is also significant but for the "greater" hypothesis. So any value bigger than 0.95 or lower than 0.05 is statically significant in our tests.

(b) **Cliff's Delta:** This value is calculated with the `cliff.delta` function in the `effsize` package. While the Wilcox p-value shows us whether or not the distributions are identical, cliff's delta is a measure on how much they differ. According to Romano and D. Kromrey, a value of $|d| < 0.147$ means "negligible", $|d| < 0.33$ is a "small" difference, $|d| < 0.474$ "medium" and otherwise it is considered "large"[35].

Both, the Wilcox p-value as well as Cliff's Delta are listed in table 4.2. The columns start with "c" or "w" respectively denoting the statistic and then the two numbers denote the quantile sets that are being compared. As an example, "c-1-2" would be the Cliff's Delta of the first quantile range (0-25%) and the second one (25-50%).

Looking at the correlation values in table 4.1, the only significant value we get from these, is the one between the average build duration and the average amount of jobs per build. With a value of 0.43 it is somewhere between a weak and a moderate uphill relationship. Looking at the box plot, that was calculated as mentioned earlier, for these variables in fig. 4.1, we can see the last group has a significant increase in the distribution. This seems to suggest that after a certain amount of jobs, the build duration seems to drastically increase. A closer look at the data sheds some light on the results.

As we can see in table 4.1, the 75% quantile of average build jobs in a build is 6.27 while the 100% quantile has a value of 93.43. This suggests that most projects have less than 7 jobs in a build and the ones that have more can have many more, resulting in longer build durations.

The Wilcox test between the 50%-75% and the 75%-100% ranges gives an extremely small value of 3.27E-44 indicating a statistically significant change. The Cliff's Delta is -0.65, meaning the difference is large. The values for Cliff's delta and the Wilcox test can be found in table 4.2.

While most of the correlations do not have statistically significant results, we can still check the other statistics for our hypotheses. We will look at each of our hypotheses now.

Variable	Correlation to Build Failure Rate	Correlation to Avg. Build Duration	Q_0	$Q_{0.25}$	$Q_{0.5}$	$Q_{0.75}$	Q_1
builds	-0.02	0.19	2.00	142.00	261.00	531.00	19447.00
avg_sloc	0.00	0.13	11.00	733.00	2198.51	8141.25	1003708.41
avg_jobs_per_build	0.16	0.43	1.00	1.39	3.04	6.27	93.43
avg_build_duration	0.25	1.00	10.07	224.12	449.56	990.88	32845.43
avg_active_team_size	-0.03	0.18	0.62	2.96	4.71	8.21	225.87
stars_watchers	-0.03	0.10	52.00	172.00	440.00	1111.00	37304.00
commits	-0.01	0.22	1.00	304.00	629.00	1434.00	69846.00
avg_by_core_team_member	-0.07	-0.06	0.00	0.84	0.92	0.97	1.00
avg_diff_src_churn	-0.01	0.04	2.84	30.06	54.95	126.97	16411.01
avg_diff_files_added	0.02	0.02	0.00	0.30	0.63	1.27	179.32
avg_diff_files_deleted	0.08	0.07	0.00	0.05	0.17	0.47	56.78
avg_diff_files_modified	0.03	0.14	0.80	2.56	3.37	5.09	162.57
avg_diff_src_files	0.03	0.12	0.09	2.44	3.30	5.30	272.03
avg_diff_doc_files	0.03	0.09	0.00	0.00	0.02	0.21	42.60
avg_diff_other_files	0.01	0.05	0.00	0.26	0.55	1.30	95.52
avg_test_cases_per_kloc	0.04	0.08	0.00	9.43	45.74	105.61	2078.73
avg_asserts_cases_per_kloc	0.03	0.11	0.00	38.91	109.81	217.19	8266.76

Table 4.1: Correlation Values between the numerical Project level variables along with the quantile values of their respective distributions.

dep/indep	w-1-2	w-1-3	w-1-4	w-2-3	w-2-4	w-3-4	c-1-2	c-1-3	c-1-4	c-2-3	c-2-4	c-3-4
build_failure_rate / builds	0.96	0.97	0.91	0.55	0.29	0.25	0.08	0.09	0.06	0.01	-0.03	-0.03
build_failure_rate / avg_jobs_per_build	0.59	0.01	0.00	0.00	0.00	0.00	0.01	-0.12	-0.30	-0.15	-0.36	-0.22
build_failure_rate / avg_build_duration	0.00	0.00	0.00	0.04	0.00	0.00	-0.25	-0.33	-0.52	-0.08	-0.34	-0.27
build_failure_rate / avg_active_team_size	0.41	0.65	0.95	0.72	0.96	0.88	-0.01	0.02	0.07	0.03	0.08	0.05
build_failure_rate / avg_sloc	0.16	0.19	0.72	0.54	0.93	0.89	-0.05	-0.04	0.03	0.01	0.07	0.06
build_failure_rate / stars_watchers	0.57	0.86	0.75	0.79	0.68	0.39	0.01	0.05	0.03	0.04	0.02	-0.01
build_failure_rate / commits	0.88	0.93	0.46	0.63	0.12	0.07	0.06	0.07	-0.00	0.02	-0.06	-0.07
build_failure_rate / avg_by_core_team_member	0.78	0.99	1.00	0.95	0.98	0.71	0.04	0.11	0.13	0.08	0.10	0.03
build_failure_rate / avg_diff_src_churn	0.11	0.67	0.54	0.95	0.84	0.33	-0.06	0.02	0.00	0.08	0.05	-0.02
build_failure_rate / avg_diff_files_added	0.88	0.99	0.12	0.91	0.01	0.00	0.05	0.11	-0.06	0.06	-0.11	-0.16
build_failure_rate / avg_diff_files_deleted	0.01	0.00	0.00	0.27	0.02	0.08	-0.10	-0.13	-0.18	-0.03	-0.10	-0.06
build_failure_rate / avg_diff_files_modified	0.07	0.19	0.04	0.71	0.33	0.17	-0.07	-0.04	-0.08	0.03	-0.02	-0.04
build_failure_rate / avg_diff_src_files	0.66	0.30	0.08	0.15	0.03	0.15	0.02	-0.02	-0.07	-0.05	-0.09	-0.05
build_failure_rate / avg_diff_doc_files	0.72	0.73	0.09	0.45	0.08	0.04	0.04	0.03	-0.06	-0.01	-0.09	-0.08
build_failure_rate / avg_diff_other_files	0.59	0.37	0.47	0.29	0.39	0.57	0.01	-0.02	-0.00	-0.03	-0.01	0.01
build_failure_rate / avg_test_cases_per_kloc	0.01	0.25	0.00	0.98	0.22	0.00	-0.12	-0.03	-0.17	0.10	-0.04	-0.15
build_failure_rate / avg_asserts_cases_per_kloc	0.24	0.35	0.04	0.67	0.15	0.06	-0.03	-0.02	-0.08	0.02	-0.05	-0.07
avg_build_duration / builds	0.03	0.00	0.00	0.00	0.00	0.00	-0.09	-0.24	-0.41	-0.16	-0.35	-0.18
avg_build_duration / avg_jobs_per_build	0.00	0.00	0.00	0.00	0.00	0.00	-0.24	-0.40	-0.88	-0.14	-0.68	-0.65
avg_build_duration / avg_build_duration	0.00	0.00	0.00	0.00	0.00	0.00	-1.00	-1.00	-1.00	-1.00	-1.00	-1.00
avg_build_duration / avg_active_team_size	0.02	0.00	0.00	0.00	0.00	0.00	-0.09	-0.23	-0.38	-0.14	-0.30	-0.17
avg_build_duration / avg_sloc	0.19	0.00	0.00	0.01	0.00	0.01	-0.04	-0.16	-0.26	-0.12	-0.22	-0.11
avg_build_duration / stars_watchers	0.33	0.10	0.00	0.19	0.00	0.00	-0.02	-0.06	-0.25	-0.04	-0.23	-0.19
avg_build_duration / commits	0.00	0.00	0.00	0.03	0.00	0.00	-0.15	-0.24	-0.51	-0.09	-0.38	-0.30
avg_build_duration / avg_by_core_team_member	0.90	1.00	0.99	0.93	0.89	0.43	0.06	0.13	0.12	0.07	0.06	-0.01
avg_build_duration / avg_diff_src_churn	0.09	0.35	0.02	0.79	0.22	0.05	-0.06	-0.02	-0.10	0.04	-0.04	-0.08
avg_build_duration / avg_diff_files_added	0.19	0.92	0.10	0.98	0.30	0.00	-0.04	0.07	-0.06	0.10	-0.02	-0.12
avg_build_duration / avg_diff_files_deleted	0.00	0.00	0.00	0.82	0.75	0.40	-0.21	-0.17	-0.19	0.04	0.03	-0.01
avg_build_duration / avg_diff_files_modified	0.00	0.00	0.00	0.01	0.00	0.04	-0.14	-0.25	-0.33	-0.11	-0.19	-0.08
avg_build_duration / avg_diff_src_files	0.03	0.00	0.00	0.01	0.00	0.08	-0.09	-0.19	-0.26	-0.10	-0.17	-0.07
avg_build_duration / avg_diff_doc_files	0.00	0.00	0.00	0.99	0.99	0.45	-0.33	-0.16	-0.17	0.16	0.16	-0.01
avg_build_duration / avg_diff_other_files	0.00	0.01	0.01	0.72	0.77	0.54	-0.14	-0.11	-0.11	0.03	0.03	0.00
avg_build_duration / avg_test_cases_per_kloc	0.01	0.08	0.00	0.87	0.01	0.00	-0.11	-0.07	-0.23	0.05	-0.12	-0.17
avg_build_duration / avg_asserts_cases_per_kloc	0.00	0.00	0.00	0.24	0.01	0.04	-0.15	-0.18	-0.26	-0.03	-0.11	-0.08

Table 4.2: Wilcox and cliffs delta values of the numerical Project level variables. The column names start with w or c for wilcox or cliff separately. The following two numbers are the two quantile ranges that are being compared.

Amount of Builds: The correlation values here are rather small, with 0.19 for the average build duration it is very weakly influenced. Looking at the Wilcox p-values and effect sizes in table 4.2, the non existent relationship against the build failure rate is strengthened, while the relationship to the average build duration is significant. The Cliff's Delta between the first and last quantile range is -0.41, so it is very close to a large difference but still considered medium. The other comparisons show a small difference with the exception of 1-2 being negligible and 2-4 being medium.

> There seems to be a small to medium influence between the amount of builds a project has in its lifetime and the duration of said builds. The more there are, the longer a build seems to take. Our data does not suggest an influence between the amount of builds and their build failure rate however.

Duration of Builds: The correlation coefficient between the duration of builds and the build failure rate is around 0.25 as can be seen in table 4.1. This indicates a close to weak uphill relationship. Meaning that the failure rate seems to slightly increase with longer build times. Looking at the Wilcox p-values in table 4.2, the differences between the distributions is significant while the Cliff's Delta goes up to -0.52 for the difference between the first and last quantile distribution. This indicates a large difference. Most of the other comparisons have a small delta except for the 2-3 comparison with a negligible one. The gradual increase of the build failure rate can also be seen in the box plot in fig. 4.1. The difference between the first and last set is clearly visible there.

> Our data suggests that the average build duration of a project has a small to medium effect on the build failure rate of it. The longer the builds get the higher the failure rate increases.

Amount of Jobs: This variable was already discussed a bit earlier. Looking at the influence to the build failure rate in table 4.2, almost all comparisons have a statistically significant p-value.

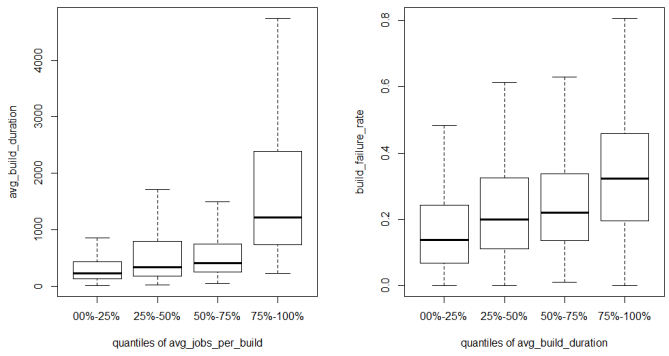

Figure 4.1: Box plots showing the distribution of the average build duration (left) / build failure rate (right) grouped into the quantile ranges of the average jobs per build (left) and average build duration (right).

The only exception is the comparison of 1-2. Looking at the effect size, the biggest one seems to be the comparison 2-4 with a value of -0.36, a medium change. The effect size for the 1-2 comparison is, even though it is minuscule, positive. Compared to all the other negative values, this can not be ignored. There might be a relationship here, but one would probably need better evidence to make a statement.

The influence on the build duration however is, as already discussed pretty significant. The effect size for the ranges 1-4 is -0.88, that is considered a large effect size.

> Our data suggests that the amount of jobs has a large influence on the duration of builds. In particular once the amount of jobs reaches seven or more, there seems to be a big increase in build times. The amount of jobs seem to not have a confident connection to build failure rate however.

Contribution Size/Churn: Here, we look at all the variables in table 4.2 which have a "diff" in their name, these variables represent the average size of the changes made to a project. We can clearly see that for the build failure rate, almost all of them have p-values that do not imply a significant change. There are however some exceptions:

1. **Average Files Added:** We can observe the comparisons of 1-3, 2-4 and 3-4 having significant differences. The effect sizes are 0.11, -0.11 and -0.16, considered negligible and small respectively. Looking at the actual quantile values in table 4.1, the 75% quantile is 1.27 and the 100% one 179.32. The small increase can be considered a result of this big variation in the last quantile range. Also the switch between "greater" and "less" significant changes does not help for a clear deduction.

2. **Average Files Deleted:** According to our calculated p-values the comparisons, 1-2, 1-3, 1-4 and 2-4 are all have significant changes. Looking at the effect sizes, the only comparison with a non-negligible value is the comparison 1-4 with -0.18, this is considered small. Since it is the comparison of the two most distant distributions and also just barely considered small, making conclusions out of this is not warranted.

Both the Average Source Files changed and Average Documentation Files changed have both a single significant change in one of the comparisons, however both of their respective effect sizes

are both negligible. Combining this with the rather insignificant correlation values in table 4.1, we can assume no significant effects here.

Looking at the build duration correlation values, they seem to look a little better, but are still mostly insignificant. They do not cross the 0.2 let alone the 0.3 barrier. There are again many comparisons with an insignificant p-value. There are more exceptions this time:

1. **Average number of lines of production code changed (avg_diff_src_churn):** This variable has only one significant comparison, namely 1-4. With a Cliff's Delta of -0.1 however the difference is negligible.

2. **Average Files Added:** This variable also only has two comparison with a significant change in the comparisons, namely 2-3 and 3-4. Again though, with an effect size of 0.10 and -0.12 it can only be considered negligible.

3. **Average Filed Deleted:** This variable has more significant changes. 1-2, 1-3 and 1-4 are all significantly different when compared. Their effect sizes are respectively -0.21, -0.17 and -0.19. All of these values are small. All of these comparisons tell us that the first quantile range seems to differ very much from the rest of them. Looking at the quantile values in table 4.1 we can see that the 25% quantile is only 0.05. This is a small number when looking at files deleted. Because we are looking at project averages, it seems like most of the projects get a value below 1 for this variable. A possible explanation for our observations might be, that projects which rarely delete files in their commits have slightly smaller build times. Since the other quantiles are so close, this explanation needs more research.

4. **Average Files Modified:** For this variable, every comparison has significant changes. The biggest effect size is in the 1-4 comparison with a value of -0.33, this indicates a medium difference. Looking at the other effect sizes, we can assume that there is indeed some influence from the average files modified here.

5. **Average Source Files Changed (avg_diff_src_files):** Looking at the values in table 4.1, all the comparisons have significant changes except for the 3-4 one. The effect size for the 1-4 comparison is with -0.26 in the small category. We can conclude a small influence here.

6. **Average Documentation Files Changed (avg_diff_doc_files):** Looking at the quantiles tells us again, that most of the projects do not have more than one documentation file change in the average. The 75% quantile is 0.21. Meaning if there are interesting results here, they should be between the last quantile range and the others. Looking at the box plots in fig. 4.2, seems to suggest a sudden increase in the second range. The p-values and effect sizes confirm the weird out-lier second range. We withdraw from making any conclusions for this variable.

7. **Average Other Files Changed (avg_diff_doc_files):** All the effect sizes here are negligible as can be seen in table 4.2.

It is important to know, that these values are averages over their respective projects. This is different from looking at the effect of a big contribution vs. a small one. We looked at the averages of a project and then at their influences on the variables.

Our data does not suggest a relationship between the average amount of contribution size/churn and the build failure rate. Our observations show however a small to medium positive relation between the average files modified and the average build duration. Also a small positive relation between the average source files modified and the build duration was also observed.

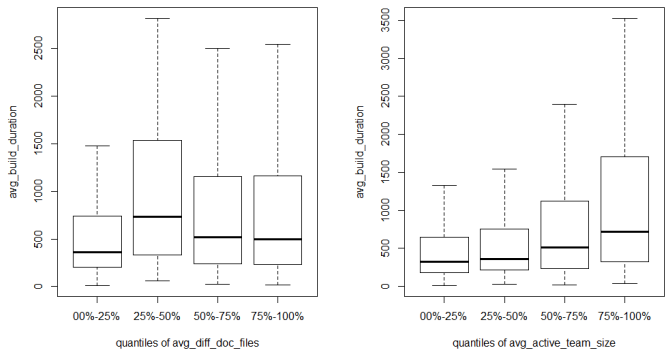

Figure 4.2: Box plots showing the distribution of the average build duration grouped into the quantile ranges of the average documentation files changed (left) and the average active team size (right).

Amount of Contributors: Here we look at the variable "avg_active_team_size", it tells us the average amount of developers in a 3 month period. Looking at the correlation to the build failure rate in table 4.1, we find a correlation value of -0.03. In table 4.2, the 2-4 comparison is significant while the 1-4 one can also be considered close to significance, however the effect sizes are both negligible. This invalidates any influences between the amount of contributors and the build failure rate our data could suggest. Looking at the build duration however, with a correlation value of 0.18 there could be more to the relationship of team size and build duration. Looking at the p-values in table 4.2, they are all smaller than 0.05 indicating statistically significant differences and their biggest effect size of -0.38 in the 1-4 comparison is also classified as medium change. Looking at fig. 4.2 makes this slight increase visible. We can also see how the increase starts being noticeable after the first two sets of quantile ranges. Looking at the p-values again confirms this observation. The 1-2 comparison has the lowest effect size with -0.09. Looking at the quantile values in table 4.1, we can see the value for the 50% quantile is 4.71 meaning that with around five or more developers the effects seem to get bigger.

> While our data does not imply a relationship with the build failure rate, there seems to be a small to medium influence to the average build duration from the average active team size. As it increases, the build times seem to increase as well. However there seems to be close to no relationship when the team size stays below five developers. Once there are more, the effects of the relation seem to start.

Project Size: Here we look at three relevant variables: Lines of production code (avg_sloc), the amount of commits (commits) or the amount of stars watchers on a project (stars_watchers). These variables are not necessarily synonymous to the project size, but could be seen as closely related. Their correlation values, when compared to the build failure rate are insignificant. A quick look at the p-values also shows not a single value being significant.

When looking at the average build duration however, we get a few more significant values:

1. **Average Lines of production code (avg_sloc):** All of the p-values show us significant results except for the 1-2 comparison. The effect sizes go up to -0.26 for the 1-4 comparison, meaning there is a small difference between the first and last quantile range. We can conclude a small relationship between the average build duration and average lines of production code.

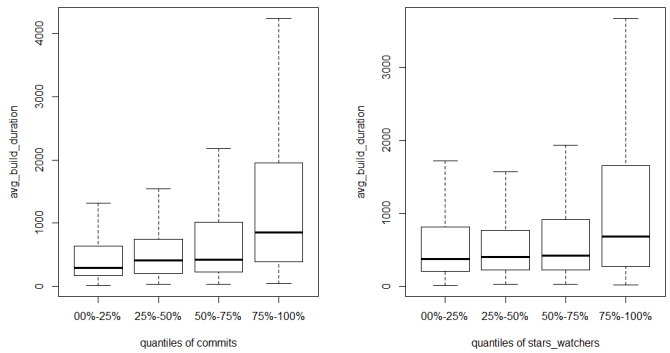

Figure 4.3: Box plots showing the distribution of the average build duration grouped into the quantile ranges of the amount of commits in the project (left) and the amount of stars_watchers (right).

2. **Amount of commits (commits):** All the p-values are below 0.05 and are therefore indicating statistically significant changes in the compared distributions here. The biggest effect size is for the 1-4 comparison again with -0.51, a large value. Looking at the effect sizes, the ones which are compared with the fourth quantile range seem the highest. Looking at the box plot in fig. 4.3 confirms our observations. The last distribution is visibly higher than the others. Looking at the quantile values in table 4.1, we can see that the 75% quantile for commits is 1434 while the 100% one is 69846. Meaning there is a much bigger range of values in the latest quantile range. Looking at the correlation value again in table 4.1, we see a value of 0.22, it is one of the bigger values in that table. We can conclude a small to medium increase up to 1500 commits and after that a large increase in build duration (together with the commits).

3. **Amount of stars watchers on a project (stars_watchers):** Only the 1-4, 2-4 and 3-4 comparisons show a significant p-value here. When looking at the effect size, the biggest one is again 1-4 with a value of -0.25, this value is considered small. We can clearly see the pattern where only the comparisons with range 4 are significant. A look at the box plot in fig. 4.3 confirms our observations. The last range shows a big difference here. The quantile values for $Q_{0.75}$ and Q_1 are 1111 and 37304 respectively, showing us again a very big range of values in the last range. We can conclude a small relationship between stars_watchers and the build duration with a bigger increase after a value of 1111 is reached.

The average build duration is affected by the amount of commits, the amount of stars watchers as well as the average lines of production code. For the average lines of production code, our data suggests a small relationship. For the amount of commits, we can conclude a small to medium increase up to 1500 commits and after that a large increase in build duration (with a bigger range of possible number of commits). A small relationship is also found for the stars watchers and the average build duration. It also seems to get way bigger after a certain point, namely 1111 here (again with a bigger range of possible values after that point). However it is important to note that the measure of commits in a project might be difficult to study as a variable as further discussed in section 5.1.1.

Test density: The correlation values for the asserts or test cases density in table 4.1 are all

pretty low. The biggest one seems to be the assert cases density when compared to the average build duration with a value of 0.1. First we look at the build failure rate:

Looking at the p-values in table 4.2, we can see that the asserts variable has just one significant value in the 1-4 comparison, but with an effect size of -0.08 it can be considered negligible. The test cases variable shows a significant p-value for the 1-2, 1-4, 2-3 and 3-4 comparisons. Their effect sizes are -0.12, -0.17, 0.10 and -0.15 respectively. These values are mostly small and negligible though. Also one of the comparisons is positive, giving us a hint on an unstable relationship here. Therefore, we withhold from making any observations here.

Next, we look at the average build duration:

Looking at table 4.2 again, we see the test cases variable has the same problem of alternating between positive and negative effect sizes. The assert variable however has generally significant p-values with the exception of the 2-3 comparison. The effect sizes for 1-2, 3-4 and 1-4 are -0.15, -0.08 and -0.26 specifically. Indicating a small relationship between the average amount of asserts per 1000 lines of code and the build duration.

> Our observations suggest a small relationship between the average amount of asserts per 1000 lines of code and the average build duration. Our data does not suggest a relationship with the build failure rate though.

Contributor Casualty: When looking at the build failure rate in table 4.2, the comparisons 1-3, 1-4 and 2-4 show significant differences. However their effect sizes are all negligible. The values for the average build duration follow a similar pattern, the 1-3 and 1-4 comparisons have significant changes, but also negligible effect sizes.

> Our data does not suggest a relationship between the contributor casualty and the build duration or the build failure rate.

Influence of Non-Numeric Variables on Build Failure Rate and Build Duration

We now look at the non-numeric values in the project level. We also wrote a little R script for these variables, but the script is less sophisticated, as it just calculates the values for each variable separately, because each of these variables were sometimes handled a little bit differently. We mention this in the following paragraphs if we needed to change something. The general idea is the following:

1. **Split the data into two sets:** Here we split the data according to the variable we want to observe. For example if we look at the programming language, we split it to Java and Ruby sets.
2. **Make both sets the same size:** Use the size of the smaller set and randomly sample the same amount from the bigger one. This way both sets are the same size for the next steps. One of our variable sets was too small with only around 100 items in it. In that case we simply duplicated the set a few times and then sampled the new size from the other set.
3. **Create the box plot:** Make a box plot for the created sets with the build failure rate and average build duration as the y-axis.
4. **Calculate the statistics:** Calculate the p-value from the Wilcox test and the effect size with Cliff's Delta as before. These values are mentioned in the texts of the specific variables and not listed in a table.

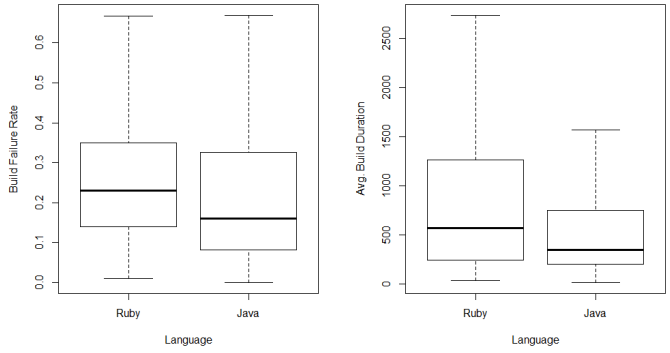

Figure 4.4: Box plots for the used language in projects showing distributions of the build failure rate as well as the average build duration.

Programming Language: While our dataset includes many different languages, there are only two with meaningful amounts of data: Java and Ruby. Because of this, we limit ourselves to these two languages. Looking at fig. 4.4 a first observation can be done. It looks like Ruby projects seem to have a bigger build failure rate as well as a longer average build duration.

Calculating the p-values with the Wilcox test gives us a value of 3.634e-06 for the build failure rate and 1.228e-06 for the build duration, meaning a statistically significant difference. The Cliff's Delta value for the build failure rate is 0.19 and therefore small. The build duration one is with 0.20 also small. We can conclude that there is a small influence between the choice of Java / Ruby and the build failure rate / build duration.

> There seems to be a small relationship between using Java/Ruby and the Build Failure Rate/Average Build duration. Ruby seems to have a bigger chance at build failures as well as longer average build times.

Code Owner: Here we looked at whether or not a project belonged to a organization on GitHub. Looking at fig. 4.5 we can observe, that there does not seem to be difference in build failure rate, however the average build duration seems to be lower once the project belongs to an organization. The p-value of the Wilcox test for the build failure rate gives us 0.60, confirming our observations of no big change in build failure rate. However the p-value for the average build duration, is close to 0 and therefore small enough to confirm a statistically significant change in the distributions. The effect size is with -0.14 negligible. This seems to indicate a very small decrease in average build duration for projects tied to organizations.

> Our data suggests a very small relationship between build duration and the project belonging to an organization. It seems like projects that are in a GitHub organization have smaller build times than the ones that are not part of an organization. There does not seem to be a relationship to the build failure rate.

Cloud Readiness: This time around we only had 110 projects that were marked "Cloud Ready" by us. Because of this, we replicated the set three times to create a set of the size 330. Then we sampled again randomly the same amount of non cloud ready rows. After that we continued normally as described earlier. Looking at fig. 4.5 we can observe a slightly lower build failure rate

distribution for cloud ready projects and also a bigger average build duration. The p-value of the Wilcox test for the build failure rate is with 0.003341 significant and the one for the build duration with 0.0001774 as well. The effect size for the build failure rate is -0.122011 and negligible while the one for the build duration is 0.1606612 and therefore considered small.

> Our data suggests a very small relationship between whether or not a project is "Cloud ready" and its build failure rate. Cloud ready projects seem to have a slightly smaller build failure rate. We also observed a small relationship between "cloud readiness" and the build duration. "Cloud Ready" projects seem to have slightly longer build times.

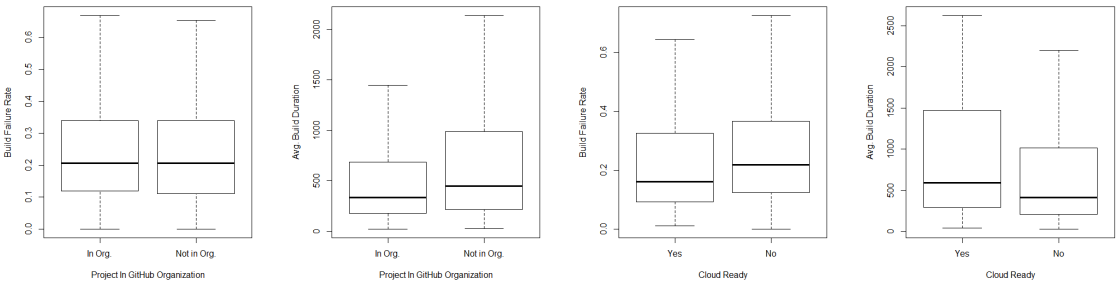

Figure 4.5: Box plots for whether the project is in an organization or not and whether the project is cloud ready or not, showing distributions of the build failure rate as well as the average build duration.

4.3.2 Build Level

For the build level, we only look at non-numeric values. Specifically the branch name of the specific builds. When calculating the statistics we only look at the build duration, because the build failure rate is a measure of many builds not a single one. To not leave it unobserved though, we calculate the build failure rate of the specific variables that we look at and then compare them to see if there are big differences. The exact steps we take are:

1. **Split the data into two sets:** Here we split the data according to the branch name we want to look at. For example all branches with the name "master" in one set, the others in to a second one.

2. **Make both sets the same size:** Use the size of the smaller set and randomly sample the same amount from the bigger one. This way both sets are the same size for the next steps.

3. **Create the box plot:** Make a box plot for the created sets with the average build duration as the y-axis.

4. **Calculate the statistics:** Calculate the p-value from the Wilcox test and the effect size with Cliff's Delta as before.

5. **Calculate the build failure rate for both sets:** This is done by counting the amount of "passed" builds divided by the amount of rows (builds) in the set.

The results of the calculations can be seen in table 4.3.

Stakeholder of Build: We look at the build data and select specific branch names or identifiers in them to compare their effect on the build duration and the build failure rate. An overview of the different approaches we did can be seen in fig. 4.6. The first plot shows the distributions on the master branch vs. any non-master branches. The second plot looks at branches that have the term "feature" in them. The third one at branches that have numbers in them, usually indicating specific version branches. The last plot looks at the branches that have the term "release" in them. The last and second seem the most visually interesting from a first look.

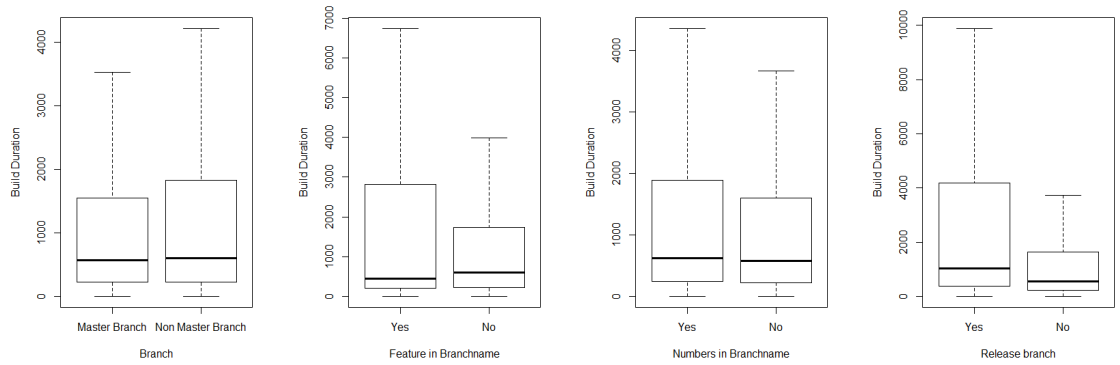

Figure 4.6: Box plots for various ways to group the builds based on branch name.

Looking at table 4.3 quickly shows us that the only relevant comparison was the branches which have the name "release" in them. So the moment a branch has the keyword "release" in it, it seems like the build duration rises a bit. With an effect size of 0.22 between the branches that have the keyword and the ones that do not, the difference is considered small. The rest of the effect sizes in table 4.3 are all negligible and therefore not looked at further. Looking at the build failure rates of the comparisons leaves most of them at a difference of about 5%. The biggest one is the comparison of branches with the keyword feature in them and the ones that do not: The feature branches have a failure rate of 0.36 while the ones that do not have the feature keyword in them have a failure rate of 0.27. The difference is with around 8% the biggest one. We consider the other 5% ones relevant as well.

Branch Filter	n	Wilcox p-value	Cliff's-delta	Failure Rate when True	Failure Rate when False
Is Master branch	214831	< 2.2e-16	-0.03	0.25	0.31
"feature" in branch name	7167	0.48	0.00	0.36	0.27
numbers in branch name	85435	< 2.2e-16	0.04	0.31	0.26
"release" in branch name	6424	< 2.2e-16	0.22	0.25	0.27

Table 4.3: Statistics for the various filters that were applied to the branches. The p-value is always the smaller one applicable and the two last columns are for the build failure rate within the filtered distributions or their counterparts that do not conform to the filter.

According to our data, branches with the keyword "release" in them seem to have larger build times than the ones that do not. The influence is small. The biggest difference in build failure rate was between branches with the "feature" keyword vs. the ones without with a difference of around 8%, where the feature branches failed more often. The master branch builds seem to fail less with a difference of around 6% while branches with numbers in them failed more with a difference of around 5%.

Chapter 5

Discussion and Future Work

In this section, we discuss the main findings our work has contributed, which older research was confirmed/contradicted and the possibilities for future research.

5.1 Addressing the Research Questions

Our first research question, *"RQ1: What (dependent) variables are used to measure effectiveness of Continuous Delivery and Integration practices?"* was tackled by first creating a big exhaustive list of variables that are related to the Continuous Delivery processes and pipelines. We created this list by looking at StackExchange questions as well as lots of earlier research. We skimmed these works for mentions of variables that could somehow influence the CI/CD processes. The result of this extensive search was a list full of variables that is table 3.1. We then looked at possible relationships between these variables to build a taxonomy of dependent and independent ones. The final visualized version can be seen in fig. 3.2. We did this because we were not able to find a taxonomy of related variables. Most of the research that was done looked at a few variables and their effects. We did find research, that made their own literature reviews though. For example Martensson et al. created the EMFIS model based on an extensive literature review of around 74 sources[26]. Ståhl and Bosch also made a rather big literature review to find the differences between industrial CI implementations [41].

Our work lays the groundwork for a more extensive and complete variable taxonomy on CI/CD pipelines and their effectiveness. Further research might extend our taxonomy with more results from interviews with developers, or even more literature research.

As for the second research question, *"RQ2: Which (independent) variables/factors impact effectiveness of Continuous Delivery and Integration practices?"* we performed a statistical analysis on mainly the TravisTorrent[3] and GHTorrent[13] datasets to answer parts of this question. Because of our reliance on these datasets, we were bound to their limitations. Mainly we only focused on the build failure rate and the build duration when analyzing the data. There are a few reasons for this. Firstly, the data has not all the dependent data we defined from our taxonomy available. As an example, we defined the software quality as one dependent variable that measures effectiveness of a CI/CD pipeline in section 3.1.3. However, there is no field called software quality in both datasets that we could analyze. We had some ideas like for example looking at the amount of issues or issues with "bug" tags, but ultimately these methods were not exact and also was there not enough of said tagged issues to create meaningful data. The test failure rate and duration of tests was already part of the variables that we already looked at, we thought it would not add much to our analysis because of that as well as those data points being mined from different logs of different analyzers with probably various different degrees of success. Limiting our dataset to just one build tool like for example Gradle could have been a possibility but reducing the amount

of projects by more than an multiple of 2 just for additional uncertain dependent variables seemed not worth it. In the following we discuss our findings of our statistical analysis on the mentioned variables.

5.1.1 Discussion of the Project Level Analysis Results

This section discusses the results of our project level analysis results which we gathered in section 4.2.1.

The amount of builds a project has in its lifetime influences the duration of said builds. The more builds a project had in its lifetime, the longer they seem to take. This contradicts our first hypothesis. An explanation of this, might be a connection between the amount of builds and the age of a project or its size. With each build the size and age of a project increases. While the relative differences are probably project specific, the general statement should be applicable. There does not seem to exist research which looks at Project Age and the Build Duration, there is research that looks at Project Size. Ståhl et al. looked specifically at Project Size and its influence to the continuity of the CI pipeline[44]. Islam and Zibran find no correlation between the project size and the build results.

Future research might look into the connection between the amount of builds and the project size/age.

The build duration has an influence on the build failure rate. The longer the builds get the higher the build failure rate becomes. This confirms our hypothesis. We did not find any research that supports or contradicts this observation. This observation makes sense though, as we mentioned earlier, the longer something runs, the more chance there is that something could fail. This assumes that the execution speed is generally the same for builds, which we can generally assume, since the hardware for open source TravisCI builds is probably very homogeneous. This gives a second incentive to shorten build times.

The amount of jobs that builds execute, influence the duration of said builds. The more jobs a build contains, the longer it takes. Our observations specifically see a big increase once the amount get greater than seven jobs per build. Our observation confirms part of our hypothesis. We did not find any indication of influence for the build failure rate.

Future research might look deeper into the seven jobs per build rule we observed. Reasoning about why it is seven jobs could be worth looking into.

There seems to be a relation between the amount of files modified and the build duration. The more changes there are, the longer the builds take. This confirms our hypothesis partially. We also hypothesized that the build failure rate would increase. Islam and Zibran also observe effects of the size/churn of contributions on the build failure rate[17]. However our observations do not support their findings. We can think of two possible reasons for this. Firstly, the authors mention that they excluded specific projects from their set and also removed all builds with the "start" status. We did neither of these things. We looked at the amount of builds that had the "started" status, but our build table had only one build with that state. The second reason is the definition of the term "Build". They mention 36.2 million builds in their study. If we look at our numbers, that corresponds to the amount of jobs that the TravisTorrent dataset looked at (they also used TravisTorrent). The way that the TravisTorrent dataset is structured is that each row is the result of one job, but with specific entries being duplicate if belonging to the same build. For example if a build has ten jobs and it failed in one of them, then all of the ten rows for that build have the build state failed even though a few of the jobs might not have failed. We checked this by looking at the build id of each row and counting the amount of differing columns. Only very few columns had differing values. The build state was not one of them. This observation lead us to create a secondary database table only consisting of the actual builds with "only" around 680 thousand builds. This could explain our differing results.

The amount of contributors influence the build duration, but not the build failure rate. The more contributors a project has, the longer the build times get. However this relationship seems to only start after around five or more contributors. In our hypothesis we mention that this might be the case since a bigger amount of contributors is usually a signal for bigger projects. There does not seem to be any research that looked at build durations based on the team size. Kerzazi et al. find in their research an influence of team size to the build failure rate[20]. Our observations do not support these findings. Islam and Zibran support our results as well with no connection found between those two variables[17]. Their research is also based on the TravisTorrent dataset while the findings of Kerzazi et al. are based on coworkers on different branches of one project instead of average team sizes of projects[17, 20]. Further research might look into the differences between these measures.

The project size has an effect on the build duration, but not the build failure rate. We mentioned the different variables that could be associated with the project size as the lines of production code, the amount of commits, the amount of star watchers or even the amount of contributors as discussed earlier. All of these variables seem to some way influence the build duration in our analysis. We did not find any correlation between build failure rate and the project size though. Islam and Zibran also do not see a correlation for these variables[17].

It is however questionable to look at the amount of commits as an indirect variable here. It is very difficult to study, since it could vary a lot depending on the project. Is for example a project with thousand small commits bigger than one with a hundred big ones? We think that only looking at this variable isolated, does not make much sense. However combining it with other variables and maybe even making it weight differently with the amount of changes in a commit could help to prevent this problem. Future research might look into how this variable could be studied better.

The assert density seems to affect the build duration. Interestingly, the test density did not seem to affect either the build duration or build failure rate. This could be an effect of the unreliability of this test measure. However the higher the assert density is, the higher the build durations seem to become. This makes sense as it is probably a better measure than the test case density. Asserts are the actual tests that are performed while a test case might contain many asserts. Another line of thinking is the reliability of counting the asserts could potentially be better than counting tests.

There does not seem to be any relationship between contributor casualty and the build duration or build failure rate. Reboucas et al. came to the same conclusion[34]. However there is also work that states that core team members do have a higher chance to pass the build tests[39]. Further research might be needed to get more data on these variables.

Ruby projects have longer build times as well as bigger build failure rates then Java projects. This observation confirms our hypothesis. Beller et al. also come to the conclusion that Ruby projects have a higher build breakage compared to Java[2]. A possible explanation for this might be, as mentioned earlier, the fact that Ruby is a dynamically typed language while Java is a statically typed. The static typing probably causes more failures to be found in compile time compared to at runtime. Further research might want to extend the languages further than just Java and Ruby as earlier research already looked at. Getting more data on more statically typed languages and dynamic ones and then comparing them could further strengthen this explanation.

Projects that belong to an organization seem to have slightly shorter build times. This could be explained by organizations prioritizing optimized build times. Our initial hypothesis of bigger build failure rates for non-organization projects was also rejected by our analysis. Further research could look into this by categorizing the organizations even more into non-profits, corporations or other types of organizations. Vassallo et al. already look at the differences between open source and industrial software CI build failures[49]. This could be a good way to compare industrial organizations doing open source and closed source software.

"Cloud Ready" projects have a slightly smaller build failure rate and also exhibit a longer build duration than other Projects. Our hypotheses for the build failure rate was correct, while the one for the build duration was not. An explanation for the longer build times might be caused by the added steps when using docker files. Another explanation for the failure rates might be, that the use of docker files might cause the build environments to be more stable and less flaky than without them.

5.1.2 Discussion of the Build Level Analysis Results

This section discusses the results of our project level analysis results which we gathered in section 4.3.2.

Builds that were run on branches which contained the keyword "release" in them, have longer build times then the others. This could be the result of having more tests run or more things happening during a release build.

Builds on branches with the "feature" keyword fail more often than ones without by a failure rate difference of 8% This could easily be explained by the fact that these builds contain new untested code for new features.

Builds on the master branch fail less often than other builds. This observation was also made by Hilton et al. where they measure a difference of around 7%. Our analysis was closer to 6%. They seem to have looked at only pull requests while we looked at all builds. This observation could be explained by the fact that it is usually the case that the maintainers of a project want the pull requests to have successful builds themselves before merging them into master.

Builds on branches with numbers seem to fail more often than the others with a difference of around 5%. Explaining this one is rather hard. One would normally assume that final release branches would be usually rock solid. This could also be further explored in upcoming research.

Chapter 6

Possible Threads to Validity

During the accumulation of variables, we clearly had to filter and group some variables already. The final table would have been much longer if that step did not happen. However it would have made the whole ordeal a lot more complicated for not much gain. Some papers like from Xia and Li used TravisTorrent variables directly [51]. All in all we tried to combine them while still keeping a lot of distinct variables. Naturally there could be errors and misjudgments on our side.

For our variables we looked mostly in to newer literature, since some of the papers we found already did literature search on other possible variables. This does however pose risks in maybe not having identified possible variables that were more relevant in earlier times.

For our datasets we used TravisTorrent as well as GHTorrent. We got a snapshot of data from a month apart for both datasets. So some data will not be at the exact same time. While both datasets are very high quality, they do have their quirks. The GHTorrent dataset even has its own paper about them by Kalliamvakou et al.[18]. We also did some restructuring of these datasets as described in section 4.1. These could naturally also have an effect on our results. Especially the notion of combining the executed jobs into builds like in the Travis terminology could be problematic. We think a clearcut definition of what a build might even be, could be needed. Our reasoning for these changes was discussed in section 5.1.1.

Also when we calculate the quantile ranges as described in section 4.3.1 we saw some variables having extremely large or small ranges of values in some quantile groups. A more detailed analysis of these ranges might be interesting for future research as well. Another thing was the amount of data we had of "Cloud Ready" builds. We only found 110 projects that were labeled "Cloud Ready" by us. We mitigated this by simply duplicating the data to be able to sample more data without the "Cloud Ready" label. While this is a viable strategy, the low amount of projects for that particular variable could be problematic.

Chapter 7

Conclusions

In this paper, we first created a large taxonomy of variables concerning the efficiency and effectiveness of Continuous Integration and Continuous Deployment practices. By performing a literature review of 42 papers and sources, we gathered 77 variables, which are then further grouped into possible dependent and independent variables. After that we also try to find the possible relationships between these variables by looking at the literature and also create our own hypotheses.

We continue by performing an empirical study. We use data provided by TravisTorrent as well as GHTorrent to find how various variables affect the build failure rate as well as the build duration. Looking at the project level and numerical variables, according to our results, the amount of builds, the amount of jobs, the average (source) files modified, the average active team size, the amount of commits, the amount of star watchers, the average lines of production code as well as the average amount of asserts per 1000 lines of code affect the build duration. The build failure rate however seems only affected by the duration of builds in our results.

Looking at non numeric variables at the project level, the choice of Java or Ruby in programming language affects both the build failure rate as well as the build duration. While, whether or not a project belongs to an organization and the "Cloud Readiness" of it seems to only affect the build duration of said project.

Looking at the build level, branches with the keyword "release" in them have larger build times than those that do not. Branches with the "feature" keyword seem to have a 8% difference in build failure rate to the branches without that keyword in it.

Our work should help identifying problematic CI/CD practices that could influence the CI/CD effectiveness. Naturally there is still a lot more to do. Our taxonomy should help with many upcoming research questions regarding the efficient and effective use of CI/CD practices. With these results, CI/CD effectiveness could be heightened in industrial as well as open source environments by manually or even automatically inspecting these variables and warning the maintainers of software projects if problematic instances of these variables are detected.

Bibliography

[1] Abigail Atchison, Christina Berardi, Natalie Best, Elizabeth Stevens, and Erik Linstead. A time series analysis of TravisTorrent builds: To everything there is a season. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.29. URL .

[2] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.62. URL .

[3] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: Synthesizing travis CI and GitHub for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.24. URL .

[4] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32 (2):50–54, mar 2015. doi: 10.1109/ms.2015.27. URL .

[5] Lianping Chen. Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, jun 2017. doi: 10.1016/j.jss.2017.02.013. URL .

[6] Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31, jan 2015. doi: 10.1016/j.infsof.2014.07.009. URL .

[7] 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 *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 55–66, 2014. doi: 10.1145/2642937.2643002. URL .

[8] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 374–377, 2012. doi: 10.1145/2351676.2351751. URL .

[9] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014. doi: 10.1145/2635868.2635910. URL .

[10] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 122:14, 2006.

[11] Alessio Gambi, Rostyslav Zabolotnyi, and Schahram Dustdar. Poster: Improving cloud-based continuous integration environments. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 797–798, 2015. doi: 10.1109/ICSE.2015.253. URL .

[12] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.38. URL .

[13] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL .

[14] Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane McIntosh. The impact of the adoption of continuous integration on developer attraction and retention. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 491–494, 2017. doi: 10.1109/MSR.2017.37. URL .

[15] 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*. ACM Press, 2016. doi: 10.1145/2970276.2970358. URL .

[16] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, 2017. doi: 10.1145/3106237.3106270. URL .

[17] Md Rakibul Islam and Minhaz F. Zibran. Insights into continuous integration build failures. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.30. URL .

[18] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2863-0. doi: 10.1145/2597073.2597074. URL .

[19] M. G. KENDALL. A NEW MEASURE OF RANK CORRELATION. *Biometrika*, 30(1-2):81–93, jun 1938. doi: 10.1093/biomet/30.1-2.81. URL .

[20] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? an empirical study. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014. doi: 10.1109/icsme.2014.26. URL .

[21] Seojin Kim, Sungjin Park, Jeonghyun Yun, and Younghoo Lee. Automated continuous integration of component-based software: An industrial experience. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 423–426, 2008. doi: 10.1109/ASE.2008.64. URL .

[22] Marko Leppanen, Simo Makinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mantyla, and Tomi Mannisto. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, mar 2015. doi: 10.1109/ms.2015.50. URL .

[23] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, mar 2018. doi: 10.1109/saner.2018.8330201. URL .

[24] Lech Madeyski and Marcin Kawalerowicz. Continuous defect prediction: The idea and a related dataset. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.46. URL .

[25] Marco Manglaviti, Eduardo Coronado-Montoya, Keheliya Gallaba, and Shane McIntosh. An empirical study of the personnel overhead of continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.31. URL .

[26] Torvald Martensson, Daniel Stahl, and Jan Bosch. The EMFIS model — enable more frequent integration of software. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2017. doi: 10.1109/seaa.2017.31. URL .

[27] Torvald Martensson, Daniel Stahl, and Jan Bosch. Continuous integration impediments in large-scale industry projects. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, apr 2017. doi: 10.1109/icsa.2017.11. URL .

[28] Mathias Meyer. Continuous integration and its tools. *IEEE software*, 31(3):14–16, 2014.

[29] Ade Miller. A hundred days of continuous integration. In *Agile 2008 Conference*. IEEE, 2008. doi: 10.1109/agile.2008.8. URL .

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

[31] Helena Holmstrom Olsson, Hiva Alahyari, and Jan 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*. IEEE, sep 2012. doi: 10.1109/seaa.2012.54. URL .

[32] Mohammad Masudur Rahman and Chanchal K. Roy. Impact of continuous integration on code reviews. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 499–502, 2017. doi: 10.1109/MSR.2017.39. URL .

[33] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.54. URL .

[34] Marcel Reboucas, Renato O. Santos, Gustavo Pinto, and Fernando Castor. How does contributors involvement influence the build status of an open-source software project? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.32. URL .

[35] Jeanine Romano and Jeffrey D. Kromrey. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? 01 2006.

[36] Eddie Antonio Santos and Abram Hindle. Judging a commit by its cover: correlating commit message entropy with build status on travis-ci. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 504–507, 2016. doi: 10.1145/2901739.2903493. URL .

[37] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous deployment at facebook and oanda. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 21–30, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4205-6. doi: 10.1145/2889160.2889223. URL .

[38] Mauricio Soto and Claire Le Goues. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 221–231. IEEE, 2018.

[39] Mauricio Soto, Zack Coker, and Claire Le Goues. Analyzing the impact of social attributes on commit integration success. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.34. URL .

[40] Rodrigo Souza and Bruno Silva. Sentiment analysis of travis CI builds. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 459–462, 2017. doi: 10.1109/MSR.2017.27. URL .

[41] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, jan 2014. doi: 10.1016/j.jss.2013.08.032. URL .

[42] Daniel Ståhl and Jan Bosch. Automated software integration flows in industry: A multiple-case study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 54–63, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591186. URL .

[43] Daniel Ståhl and Jan Bosch. Industry application of continuous integration modeling: a multiple-case study. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, pages 270–279, 2016. doi: 10.1145/2889160.2889252. URL .

[44] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, 127:150–167, may 2017. doi: 10.1016/j.jss.2017.02.003. URL .

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

[46] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014. doi: 10.1109/icsme.2014.62. URL .

[47] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, 2015. doi: 10.1145/2786805.2786850. URL .

[48] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous delivery practices in a large financial organization. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, oct 2016. doi: 10.1109/icsme.2016.72. URL .

[49] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of CI build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, sep 2017. doi: 10.1109/icsme.2017.67. URL .

[50] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. I'm leaving you, travis: A continuous integration breakup story. 2018.

[51] Jing Xia and Yanhui Li. Could we predict the result of a continuous integration build? an empirical study. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, jul 2017. doi: 10.1109/qrs-c.2017.59. URL .

[52] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, oct 2017. doi: 10.1109/ase.2017.8115619. URL .

Appendices

A Papers Used for Taxonomy Definition

DOI	Title	Bibliography
10.1109/ICSME.2017.67	A Tale of CI Build Failures: an Open Source and a Financial Organization Perspective	[49]
10.1109/MSR.2017.29	A Time Series Analysis of TravisTorrent Builds: To Everything There is a Season	[1]
10.1109/QRS-C.2017.59	Could We Predict the Result of A Continuous Integration Build? An Empirical Study	[51]
10.1109/MSR.2017.62	Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub	[2]
10.1145/2786805.2786850	Quality and Productivity Outcomes Relating to Continuous Integration in GitHub	[47]
10.1145/2635868.2635910	Techniques for Improving Regression Testing in Continuous Integration Development Environments	[9]
10.1109/Agile.2008.8	A Hundred Days of CI	[29]
10.1145/2970276.2970358	Usage, Costs and Benefits of Continuous Integration in Open-Source Projects	[15]
10.1109/ICSME.2014.62	Continuous Integration in a social-coding world: Empirical evidence from GITHUB	[46]
10.1109/ICSME.2014.26	Why do Automated Builds Break? An Empirical Study	[20]
10.1109/MSR.2017.32	How Does Contributors' Involvement Influence the Build Status of an Open-Source Software Project?	[34]
10.1109/MSR.2017.30	Insights into Continuous Integration Build Failures	[17]
10.1109/MSR.2017.54	An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software	[33]
10.1109/ASE.2017.8115619	The impact of continuous integration on other software development practices: a large-scale empirical study	[52]
10.1109/MSR.2017.31	An Empirical Study of the Personnel Overhead of CI	[25]
10.1109/MSR.2017.34	Analyzing the Impact of Social Attributes on Commit Integration Success	[39]
10.1109/seaa.2017.31	The EMFIS Model – Enable More Frequent Integration of Software	[26]
10.1109/icsa.2017.11	Continuous Integration Impediments in Large-Scale Industry Projects	[27]
10.1016/j.jss.2013.08.032	Modeling continuous integration practice differences in industry software development	[41]
10.1016/j.jss.2017.02.003	The continuity of continuous integration: Correlations and consequences	[44]
10.1145/3196398.3196422	I'm Leaving You Travis: A Continuous Integration Breakup Story	[50]
10.1109/msr.2017.38	An Empirical Study of Activity, Popularity, Size, Testing, and Stability in Continuous Integration	[12]
10.1145/2642937.2643002	Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation	[7]
10.1145/2351676.2351751	Communicating Continuous Integration Servers for Increasing Effectiveness of Automated Testing	[8]
10.1109/ICSE.2015.253	Poster: Improving Cloud-based Continuous Integration Environments	[11]
10.1109/MSR.2017.46	Continuous Defect Prediction: The Idea and a Related Dataset	[24]
10.1109/ASE.2008.64	Automated Continuous Integration of Component- Based Software: an Industrial Experience	[21]
10.1145/2889160.2889223	Continuous Deployment at Facebook and OANDA	[37]
10.1007/978-3-319-06862-6_12	Visualizing Testing Activities to Support Continuous Integration: A Multiple Case Study	[30]
10.1145/2591062.2591186	Automated Software Integration Flows in Industry: A Multiple-Case Study	[42]
10.1109/AGILE.2009.16	Enabling Agile Testing Through Continuous Integration	[45]
10.1109/MS.2015.27	Continuous Delivery: Huge Benefits, but Challenges Too	[4]
10.1016/j.jss.2017.02.013	Continuous Delivery: Overcoming adoption challenges	[5]
10.1016/j.infsof.2014.07.009	On the journey to continuous deployment: technical and social challenges along the way	[6]
-	Continuous Integration	[10]
10.1109/MSR.2017.37	The Impact of the Adoption of Continuous Integration on Developer Attraction and Retention	[14]
10.1145/3106237.3106270	Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility	[16]
10.1109/MS.2015.50	The Highways and Country Roads to Continuous Deployment	[22]
10.1109/MS.2014.58	Continuous Integration and Its Tools	[28]
10.1109/SEAA.2012.54	Climbing the "Stairway to Heaven" - A multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software	[31]
10.1109/MSR.2017.27	Sentiment Analysis of Travis CI Builds	[40]
10.1145/2901739.2903493	Judging a Commit by Its Cover - Correlating commit message entropy with build status on Travis-CI	[36]

B StackExchange Posts Used for Taxonomy Definition

Question	Link
optimize compiletime in Continous Integration	https://stackoverflow.com/questions/8646256/optimize-compiletime-in-continous-integration
Best practice on how to configure two local maven directories for continous integration with Hudson	https://stackoverflow.com/questions/3258323/best-practice-on-how-to-configure-two-local-maven-directories-for-continous-inte
Jenkins and Sonarqube - where to run unit tests	https://stackoverflow.com/questions/37405178/jenkins-and-sonarqube-where-to-run-unit-tests
Continuous integration - Best practices	https://stackoverflow.com/questions/1351755/continuous-integration-best-practices
NPM Best Practices for Continuous Integration	https://stackoverflow.com/questions/39138826/npm-best-practices-for-continuous-integration
Best-practice for continuous integration and deployment	https://stackoverflow.com/questions/9105459/best-practice-for-continuous-integration-and-deployment
Build management/ Continuous Integration best practices	https://stackoverflow.com/questions/419181/build-management-continuous-integration-best-practices
Continuous integration - build Debug and Release every time?	https://softwareengineering.stackexchange.com/questions/142604/continuous-integration-build-debug-and-release-every-time
Continuous Integration Feedback Cycle	https://softwareengineering.stackexchange.com/questions/189904/continuous-integration-feedback-cycle
Speeding up PHP continuous integration build server on Hudson CI	https://stackoverflow.com/questions/3696629/speeding-up-php-continuous-integration-build-server-on-hudson-ci
To Clean or not to Clean	https://stackoverflow.com/questions/5812872/to-clean-or-not-to-clean
maintaining a growing, diverse codebase with continuous integration	https://softwareengineering.stackexchange.com/questions/87723/maintaining-a-growing-diverse-codebase-with-continuous-integration
Improving CI build time (.NET)	https://stackoverflow.com/questions/8633313/improving-ci-build-time-net
Best practices for the best max length of time for running unit tests in CI	https://stackoverflow.com/questions/6737387/best-practices-for-the-best-max-length-of-time-for-running-unit-tests-in-ci
Different Ways to create an effective Ci pipeline for sonarqube analysis	https://stackoverflow.com/questions/43527267/different-ways-to-create-an-effective-ci-pipeline-for-sonarqube-analysis
What is a good CI build-process	https://stackoverflow.com/questions/102902/what-is-a-good-ci-build-process
Advice on Rails and CI, how often does this run exactly? or what is common practice	https://stackoverflow.com/questions/7621633/advice-on-rails-and-ci-how-often-does-this-run-exactly-or-what-is-common-pract
CI tests to enforce specific development rules - good practice?	https://softwareengineering.stackexchange.com/questions/143030/ci-tests-to-enforce-specific-development-rules-good-practice
Jenkins CI workflow implementation	https://stackoverflow.com/questions/37706713/jenkins-ci-workflow-implementation

C Taxonomy of Variables for CI/CD Effectiveness

Stage	Variable	Previous work found possible influential factors	Unstudied potential factors that could be investigated
Define	Developer Productivity		Code Owner, Contributor: Amount of Followers, Contributor Experience, Contributor Project Knowledge, Communication, Developer Motivation, Notification Quality, Project Architecture/Type, Project Organization, Project Size, Team Localisation
Develop	Developer Productivity	Amount of Contributors[37], Project Size[37], Duration of Builds[16], Duration of Tests[28]	Amount of Build Targets, Amount of PR's (accepted), Amount of Tests, Amount of Dependencies, Build Failure Rate, Build Failure Type, CI Server used, Code/Software Quality, Contributor: Amount of Followers/Casualty/-Commit Frequency/Experience/Project Knowledge, Communication, Defect Detection & Localisation Time, Developer Motivation/Education/Sentiment, Feedback Time, Incremental Builds, Notification Quality, PR Latency/Workflow, Programming Language, Project Age/Maturity/Architecture/Type/Organization, Team Localisation, Testing, Tests Failure Rate/Coverage, Tools Used, Tools Integration, Usage Of CI, Usage of static code analysis, Version Control System, Work Breakdown / Guidelines
	Development Time		Same variables as Developer Productivity
Commit	Developer Productivity		Contribution Complexity/File Type/Size/Churn/Type/Work Item, Contributor: Amount of Followers/Casualty/Commit Frequency/Experience/Project Knowledge, Commit Message Entropy, Developer Motivation/Education/Sentiment, Version Control System, Work Breakdown/Guidelines
Build/Integrate	Build Failure Rate	Test Failure Rate[2], Contribution Type[2, 33], Contribution Size/Churn[17], New Commits in Build[17], Amount of Contributors[20], Contribution Work Item[20], Stakeholder of Build[15, 20], Build Failure Rate[33], Contribution Complexity[33], Contributor Commit Frequency[33], Contributor: Amount of Followers[39], Contributor: Amount of Contributions[39], Contributor Casualty[39], Developer Sentiment[40], Commit Message Entropy[40] Project Age[46], Programming Language[46]	Amount of Builds, Amount of Build Targets, Amount of Contribution Types, Amount of Contributors, Amount of PR's (accepted), Amount of Dependencies, CI Server used, Code/Software Quality, Configuration Complexity, Contribution File Type/Work Item, Contributor Amount of Followers/-Commit Frequency/Experience/Project Knowledge, Communication, Communication between CI servers, Developer Motivation, Developer Education, Development Time, Environment, Incremental Builds, New Commits in Build, Project Maturity/Architecture/Type/Organization/Popularity/Size/-Growth, Stakeholder of Build, Testing, Time & Date of Builds/Commits, Tools Used, Usage of static code analysis
	Build Failure Type	Code Owner[49]	Same variables as Build Failure Rate
	Duration of Builds	Amount of Jobs[2], Project Size[44]	Amount of Builds, Amount of Build Targets, Amount of Tests, CI Server used, Configuration Complexity, Contribution Size/Churn, Communication between CI servers, Environment, Hardware, Incremental Builds, Job Structure, New Commits in build, Programming Language, Tools used, Usage of static code analysis, Using Parallel Computing
Test	Test Failure Rate	Programming Language[2], Environment[2]	Same variables as Build Failure Rate in the Build / Integrate Stage + Amount of Tests, Tests: Amount of Environments/Failure Rate/Coverage/Strategy/-Generation for classes/Given Time
	Duration of Tests	Programming Language[2]	Same variables as Duration of Builds in the Build / Integrate Stage + Amount of Tests, Tests: Amount of Environments/Coverage/Strategy/Generation for classes/Given Time
Release/Deploy	Release Frequency	Usage of CI[15, 16]	Amount of Tests, Amount of Hotfixes, Build Failure Rate, CI server used, Code Owner, Code/Software Quality, Contributor Experience/Project Knowledge, Communication, Defect detection & localisation time, Developer Attraction/Retention/Productivity/Efficiency/Motivation/Education, Development Time, Feedback Time, Programming Language, Project Age/-Maturity/Organization/Size, Team Localisation, Testing
	Feedback Time	Usage of CI[2, 22], Duration of Builds[2], Test Failure Rate[33]	Amount of Builds, Build Failure Rate, CI Server used, Communication between CI servers, Defect detection & localisation time, Duration of Tests, Environment, Incremental Builds, Job Structure, Hardware, Notification Quality, Preconfigured Testing VM's used, Tests: Failure Rate/Given Time, Time & Date of Builds/Commits, Tools Used, Usage of static code analysis
Other	Project Size/Growth		Amount of Builds, Amount of Build Targets, Amount of Contributors, Amount of Jobs, Amount of Tests, Amount of Dependencies, Contributor: Amount of Contributions, Project Age/Maturity/Popularity/Organization
	Code/Software Quality	Amount of Contributors [37], Project Size [37], Usage of CI[47]	Amount of Contributors, Amount of PR's (accepted), Amount of Tests, Amount of Dependencies, Amount of Hotfixes, Code Owner, Contributor: Casualty/Experience/Amount of Followers, Communication, Developer Education, Development Time, PR Work-flow, Programming Language, Project Age/Maturity/Architecture/Type/Organization, Testing, Tests: Coverage, Tools Used, Usage of static code analysis, Work Breakdown/Guidelines