Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Severi Haverila

Impacts of Continuous Delivery in Software Projects

Master's Thesis
San Jose, September 10, 2016

Supervisor: Professor Marjo Kauppinen
Advisor: Marko Klemetti M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Severi Haverila		
Title:	Impacts of Continuous Delivery in Software Projects		
Date:	September 10, 2016	**Pages:**	vii + 87
Major:	Software Engineering	**Code:**	SCI3043
Supervisor:	Professor Marjo Kauppinen		
Advisor:	Marko Klemetti M.Sc. (Tech.)		

Continuous delivery views software development holistically. It breaks down barriers that have traditionally been between development, testing and operations and combines them to a single working unit. The idea of continuous delivery is to develop software in such a way that the software is always in a releasable state and can be deployed into production at any time.

This thesis studies continuous delivery from the perspective of the following research problem: How does continuous delivery affect software projects? The topic is researched based on both a literature review and an empirical study performed. The latter consists of three interviews, a presentation and a follow-up survey based on the presentation.

The results of this thesis indicate that automation of manual tasks – such as building, testing and deploying the software – is a key building block of continuous delivery. In addition, the following principles facilitate a successful adoption of continuous delivery: build quality in, shared responsibility and creation of a reliable and repeatable software release process. These principles include different practices – such as test-driven development, frequent integration of code and frequent deployments – that will steer the software projects to the right direction.

Continuous delivery makes software deployments more reliable and flexible. It also increases the transparency of the development process and can have a positive effect on customer satisfaction. On the other hand, continuous delivery requires a higher initial investment and can require a fundamental change in how the software is being developed. Developers utilizing continuous delivery have to be skilled not just in development practices, but they also have to have knowledge over test automation and the operational side related to software deployments.

All in all, continuous delivery impacts the whole process of developing software starting from how software is to be developed and deployed until how the process should be supported on an organizational level. This makes continuous delivery a challenging process to adopt, but often this effort is worth due to all the benefits it introduces.

Keywords:	Continuous Delivery, Test Automation
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Master's Programme in Computer, Communication and Information Sciences

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Severi Haverila

Työn nimi:
Jatkuvan toimittamisen vaikutus ohjelmistoprojekteissa

Päiväys:	10. syyskuuta 2016	**Sivumäärä:**	vii + 87
Pääaine:	Ohjelmistotuotanto	**Koodi:**	SCI3043
Valvoja:	Professori Marjo Kauppinen		
Ohjaaja:	Diplomi-insinööri Marko Klemetti		

Jatkuva toimittaminen käsittelee ohjelmistokehitystä kokonaisvaltaisesti ja yhdistää ohjelmistokehityksen, ohjelmistotestauksen, sekä tuotantotoiminnot yhdeksi toimivaksi kokonaisuudeksi. Jatkuvan toimittamisen tärkein tavoite on mahdollistaa ohjelmiston julkaisu koko sen kehityksen elinkaaren ajan.

Tämä diplomityö tutkii jatkuvan toimittamisen ohjelmistoprojekteihin tuomia vaikutuksia. Tähän tutkimusongelmaan vastataan kirjallisuuskatsauksen sekä empiirisen tutkimuksen kautta. Empiirinen tutkimus sisältää kolme haastattelua, esityksen sekä esitykseen pohjautuvan kyselyn.

Diplomityössä suoritetun tutkimuksen tulokset osoittavat, että manuaalisten työvaiheiden – kuten ohjelmiston kääntämisen, testaamisen ja julkaisun – automatisointi on keskeinen osa jatkuvaa toimittamista. Lisäksi jatkuvan toimittamisen käyttöönotto vaatii erilaisten periaatteiden noudattamista, joista tämä diplomityö nostaa esiin seuraavat: laadun sisäänrakentaminen, jaettu vastuu, sekä luotettavan, että toistettavan julkaisuprosessin luominen. Nämä periaatteet sisältävät käytäntöjä kuten testivetoinen ohjelmistokehitys, jatkuva integraatio sekä toistuva ohjelmiston julkaiseminen, jotka ohjaavat ohjelmistoprojekteja oikeaan suuntaan.

Jatkuva toimittaminen tekee ohjelmiston julkaisemisesta luotettavampaa ja joustavampaa. Tämän lisäksi se parantaa ohjelmistokehitysprosessin läpinäkyvyyttä ja voi vaikuttaa positiivisesti asiakastyytyväisyyteen. Jatkuvan toimittamisen käyttöönotto kasvattaa projektien aloituskustannuksia ja se saattaa vaatia perinpohjaista muutosta käytössä oleviin ohjelmistokehityskäytäntöihin. Sen hyödyntäminen edellyttää myös kehittäjiltä laaja-alaista osaamista, sillä heidän täytyy ohjelmistokehityksen lisäksi pystyä työskentelemään testiautomaation sekä tuotantotoimintojen parissa.

Jatkuva toimittaminen vaikuttaa niin yksittäisten kehittäjien toimintatapoihin kuin myös siihen, kuinka organisaatioiden tulee tukea projekteja. Tämä tekee jatkuvan toimittamisen käyttöönotosta haastavaa, mutta usein tähän käytetty aika on sen tuomien hyötyjen takia vaivan arvoista.

Asiasanat: jatkuva toimittaminen, testiautomaatio

Kieli: Englanti

Acknowledgements

Firstly, I want to thank my thesis supervisor Marjo Kauppinen and my thesis advisor Marko Klemetti for their support and feedback throughout this process. I also want to thank them for their help in kick-starting this process on such a short notice and making it possible for me to complete this thesis in three months. In addition, I want to thank Tatu Kairi for taking the time to review my thesis and for providing valuable feedback.

Lastly, I want to thank Minna, my family and !nerdclub for their support during and outside of my studies.

San Jose, September 10, 2016

Severi Haverila

Abbreviations and Acronyms

ATTD	Acceptance Test-Driven Development
DevOps	Development and Operations
CD	Continuous Delivery
CI	Continuous Integration
QA	Quality Assurance
NPM	Node Package Manager
TA	Test Automation
TDD	Test-Driven Development
VCS	Version Control System

Contents

Chapter 1

Introduction

1.1 Motivation

Releasing software has traditionally been a manual, time-consuming and error-prone process that has caused a lot of stress to the people involved [1, p. xxiii-xxv]. Releasing the software has only been one part of the problem, as already during the development phase challenges have been introduced by changing requirements and a time-consuming testing phase just before the release. Modern software development practices have started to tackle this problem in multiple ways. Practices like agile, continuous integration and test automation are aiming to make development process more efficient and to enable developers to build software that actually meets customers needs.

Continuous delivery is a process that aims to combine the good development practices and techniques into one working unit [1, p. xxiii-xxv]. The core motivation of continuous delivery is keeping the software in a deliverable state throughout its development process. This enables flexible deployments and minimizes the risk of big failures.

The aim of this thesis is firstly to study and document different development processes which are central for successful usage of continuous delivery. Secondly, this thesis presents benefits and challenges that continuous delivery introduces.

1.2 Problem Statement and Research Questions

The research problem of this thesis is:

How does continuous delivery affect software projects?

The research problem will be answered from the perspective of the following research questions:

RQ 1 What kind of changes to software projects are required by continuous delivery?

RQ 2 What kind of benefits and challenges continuous delivery has in software projects?

1.3 Structure of the Thesis

This thesis begins with an introduction chapter in which the motivation for this thesis is presented together with the problem statement and research questions. In the second chapter, the used research methods are presented. Chapter three presents the findings of the literature review and after that in chapter four, the results of the empirical study are displayed. In chapter five, the results of this thesis are analyzed after which, in chapter six, the conclusions are presented. Figure 1.1 visualizes how the different chapters and sections of this thesis answer the research questions. The left side – i.e. the green area – depicts the RQ 1 and the right side – i.e. the blue area – depicts the RQ 2. The chapters and sections answering the research questions are then positioned in a way that they are located within the areas defined by the research questions which they answer to.

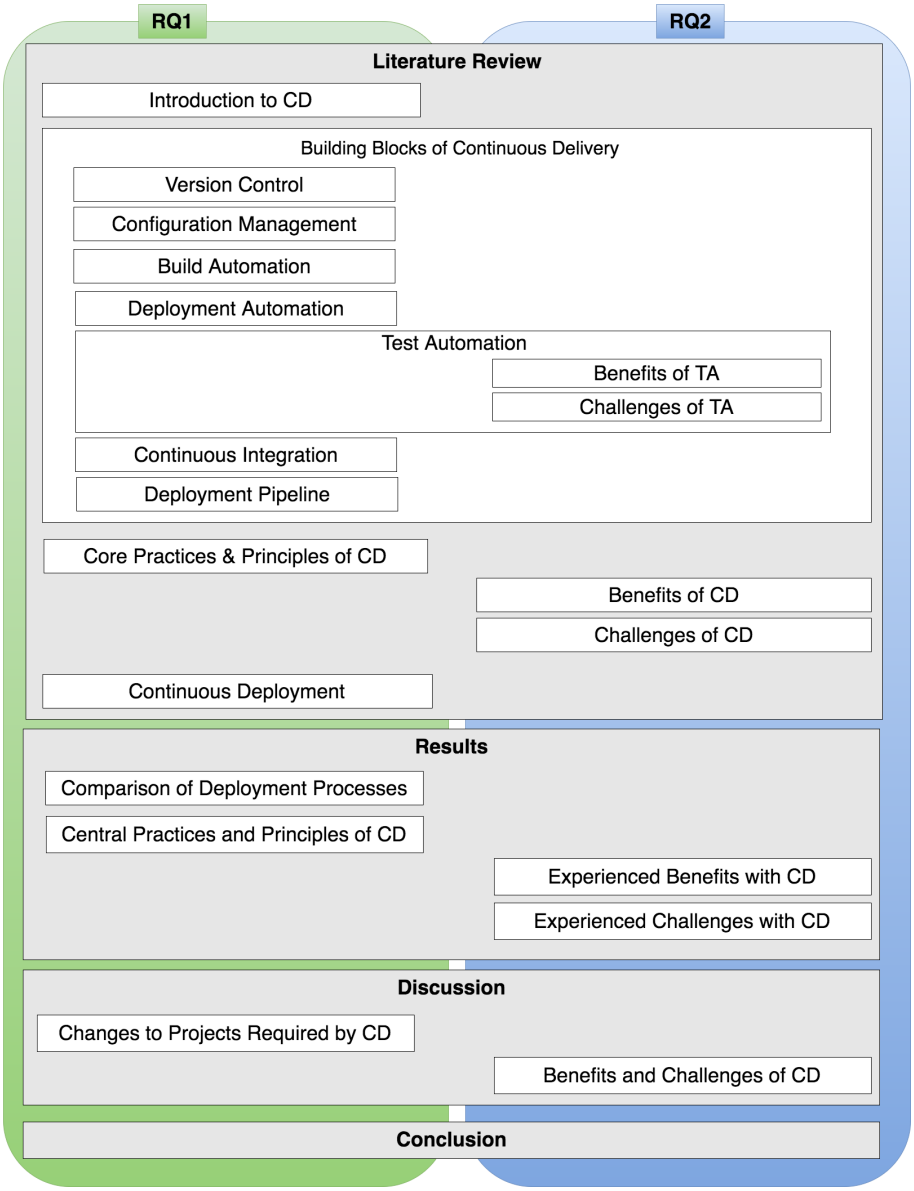


Figure 1.1: Structure of the Thesis

Chapter 2

Research Methods

The research of this thesis consists of a literature review and an empirical study. The research process is depicted in Figure 2.1. The research process started with a thorough literature review, whose findings were then used to plan the empirical study. The results of the empirical study were then compared with the existing research found in the literature.

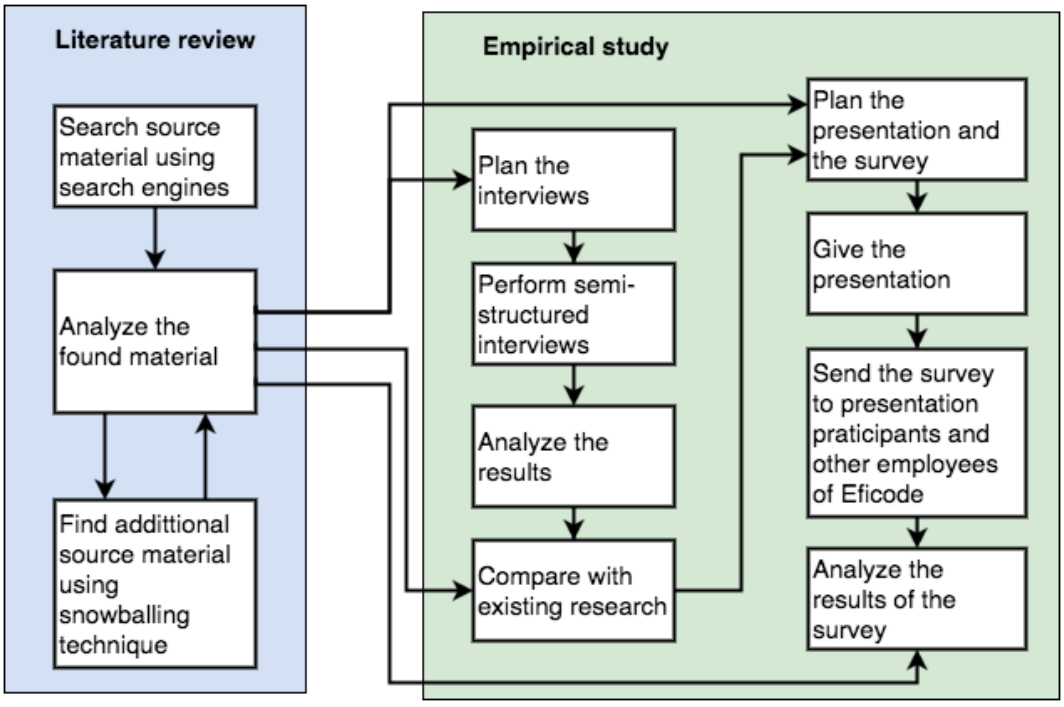


Figure 2.1: Overview of the Research Process

2.1 Literature Review

The literature review gives an overview on the pre-existing research done on this topic. Since the topic is fairly new, the amount of academic papers about the topic is still limited. Besides academic papers, books and web articles have been used as source material. The credibility of the non-academic sources has been maintained by mainly using sources that are commonly approved, referenced or written by credible authors.

The process how source material was gathered is presented in Figure 2.1. Google Scholar[1] was the main search engine used to find scientific articles. In addition, Safari Books[2] was used to search books related to the topic and Aalto Doc[3] to search master theses. The following keywords were used for searching source material:

1. Continuous Delivery
2. Continuous Integration
3. Test Automation

After analyzing the different source materials found, a technique called snowballing was applied. This means the source material of the different articles was analyzed and interesting or commonly referenced sources were further evaluated. A great deal of the source material used in this master's thesis was found using this technique.

2.2 Empirical Study

The research process of the thesis is presented in Figure 2.1. The empirical study consisted of two parts. First, semi-structured interviews were performed. The topic of the interviews was to discuss continuous delivery from the perspective of the two research questions. The discussions were focused on comparing two projects with each other, where in the first project continuous delivery was being applied and in the other project it was not. The interview results were analyzed and the findings compared with existing research found in literature. The second part of the empirical study consisted of a presentation and a survey. The aim of the presentation was to present

[1]https://scholar.google.com
[2]https://www.safaribooksonline.com
[3]https://aaltodoc.aalto.fi

the findings related to RQ 1 to professional software engineers and using the survey gather feedback and quantitative data.

2.2.1 Case Description

The projects have been implemented by Eficode. Eficode is a Finnish company offering software services specialized in software development, DevOps and consulting. It has been founded in 2005 and at the time of writing Eficode has around 100 employees working in offices in Helsinki, Tampere and Copenhagen. Customers range from telecom- and banking industry all the way to small startups.

2.2.1.1 Project A

Project A is a project utilizing continuous delivery and it is being developed for a fresh startup. At the time of the interview, the product development had been going on for around three months and was about to go to a beta testing phase. In total three developers have participated in the development of the project.

The project uses JavaScript-based technologies for its applications. The front end application is built using AngularJS[4] and the back end application uses NodeJS[5] together with Koa[6].

The project uses Jenkins CI server[7] for its deployment pipeline. The deployment pipeline on the CI server consists of nine different steps, and takes on average three minutes to execute. The deployment pipeline was build upon an existing generic deployment pipeline that had been developed in the company. The product had not yet been officially published and therefore the release stage in the figure means releasing the product to the customer – i.e. to the startup the product was being developed for – but not to its end users.

[4]https://angularjs.org
[5]https://nodejs.org/en
[6]http://koajs.com
[7]https://jenkins.io/2.0

Configuration management in this project is done with three different components:

1. Vagrant[8]

   Vagrant is a software that manages virtual machine life cycles. It is used for setting up and managing a standard virtualized environment in which Docker is going to be run. The virtualization software used in this project is KVM[9]. The configurations applied to the environment are stored in a file called Vagrantfile and the dependencies at this stage are installed and documented using Ansible[10]. Vagrant and virtualization is utilized in the deployment pipeline only for setting up the test environment, in production this step is not needed.

2. Docker[11]

   Docker is a software containerization platform and it is used to install project dependencies to isolated containers in which the different parts of the application are run. These containers can then be run on the different environments and this way ensure that both development and testing environments are identical to the production environment.

3. Node Package Manager (NPM)[12]

   NPM is a dependency management tool for JavaScript projects. NPM is used to install third party packages that a project depends on. NPM is used for back end and front end applications. The NPM dependencies are stored in a file called package.json.

2.2.1.2 Project B

Project B is a project whose development started in year 2012 and is currently under a support contract. During the active development phase the project was developed by three to four developers. The version running in production had been implemented over a year ago. Some additional development has been done after that but it was never deployed into production. A bug was found in the software during the support phase in 2016. The developers

[8]https://www.vagrantup.com
[9]http://www.linux-kvm.org
[10]https://www.ansible.com
[11]https://www.docker.com
[12]https://www.npmjs.com

who originally participated in the project are either not working for Eficode anymore or had continued with other projects.

The project was developed using Java[13] and it runs on a Tomcat server[14]. The project has dependencies on external web services. There is no knowledge whether a CI pipeline had been in use when the project was actively developed, but during the support phase of the project no CI pipeline existed. The project build and deployment was documented in Confluence[15] but the instructions were either outdated or inaccurate.

The project's quality assurance relied on unit tests and manual exploratory testing. A few end-to-end acceptance tests had been created using Robot Framework[16], but they were not comprehensive enough to be of actual value.

Configuration management exists in the project to some extend as project's external Java library dependencies are documented and managed using Maven[17]. The rest of the dependencies, such as required version of Java, were not documented.

The project is built using Maven. Because the test and production environments were not identical, the project had to be separately built for both of these environments using different build-profiles. The building of the project was executed by the developers on their local development environment.

2.2.2 Data Collection and Analysis

Three persons were interviewed. Two of the interviewees (referred as Developer 1 and Developer 2) had participated in both of the projects. Developer 1 worked as a lead architect in Project A and was highly involved in setting up the continuous delivery pipeline. In Project B he had more of a managerial role but was also involved in solving the technical tasks. Interviewee 2 worked as a front end developer in Project A. He was not involved in building the continuous delivery pipeline, but he is the most active developer working on the project and utilizes the pipeline actively. In Project B he worked in a support role, meaning he was responsible for fixing a bug that was found in the software after it had been running in production for over

[13]https://www.oracle.com/java/index.html
[14]http://tomcat.apache.org
[15]https://www.atlassian.com/software/confluence
[16]http://robotframework.org
[17]https://maven.apache.org

Interviewee	Role in Project A	Role in Project B
Developer 1	Lead Architect	CTO
Developer 2	Developer	Support Engineer
Ops 1	-	Platform Specialist

Table 2.1: Interviewees and Their Roles

a year. The third interviewee (referred as Ops 1) was involved in designing the base for the continuous delivery pipeline used in Project A. An overview of the persons interviewed and their roles is presented in Table 2.1.

The interviews with Developer 1 and Developer 2 were performed between 28.06.2016 – 29.06.2016 and both of them lasted a bit less than an hour. The interview with Ops 1 was performed 08.07.2016 and it lasted 16 minutes. The interviews were performed as semi-structured, meaning the questions were planned beforehand, but the conversations were let to divert to interesting aspects that came up during the interview [2]. Semi-structured interviews are common in case studies [2]. The interview questions are presented in Appendix A and Appendix B. Besides official interviews, the topic was further discussed with the interviewees during the analysis of the interview results.

The project interviews had two goals. First goal was to get a high-level understanding of the two projects (questions 1 & 11). The second goal was to gather data related to the research questions of this thesis. For RQ 1, one the goals was to understand how continuous delivery was used in Project A and how the different parts of it were implemented (questions 2,3 5 & 6). Also, the aim was to compare how the deployment process differed from Project B (questions 12,13 & 15). The goal of question 22 was to find out what development practices the developers considered as being important in Project A.

The goal of questions 4, 7, 8–10, 14, 16–21 & 23 was to find out the benefits and challenges experienced in the projects. The aim of this was to gather qualitative data that can be analyzed and used to answer RQ 2.

The goal of the operations (Ops) interview was to gather a better understanding of the implementation process of the deployment pipeline infrastructure (question 1, 2, 3, 9). In order to gather more qualitative data for RQ 2, questions 4, 5, 6 and 7 were asked. This was important in order to get a better understanding of the challenges related to the process of setting up the deployment pipeline. The aim of question 8 was to gather data for RQ 1 from the operations' perspective.

The presentation was held on 22.07.2016 for employees of Eficode. The presentation lasted around 23 minutes and the topic of the presentation was "Continuous Delivery: What is needed to succeed?". In the presentation, a short overview on the topic was presented after which the different best practices of continuous delivery were discussed i.e.the findings related to RQ 1 were presented.

After the interview a survey was sent out to all Eficode's employees. The purpose of the survey was firstly to gather quantitative data to validate the findings presented in this thesis but also to get differentiating opinions that can be analyzed. The survey was constructed using Google Forms[18] and its content is presented in Appendix C.

In total 20 responses to the survey were received. All of the respondents are professional software engineers and therefore their opinions can be regarded as being substantial. Out of the 20 respondents 13 participated in the presentation given on the topic. The respondents answered the survey anonymously. The presentation was also video recorded and a recording of the presentation was sent out to all employees of Eficode. The responses are presented in Appendix D.

The data collected from the interviews was analyzed by reading the notes written during the interview and listening to the interview recordings and writing down the important statements made during the interview. These written statements were then compared with each other in order to form a high level understanding.

The survey data was analyzed by exporting the answers to Google Sheets[19] and ordering the answers in a way that best visualizes the common patterns between respondents' answers to different questions. This is visualized in Figure 2.2.

	Question 1	Question 2	Question 3	Question 4
Respondent 1	Agree	Agree	Agree	Agree
Respondent 2	Agree	Agree	Agree	Disagree
Respondent 3	Agree	Agree	Disagree	Disagree
Respondent 4	Agree	Agree	Disagree	Disagree
Respondent 5	Agree	Disagree	Disagree	Disagree

Figure 2.2: Visualization on Performed Data Analysis of the Survey Results

[18]https://www.google.com/forms/about
[19]https://www.google.com/sheets/about

Chapter 3

Literature Review

3.1 Introduction to Continuous Delivery

Continuous delivery is a software engineering approach which aims to tackle challenges traditionally experienced with software deployments [1, p.xxiii-xxv]. In the literature, continuous delivery is defined as follows:

> We describe continuous delivery as the ability to release software whenever we want.
> - *Neely and Stolt [3]*

> Continuous delivery is a software engineering approach in which teams keep producing valuable software in short cycles and ensure that the software can be reliably released at any time.
> - *Chen [4]*

> Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.
> - *Fowler [5]*

As can be seen from the definitions, the key principle of continuous delivery is keeping the software continuously in a deployable state. However, the definitions lack one important principle of continuous delivery that facilitates the process of being continuously able to deploy the software in a reliable and effective way: automation.

The importance of automation is highlighted in the title of a book by Humble and Farley [1], which originally popularized continuous delivery: *Continuous*

Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. In this thesis continuous delivery is summarized as being a software development discipline which gives high priority to automation of manual tasks – such as building, testing and deployment – and in which the software can be released at any time reliably. Figure 3.1 visualizes what continuous delivery consists of and how the different parts relate to each other. In practice, continuous delivery consists of its technical implementation – the deployment pipeline – and different principles and practices that have to be followed.

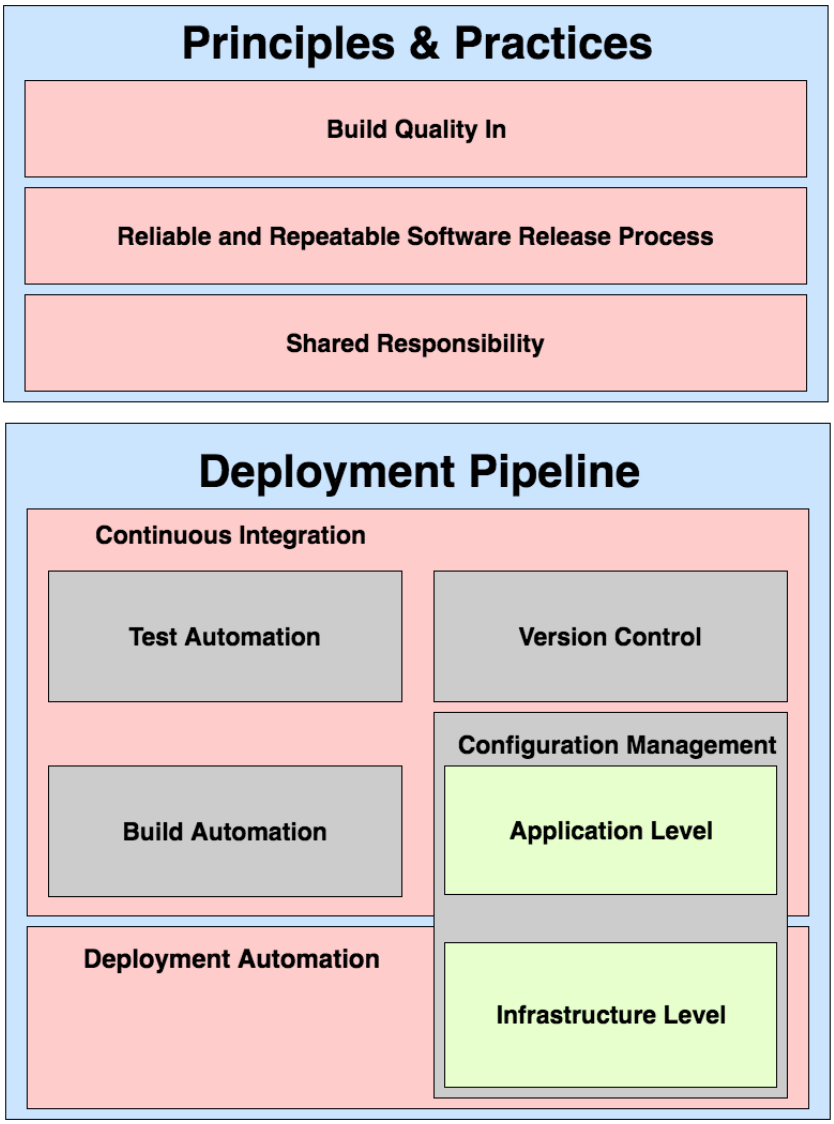


Figure 3.1: Overview of Continuous Delivery

Figure 3.2 shows a high-level overview of the different steps that are included in continuous delivery starting from building the software until deploying it into production. The figure also visualizes the relation and differences between continuous delivery and two other closely related practices: continuous integration and continuous deployment. These two practices are discussed in more detail later in this chapter.

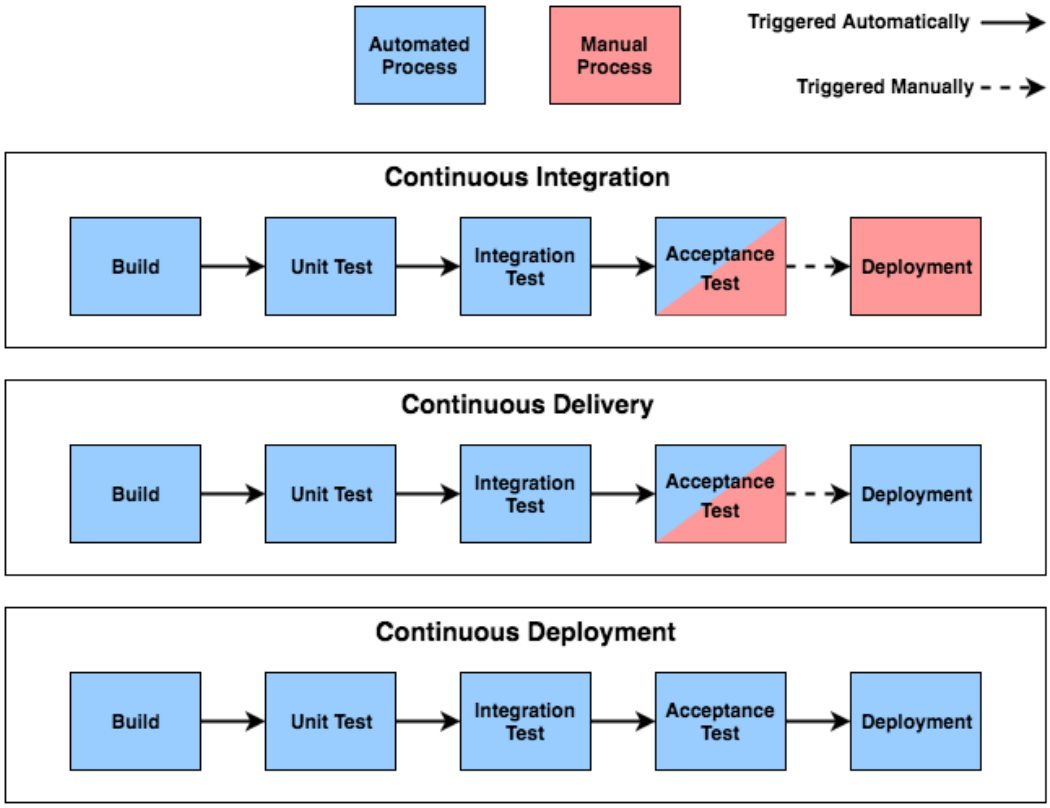


Figure 3.2: Comparison Between Continuous Integration, Continuous Delivery and Continuous Deployment

3.2 Building Blocks of Continuous Delivery

In this section, the core building blocks of continuous delivery are introduced. These building blocks create the base for a successful deployment pipeline. The relation of all these building blocks is visualized in Figure 3.1.

3.2.1 Version Control

Version control is the key element of any software development project [6]. Version control is handled by a version control system (VCS). Everything that is required to build, run and test a project has to be stored in the version control system [1, p.56-57].

A version control system has two important purposes. Firstly, it is used to provide access to all versions of all files [1, p.32-33]. This means that developers have to be able to see what changes have been introduced to which files, when and by whom i.e.the changes made to the software have to be traceable. Secondly, it enables collaboration [1, p.32-33]. Multiple developers have to be able to work on the same software together despite of the developers physical location or time zones they are working in.

3.2.2 Configuration Management

According to Pulkkinen [7], configuration management means automation of setup and configuration processes of different environments and infrastructure. Humble and Farley [1, p. 32] extend this definition to include application configurations and other artifacts needed in the project as well.

This thesis divides configuration management into two levels:

- Application level

  Application level configuration management includes storing all the application related configurations needed to build, install and run the application.

- Infrastructure level

  Infrastructure level configuration management includes storing all configurations needed to set up an infrastructure on which the application or applications are run. This includes information such as network configurations or target operating systems.

In order to create a clear separation, this thesis separates configuration management from the automation of setting up the environments and building the application based on the configuration. However, these steps are closely related; as can be seen from the definitions of configuration management above, many sources do not make this distinction.

The motivation for proper configuration management is keeping the different environments and configurations consistent, and tracking changes made to them. If the different environments are not consistent or changes made to the environments have not been tracked, environment related problems can be introduced to the software which can be difficult to debug and reproduce. This is because even if a software failed in production, the same software could still work in a test environment due to inconsistent environments [8] [1, p. 18].

Version control is a central part of configuration management. Version control is not just for storing the application source code, but also for (at minimum) everything that is needed to re-create application's binaries and setup its running environment from scratch [1, p. 33]. According to Humble and Farley [1, p. 50], creating a new environment should always be more feasible than repairing old ones. Fowler [9] calls this "Phoenix Servers", which is a rather matching name.

Humble and Farley [1, p. 32] highlight that an important aspect of configuration management in continuous delivery is that it should serve as a source for information. In addition, it should be accessible and modifiable to all team members. Without proper configuration management, the overview of what has been installed on a server is easily lost [10].

3.2.3 Build Automation

Build automation is the process of building the software automatically. In practice, this means that one should be able to build the application executing one command [6]. Humble and Farley [1, p. 143] also use the phrase build scripting to explain how this is achieved. Build automation lowers the risks introduced by human errors. This way, it enables building the application in a consistent and repeatable way.

3.2.4 Deployment Automation

Humble and Farley [1, p. 143] use the phrase "deployment scripting" for describing the process of automating the deployment of an application. They state that deployment scripting should cover both the process of upgrading an application, as well as installing it from scratch. In addition, they argue that the same deployment script should be used for deploying the application into different environments, which means environment specific configurations

have to be retrieved from configuration files. This builds the connection between deployment automation and configuration management.

Another important factor is that reverting deployed changes, in case errors are introduced, has to be possible. According to Humble and Farley [1, p. 131-132], the process of deploying and reverting the changes should not differ, because the reverting process is rarely tested and is therefore error-prone. They state that the reverting process can be either achieved by keeping the previously-deployed version of the software stored or by re-deploying it from scratch.

This thesis considers deployment automation as being the process of automatically setting up the environment and infrastructure on which the software is run, as well as deploying the application to that environment. The process of setting up the environment is also referred as environment management in the literature [1, p. 154-155].

3.2.5 Test Automation

Myers et al. [11, p. 6] define software testing as "the process of executing a program with the intent of finding errors". This is a good definition as it highlights that the idea of testing is to find bugs in software, not to show that they do not exist.

Test automation is a way for software quality assurance to keep up with the growing delivery pace in software development [12, p. 24]. Multiple definitions of test automation exist. Dustin et al. [13, p. 4] defines test automation as follows:

> The management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool.
> - *Dustin et al. [13]*

This is used as definition for test automation also by Karhu et al. [14]. Another definition is provided by Koirala and Sheikh [15, p. 127]:

> Automation is the integration of testing tools into the test environment in such a manner that the test execution, logging, and comparison of results are done with little human intervention.
> - *Koirala and Sheikh [15]*

Bach [16] extends test automation to cover the automation of the following items as well: test generation, system configuration, simulators, probes, oracles, activity recording and coverage analysis. This shows that test automation is a broad topic and depending on the environment can have varying meanings what is actually covered with it. As a summary, this thesis defines test automation as the automation of test execution, result evaluation and related supportive tasks.

Test strategy plays a crucial role in testing and is often not implemented correctly in test automation [17]. Knowing what to test and to what extent, allows testers to focus on the critical parts of the software in which bugs are more likely to happen. According to Boehm and Basili [18], 80% of the defects in a program come from 20% of its modules and 90% of software's downtime is caused by 10% of its defects. This underlines the need of allocating testing resources to those parts of the system where they are most needed and thus bring the biggest return of investment. This applies to test automation as well. Not all tests should be automated; tests which require intensive manual work and are run often are usually good candidates to be automated [12, p. 27].

According to Berner et al. [17], tests are repeated far more often than expected, which has to be taken into consideration when evaluating whether test automation should be applied in a project. They state that if a test case is to be executed more than 10 times, its automation should already be considered. They also argue that one important thing that has to be taken into account is that tests should be implemented on the correct level. For example, internal program logic should be tested on unit test level rather than through the user interface. Testing the right things on the right level makes debugging of a failed test case easier and also makes test creation more simple.

3.2.5.1 Benefits of Test Automation

An overview of main benefits of test automation found in literature is shown in Table 3.1. Many of the benefits of test automation are closely related. For example, lower long-term costs can be seen as the effect caused by other benefits such as reduced testing time or better use of resources. Therefore also in the literature different categorizations of the main benefits exist. Some examples are provided by Dustin et al. [12, p. 40], Rafi et al. [19] and by Graham and Fewster [20, p. 9].

Benefit	Description	References
Better use of resources	TA frees human resources from repetitive tasks, enables higher accuracy in those and allows manual testers to focus on more complex and advanced tests. Tests can be run even during out-of-office hours.	[17, 21] [12, p. 40] [20, p. 9]
Enables new types of testing	TA allows testing of things that cannot be tested test manually. One example of this is performance testing.	[12, p. 40] [20, p. 9]
Faults are easier reproduce	When doing manual (especially exploratory) testing, it can be hard to reproduce detected faults. TA makes it simple to rerun the test case and this way makes debugging the fault more simple.	[12, p. 40] [20, p. 9]
Increased confidence	Having extensive TA increases confidence in the product's quality.	[20, p. 10] [22]
Reduced testing time	TA increases significantly the efficiency of running tests. More tests can be run in less time compared to manual testing.	[14, 17, 23] [20, p. 9]
Test coverage increases	Automated tests enable more comprehensive testing of the software, which leads to improved software quality.	[14, 24–26] [12, p. 40]
Reuse of tests	The cost of TA can be distributed over multiple test executions. When automated test cases are executed frequently, the cost of one execution decreases. In the literature this is referred as "reusing the tests".	[20, p. 10] [14]
Lower long-term costs	Successful TA lowers the testing costs because testing can be done more effectively and with less human interaction.	[14, 25, 27]

Table 3.1: Benefits of Test Automation

Test automation has multiple benefits compared to manual testing. Test automation increases software quality by enabling better test coverage [14]. Test coverage measures how thoroughly the code has been tested [28]. An important thing to note is that good test coverage, when defined as lines of code that have been tested, can be achieved with low-quality tests that do not bring any additional value [29]. Therefore, test coverage by itself is not enough, also good quality tests are needed [29].

Test automation reduces testing time meaning the time it takes to run a certain amount of tests decreases [14]. This enables shorter testing periods and tests can be run more often [17]. When the efficiency of testing increases, more thorough and comprehensive testing can be implemented [20, p. 9]. This leads to improved software quality [17] which again can increase the customer satisfaction [21].

Test automation facilitates better use of resources. When repetitive manual tasks are automated, testers can focus on more complex scenarios and better tests [21]. It also enables running regression tests during out-of-office hours such as during the weekends or nights. In the long-term, this leads to lower software testing costs [25], which is a significant factor since testing is a major part of time consumption and cost in software projects [12, p. 27]

Reusability of tests is a central benefit that is introduced by test automation. The effort required to implement a test can be distributed among the times the test is run [20, p. 10], which in the long run will make the creation of automated tests economically competitive compared to manual testing [14].

By having a solid test automation in place, the confidence towards software's quality increases. Also, changes and releases can be made with better confidence, which however is only possible with comprehensive and good quality automated testing. Having low-quality tests can lead to false sense of security. [20, p. 10-11]

Faults are easier to reproduce when the test execution is automated [12, p. 40]. Graham and Fewster [20, p. 10-11] state that automation makes the execution of tests consistent and facilitates repeatability, which is difficult to achieve manually. According to them, repeating the same tests on different hardware configurations is one example in which automated testing outperforms manual testing. Test automation also enables new types of testing which cannot be done manually, such as performance testing [20, p. 9].

3.2.5.2 Challenges of Test Automation

A summary of the challenges of test automation is presented in Table 3.2. One thing that is not visible in the table is resistance to change because it was not explicitly mentioned in the analyzed literature. However, Karhu et al. [14] stated that motivating the employees to adopt new practices – such as test automation – seems to be important. Similar findings were also recognized by Fecko and Lott [21]. Having people changing their ways of working forces them out of their comfort zones and easily causes resistance to change [30].

One fundamental challenge with test automation is that it comes with a higher short term cost than manual testing [14]. Implementation of automated test cases requires initially more effort than documenting manual test cases [12, p. 56]. Test automation also requires a specific skill set and therefore testers need training before they can be successful in test automation [19]. Besides traditional testing skills which were required for testers such as deep domain knowledge and ability to use different testing tools, also development skills are needed when test automation is used [21].

False or unrealistic expectations are another challenge that companies face with test automation [17]. Companies might have false expectations towards the benefits of test automation and think it automatically solves all testing related needs, whereas in reality, it tackles unproductive test activities which are inefficient or impossible to do manually [17]. Another false expectations that exists is that test automation would automatically reduce the effort required by testing [12, p. 74].

False expectations is one cause for organizational challenges that companies face with test automation. If not enough time and resources are allocated to the implementation of test automation in a project, people will ignore it in order to get the software features done within the allocated budget [14]. One organizational problem mentioned by Graham and Fewster [20, p. 12] is that test automation is often viewed on a project level. They state that this is a problem especially in large organizations, which do not have standards how test automation should be implemented. This can lead to different implementations of test automation and therefore cause higher initial costs for all the projects and make moving testers to different projects difficult.

Organizational challenges are not just related to tight schedules and lack of standardized ways of working. Administrative issues can also have a negative effect on adopting test automation in a project [20, p. 12]. For example, if not

Challenge	Description	References
Automated test cases need maintenance	Automated test cases need to be maintained when the software changes. If automated tests start failing due to requirements- or technical changes and are not fixed, their trustworthiness decreases and people start ignoring them. This way the actual benefits of having TA is also lost.	[14, 17, 22] [20, p. 11]
Manual testing cannot be fully replaced	Automating 100% of software testing cannot be achieved. Manual testing is still needed for finding more complex defects.	[14, 16, 17, 27, 31]
False expectations	False expectations arise if TA is not understood correctly. If expectations are not met in a project, TA is easily abandoned.	[12, p. 74] [17] [20, p. 27]
Lack of proper test strategy	Test strategy defines which types of tests should be tested and how. Poor testing strategy means tests are not able to find faults efficiently.	[20, p. 11] [16, 17]
False sense of security	TA can provide a false sense of security in the quality of the product. If the automated tests are not testing the product correctly according to its requirements or themselves have defects, the test results can be misleading.	[20, p. 11] [22]
Technical challenges	Complex and highly customized products which are not designed with testability in mind can be challenging to test automatically.	[14, 16] [20, p. 11]
Training and skilled personnel needed	Test automation requires skilled testers. Besides traditional testing skills, also development skills are needed in order to create high quality automated test cases.	[14, 21]
Higher short-term costs	TA comes with a higher short-term cost. Implementing automated test cases takes longer than the creation of manual test cases. Also other factors such as maintenance and training costs have to be taken into account when calculating the TA costs.	[12, p. 56] [14, 27]
Organizational issues	Due to higher short-term costs automation requires time and support from management. Tight schedule and budget have a negative effect on TA, because it can be difficult to allocate resources to TA.	[14] [20, p. 12]

Table 3.2: Challenges of Test Automation

enough licenses or proper tools are given to people who need them, the adaptation of test automation might end up being more costly [20, p. 12].

Test strategy is a central part of test automation and testing in general [12, p. 129-130]. If no sound test strategy exists, automating the tests will not be enough. Graham and Fewster describe the problem in the following quote:

> Automating chaos just gives faster chaos.
> - *Graham and Fewster [20, p. 11]*

Lack of proper test strategy can thus lead to false sense of security. Software's requirements must be tested according to their actual intention [22] and the tests have to be up-to-date [20, p. 11], or else, they do not reliably assure the software's quality. This can cause a problem where the software contains bugs or does not meet its defined requirements, but the problem is not recognized by the automated tests. It also highlights that automated test cases need maintenance in order to keep up with changing requirements and added functionality. This is also caused by defective test cases [20, p. 11]. If this problem is not acknowledged, developers and testers might continue the development on top of buggy software or deliver it to the customers just because the automated tests did not find any defects.

One thing that is important to realize is that everything cannot be tested automatically. An automated test case cannot find complex defects that an experienced tester is able to reveal [17]. The purpose of test automation is to ensure that existing functionality works after new code changes have been made or if the software is run in a new environment [20, p. 11]. According to Berner et al. [17], most of the defects found in automated testing are actually found during the creation of the automated test cases. They validate their argument based on findings by Kaner [31], who states that 60% to 80% of bugs are found during the development phase of automated test cases.

Not everything can easily be tested automatically due to technical challenges. If the software is complex [14], has dependencies to third-party-products [20, p. 11] or in general is not implemented with testability in mind, automated testing can be difficult to achieve. For this reason, test automation is not something that just affects the work of testers, but also developers have to create testable software.

3.2.6 Continuous Integration

The concept of continuous integration was originally introduced by Beck [32] in his book *Extreme programming explained* as part of the primary practices of extreme programming. Continuous integration is today widely used, but the ways it is applied in projects varies a lot [33]. It is considered as one of the key elements of agile development [34] and one of the building blocks of continuous delivery [1, p. 4].

The integration phase has traditionally occurred at the end of software projects [6]. Deferring the integration of different parts of a software increases the risk and pressure of the project just before the release. Fowler [6] provides an example of problems which were caused by a deferred integration in a project he was involved in. In his example, the software had been under development for a few years and the integration phase had already taken couple of months. He stated that even at this stage nobody had an idea when the integration was going to be finished.

Continuous integration is a software development practice in which developers integrate their code changes frequently with others. This is achieved by committing changes to the same mainline of a version control system at least daily [6, 33]. One criteria of continuous integration is that every time changes are committed to the version control, the application gets automatically built and tested [1, p. 55].

The motivation for continuous integration is to keep the software cohesive throughout its development phase so that integration would not be a time consuming and error-prone phase at the end of the project. It should rather be done continuously, thus the name continuous integration. If developers do not integrate their code on a frequent basis, it will be hard to predict how long the integration is actually going to take [6]. Also, if the different parts of the software have not been integrated, one cannot be sure whether the software is actually going to work [1, p. 56].

One benefit of continuous integration is that developers get rapid feedback if new changes have introduced bugs. If the build pipeline breaks, the developers have to put all their focus into fixing the build in order to keep the software in a working state [1, p. 55]. This pinpoints the importance of proper automated tests in successful continuous integration. If thorough test automation is not in place, changes to the software can introduce bugs to the existing functionality.

The technical implementation of continuous integration consist of two compo-

nents: Automated workflows and a feedback system. Automated workflows enable execution of user defined actions, which usually fetch newest code changes from the version control system, build the software and run its automated tests. The feedback system then notifies the developers whether the build was successful or not. [1, p. 63]

Figure 3.3 presents an example how continuous integration can look like in practice. First developer fetches the newest changes from the version control system (Step 1). After that he or she builds and tests the newly made changes locally (Step 2 and 3). If the local build passes, the developer then pushes the changes to the version control system (Step 4). This triggers a new build process on the CI server (Step 5). First the CI server fetches the newest changes from the version control system and then starts to automatically build a fresh instance of the application (Step 6). When the application is successfully built, the automated tests are run (Step 7). When the application has passed the CI pipeline, the results will be published (Step 8) in a way that the developers can see it. The developer who triggered the build process then assures that the build was successful (Step 9). If the freshly committed changes have passed the CI pipeline successfully, no further actions are needed. Other developers can later fetch the newest changes from the version control system (Step 10). If the build fails i.e.either the build phase does not succeed or errors are detected during the testing phase, the developer is notified. No further changes except build fixes are accepted to the version control system until the software has passed the CI pipeline successfully.

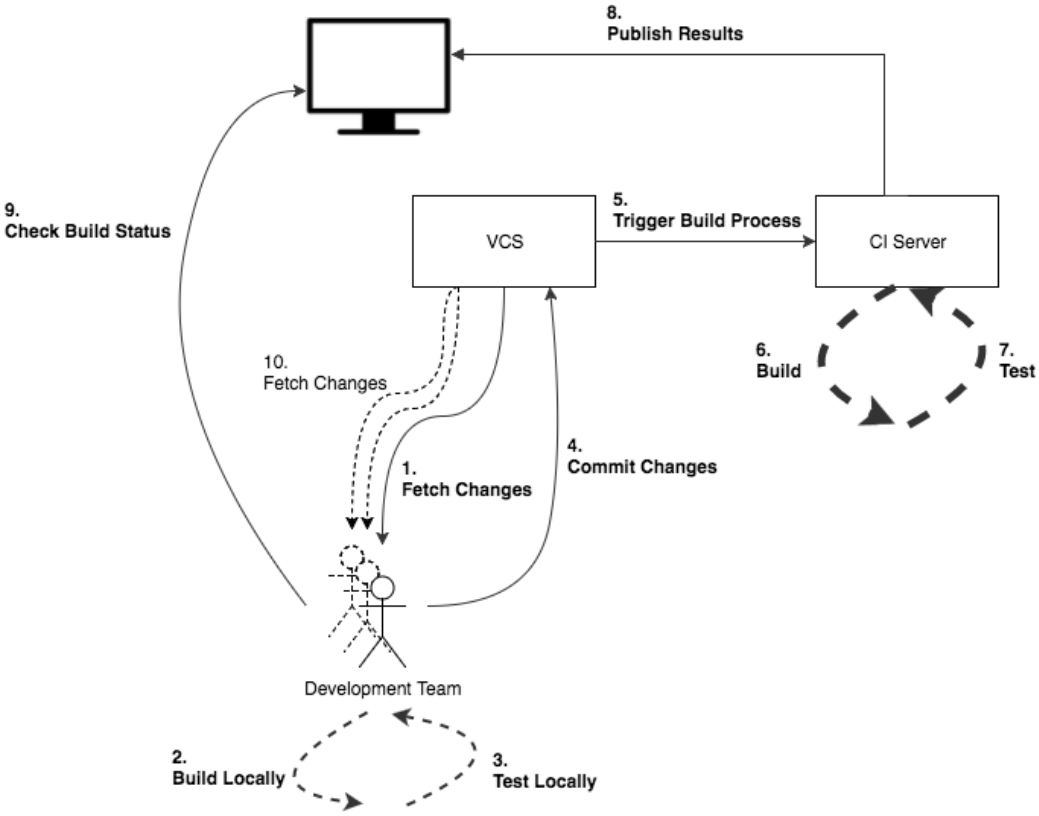


Figure 3.3: Example CI Flow

3.2.7 Deployment Pipeline

Humble and Farley [1, p. 4] describe the deployment pipeline – also called continuous delivery pipeline [4] – as being a logical conclusion of continuous integration and a central aspect of continuous delivery. According to Fowler [5], the difference between continuous integration and continuous delivery is that whereas continuous integration includes integrating, building and testing the code within a development environment, continuous delivery focuses on ensuring that the software is constantly in a deliverable state. This means that the software's quality has to be assured also in a more production-like environment. This definition gives the right idea, but is not generalized enough. Depending on the project, continuous integration can include tests run in a production-like environment but still cannot be called continuous delivery. One important difference is that in continuous delivery, the deployment process is automated and only needs to be triggered manually, whereas continuous integration does not take the deployment process

into account.

The deployment pipeline describes the different stages of a project's delivery process [35]. Each stage increases the confidence in the build by running more high-level tests in a more production-like environment [35]. The idea is to provide quick feedback to the developers in the early stages and run more time-intensive tests only if the build has passed the previous stages successfully [1, p. 110]. This trade-off between confidence and time-cost is visualized in Figure 3.4. The build is only passed to the next stage if it passes the previous stage successfully.

According to Humble and Farley [1, p. 109-110], the deployment pipelines and their stages differ and cannot be standardized in a way that would be applicable to all projects. They argue that even though additional testing stages can be present, all projects implementing continuous delivery have the following stages in common:

1. The commit stage

   The software is built and a technical verification of the software's quality is made. Technical verification in this context means that a collection of automated lower level tests (such as unit tests) and code analysis are performed. The commit stage provides initial feedback to the developers on the code changes they have made [4].

2. Automated acceptance test stage

   The software is automatically tested according to its functional and non-functional requirements [1, p. 110] [4]. Often the production environment is significantly different from the development- and test environments [1, p. 117]. Therefore, the acceptance tests should be run in a production-like environment, which is automatically set up and configured [4].

3. Manual test stage

   The software is tested manually in order to catch bugs that have slipped through the automated testing phase Humble and Farley [1, p. 109-110]. This can be done for example using exploratory testing.

4. Release stage

   The software is released to the end users or deployed to a staging environment. In continuous delivery, the decision to deploy is "a manual process" i.e. the deployment has to be triggered manually, but the deployment itself should be performed automatically [4] [1, p. 106].

> Continuous deployment takes this stage one step further: all changes are deployed without a manual approval.

This statement is mostly supported by examples from the literature which describe deployment pipelines used in real world projects [4, 8, 36]. Chen [4] states that the manual test stage is not always necessary, which is also supported by Udd [37]. The different stages are presented in Figure 3.4. Automation plays a crucial role in the deployment pipeline. In order to assure the product's quality through every build, test automation is highly utilized. Test automation enables fast feedback if a build breaks existing functionality or requirements and therefore is an essential part of continuous delivery.

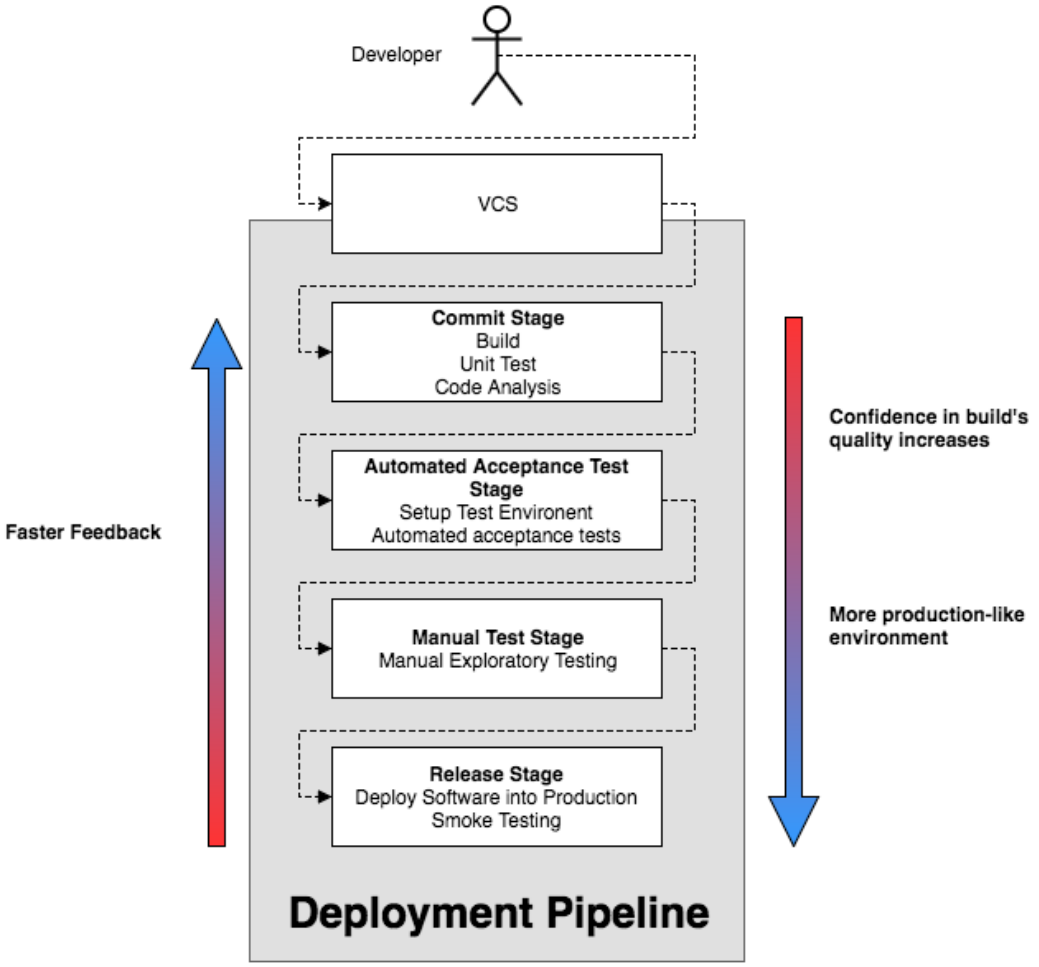


Figure 3.4: An Example Deployment Pipeline Inspired by Humble and Farley [1, p. 110]

3.3 Core Principles & Practices of Continuous Delivery

This section summarizes different principles and related practices, which are essential for a successful usage of continuous delivery.

3.3.1 Build Quality In

Detecting defects early reduces the cost of fixing them. Continuous delivery aims to find defects early, but discipline is required from the developers in order to be successful [1, p. 27]. Defects have to be fixed as early as possible. Building quality in is an ongoing process meaning that testing should be done at the same pace as development proceeds and that everyone is responsible for the quality of the software [1, p. 27]. It also includes the principle of continuously improving the test automation strategy [1, p. 83]. Testability is a non-functional requirement that is often forgotten [17]. When testing becomes a part of the software development process it also forces the developers to take the testability of the software into account.

Prioritize deployable software over features. The idea of continuous delivery is to keep software continuously in a releasable state. If this principle is not followed, one cannot state that continuous delivery is being used. This requires discipline from the developers. If a commit breaks the build, no additional features should be committed to the version control system before the build is fixed [1, p. 66]. The focus of the developers in this state needs to be fixing the software. This is called stop the line [1, p. 66].

Test-driven development. Having thorough and extensive testing in place is essential for continuous delivery [1, p. 66]. Good unit test coverage is crucial for enabling fast feedback during the commit stage [1, p. 66]. In order to assure the software's quality when new changes are introduced, tests should be created at the same time as the actual code [38]. Test-driven development (TDD) is a software development practice in which for all changes made in the implementation code, unit tests have to be created first [39]. Acceptance test-driven development (ATDD) is one version of TDD. In ATDD, tests for the projects requirements are implemented before the actual development of these features starts [40, p. 4]. This way ATDD also facilitates the communication between the projects stakeholders [40, p. 4].

Fast test feedback. Test automation plays an important role in continuous delivery. Even with comprehensive automated testing, developers should be able to get feedback on their changes quickly. If the duration of running the project's test suite grows, developers have to wait longer until they can be sure their changes have not introduced defects into the system. One way this problem is tackled in continuous delivery is distributing the different tests suites to multiple stages in the deployment pipeline [1, p. 171]. Also notable here is that the early stages should provide quick feedback in order to more easily detect what has caused the defect, because later more time-consuming stages in the deployment pipeline might batch multiple changes together, making pinpointing the problem more difficult [1, p. 171].

3.3.2 Reliable and Repeatable Software Release Process

Automate and standardize deployment process. The software should be deployed into different environments in a consistent way [1, p. 113-120]. This can be achieved by automating the deployment process. Fowler [5] actually states that being able to do such "push-button deployments" is one criteria that has to be fulfilled in order to state that continuous delivery is being used.

Deploy frequently. Software should be deployed into production. By having frequent releases, the risks involved with a single release decreases [5]. Frequent deployments also facilitate frequent customer feedback, which provides information whether the changes made meet the customer needs [4].

Frequent integration. The software should be developed in a way that changes committed to the version control system iteratively in small batches [1, p. 57]. Humble and Farley [1, p. 390-391] state that if developers are not integrating their code changes in to the mainline of the version control system at least once a day, the process cannot be called continuous integration. Doing frequent integrations to the mainline clashes with the idea of feature branches unless the branches are merged with the mainline at least daily [41] [1, p. 390-391]. Feature toggles are a way of dealing larger features and continuous integration [3].

Build binaries once. Binaries should only be build once during the deployment pipeline. This has two reasons. Firstly, building binaries can be a time-consuming. In order to keep the deployment pipeline as efficient as possible, such bottlenecks have to be eliminated. Secondly, rebuilding binaries intro-

duces risks that they will not be consistent during the different deployment stages. This can lead to problems where the software successfully passes the different testing stages of the pipeline, but introduce defects in production that have been caused by the inconsistency of the binaries. These inconsistencies can be caused by different library versions, for instance. The same binaries should be deployable to different environments used in the project. This can be achieved by separating environment dependent configurations from the code. [1, p. 113-115]

3.3.3 Shared Responsibility

The key factor of successful continuous delivery is that everyone is responsible for keeping the software in a deliverable state and for the adaptation on continuous delivery in general. This applies both on a team- as well as organizational level [1, p. 28]. All in all, continuous delivery should be viewed as a common goal that requires cooperation and cannot work with organizational silos not properly communicating with each other [1, p.28-29] .

Continuous improvement. Adoption of continuous delivery should be viewed as an incremental process that – especially in large organization – is difficult to adopt at once [1, p.420]. Instead, the adoption process should be done incrementally and its impact should be measured at the same time[1, p.420]. For example, Neely and Stolt [3] state that when automating the deployment process one should start with its slowest parts first. Even when continuous delivery has successfully been taken into use, teams should have retrospectives in which they can discuss problems and propose improvements [1, p.28].

3.4 Benefits of Continuous Delivery

Multiple benefits for continuous delivery are listed in the literature. Some sources such as Leppänen et al. [42] and Claps et al. [43] write about continuous deployment rather than continuous delivery, but many of the benefits mentioned in these sources can be regarded as being also valid for continuous delivery as well. The benefits of continuous delivery found in literature are summarized in Table 3.3 and discussed in more detail below.

Benefit	Description	References
Flexible deployments	Because software is continuously kept at a deployable state, its deployments to production can be done when needed.	[1, p. 21]
Reduced cycle time	Reduced cycle time means that the time it takes from starting the implementation of a feature until deploying it into production is shorter	[4, 42] [1, p. 17]
Faster feedback	Having the software continuously in a releasable state facilitates frequent deployments which again enable faster feedback. Faster feedback lowers the risk of developing software that creates no value for the end users.	[4, 5, 37, 42] [1, p. 11]
Increased productivity	Continuous delivery increases productivity by relying on heavy automation of manual tasks and allowing developers to focus in things that are important.	[4, 37, 42] [1, p. 18]
Increased process transparency	With more frequent releases, it is easier for the customer to believe that something has been done if it runs on a production-like environment.	[5, 37, 42]
Improved product quality	Continuous deployment has a positive effect on the product quality. This is achieved by continuous and thorough testing of the product after every change together with automation of manual steps which traditionally have caused human errors.	[4, 37] [1, p. 18]
Reliable releases	Software releases become more reliable because the software gets deployed frequently and automatically with smaller changes. This also lowers stress related to deployments.	[4, 5, 37] [1, p. 11]
Improved customer satisfaction	Continuous delivery can increase customer satisfaction because customer feedback can be reacted on quickly.	[3, 4, 42]
Releases cause less stress	Because the release process is automated, there are less possibilities for human errors. The decrease in stress is especially relevant when the software is released frequently with smaller changes.	[4, 37] [1, p. 20]

Table 3.3: Benefits of Continuous Delivery

Trustworthy releases is a benefit that is mentioned in multiple sources such as Humble and Farley [1] and Chen [4]. Because the software is kept at a releasable state continuously and can be deployed to production in an automated way, there is less possibilities for human errors [4]. As the deployment process is automated, also risks involved with losing employees from a project are lower because everyone has the ability to deploy the application [37]. Automated deployment process also enables flexible software deployments i.e.the software can be deployed at will [1, p. 21].

Releasing software frequently is encouraged (but not required) in continuous delivery. By releasing the software frequently smaller changes can be deployed to production, which also means that fewer concerns can go wrong [4]. The importance of releasing frequently is highlighted by Humble and Farley [1, p. 12], who state that frequent releases significantly reduce risks associated with releases and enables gathering of quick feedback and therefore is a central part of continuous delivery. Having quick feedback also motivates developers [42].

Trustworthy releases also lower the stress for developers and other project's stakeholders [1, p. 20] [4, 37]. According to Humble and Farley [1, p. 26], if some part of the development process is painful, it should be done more often, which is also the case with releasing frequently. The trustworthiness of the releases is reinforced by configuration management, which reduces errors caused by infrastructure and project dependencies [1, p. 18-19]. Automatically setting up environments is a key factor increasing productivity [4] and is enabled by configuration management and virtualization.

Continuous delivery reduces cycle time [4, 42] [1, p. 17], meaning the time it takes to get a change to production from start of the implementation. This makes the development process more adaptable. For example, if a critical bug is found, it can be fixed and delivered faster than before to the customers. This also has a positive effect on the customer satisfaction, because the response time to customer feedback gets shorter [42].

Continuous delivery makes the software development process more transparent to all stakeholders. Having a well defined definition of done, such as that the changes have been deployed to a production-like server, helps to follow how the development process is actually proceeding [5].

3.5 Challenges of Continuous Delivery

Besides its numerous benefits, continuous delivery also has its challenges. The challenges related to continuous delivery are summarized in Table 3.4 and discussed in more detail below.

Challenge	Description	References
High short-term cost	Continuous delivery comes with a high short-term investment, because setting up the continuous delivery pipeline can be time consuming.	[4, 8, 42, 43]
Complex and large software or legacy code	Continuous deployment can be difficult to apply if the software is large and complex or has multiple dependencies to other projects.	[4, 42]
Customer- and third party-related challenges	Customers might have internal QA criteria for the software used that might not support continuous delivery. Also customers might not like more frequent deliveries and it might negatively affect third parties which rely on a stable product.	[43]
Environment inconsistencies	Different production-, test- and development environments can cause challenges. Also insufficient configuration management is one cause of environment problems.	[1, 8, 42]
Organizational challenges	Organizational challenges include barriers caused by inflexible processes, responsibility silos or bureaucracy for instance.	[4, 8]
Test automation-related challenges	Successful test automation is critical in continuous delivery. Challenges related to test automation are presented in more detail in Section 3.2.5.2.	[8, 42, 43]
Resistance to change	Continuous delivery can be a major change for developers workflow and this way can force them out of their comfort zones.	[42]
Skilled developers needed	Continuous delivery requires experienced and skilled development teams.	[43]

Table 3.4: Challenges of Continuous Delivery

Getting started with continuous delivery has its technical challenges because no out-of-the box solutions exists for the deployment pipeline [4]. Because of this, establishing the deployment pipeline is time consuming and comes with a high short-term cost [8]. Therefore it is crucial to have a solid support from management in order to have the needed resources and patience for getting the deployment pipeline implemented [8].

Test automation plays a central role in the QA process of continuous delivery. Test automation itself is not a simple thing to implement and comes with its own challenges. These challenges are presented in Table 3.2.

Complex and large software or legacy code base can be a challenging environment for continuous delivery [4]. Large software with numerous dependencies can cause slow build- and testing times for the project, which again slow down the feedback cycle [42]. Large poorly tested legacy software also creates a challenge for the projects quality assurance [42].

Bugs caused by inconsistencies in the environments in which the program is run can make finding them difficult and time consuming [1, p. 18]. According to Leppänen et al. [42], keeping all the environments and their configurations consistent enough can be a challenging task.

As can be seen from the different challenges listed in this section, continuous delivery is not a simple thing to achieve and requires a different mindset compared to traditional software development. Continuous delivery consists of multiple parts which have to be in correct shape in order to be successful. To achieve this, skilled developers are needed [43]. The transformation that continuous delivery introduces to the project and the development processes can be challenging to be motivated to everyone involved and actual resistance to change can be present [42].

Resistance to change is surely one cause for organizational challenges that might be encountered when continuous delivery is taken into use. Organizational challenges can be challenges caused by difficulties in getting access to resources which are needed for getting the deployment pipeline work [4]. If organizational structure is split into different responsibility silos, it might take time to open these up. According to Gmeiner et al. [8], a collective responsibility should be taken for the deployment pipeline, which means continuous delivery should have project wide support.

3.6 Continuous Deployment

The term continuous deployment was popularized by Fitz [44]. The difference between continuous deployment and continuous delivery is that in continuous deployment, every change that passes the automated tests is deployed into production automatically [7, 45], whereas in continuous delivery this step is triggered manually. One could also state that continuous deployment can be regarded as being a version of continuous delivery, but this does not apply vice versa [45]. However, Humble [45] states that if continuous delivery is applied correctly, the decision to move towards continuous deployment should not require any additional effort from the development point of view. In short, the essential difference between continuous delivery and continuous deployment is who makes the decision when new features should be deployed. In continuous deployment, it is decided by the IT, whereas in continuous delivery it is a business decision [45]. The difference between continuous integration, continuous delivery and continuous deployment is visualized in Figure 3.2.

As continuous deployment automates the complete deployment pipeline, the manual test stage is missing even though a short QA gate might still exist [45]. Because of this, the relevance of good quality and comprehensive automated testing is emphasized in continuous deployment [7].

The motivation behind continuous deployment is to force frequent software deliveries, in order to be able to fail fast if something goes wrong [44]. Frequent deliveries also can create faster customer feedback.

Continuous deployment has its benefits but also its challenges. for example, not all customers want their software to be updated frequently [42] or it might even be impossible, for instance as is generally with embedded systems [45]. Another challenge mentioned by Claps et al. [43] is that if features would be continuously deployed to the customers, they might not notice the new features easily. The challenges of continuous delivery in general are discussed more thoroughly in Section 3.5.

3.7 Summary

Continuous delivery is software development process which gathers multiple different software development practices into a single working unit. The technical cornerstone of continuous delivery is the deployment pipeline. Deployment pipeline is a manifestation of the deployment process starting from the moment when a developer commits changes to the source code until the moment when the changes are deployed into production. Continuous delivery requires skilled developers who have to follow certain principles and practices that enable the successful usage of the deployment pipeline.

The key principles of continuous delivery are good quality and comprehensive tests, automation of manual work, frequent integration and having all the necessary information needed for the deployment stored in a version control system. Besides that, quick deployment pipeline feedback plays an important role. Many of these practices have their roots in continuous integration.

Chapter 4

Results

In this chapter, the results of the performed interviews are presented from the perspective of the research questions of this thesis. Section 4.1 compares the deployment processes of Project A and Project B. Section 4.2 describes different principles and practices that were applied and considered important. This section also presents the results of the survey. After that, in Section 4.3 and in Section 4.4 the experienced benefits and challenges of continuous delivery are presented. Lastly, in Section 4.5, a summary of the results is given.

4.1 Comparison of Deployment Processes

Figure 4.1 visualizes the deployment process of Project A and more specifically how the different steps involved can be divided into the deployment pipeline stages. The commit- and automated acceptance test stages were performed and triggered automatically after new code was pushed into the version control system. If these stages passed successfully the code was automatically deployed into a staging server in which manual exploratory testing could be performed. The deployment into production was not yet in its full use as the software had not yet been published, but the goal was to automate the deployment process for the deployment stage as well.

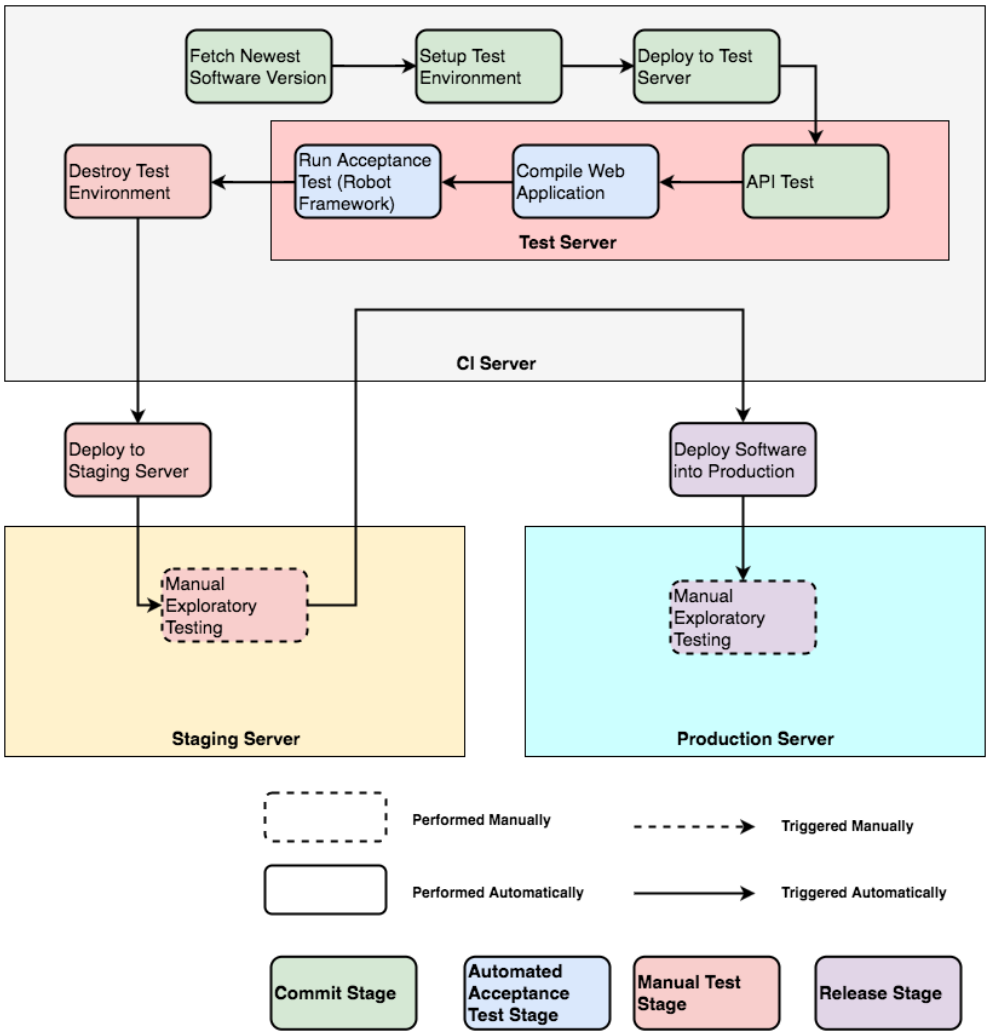


Figure 4.1: Deployment Process of Project A

The deployment process of Project B is visualized in Figure 4.2. First, the developers ran the unit tests and performed manual exploratory testing on their local machines. Then, on their local machines, they built the project using maven and copied the binaries manually to the test server, on which the old version of the software was manually replaced with the new one. When the new version of the software was running on the test server, manual exploratory testing was performed. When the quality of the software had been assured on the test server, the developers locally build the application for the production environment. After that the same steps were performed on the production server as on the test server.

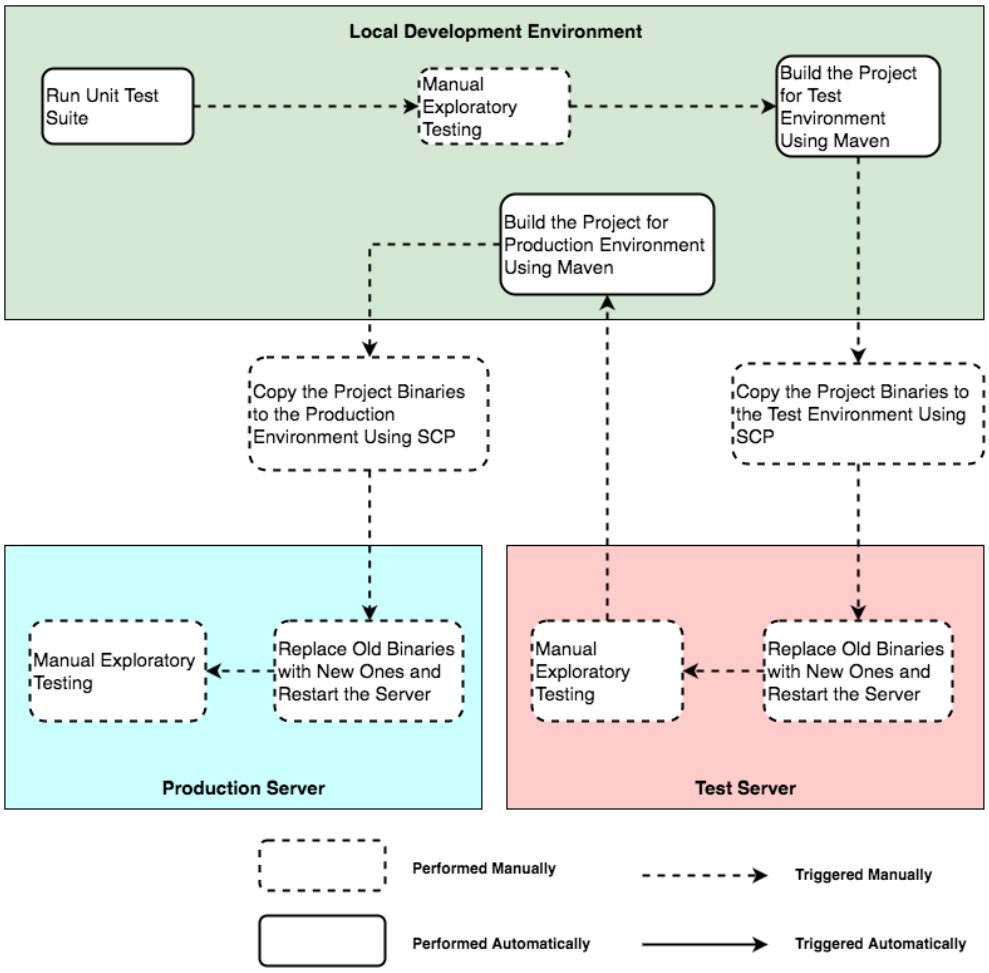


Figure 4.2: Deployment Process of Project B

As can be seen, the deployment process involved multiple manual steps. As the software did not have comprehensive end-to-end test automation in place, developers had to spend time manually testing the software in all the environments. The build process itself was automated, but had to be triggered manually. Also, copying the new binaries and replacing the existing ones on the production and test environments was performed manually.

The biggest difference between the deployment processes of Project A and Project B is the amount of manual steps involved. Whereas Project A had an automated deployment process starting from when a change had been made until when it was running on the staging server, in Project B getting the software running on the test server included multiple manual steps. The same difference is also visible in how the projects are deployed into production.

This part of the deployment is also planned to be automated in Project A, whereas in Project B it has to be done manually. As Project A used Docker for running its components, its environments were automatically set up and all the needed dependencies were documented. This was not the case for Project B, which can cause inconsistencies between the different environments and this way lead to environment related software bugs that are hard to debug. Both of the project had automated their build process for building the application for the different environments and was regarded as being a good practice.

4.2 Central Principles and Practices of Continuous Delivery

This section presents the principles and practices that are regarded as being central when using continuous delivery. First the findings from the interviews are presented and after that the survey results are discussed.

4.2.1 Findings Based on the Interviews

The most essential development principle that came up during the interviews and was also highlighted in the literature was that continuous delivery only works if software is done with good quality in mind. Humble and Farley [1] call this principle building quality in. In the interviews, Developer 1 stated that developers have to feel proud of the code they produce. When developers are proud of their work, they want to produce good quality code that has been thoroughly tested. According to the interview, it also enforces shared responsibility. Developer 1 stated that if shared responsibility is enforced by the development practices (such as continuous delivery), one simply does not want to commit badly tested code into the version control system because of the embarrassment that would be caused if the code breaks in the staging or production environment.

> Developers have to feel responsible for the changes they have made to the software. Already the embarrassment [of breaking existing functionality or introducing bugs] is often enough to make tests starting to appear, meaning it in a way steers [the development] to the right practices. And if it [continuous delivery] is not being used ... in traditional organizations developers and testers are separated ... developers do not have any kind of responsibility for whether the product works because they always can shove the changes to the testers and state "try out whether it works".
> *–Developer 1*

In the literature, TDD was mentioned as being a highly important practice in continuous delivery. The interviews also highlighted the importance of tests and particularly the fact that for all new code changes that are committed to the version control system, also automated test cases should to be included. According to the interviewees, TDD was not being followed in its purest form in Project A. The approach used was more based on trusting the developers. Developer 1 explained that too many pre-defined processes decrease the developers' feeling of responsibility and that more processes are usually needed for less-skilled developers.

Developer 2 explained that he is using a tests-first approach when developing code for Project A's back end application, but this process was the developers personal choice rather than a pre-defined process that all developers would have to follow. Acceptance tests had been created, but the interviewee felt that there often was too little time to implement those. This can be explained by how end-to-end tests often take more time to implement on the user interface level. In the back end application, the testing was rather a fast task as it was implemented with a unit testing framework. This finding suggests that even though the interviewees understood the need of having thorough tests, the practice was currently not being followed to its full extent.

The interviews showed that one of the key things the developers benefited from was fast feedback from the deployment pipeline. This is a principle that is also recognized in the literature as being a central aspect of successful adaptation of continuous delivery. Project A's different deployment pipeline steps and their duration are visualized in Figure 4.3. From this, it can be seen that for the first stage (commit stage) the duration is around 1 minute and 21 seconds and for the second stage (automated acceptance test stage) is around 1 minute and 16 seconds.

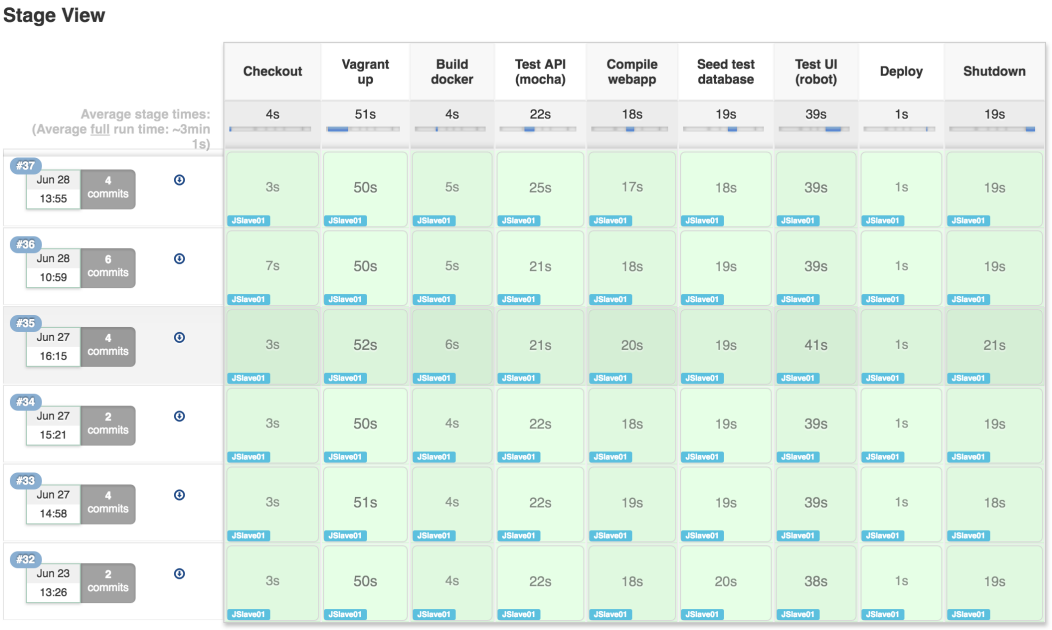


Figure 4.3: Deployment Pipeline of Project A

According to the interviews, frequent deployments are a central part of continuous delivery for two reasons. Firstly, frequent deployments enable catching software errors early on. When the software is deployed frequently, the deployment process was experienced to be less error-prone because only small set of changes were introduced and in case a bug was found the changes could be easily reverted and the bug was easier to find. Secondly, frequent deployments facilitate communication with other project stakeholders. Because Project A was not yet publicly available, stakeholders in this case mean other people than developers who are involved with the project. Frequent deployments seemed to help communicating the state of the project to the stakeholders as well as gather their opinions on newest changes. The findings of the literature review support this finding as having a positive affect on the continuous delivery process for the same reasons mentioned here.

Push-button deployments were considered as being an important principle. Developer 2 stated that the deployment pipeline did currently not support automated deployments to production, but it was seen as a goal that will be introduced to the pipeline later. In literature, this is considered as being a central part of the continuous delivery process.

According to the interviews having many branches in VCS makes software development more complex and is not a good software development practice

in general. Developer 1 stated that there should only be one branch used for development practices.

Having a technologically agnostic structure for the deployment pipeline was considered as a crucial aspect of successful continuous delivery. Building this structure had been a project of its own and Project A was the first actual project that had taken the pipeline into use. When asking whether building of such a pipeline is worth of the effort, Ops 1 stated the following:

> When thinking that if by this pipeline we can save five minutes of a developers work everyday, the pipeline pays itself back quickly. Actually I think it already has.
> –*Ops 1*

A technologically agnostic deployment pipeline will pay itself back, if it can be utilized in other projects. However, to actually prove the speed of this process, further studies would be needed. Having such a pipeline in place will decrease the starting costs for all projects that take it into use as the basic structure already exists and projects only have to build upon that based on their needs.

One of the interviewees stated that continuous delivery should be applied to projects from the start. When the infrastructure and architecture is built to support continuous delivery from day one, it is far more easier to continue applying the practice when the software really grows.

Principles & Practices	**Interviewee**		**Lit.**
	1	**2**	
Build quality in	X	X	X
Create automated tests before pushing new changes to VCS	X	X	X
Fast pipeline feedback		X	X
Deploy frequently	X		X
Automate deployment process	X		X
The less branches are in VCS the better	X		X
Build a technologically agnostic deployment pipeline in order to facilitate reuse in other projects	X	X	
Start using continuous delivery from the start of the project	X		

Table 4.1: Interview Results – RQ 1

4.2.2 Survey Results

The data of the survey results is presented in Appendix D. The survey results indicated that people mostly agreed with the statements made, though some exceptions existed.

First part of the survey consisted on questions on the different building blocks of continuous delivery. Participants had to express their opinions whether the following items can be considered as being an important part of continuous delivery:

1. Version Control
2. Build Automation
3. Test Automation
4. Continuous Integration
5. Configuration Management
6. Deployment Automation

According to the results, the common understanding was in line with the findings of this thesis as the majority of the people either strongly agreed or agreed with the statements. The only item which had one differentiating opinion was deployment automation as one of the survey respondents strongly disagreed with the statement. The same participant also disagreed with the following statement:

- Deployment process should be automated

One reason for this could be misunderstanding the statement as it is possible that the respondent had understood the statement in a way that the deployment should be triggered automatically as it is done in continuous deployment. The respondent did not clarify his answer. Nevertheless, the majority agreed with both of these statements and as they are also strongly supported in the literature one can conclude that they are true.

The second part of the survey consisted on listing best practices of continuous delivery found in the literature and collecting people's opinions on those. A rough division based on the answers is visualized in Table 4.2.

As can be seen, the two most controversial statements were related to continuous integration and more precisely to branching practices that should be applied. Generally many of the respondents preferred using feature branches

Majority of the respondents disagreed with the following statements:
1. Feature branches should be avoided
2. Everyone should commit their work directly to the mainline/trunk/main-branch of the version control system
Majority of the respondents agreed with the following statements, but also some of the respondents disagreed:
3. Untested code should not be committed to the version control system
4. Test driven development is an essential practice in continuous delivery
5. Everything needed to build and deploy an application should be stored in version control
6. Everyone should integrate their work with the mainline at least daily
The statements below were commonly regarded as being true:
7. The software should be deployed frequently into production
8. Developers should integrate their changes frequently
9. Setting up an environment in which the software is run should be an automated process
10. Deployment process should be automated
11. Build process should be automated
12. Everyone is responsible for keeping the software in a deliverable state
13. Fixing of broken builds has the highest priority
14. Deliverable software should be prioritized over new functionality
15. Deployment should be a standardized process i.e. the software should be deployed in the same way to different environments
16. Organizational support is needed in order to successfully adopt continuous delivery
17. Successful usage of test automation requires a test strategy

Table 4.2: Survey Results Categorized by the Responses Gathered

as part of their development and did not see the benefits of committing their code changes directly to the mainline of the version control system. This controversy is also recognized in the literature [46]. However, the majority of the respondents agreed that the code changes should be integrated with the mainline at least daily. Therefore, this does not necessarily conflict with the findings from the literature review. However, it can be argued that having feature branches easily discourages this behavior as incomplete changes can be committed to the feature branches which can cause that the code is not constantly in a deliverable state. Also working in branches does not as such encourage frequent integrations with the mainline, and therefore this practice would require discipline from the developers.

One of the respondents stated that extensive use of branches should be applied. He had two arguments for this. Firstly, he stated that this workflow would allow keeping up better code quality through code reviews. One way how they can be applied is having people reviewing the code changes made to a branch before the branch is merged with the mainline. This process is supported by different version control providers such as Github[1] and Deveo[2]. However, code reviews can also be performed without the use of branches, more precisely on a commit level with tools such as Gerrit[3]. In this workflow, commits are first pushed to a repository containing pending changes. Only when the pending changes have been reviewed and accepted, they are automatically submitted to the actual mainline of the project. The usage of Gerrit is not in the scope of this thesis, but more information about this workflow can be found on Gerrit's web page[4].

The other argument that the respondent stated in favor of extensive usage of branching was isolation of problem when they arise. This, however, is something that the literature suggest is better achieved the opposite way - by avoiding feature branches. This requires that also other best practices of continuous delivery are being followed, such as frequent deployments and thorough test automation. If the software's deployment pipeline and testing that is executed on it is on good level, the errors that single changes introduce are found early if it breaks tested functionality. Frequent deployments enable having smaller changes deployed into production at once, so errors that might occur there can also be easier isolated.

[1] https://help.github.com/articles/using-pull-requests

[2] http://blog.deveo.com/code-review-workflow-in-deveo

[3] https://www.gerritcodereview.com

[4] https://gerrit-documentation.storage.googleapis.com/Documentation/2.12.3/intro-quick.html

Two closely related statements that most of the respondents agreed on but which also got some disagreeing responses were related to when the code should be tested (Statements 3 and 4). The majority of the respondents agreed that untested code should not be committed to the version control system and that a TDD approach should be applied. During the presentation one of the participants also highlighted that committing changes locally can be done without testing the changes immediately, but when the code changes are pushed to the central version control repository they should also include tests.

All the respondents who disagreed with this were also in favor of developing code in feature branches rather than directly in the mainline. This relation is shown in Figure 4.4. From this, an assumption can be made that these respondents prefer developing their features in separate branches and also test the features before the branch is merged with the mainline i.e. testing can be done after the changes have been implemented but before integrating them with others or releasing the code into production. Whichever the reasoning behind this is, it conflicts with the main idea behind continuous delivery: keeping the software continuously in a releasable state.

The last statement that received a few disagreeing responses was about configuration management (statement 5 in Table 4.2). However, the vast majority of the respondents (13/20) strongly agreed with the statement. One important thing to note here is related to a comment that was written by one of the respondents, who stated that in his opinion binaries of tools which are publicly available should not be stored in version control. Instead of actual binaries, configuration files and scripts should be stored, which document the version numbers of the tools used and possibly automate their installation process in which the binaries itself are fetched from an other location.

One thing that was highlighted by two of the respondents was that proper communication and cooperation is essential for successful usage of continuous delivery. This is most certainly true. In order to facilitate an environment in which everyone feels responsible for keeping the code in a deliverable state and in which the code is deployed frequently into production, communication between the different people involved in the project has to be on a good level. Also, cooperation between developers and IT operation people has to be working. Fowler [5] extends this statement to everyone involved in the delivery.

Respondent ID	Version Control: Branching		Testing		Integration	
	Everyone should commit their work directly to the mainline of the version control system	Feature branches should be avoided	Test driven development is an essential practice in continuous delivery	Untested code should not be committed to the version control system	Everyone should integrate their work with the mainline at least daily	Developers should integrate their changes frequently
1	Strongly disagree	Disagree	Strongly disagree	Disagree	Disagree	Strongly agree
2	Disagree	Disagree	Strongly disagree	Disagree	Disagree	Agree
3	Strongly disagree	Disagree	Disagree	Strongly disagree	Strongly agree	Strongly agree
4	Strongly disagree	Strongly disagree	Disagree	Strongly agree	Neither agree nor	Agree
5	Disagree	Disagree	Agree	Agree	Agree	Agree
6	Disagree	Disagree	Agree	Neither agree nor	Agree	Strongly agree
7	Strongly disagree	Strongly disagree	Neither agree nor	Neither agree nor	Neither agree nor	Agree
8	Strongly disagree	Disagree	Strongly agree	Agree	Neither agree nor	Agree
9	Disagree	Neither agree nor	Agree	Agree	Agree	Strongly agree
10	Disagree	Neither agree nor	Agree	Agree	Agree	Agree
11	Disagree	Agree	Neither agree nor	Strongly agree	Agree	Strongly agree
12	Strongly agree	Strongly agree	Strongly agree	Agree	Strongly agree	Strongly agree
13	Agree	Agree	Agree	Agree	Agree	Strongly agree
14	Agree	Agree	Agree	Agree	Agree	Strongly agree
15	Agree	Agree	Agree	Agree	Agree	Strongly agree
16	Agree	Strongly agree	Neither agree nor	Strongly agree	Strongly agree	Strongly agree
17	Strongly agree	Strongly agree	Strongly agree	Neither agree nor	Strongly agree	Strongly agree
18	Neither agree nor	Neither agree nor	Strongly agree	Neither agree nor	Strongly agree	Strongly agree
19	Neither agree nor	Neither agree nor	Agree	Neither agree nor	Neither agree nor	Agree
20	Neither agree nor	Neither agree nor	Strongly agree	Strongly agree	Neither agree nor	Neither agree nor

Figure 4.4: Relation Between Answers on Branching-, Testing- and Integration Approaches

4.3 Experienced Benefits with Continuous Delivery

In this section, the benefits of continuous delivery are presented based on the interviews performed. First, in Section 4.3.1, the experienced benefits of continuous delivery are shown and after that, in Section 4.3.2 the challenges experienced in Project B – which could have been solved by the usage of continuous delivery – are discussed.

4.3.1 Benefits of Continuous Delivery Experienced in Project A

The interviewees felt that continuous delivery increases the confidence towards the product and its development- and deployment process. The increased confidence was explained by well-built automated regression tests and identical development-, test- and production environments.

The interviews suggested that continuous delivery has a positive effect on the transparency of the development process. This is explained by how continuous delivery facilitates gathering of different metrics related to the process. One example given by Developer 1 was how requirements and their acceptance tests could be directly connected with each other. This way different stakeholders would be able to follow how well the project is proceeding.

Reliable deployments were listed as one important benefit in the interview. The reason was that because the product deployment is done automatically and more frequently, there is less space for human errors during the process and the bugs are found early on. Flexible deployments were not explicitly mentioned in the interviews. One of the reasons for this can be that the software has not yet been published to its end users and therefore the need for deploying the software when ever needed has not yet come forwards. On the other hand, the respondents would most likely agree that flexible deployments are one benefit of continuous delivery if explicitly asked.

> By maximizing the small errors by deploying frequently, you end up minimizing the big failures.
> *–Developer 1*

According to the interviews, continuous delivery facilitates better use of resources. More precisely, the benefits of test automation were discussed. By having a thorough automated regression test suite in place, the developers were able to focus on exploratory testing of new features rather than assuring that the old functionality still functioned. Also automation of the deployment process was mentioned as one benefit.

Interviewees felt that continuous delivery affects customer satisfaction positively. This has a close relation to other benefits which were contributed to continuous delivery. For example, reduced cycle time and process transparency were listed as reasons for good customer satisfaction. Also faster feedback was mentioned as a benefit. This applies on multiple levels. Firstly, the developers get quick feedback whether the newest changes have broken the existing functionality and that their changes also work in more

production-like environments. Secondly, the developers are able to get continuous feedback from the customers or other stakeholders about whether the newest functionality developed meets its original requirements.

One benefit that came up during the interviews was that continuous delivery, more precisely its deployment pipeline, works as a documentation of the project. This was not mentioned in the literature as such, however Udd [37] stated that continuous delivery lowers the risks of losing an employee from the project since the deployment and testing are highly automated. All in all, one can state that both literature and the interviews agree that by applying continuous delivery, project is less dependent on the persons working with the project, because all the critical steps are automated – and this way also documented.

> No written documentation can replace a human who is explaining how something is done or a working deployment automation whose configuration works as an up-to-date documentation.
> *–Developer 1*

Reduced long term costs is a benefit that was mentioned explicitly in the interviews and is also one important factor listed in literature. This is a result of the impact of the several other benefits that continuous delivery introduces, such as high level of automation and reliability.

> Creation of a continuous delivery pipeline takes time and requires technical skills, but it will pay itself back and can be reused in other projects.
> *–Developer 1*

One thing that was mentioned multiple times during one interview was that continuous delivery facilitates good software development practices. According to the Developer 1, developers have to be proud of the code they write and therefore feel responsible for the quality also. If a developer produces bad quality or untested code, it will rather soon be recognized when the product is deployed, which should be an embarrassing moment for the developer.

Developer 2 stated that the speed and easiness are visible benefits when comparing the Project A with Project B. The developers are able to get fast feedback on their newly created changes and this way continuously assure that they have not broken existing functionality of the software.

> The best thing has been the easiness and the incredible speed of feedback.
> *–Developer 2*

An overview of the benefits that came up during the interviews are listed in Table 4.3.

Benefit	Interviewee			Lit.
	Dev 1	Dev 2	Ops 1	
Flexible deployments				X
Reduced cycle time	X			X
Faster feedback	X	X		X
Better use of resources	X			X
Decreases costs in the long run	X		X	X
Increased process transparency	X	X		X
Increased confidence	X	X		X
Reliable releases	X			X
Improved customer satisfaction	X			X
Quality of regression testing done by computers is better than by human	X			X
Documented and less developer-dependent deployment process	X			X
Embraces good software development practices	X			
Development is easier		X		

Table 4.3: Interview Results – RQ 2 Benefits

4.3.2 Challenges Experienced in Project B

The interviews revealed multiple challenges with Project B. The biggest challenges mentioned was that the deployment process of the project involved multiple manual steps and was considered error-prone and stressful. The projects deployment process is visualized in Figure 4.2.

Because the development-, test- and production environments were not identical, ensuring that the software will function in production after a change has been made was not bullet-proof even after testing it in the test environment. Testing of the software was time consuming, because no comprehensive end-to-end regression tests existed and the software therefore had to be tested manually after every change.

> This [Project B] was a typical example of a ”works on my machine”-project.
> *–Developer 2*

As all the previous developers who had implemented the software were not working at the company anymore, the only source of information about the project was its documentation and source code. First challenge experienced was that around 50 branches existed in the version control system, but it was not documented which branch or which version of the product had been deployed into production. Besides that, the documentation for the deployment and build process was inaccurate. This increased the cycle time of fixing the bug.

> If continuous delivery would have been in use, it would have taken us one hour instead of three weeks to get the bug fixed and deployed into production.
> *–Developer 1*

The challenges of Project B which according to literature and opinions of the interviewees could have been solved by applying continuous delivery are listed in Table 4.4.

Challenge	**CD would have solved**	
	Interview	**Literature**
Error-prone releases	X	X
Long cycle time	X	X
Time consuming manual testing before release	X	X
Problems caused by leaving employees	X	X
Stressful releases		X

Table 4.4: Challenges of Project B

4.4 Experienced Challenges with Continuous Delivery

The main challenge mentioned in the interviews was that continuous delivery is not an easy process to set up. Setting up a proper deployment pipeline is technically challenging and besides ordinary development tasks, requires operational skills such as knowledge from virtualization and containerization technologies.

> For an average developer it [setting up a deployment pipeline from scratch] is a pretty difficult process.
> *–Ops 1*

The continuous delivery pipeline infrastructure that was used in Project A was the result of a third attempt of building such a pipeline infrastructure, which highlights the technical challenges that continuous delivery introduces. Besides technical challenges also skilled developers are needed. According to Developer 1, it is not enough to have continuous delivery working at a technical level, also good development practices have to be applied. Due to these reasons new developers who have not applied continuous delivery before may need training, either to the development practices or technologies used.

The delivery pipeline was still under development when Project A took it in to use. This caused some problems and additional work, because Project A was not always up-to-date with the newest pipeline setup. Even though this problem would not exist if the setup of the pipeline did not evolve, but it highlights that also with continuous delivery maintenance is required.

Even though Project A adapted continuous delivery from early on and people were satisfied with the level of test automation, issues still existed. Developer 2 explained that the coverage of end-to-end acceptance tests could be larger, but he felt that there is not always enough time to implement those. In the literature, this is categorized as an organizational issue. The technical debt of a product grows if new features are prioritized over quality, which can cause problems with continuous delivery which heavily relies on proper automated testing.

One challenge that was highlighted by Ops 1 was how could continuous delivery, and more precisely the pipeline that had been created, be taken into use in more projects in the company. Currently the pipeline had been taken into use mainly in project which included active involvement by people who contributed to the creation of the deployment pipeline. The interviewee also stated it is not trivial to create a deployment pipeline infrastructure that would be easy to use even for people with no experience with the used technologies.

> On the other side [one challenge is] how we could develop it [the deployment pipeline] even more further because at this moment it is still a bit difficult to use. There are certain things one has to know and anyway, we can't hide everything [technical complexity] there in the background.
> *–Ops 1*

The challenges that were experienced while applying and adopting continuous delivery in Project A are listed in Table 4.5.

Challenge	**Interviewee**			**Lit.**
	Dev 1	**Dev 2**	**Ops 1**	
High short-term cost	X	X	X	X
Technically challenging	X	X	X	X
Skilled developers needed	X			X
Thorough acceptance testing requires time, which is not always available		X		X
Evolving deployment pipeline		X		
Environment inconsistencies				X
Organizational challenges				X
Customer- and third party-related challenges				X
Complex and large software or legacy code				X

Table 4.5: Interview Results – RQ 2 Challenges

4.5 Summary of the Results

The deployment processes of the two projects differentiated mainly on how much automation was involved. The deployment process of Project A was highyl automated, where as in Project B it involved multiple manual steps. Also, in Project A developers were able to automatically create identical environments from scratch and all the needed dependencies were documented. This was not the case for Project B in which the different environments were not identical and had to be set up manually.

The interviews highlighted that one key needed to succeed with continuous delivery is having the developers feeling responsible for the code they de-

velop. This feeling of responsibility enforces the creation of good quality code and proper testing. TDD was experienced as being a good approach when applying continuous delivery, but the results of the interviews also suggested that the decision of using such development approaches should come from the developers themselves, as too many predefined processes can have a negative effect on the developers' feeling of responsibility.

The results also indicated that effort should be put into creating a deployment pipeline that can be utilized in other projects as well. This way other projects can benefit from the work done on the existing deployment pipeline, and lower their starting costs and effort. In addition, fast feedback, frequent deployments and automation of the deployment process were considered as being important.

The results of the survey supported many findings of the literature review and interviews. The most controversial statements were related to continuous integration practices. Many respondents disagreed with the statement that branching should be avoided. However, the statement that code should be integrated frequently with the mainline of the version control system was generally agreed upon.

According to the interviews continuous delivery was considered to increase the confidence towards the development- and deployment process. It was also stated that continuous delivery improves the use of the available resources as many of the repetitive and and error-prone processes have been automated. This also makes the project easier to maintain later on when the knowledge needed to build and deploy the application is documented in the automation scripts. In addition, the interviewees felt that continuous delivery had an positive effect on customer satisfaction and reduced long term costs. Lastly, the interviewees stated that continuous delivery embraces good software development practices.

The results of the interviews indicated that continuous delivery has a higher initial cost, is technically challenging and requires skilled developers. As the creation of comprehensive automated tests requires more than manual testing in the short term, one of the interviewees felt that the coverage of automated end-to-end test cases should be higher. This pinpoints the problem of how difficult it can be to define a test strategy that is both feasible to implement and comprehensive enough.

Chapter 5

Discussion

This chapter discusses the research questions of this thesis based on results of the empirical study and literature review. RQ 1 is answered in Section 5.1 and RQ 2 in Section 5.2.

5.1 Changes to Projects Required by Continuous Delivery

Successfully adopting continuous delivery in a software project requires fulfilling manifold requirements. This is because continuous delivery affects the whole process of developing a software – starting from implementing a change to having the change deployed into production. For example, continuous delivery affects how the software needs to be tested, how features have to be developed and how the software is to be built and deployed.

The different building blocks of continuous delivery can be summarized as version control, configuration management, build-, deployment- and test automation. In addition, the following three principles guide the development process to a direction that enables the usage of continuous delivery: building quality in, creation of a reliable and repeatable software release process and embracing shared responsibility. These principles include multiple practices, many of which originate from continuous integration which can be seen as a base and prerequisite for continuous delivery.

All in all, automation of manual tasks plays an important role in continuous delivery. Having time consuming and error-prone steps automated, the risks

related to these decrease. Also, it makes the project less dependent on individuals. The benefits of automation were clearly visible when comparing the deployment processes of the studied projects of this thesis.

To fully automate the quality assurance process of a project can be a challenging task. Many literature sources – such as Udd [37], Chen [4], Gmeiner et al. [8], Laukkanen et al. [47], Leppänen et al. [42] and Neely and Stolt [3] – state that manual quality assurance of some sort was performed before deploying the software into production in the studied projects. However, some examples of projects which have fully automated their quality assurance process exist in the literature [42, 48]. This indicates that removing all manual quality assurance steps is possible, but has to be separately considered depending on the project and its quality assurance requirements.

One of the building blocks of successful adoption of continuous delivery is having everything that is needed to build, run, test and deploy the application stored in version control. This puts requirements on having a comprehensive configuration management in place. This way, it is always clear what applications, dependencies and operating systems are running on different environments. Having all of these configurations managed enables creation of identical environments. When configuration management is combined with automated setup and -provisioning of the servers, one can be sure that different environments are running exactly what has been defined. This lowers the risk of having bugs that are only visible in a certain environment. The deployment process itself should be standardized so that the software can be deployed to all environments in a consistent way.

Creating a reliable and repeatable deployment process is essential. This can be achieved by releasing the software frequently and by applying continuous integration as part of the development process. Deploying the software frequently into production enables frequent gathering of user feedback and solving deployment related issues early on. Nevertheless, they are necessarily not suitable for all projects. In some projects it can be technically impossible to deploy the software frequently, but in others customers might not even want it. Although frequent software releases are not applicable to all projects, even then the software should be frequently delivered to a production-like environment, such as a staging server. The staging server can then be used to perform manual quality assurance on the software before it is eventually released to its end users.

Continuous integration is a practice that is nowadays generally used in software projects. Then again, what is actually defined as being continuous integration varies. Having a CI server running does not mean that continu-

ous integration is being correctly performed. The key is frequent integration of code. When it comes to continuous integration and continuous delivery, the results of this thesis show that version control practices seem to be something that people have the strongest opinions on. Even though the literature reviewed in this thesis indicates that development should be done in a single branch in order to get the changes integrated frequently, many developers disagree with this branching strategy and prefer using feature branches instead. This controversy is also recognized in the literature.

Building quality in is a principle that has to be understood by all stakeholders. It affects how software is to be developed in a way that developers have to invest time into quality assurance from the start of the project. In practice this means that before changes are integrated with the mainline of the version control system, automated tests have to be created as well. In addition, the priority of new features should always be considered lower than keeping the software in a releasable state. This requires a new mindset from the managers and customers as well, since they have to understand that by putting pressure on the developers in order to get some features implemented before a deadline easily can have a negative effect on the quality of the software and this way compromise its deliverability.

Continuous delivery facilitates better communication and cooperation between different stakeholders, both internally and externally. Nevertheless, it is important to note that proper communication can also be seen as a prerequisite for a successful adoption of continuous delivery. In other words, in order to successfully utilize continuous delivery, project's stakeholders have to put effort into having a proper communication in place. Continuous delivery both supports and requires this. Good communication and spirit of collaboration embrace shared responsibility for the product, which is one of the key principles involved with continuous delivery. Shared responsibility means that all developers have to take care that the software is continuously in a deliverable state and that errors occurring during builds or deployments get fixed and enjoy top priority. For this reason continuous delivery can be an especially challenging process to adopt in large, bureaucratic organizations in which software development is divided to functional silos with little collaboration and communication. Then again, successfully adopting continuous delivery can help these organizations to adopt modern software development practices.

When adopting continuous delivery it is important to note that it does not have to – and often should not – occur at once. Even if continuous delivery was successfully being used in a project, its current practices would have to be

frequently evaluated. Improving the development and deployment processes is a constant effort and should adapt when the project matures. The best way to get people to understand the benefits of a process is to try it out in a project. This way the process can be analyzed whether it fits to the company's way of working or not. Best way to get people understand the benefits of a certain process is to see it in action in their domain.

Continuous delivery is a holistic approach. Project B already had adopted some practices of continuous delivery – such as build automation and configuration management to some extent – but still experienced problems. This underlines the importance of how continuous delivery combines all the practices into one working unit. Still, the adoption of a subset of the practices already has a positive effect on projects as was recognized in Project B. Nevertheless, in order to get all the benefits of continuous delivery, adopting single practices "here and there" is not enough.

The literature review provided a comprehensive overview on the changes to software projects that are required when adopting continuous delivery. Many of these results were also recognized in the interviews and agreed upon in the survey. However, a more long-lasting case study would have been needed in order to analyze, how the different processes and practices involved in the project evolve and mature. It is also important to note that the mentioned principles and practices serve as a good guideline, but the concrete actions that have to be taken into use may vary depending on the project and company. In addition, Project A had not yet been officially published and was of small size, and therefore the impact that releases or large projects have on the development and deployment practices could not be analyzed based on it. Then again, the empirical study involved people who have previous experience with continuous delivery from other projects and a strong background in software engineering and DevOps, and therefore their opinions on these topics are relevant.

5.2 Benefits and Challenges of Continuous Delivery

As can be seen from the results of this thesis, continuous delivery definitely has its benefits but also its challenges. The findings of the literature review are more comprehensive compared to the results of the interviews. This was an expected results since the existing research found in the literature is based on several projects, whereas the interviews were focused on one project which implemented continuous delivery. However, the results of the interviews and literature review support each other as all the benefits and challenges mentioned in the interviews have also been identified in the literature – either as part of continuous delivery or test automation.

Improved reliability and flexibility of deployments are central benefits of continuous delivery. Traditionally, releasing software has been a stressful and error-prone process, which are the key things continuous delivery tackles. In addition, continuous delivery decreases the cycle time in software projects. These benefits together make the software projects more adaptive, as customer feedback and bugs can be reacted on quickly. Being able to adapt to customer feedback quickly is especially useful in projects which are still under development, whereas being able to deploy bugfixes flexibly into production is practical also in projects, which are not under active development but under a support contract. From customer's point of view, visible benefits of continuous delivery are increased process transparency and improved product quality, which can also lead to improved customer satisfaction. In the literature at least the following sources reported an increase in the customer satisfaction in the studied projects: Leppänen et al. [42] and Neely and Stolt [3]. Nevertheless, more precise studies on this topic are missing and therefore this subject forms an interesting topic for further research.

Continuous delivery enables better use of resources and thus increases productivity and makes development more cost-efficient. However, continuous delivery is not just a switch that can be turned on with all its benefits immediately present. Taking it into use is a technically challenging process and might require an extensive change in how software is being developed in the organization. Thus, adopting continuous delivery in a project requires patience and a higher initial investment.

Due to technical challenges and changes in how the software is being developed, skilled and disciplined developers. The process also requires support at the organizational level: if not enough time and resources are allocated

to follow the different practices – such as comprehensive test automation – the technical debt grows and the trust in the product's quality suffers. Then again, if the requirements of the software are still evolving and thus frequently changing, a lot of rework and maintenance needs to be allocated in order to keep the tests up-to-date, which can be a time-consuming task. If automated tests are not maintained in the same pace as the software is updated, they cannot be trusted anymore and people will start ignoring them. On the other hand, in some cases it can be a reasonable decision to postpone, for example, the automated acceptance testing until the requirements are clear enough.

It is important that the stakeholders of a project understand that the benefits of continuous delivery appear in the long run. For example, creating automated test cases is often more time consuming than once executing the test manually, but will quickly outperform manual testing when the tests are executed on a frequent basis.

The extensive literature research raised manifold benefits and challenges, many of which were also mentioned in the interviews. In order to gather more detailed information on the benefits and challenges it would have been interesting to study a project which had been applying continuous delivery for a longer time and had more developers and stakeholders involved. Now the results of the empirical study related to the long-term benefits were based on the opinions of few interviewees and their experiences gained during a few months. However, one of the interviewees had utilized continuous delivery in other projects and this way was able to provide opinions on the long-term benefits as well. It is also noteworthy that most likely many of the benefits and challenges mentioned in the literature but not in the interviews, are factors that the interviewees would have agreed upon if explicitly asked.

Chapter 6

Conclusions

Continuous delivery affects software projects in a comprehensive way. It has an impact on how developers have to implement and test the software, how different teams have to collaborate with each other and lastly, how the software should be deployed into production. It discourages organizational silos and encourages shared responsibility and a mindset that quality has to be built in to the software rather than trying to add it afterwards at the end of the project.

Continuous delivery introduces different building blocks that are needed, many of which originate from continuous integration. Continuous delivery relies heavily on automation of manual tasks. For example, build-, test and deployment automation are an essential part of continuous delivery. In addition, configuration management plays an important role. The information related to everything that is needed to build, run, test and deploy the software has to be version controlled.

Continuous delivery makes the process of deploying software reliable and repeatable. By deploying software frequently, less risks are involved with a single deployment. Frequent deployments also facilitate gathering of customer feedback, which again helps developing software that actually creates value to the customer. Customer feedback can also be reacted on more quickly as continuous delivery reduces the cycle time of software projects.

Adopting continuous delivery comes with a higher initial investment and can be a challenging process to implement successfully. Nevertheless, it also comes with many benefits that often in the long run make the effort put into adopting the practice worthwhile. Even though continuous delivery gives a high priority on test automation, also here the effort needed to automate

all the testing has to be evaluated. Often some form of manual quality assurance is still needed before software can be released into production, as automating the whole quality assurance process would not be possible or feasible to implement.

One interesting topic for further research would be analyzing how quickly investments made for adopting continuous delivery pay themselves back in different sized and different kind of projects. This could give insight on what are the most costly practices to adopt and difficult technical challenges to solve in different domains, and this way also provide instructions on how this can be achieved. Furthermore, it would be interesting to get more case studies of companies adopting continuous delivery and to compare how their processes differ from each other and to analyze what have been the practices that developers have found most helpful. Additional case studies could also give insight on how well different projects have been able to automate their testing and what – if any – have been the manual quality assurance stages of the projects.

Bibliography

[1] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.

[2] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples.* John Wiley & Sons, 2012.

[3] Steve Neely and Steve Stolt. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In *Agile Conference (AGILE), 2013*, pages 121–128. IEEE, 2013.

[4] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015.

[5] Martin Fowler. Continuous delivery. `http://martinfowler.com/bliki/ContinuousDelivery.html` Accessed 08.06.2016, May 2013.

[6] Martin Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html` Accessed 08.06.2016, May 2006.

[7] Ville Pulkkinen. Continuous deployment of software. In *Proc. of the Seminar no. 58312107: Cloud-based Software Engineering*, pages 46–52, 2013.

[8] Johannes Gmeiner, Rudolf Ramler, and Julian Haslinger. Automated testing in the continuous delivery pipeline: A case study of an online company. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–6. IEEE, 2015.

[9] Martin Fowler. Phoenixservers. `http://martinfowler.com/bliki/PhoenixServer.html` Accessed 16.06.2016, July 2012.

[10] Martin Fowler. Snowflakeserver. `http://martinfowler.com/bliki/SnowflakeServer.html` Accessed 16.06.2016, July 2012.

[11] Glendford J. Myers, Tom Badgett, and Corey Sandler. *The art of software testing*. John Wiley & Sons, Inc., 2012.

[12] Elfriede Dustin, Thom Garrett, and Bernie Gauf. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education, 2009.

[13] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[14] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *2009 International Conference on Software Testing Verification and Validation*, pages 201–209. IEEE, 2009.

[15] S. Koirala and S. Sheikh. *Software Testing*. Jones & Bartlett Learning, 2009.

[16] James Bach. Agile test automation. `http://satisfice.com/articles/agileauto-paper.pdf` Accessed 06.06.2016, 2003.

[17] Stefan Berner, Roland Weber, and Rudolf K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th International Conference on Software Engineering*, pages 571–579. ACM, 2005.

[18] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Computer*, 34:135–137, 2001.

[19] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[20] Dorothy Graham and Mark Fewster. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley, 1999.

[21] Mariusz A Fecko and Christopher M Lott. Lessons learned from automating tests for an operations support system. *Software: Practice and Experience*, 32(15):1485–1506, 2002.

[22] Borge Haugset and Geir Kjetil Hanssen. Automated acceptance testing: A literature review and an industrial case study. In *Agile, 2008. AGILE'08. Conference*, pages 27–38. IEEE, 2008.

[23] Tom Wissink and Carlos Amaro. Successful test automation for software maintenance. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 265–266. IEEE, 2006.

[24] Roy Patrick Tan and Stephen Edwards. Evaluating automated unit testing in sulu. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 62–71. IEEE, 2008.

[25] Mohammad Alshraideh. A complete automation of unit testing for javascript programs. *Journal of Computer Science*, 4(12):1012, 2008.

[26] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a. IEEE, 2007.

[27] Paul Pocatilu. Automated software testing process. *Economy Informatics*, 2(1):97–99, 2002.

[28] Brian Marick. How to misuse code coverage. *Testing Foundations* `http://www.exampler.com/testing-com/writings/coverage.pdf`, 1997.

[29] Martin Fowler. Testcoverage. `http://martinfowler.com/bliki/TestCoverage.html` Accessed 20.06.2016, April 2012.

[30] Torben Rick. Top 12 reasons why people resist change. `http://www.torbenrick.eu/blog/change-management/12-reasons-why-people-resist-change/` Accessed 07.07.2016.

[31] Cem Kaner. Improving the maintainability of automated test suites. *Software QA*, 4(4), 1997.

[32] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[33] Daniel Ståhl and Jan Boschb. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

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

[35] Martin Fowler. Deploymentpipeline. `http://martinfowler.com/bliki/DeploymentPipeline.html` Accessed 08.06.2016, May 2013.

[36] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[37] Raoul Udd. Adopting continuous delivery: A case study. Master's thesis, Aalto University, 2016.

[38] n.a. Continuous delivery: Unit tests. `https://technologyconversations.com/2014/06/10/continuous-delivery-unit-tests/` Accessed 21.06.2016, June 2014.

[39] Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and software Technology*, 46(5):337–342, 2004.

[40] Ken Pugh. *Lean-Agile Acceptance Test-Driven Development: Better Software Through Collaboration*. Pearson Education, 2010.

[41] Martin Fowler. Featurebranch. `http://martinfowler.com/bliki/FeatureBranch.html` Accessed 21.06.2016, September 2009.

[42] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The speedways and country roads to continuous deployment. *IEEE*, 32(2):64–72, 2015.

[43] 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, 2015.

[44] Timothy Fitz. Continuous deployment. `http://timothyfitz.com/2009/02/08/continuous-deployment/` Accessed 16.06.2016, February 2009.

[45] Jez Humble. Continuous delivery vs continuous deployment. `https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/` Accessed 16.06.2016, August 2010.

[46] Alanna Brown, Nigel Kersten, Nicole Forsgren, Gene Kim, and Jez Humble. State of devops report, 2016.

[47] Eero Laukkanen, Timo O.A. Lehtinen, Juha Itkonen, Maria Paasivaara, and Casper Lassenius. Bottom-up adoption of continuous delivery in a stage-gate managed software organization. Technical report, 2016.

[48] Timothy Fitz. Continuous deployment at imvu: Doing the impossible fifty times a day. `http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day` Accessed 17.08.2016, February 2009.

Appendix A

Project Interview Questions

Project A

1. Describe the project in you own words
2. Describe the process of deploying a change from an idea to production
3. Describe the project's deployment pipeline
4. How much effort did it require to set up the deployment pipeline
5. How were the different environments and the software's dependencies managed?
6. How was quality assurance done in the project?
7. Were you able to confidently deploy a change to production if it passed the tests
8. What went well in the project?
9. What were the challenges in the project?
10. How could have these challenges been avoided?

Project B

11. Describe the project in you own words
12. Describe the process of deploying a change from an idea to production
13. What manual stages were involved?
14. What were the most error-prone stages?

15. How were the different project dependencies managed in the different environments?
16. How was quality assurance done in the project?
17. Were you able to confidently deploy a change to production if it passed the tests
18. What went well in the project?
19. What were the challenges in the project?
20. How could have these challenges been avoided?

Comparing Project A with Project B

21. What benefits were introduced by continuous delivery in Project A?
22. What development practices had the biggest impact on the project's success in Project A?
23. How would you compare process transparency towards the customer visibility and customer satisfaction in these two projects?

Appendix B

Ops Interview Questions

1. What was the goal?
2. Explain how the pipeline works
3. Did you have any alternative technologies or implementations that you were considering for the pipeline
4. How challenging was it to implement the pipeline?
5. Would an average developer be able to implement it?
6. What were the most challenging parts to implement?
7. How long did the whole process take?
8. What were the most important design decisions?
9. How well was the goal achieved?

Appendix C

Survey

Did you see the presentation about continuous delivery on Friday 22.7.2016?

yes	☐
no	☐

Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "

	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly Disagree
Version Control	☐	☐	☐	☐	☐
Build Automation	☐	☐	☐	☐	☐
Test Automation	☐	☐	☐	☐	☐
Continuous Integration	☐	☐	☐	☐	☐
Configuration Management	☐	☐	☐	☐	☐
Deployment Automation	☐	☐	☐	☐	☐

When talking about continuous delivery, select the option that best describes your opinion on the statement?

	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly Disagree
Feature branches should be avoided	☐	☐	☐	☐	☐
Everyone should commit their work directly to the mainline/trunk/main-branch of the version control system	☐	☐	☐	☐	☐
Everyone should integrate their work with the mainline at least daily	☐	☐	☐	☐	☐
Everything needed to build and deploy an application should be stored in version control	☐	☐	☐	☐	☐
Setting up an environment in which the software is run should be an automated process	☐	☐	☐	☐	☐
Deployment process should be automated	☐	☐	☐	☐	☐
Build process should be automated	☐	☐	☐	☐	☐

Untested code should not be committed to the version control system	☐	☐	☐	☐	☐
Everyone is responsible for keeping the software in a deliverable state	☐	☐	☐	☐	☐
Fixing of broken builds has the highest priority	☐	☐	☐	☐	☐
Deliverable software should be prioritized over new functionality	☐	☐	☐	☐	☐
Deployment should be a standardized process i.e. the software should be deployed in the same way to different environments	☐	☐	☐	☐	☐
The software should be deployed frequently into production	☐	☐	☐	☐	☐
Organizational support is needed in order to successfully adopt continuous delivery	☐	☐	☐	☐	☐
Test driven development is an essential practice in continuous delivery	☐	☐	☐	☐	☐
Successful usage of test automation requires a test strategy	☐	☐	☐	☐	☐
Developers should integrate their changes frequently	☐	☐	☐	☐	☐

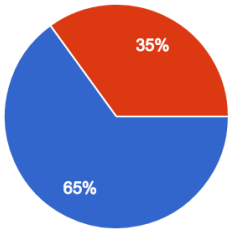
(Optional) Provide clarifications to your answers. Please also write down which answer the clarification is connected to

If not already listed in this survey, mention practices or principles that you consider being essential for successful adoption and usage of continuous delivery

Appendix D

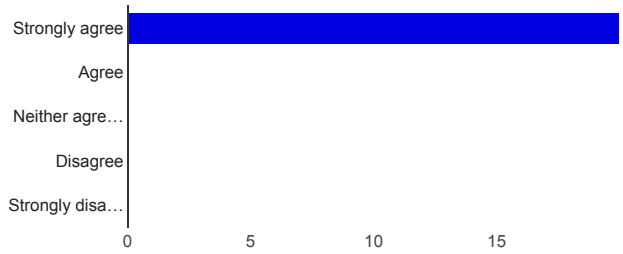
Summary of the Results of the Survey

Did you see the presentation about continuous delivery on Friday 22.7.2016?



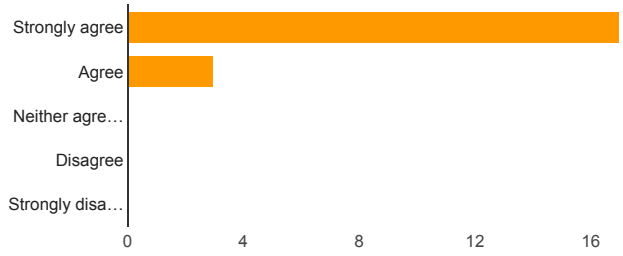
Yes	13	65%
No	7	35%

Version Control [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



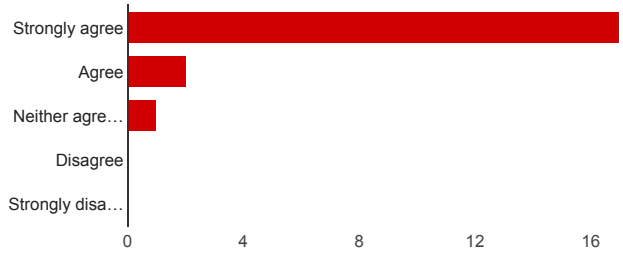
Strongly agree	**20**	100%
Agree	**0**	0%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Build Automation [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



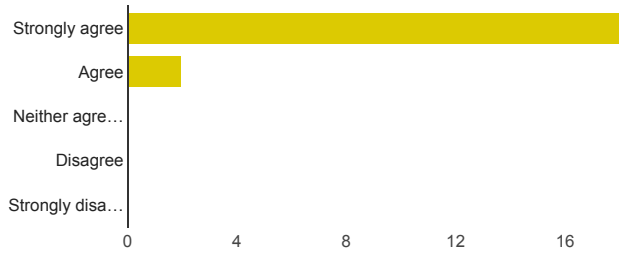
Strongly agree	**17**	85%
Agree	**3**	15%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Test Automation [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



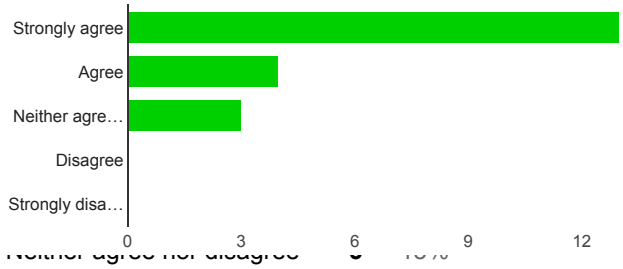
Strongly agree	**17**	85%
Agree	**2**	10%
Neither agree nor disagree	**1**	5%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Continuous Integration [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



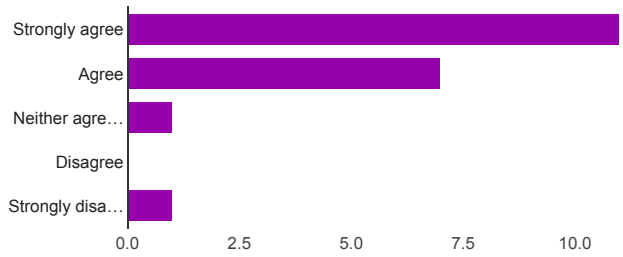
Strongly agree	**18**	90%
Agree	**2**	10%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Configuration Management [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



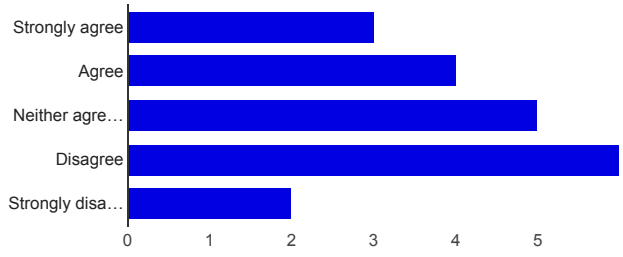
Disagree	**0**	0%
Strongly disagree	**0**	0%

Deployment Automation [Select an option that best describes your opinion on the statement: "The following topic is an important part of continuous delivery: ... "]



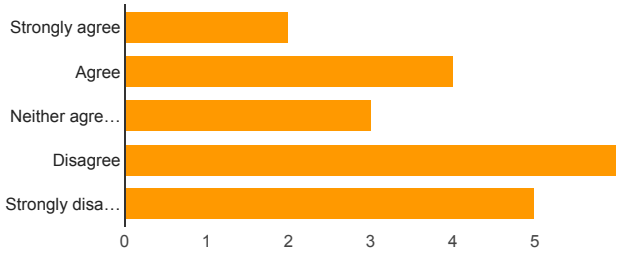
Strongly agree	**11**	55%
Agree	**7**	35%
Neither agree nor disagree	**1**	5%
Disagree	**0**	0%
Strongly disagree	**1**	5%

Feature branches should be avoided [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



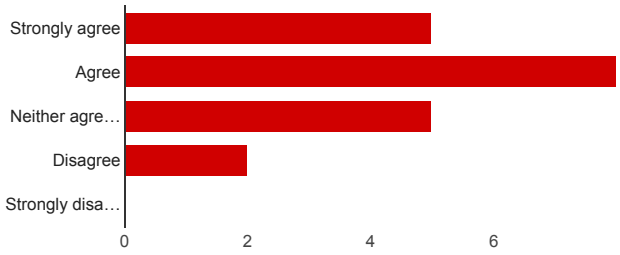
Strongly agree	**3**	15%
Agree	**4**	20%
Neither agree nor disagree	**5**	25%
Disagree	**6**	30%
Strongly disagree	**2**	10%

Everyone should commit their work directly to the mainline/trunk/main-branch of the version control system [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**2**	10%
Agree	**4**	20%
Neither agree nor disagree	**3**	15%
Disagree	**6**	30%
Strongly disagree	**5**	25%

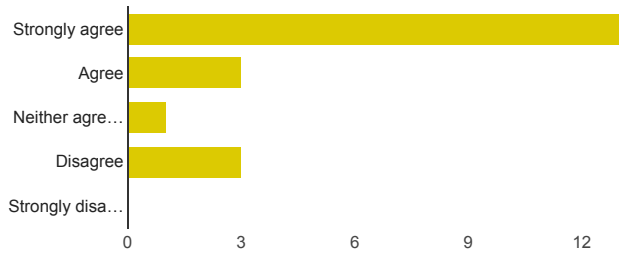
Everyone should integrate their work with the mainline at least daily [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**5**	25%
Agree	**8**	40%
Neither agree nor disagree	**5**	25%
Disagree	**2**	10%

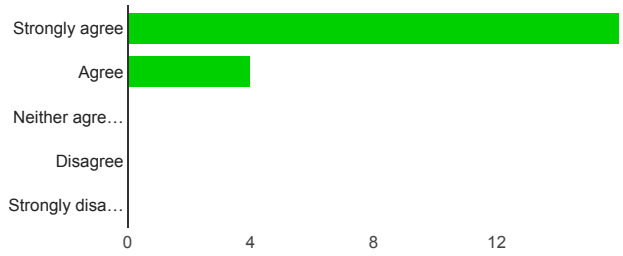
Strongly disagree	**0**	0%

Everything needed to build and deploy an application should be stored in version control [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



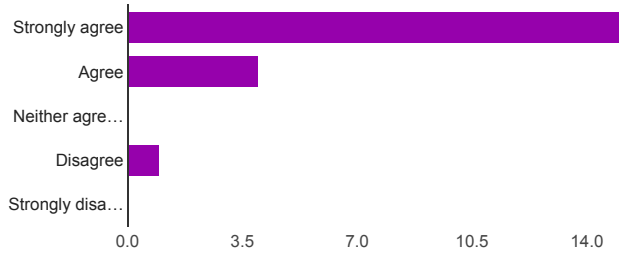
Strongly agree	**13**	65%
Agree	**3**	15%
Neither agree nor disagree	**1**	5%
Disagree	**3**	15%
Strongly disagree	**0**	0%

Setting up an environment in which the software is run should be an automated process [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



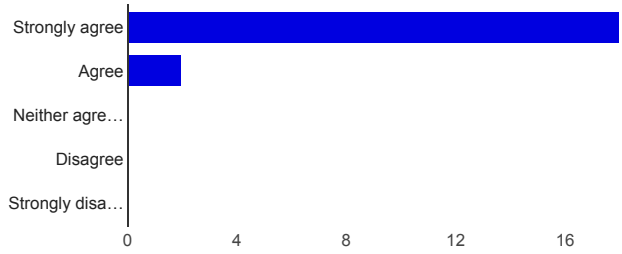
Strongly agree	**16**	80%
Agree	**4**	20%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Deployment process should be automated [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



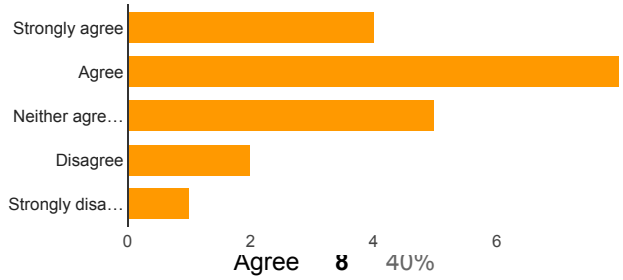
Strongly agree	**15**	75%
Agree	**4**	20%
Neither agree nor disagree	**0**	0%
Disagree	**1**	5%
Strongly disagree	**0**	0%

Build process should be automated [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



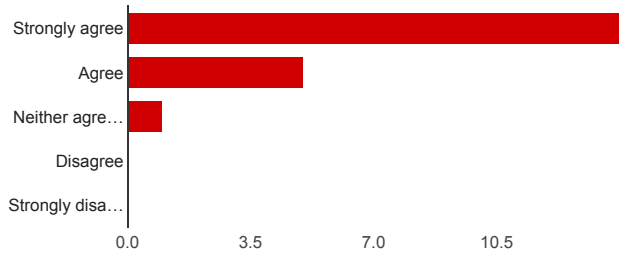
Strongly agree	**18**	90%
Agree	**2**	10%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Untested code should not be committed to the version control system [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



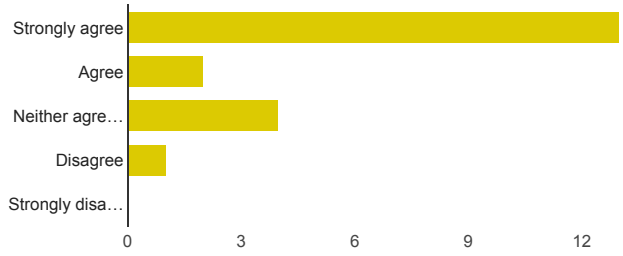
Agree	**8**	40%
Neither agree nor disagree	**5**	25%
Disagree	**2**	10%
Strongly disagree	**1**	5%

Everyone is responsible for keeping the software in a deliverable state [When talking about continuous delivery, select the option that best describes your opinion on the statement?]

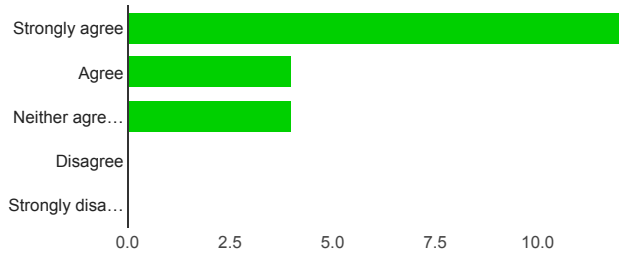


Strongly agree	**14**	70%
Agree	**5**	25%
Neither agree nor disagree	**1**	5%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Fixing of broken builds has the highest priority [When talking about continuous delivery, select the option that best describes your opinion on the statement?]

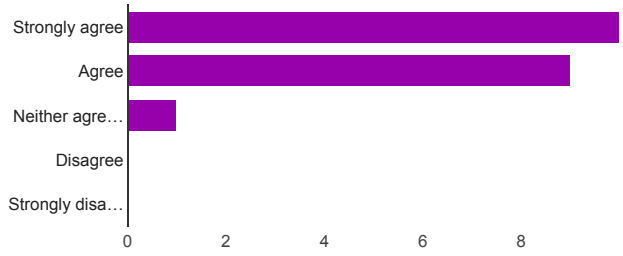


Deliverable software should be prioritized over new functionality [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**12**	60%
Agree	**4**	20%
Neither agree nor disagree	**4**	20%
Disagree	**0**	0%
Strongly disagree	**0**	0%

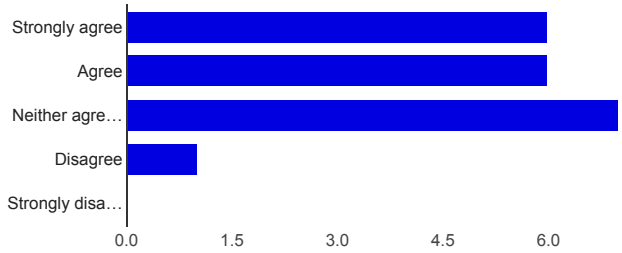
Deployment should be a standardized process i.e. the software should be deployed in the same way to different environments [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**10**	50%

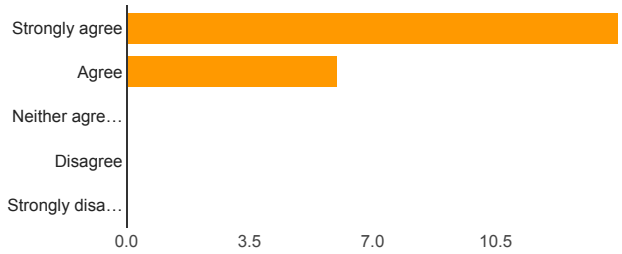
Agree	**9**	45%
Neither agree nor disagree	**1**	5%
Disagree	**0**	0%
Strongly disagree	**0**	0%

The software should be deployed frequently into production [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



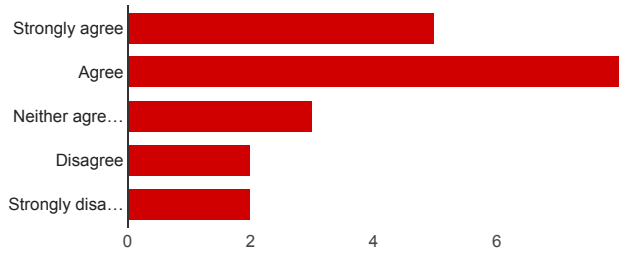
Strongly agree	**6**	30%
Agree	**6**	30%
Neither agree nor disagree	**7**	35%
Disagree	**1**	5%
Strongly disagree	**0**	0%

Organizational support is needed in order to successfully adopt continuous delivery [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



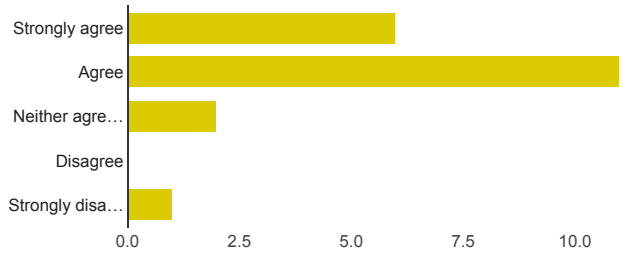
Strongly agree	**14**	70%
Agree	**6**	30%
Neither agree nor disagree	**0**	0%
Disagree	**0**	0%
Strongly disagree	**0**	0%

Test driven development is an essential practice in continuous delivery [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**5**	25%
Agree	**8**	40%
Neither agree nor disagree	**3**	15%
Disagree	**2**	10%
Strongly disagree	**2**	10%

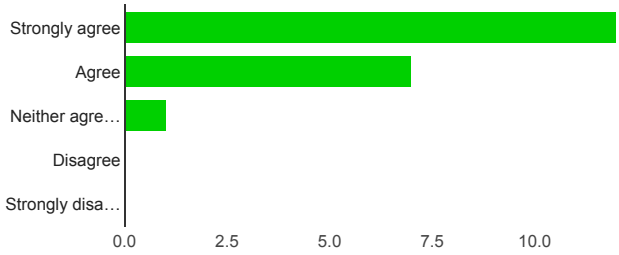
Successful usage of test automation requires a test strategy [When talking about continuous delivery, select the option that best describes your opinion on the statement?]



Strongly agree	**6**	30%
Agree	**11**	55%
Neither agree nor disagree	**2**	10%
Disagree	**0**	0%
Strongly disagree	**1**	5%

Developers should integrate their changes frequently [When talking about continuous delivery, select the option that best describes your opinion on the

statement?]



Strongly agree	**12**	60%
Agree	**7**	35%
Neither agree nor disagree	**1**	5%
Disagree	**0**	0%
Strongly disagree	**0**	0%

(Optional) Provide clarifications to your answers. Please also write down which answer the clarification is connected to

Extensive use of branches with commits should be used, instead of committing directly to main line. This is for purpose of code reviews and easier isolation of problems when they arise. Then these changes should be merged into the mainline as often as reasonable. This helps keeping up the code quality.

I think there could possibly some different development branch, which is merged into the master branch at least daily. Long term branching is a bad idea, but I feel people could create their own branch when they are testing their system and then integrate it immediately to the main branch and not stick with their own branches.

Build & deploy configurations should be in version control, but I wouldn't put there e.g. binaries for the needed tools in case those are commonly available, or at least I would have a separate repository for those. Testing code before

If not already listed in this survey, mention practices or principles that you consider being essential for successful adoption and usage of continuous delivery

Seamless cooperation with development and It-Operations.

Needless to say, nothing works without proper communication. Even though everyone is responsible for their own changes and also responsible for fixing the builds, communication needs to be good in order to get into the state where at least daily deployments can be done.

You have a pretty good list here, and if you have been reading Humble&Farley you probably haven't missed anything important. :)