Development of Continuous integration for a scalable progressive web application

Peter Parikka

Degree Thesis

Information technology

2019

EXAMENSARBETE	
Arcada	
Utbildningsprogram:	Informationsteknik
Identifikationsnummer:	7054
Författare:	Peter Parikka
Arbetets namn:	Development of Continuous integration for a scalable progressive web application
Handledare (Arcada):	Dennis Biström
Uppdragsgivare:	

Sammandrag:

Små företag och startups vill komma ut på marknaden så forts som möjligt. Detta orsakar att företagen inte prioriterar goda utvecklingsstandarder eller kontinuerlig integration- och leverans. Examensarbetets syfte är att utforska hur en startup kan utveckla kontinuerlig integration- och leverans från första början. Det första kapitlet utforskar konsensus av tidigare litteratur och ger läsaren en översikt om vad kontinuerlig integration- och leverans är. Kapitlet börjar med en genomgång av hur kommunikation påverkar processen och nyttan av att utveckla kontinuerlig integration- och leverans. Medan den andra delen utforskar problem med att utveckla kontinuerlig integration- och leverans. Examensarbete fortsätter med att presentera hur applikationen är planerat och vad som skall beaktas i planeringen av kontinuerlig integration- och leverans för en startup. I planeringen tas också upp olika arbetsredskap och varför detta arbetsredskap valdes till applikationen. Versionskontroll och kodgranskning utforskas i detalj, för det är en essentiell del för att uppnå lyckad kontinuerlig integration- och leverans. Det sista kapitlet går djupt in på implementationen, arkitekturen och olika komponenterna som används i applikationen. Applikationen utvecklades med hjälp av Amazon Web Services och GitHub. Amazon Web Services används för utvecklingen av ett kontinuerligt integration- och leveransrör, medan GitHub används som versionskontroll. En detaljerad beskrivning av de olika komponenterna som används inom Amazon Web Services ges. Illustrationer och kodsnutt presenteras för att ge läsaren en helhetsbild av processen. Kapitlet slutar med en genomgång av problem som uppkommit i implementationen. Examensarbetet slutar med att diskutera nyttan och problem med kontinuerlig integration- och leverans samt utforskar examensarbetet kritiskt. Förslag till kommande arbeten ges.

Nyckelord:	Kontinuerlig integration, Kontinuerlig leverans, Amazon Web Services, GitHub
Sidantal:	33
Språk:	Engelska
Datum för godkännande:	31.05.2019

DEGREE THESIS	
Arcada	
Degree Programme:	Information technology
Identification number:	7054
Author:	Peter Parikka
Title:	Development of Continuous integration for a scalable progressive web application
Supervisor (Arcada):	Dennis Biström
Commissioned by:	

Abstract:

Many small companies and startups feel a pressure to increasingly develop applications with the latest standards and principles. The problems arise when a startup should launch their application for the market as fast as possible, whilst not having time to design good coding practices or set up complex Continuous integration and delivery pipeline. The purpose of this thesis is to tackle how to set up Continuous integration and Continuous delivery in a small company or a start-up. The thesis is based on practical work which allows applications to easily be maintained, integrated and deployed for production. The beginning of this thesis will cover the consensus of past literature on Continuous integration and delivery, what the key takeaways are, what benefits they produce and how implementing these will affect the whole organization. It will then expand on planning and tools used to achieve expected result and dive deep into the practical implementation phase of the project. Illustrations of code and Amazon web service systems are included, for the reader to get a better picture of how the system works as a hole. The thesis will conclude with a discussion about the challenges, findings and improvements to be considered for future studies.

Keywords:	Continuous integration, Continuous Delivery, Amazon web services, Git , GitHub
Number of pages:	33
Language:	English
Date of acceptance:	31.05.2019

INNEHÅLL / CONTENTS

1 INTRODUCTION 6

1.1 Purpose 7

2 BACKGROUND 8

2.1 Continuous integration 9
2.2 Continuous delivery 11
2.3 Build pipeline 11
2.4 Communication and transparency 13
2.5 Benefits of adopting CI/CD 14
2.6 Challenges in adopting Continuous integration and delivery successfully 16

3 Planning the application 18

3.1 CI and CD tools 19
3.2 Version Control 21
3.3 Code Review 21

4 Implementation 22

4.1 Architecture 22
4.2 Development Flow Model 24
4.3 A deep dive into to AWS CI/CD 24
4.3.1 Amazon Simple Notification Service (SNS) 26
4.4 Challenges of implementation 28

5 Conclusion 29

5.1 Future studies 29

KÄLLOR / References 31

Figures

Figure 1. Continuous integration (CI) cycle. Photo C. Aaron Coins. Devops 2015 10
Figure 2 How AWS CodePipeline works. Amazon web services.................................. 20
Figure 3 GitHub workflow. GitHub 2017 22
Figure 4 Environment model... 23
Figure 5 Continuous delivery flow model... 24
Figure 6 buildspec YAML file describing CodeBuild process 25
Figure 7 How AWS Simple notification service works. Amazon web services 26
Figure 8 Publish messages to SNS topic 27
Figure 9 HTTP POST to Discord 28

1 INTRODUCTION

Software development has changed in the past decade from the more traditional waterfall methods to require more flexibility, increased feedback loops and customer centricity. This has increased the need for frequent releases and thus more production deployments.

Agile development emphasizes Individuals, interactions, working software, customer collaboration and responding to change (Beck et al. 2001). The lean ideology builds on principles such as eliminating waste, constant learning, late decision making, fast delivery, and empowerment of developers (Poppendieck & Poppendieck 2003). Both agile and lean methodologies are designed to provide quick and tangible results for the customer. Software organizations need to deliver in shorter lead time, better quality, and with a lower budget (Nurdiani et al. 2016 p.162).

Companies delivering their products to customers is where value is created. This means that customer feedback and insight is valuable, therefore increasing the need for companies to be agile and respond fast to customer's needs. Code written needs to be released in fast cycles, with confidence and with the ability to change rapidly. This is some of the reasons agile and lean ways of working has gained traction. Agile and lean software development principles have become the norm especially in software development companies. 85.4% of 58,981 developers surveyed by (Stack Overflow 2018) used agile methodologies in their work.

Continuous integration (CI) is a software development practice that enables fast, robust integration of code. It enables teams to achieve agile and lean principles more easily. CI has been reported to improve release frequency and predictability (Goodman and Elbaz 2008), increase developer productivity (Miller 2008) and improve communication (Downs et al. 2010).

CI is an essential part of the modern software development. Frequent deployment enables users to get new features more rapidly, give feedback faster and thus enabling users to be in an essential role in the development process. This lifts the barriers between customers

and deployment. Which has been described to be one of the biggest barriers to successful software development. (Fowler 2006)

The practice of CI is to routinely integrate code changes into a repository, testing the changes, as early and often as possible and for developers to integrate their code daily, or even multiple times a day. The faster you deliver code to production, the faster customers get the new features, the company at hand increases their market share and are better equipped to beat their competition.

1.1 Purpose

This study will investigate the benefits of adopting Continuous integration (CI) and Continuous delivery (CD) from the get-go. The first part of this study will go in CI and CD practices. The second part of this study will be a walkthrough of an easy to setup CI/CD stack for small businesses and startups. Studies performed in the field of CI so far has focused on bigger companies or corporations, where some of the most common challenges has been testing, legacy code and communication (Hukkanen 2015). One study found with the search criteria was written on Implementing Continuous Integration in a Small Company (Rejström 2016). This study still had many obstacles from old implementations of coding standards, that heavily weighted on the tools they chose to use in their CI.

This thesis does not include code testing or how to implement them as the main point of this thesis is to describe the benefits of implementing CI and CD from the get-go.

This thesis is divided into 4 chapters. Chapter 2 focuses on the background of CI and is designed to give the reader a complete understanding on the main principles of CI and knowledge of how CD ties into itChapter 2 will also touch on the benefits, challenges and communication & transparency of CI and CD. The third chapter will go through how to plan a CI/CD pipeline, what tools you need for it and a walkthrough on how to implement one. Finally, chapter four will include discussion and conclusion of this paper.

This thesis is based on an application designed in collaboration with a fellow IT student, to help users track their daily supplementation and get tangible results of tracking how their subjective feeling changes depending on supplementation. (Wiksten 2019) investigates how to select different technologies for the development of a modern web application. A more detailed description will be presented in the beginning of chapter 3.

2 BACKGROUND

When creating a new product, we usually design it in a way that gives the most value to the customer. Customers are in the centre role of every company developing applications and the faster interaction we can get between the customer and the development team the better. This is where CI and CD comes in.

They provide us a way to test, integrate and automate the pipeline in a fashion that decreases manual work, increases productivity, minimize risk of mistakes, drastically limits bugs and leaves the developers a sense of security when deploying new changes to the customers (Fowler 2006). This enables a fast feedback cycle between the customer and the development team. Developers can make and deploy new features with ease and customers can respond to them faster. This will create a sense of importance for the customer and it will help the company to get ahead of its competitors.

This chapter will go deeper into CI and CD based on earlier studies made in the field of CI and CD. A general description of CI is presented in section 2.1.1 followed by CD in section 2.1.2. Section 2.1.3 will go in to the build pipeline and how it functions. Section 2.1.4 will go into how important communication and transparency is for adopting successful CI and CD practices. Benefits of adoption CI/CD principles are described in section 2.1.5 and section 2.1.6 will go into challenges of CI/CD.

The following search criteria was used to identify relevant literature in CI/CD. Studies were included that where up-to date, relevant and established. Because no studies had been made on startups implementing CI/CD practices, there was a need to find more relevant literature through blog posts, seminars and articles. Several companies have written

about CI/CD practices in blog posts and articles. This was a great way to get some frame of reference and insight to this study. Since they are not academic journals, they are still subject to validity threats.

2.1 Continuous integration

CI was first mentioned in 1994 in terms of developing micro processes (Booch 1994 p. 273). The term was later adopted by (Beck 2000 p.16) in his definition of CI which states: "The latest code is built every night. The nightly builds provide us with insights about cross-component integration problems. Once per week we do an integration build where we ensure integrity across all components"
Approximately 12 years later of the first mention of Continuous integration (CI), Martin Fowler (Fowler 2006) Chief scientist at ThoughtWorks, and a man who have been credited with defining what the current practices, of CI is stated:

"*Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*"

Through the years our expectations and need for CI has changed. Software companies needs to integrate their code early and often, test the code for errors and report them as quickly as possible. Integrating code early and often, checking it for bugs and/or build errors has been shown to reduce integration problems significantly and develop working software more rapidly (Fowler 2006).

Figure 1 demonstrates how CI can be set up. When a developer has new changes to be added to the main branch of the repository worked on, the developer check in their changes (Step 1). The CI server is listening for changes in the source control server and when new changes are present the CI server fetches them (Step 2). This triggers a build process that builds a fresh instance of the application (Step 3). When the build is successfully completed the server will start to run automated tests with predefined test criteria (Step 4). The new instance will fail or succeed (Step 5) and the developers and managers

will be notified of the progress (Step 6). If the build would fail, no further changes should be checked in and the bug needs to be fixed immediately. (Fowler 2006)

This enables developers find the errors faster, resolve them quicker and continue building good quality software. The fastest way to resolve a broken build is to revert to the latest commit from the mainline, reverting the system to the last stable build and fixing the problem locally. In the case of the build succeeding, the team will continue their work with no further action needed. When the build CI succeeds the other developer can fetch the latest changes from the mainline. The CD part of the process starts from here and will be discussed deeper in the next chapter.

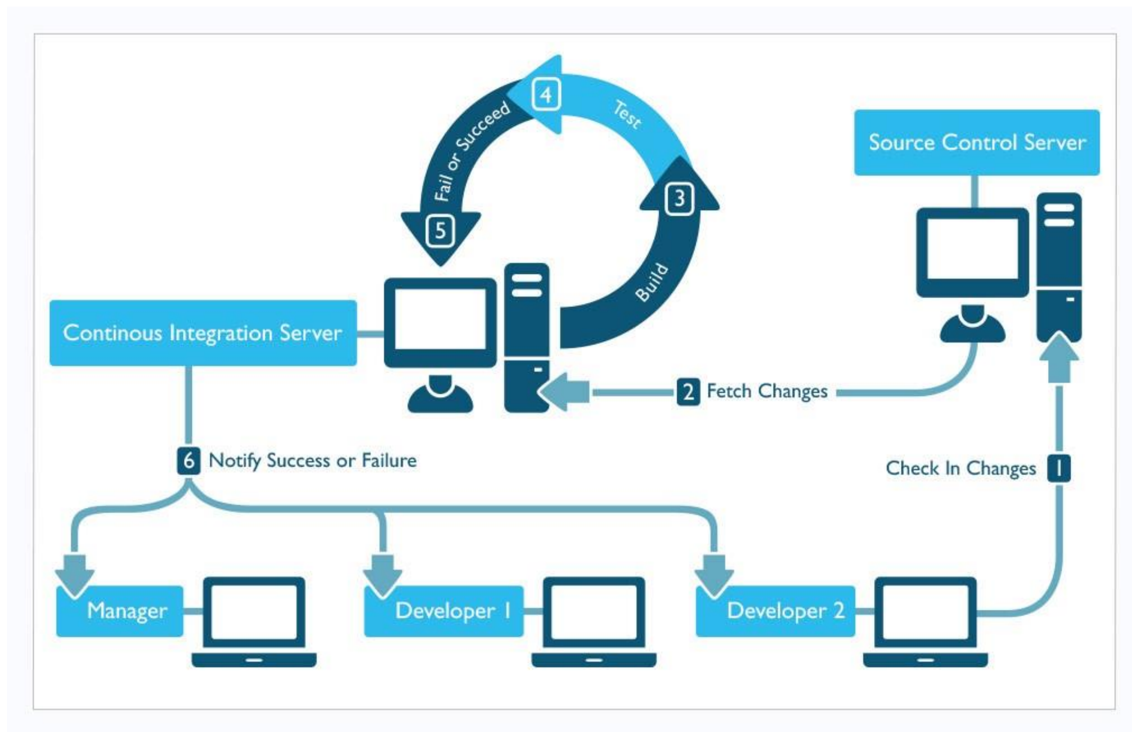

Figure 1. Continuous integration (CI) cycle. Photo C. Aaron Coins. Devops 2015

2.2 Continuous delivery

CD builds on the groundwork of CI. The main principle of CD is to build software in a way, that the software can be released to production at any time. The software should be deployable at any time without nobody batting an eyelid or panicking (Fowler 2006). Continuous delivery is often confused with Continuous development. Continuous delivery means that every change made in the Version control system (VSC) goes through the deployment pipeline and automatically gets deployed into production. The process of Continuous Delivery is to constantly have a deployable version of the software, but you can determine when it is deployed. Continuous delivery is therefore usually preferred in business cases where a company wants to postpone the release of new software. It shall be noted that the full potential of adopting CI is only obtained once Continuous delivery is added to the equation (Humble & Farley 2010; Olsson et al. 2012). (Fowler 2006) also states that the principal benefits of Continuous delivery are:

- Reduced deployment risk since deploying in smaller batches
- More believable progress. It is believed in a greater extent when a new version is in production than when a developer is noting it to be “done”
- User feedback. The faster you deploy new versions to production the faster you get feedback from the customer.

2.3 Build pipeline

A CI pipeline needs to be fast and not act as a bottleneck to the development team. The team needs to obtain the build status fast for them to deliver frequent builds. There are two sides to the build progress. As (Ståhl & Bosch 2014) notes, testing of the code quality needs to be comprehensive. This imposes a conflict between these two goals. On the other hand, we want to make sure the build is fast, but the code quality and other tests needs to be performed also. One of the techniques used to make the builds faster is a build pipeline. With a build pipeline the process can be split up to multiple stages.

The first stage as described by (Fowler 2006), the commit build can only include the fast and most crucial tests. After or parallel to the commit build there can be as many stages as the project needs, running more complete tests. These tests may include more time-

consuming tests as database- and end-to-end behavior tests. This suite of test can take hours to run (Fowler 2006). The time-consuming test suite runs when it can, taking the last successful build and running that through the more comprehensive test. If the secondary build fails, the team does not need to stop everything and fix it, as when the main commit build fails. Still the bugs need to be fixed as fast as possible while keeping the commit build running.

In the case where the secondary build fails, the team needs to consider adding some further tests to the commit build. This could strengthen the commit build and lead to less failures in later test suites, while still assuring that the commit build stays fast. The build pipeline should model your process for building, testing and releasing your application, and this pipeline ensures that each change can pass through this process independently in as an automated fashion as possible (Humble & Farley 2010).

When working in a team that utilizes CI and CD, testing your code changes before committing them to the mainline is important. When done right, it will reduce CI build failures and therefore make your teams process faster. Before committing to the mainline, the developer should pull the latest changes to their local environment and run a complete test suite locally. This ensures that the developer has always the latest code changes and reduces merge conflicts. Depending on the project there can be cases where the test suites are too large to run locally, or the developer is not able to mimic the complete environment locally (Hukkanen 2015 p.8). However, at least a reasonable and fast subset of tests should be run (Humble & Farley 2010 pp. 60-62). As (Humble & Farley 2010) describes, below are the 5 steps to achieve a build pipeline:

1. Model your value stream and create a walking skeleton.
2. Automate the build and deployment process.
3. Automate unit tests and code analysis.
4. Automate acceptance tests.
5. Automate releases.

2.4 Communication and transparency

To achieve a highly functioning CI/CD system, there needs to be an increased level of communication and transparency. You want to ensure that everyone can easily see the status of the system and changes made to it (Fowler 2006). Setting up information mediums showing at least the following: build times, test coverage, build status and open pull requests, are essential to successfully adopting the CI practices (Neely & Stolt 2013, pp. 121–128)

There are many different tools that are used for this. The most common are integrating success/failure messages to email, instant messaging, IDE monitors, RSS feed or displayed on a website. Also monitors physically located in the office, from where the team can see the progress of failing build, changes to the test suite and new pull request to be reviewed. More extreme solutions have been implemented by teams, everything from red and green lava lamps to robotic dogs walking up to the responsible developers, *"[displaying] to the team that it is not happy with that developer, in a friendly, funny and playful way*" (Ablett et al. 2007 pp. 931–936).

It has also been concluded that multiple channels of communication have a great impact on awareness and responsiveness to broken builds (Downs et al. 2012 pp. 507-517). One interesting way of recording the longtime improvement of the build and therefore communicating the success that has been achieved through adopting CI was described by (Fowler 2006):

> We put a calendar on the wall that showed a full year with a small square for each day. Every day the QA group would put a green sticker on the day if they had received one stable build that passed the commit tests, otherwise a red square. Over time the calendar revealed the state of the build process showing a steady improvement until green squares were so common that the calendar disappeared - its purpose fulfilled.

An essential part of communication effectively through these mediums are feedback that is tailored to the team's needs. The team needs adequate information to act rapidly and appropriately. Error messages needs to be specific and define which test failed, how it failed and provide support for fixing it. When tailoring information, it's important to not be overloading the team with information. If one displays too much information to the

team, the important information will easily get lost in the sea of information. Furthermore, the information from the CI build needs to be displayed immediately so it can be fixed as fast as possible. (CollabNet 2013)

Actions need to be transparently described, so each team member knows their responsibility and how to act when an event happens. E.g. there should be no question on what to do when there is a broken build or drop in code coverage in production. CD has a positive effect on transparency, because it facilitates gathering different metrics of the development progress (Haverila 2016 p. 49). CI and CD can also expose concerns in business stakeholders and VPs of the product you are developing. Some concerns noted in this Plutora article (Siroky 2019) was that now when releasing daily there are also continuous responsibility, concerns for quality, what the team is shipping and who makes decisions about the features released. All the concerns are valid and magnifies the important role of communication and transparency when using CI practices. It's proposed to every two weeks send a complete summary of release changes and release activity. Also, a good idea is to send a record of how releases are tested before and after release. In summary, both keeping the team and the stakeholders informed of the progress is key for successful adaptation of CI.

As Conway's Law (Conway 1968) states that:
"organizations that design systems are constrained to produce systems which are copies of the communication structures of these organizations".

2.5 Benefits of adopting CI/CD

Described below is some of the common benefits of adopting CI and CD practises. Many of them can be also tied in with adoption of Agile and lean principles. The bulk of these benefits will help an organization to get their coding practises to a new level and to get an advantage on their competition.

Increased confidence. As there is an automated process constantly ensuring that errors in the code are found and fixed before errors start to accumulate, fixes can be applied

faster without breaking other parts of the code (Fowler 2006). Increased confidence towards the product, its development and deployment practises has been observed through utilizing regression tests (Haverila 2016 p. 49). When old code is tested frequently and automatically, developers and testers are confident that their new changes will not break the old code and therefore has the time to perform more exploratory test on the new features. Risks of the team having a large integration process in the end with untested code, is avoided through constantly keeping the software in a working state.

Increased process transparency. Everything regarding the process of Continuously integrating and delivering should be monitored in detail, such as: build times, test coverage, build status, open pull requests and feature deployment. This increases the transparency of the whole practise and will expose the details of CI/CD to everyone involved. When practising CI and CD the deployment pipeline acts as documentation for the project (Haverila 2016 p. 50). As there is clear documentation of how CI and CD works and through that how the team's workflow regarding coding standards, testing standards and deployment works, it's also easier for new developers to contribute safely.

Better use of resources. Automated testing practises facilitates better use of resources. In testing there are many repetitive tests which can be automated while testers and coders can focus on more complex tests. When manual repetitive work decreases, productivity increase. The short-term costs will increase due to the setup of the deployment pipeline, setting up automated testing, maintenance and training. All these initial costs and large amount of effort will be paid back in the long run (Haverila 2016 pp. 55-60). Creating test will pay itself back, because they reduce manual work and they can be used in future projects and therefore reduce the starting costs of future projects significantly.

Shorter time to market. The more often you deploy, the shorter the feedback cycle becomes and the faster you will get feedback from your team and the customer. As the new version of software is constantly kept up to date, we avoid the risk of having a large chunk to deploy at a time and have much smoother integrations to production. With fast and frequent deployments, the organisation gains better understanding of the customers and the market (Neely & Stolt 2013). Frequent releases also help promote experimentation as there is lower risk deploying and one can always revert a deployment.

Code quality. With the practice of CI, the code quality will improve. As there is a structured way to test code quality and how the code integrates with the existing code, the standards of writing code will increase. This is partly due to transparency as all the code

written will go through scrutiny. Pastises such as peer reviews, linters and automated tests will keep the code quality consistent and developers will need to feel proud of the code they have written before committing it to the mainline (Haverila 2016 p. 40). This automatically results in less merge conflicts, better code quality and a better product in general.

All these benefits are incremental benefits that together will have a great impact on the success of a software teams development process. When adopting CI/CD practises, there is a great possibility to solve some concrete problems. In such cases there will be other project specific benefits not stated in the literature. To achieve these proposed benefits, you need to adopt CI/CD successfully. Presented in the following section are the common challenges of adopting CI/CD.

2.6 Challenges in adopting Continuous integration and delivery successfully

Described below is some of the common challenges of adopting CI and CD practises. Some of these are common challenges when adopting new ways of working in a project. Other challenges are more specific to CI/CD process.

Change resistance. Adopting new tools or frameworks in an organization is hard. Humans generally so not like change. Adopting to CI and CD practises requires change in the whole organization. Everyone from management to developers has to be on the same page and accept the new changes in the ways of working. Developers resist adaptation of CI/CD practises because their code will be exposed earlier for review and therefore the added transparency will bring some change resistance. Developers can feel humiliated if their code fails the tests. One of the most common challenges in successfully adopting CI and CD is change resistance (Debbiche et al. 2014).

Management support. The hole adaptation of successful CI and CD practices hangs in the hands of getting enough management support (Neely & Stolt 2013 pp.121-128). If one

does not get enough support from the management and have the needed resources in regards of tools and sufficient developer resources, it will be impossible to create successful CI and CD practices.

Legacy code. Big challenges also arrive with large amount of legacy code. It's nearly impossible to design automated tests for the legacy code, therefore leaving a great amount of code untested. The team will encounter problems in the new code, that interacts with legacy code and get unexpected behaviors. It's nearly impossible to make sure that the new code will not break the old legacy code, so one cannot blindly trust on the automated tests. Legacy code might also be owned by another team or third party, shifting the testing responsibilities to other developers and hindering the process of deployment. (Rodriguez et al. 2016)

Environmental management. One of the major challenges in CI/CD is the environment and its dependencies. Differences in development, testing and production environments can have a huge impact on the reliability of the production pipeline. If there are differences in environments, a bug can slip through and manifest itself in production. Good management and enabling automated provisioning of environments are essential to a successful implementation of CI/CD. (Leppänen et al. 2015 pp. 64-72)

Test automation. Designing and writing automated tests is challenging to get used to if this step was not required earlier. Having people change their development progress, forces them out of their comfort zone, which in turn causes change resistance (Torben 2011).

Test automation is also hard and requires a large initial investment both financially and from the developers. Developers need training to design and write tests for their own code. There are also many false or unrealistic expectations of what the benefits of automated tests are. Stakeholders may believe that automated testing automatically solves all testing needs. But automatic tests tackle unproductive or impossible tests to be performed manually. Test strategy needs to be thought out carefully and management needs to allocate enough time for developers to write the tests in a manner that they are valid reliable tests. If tests are not written carefully in a manner that developers trust in these tests, they will lower developer's confidence and thus undermine the result of a faulty build as the

developers will assume the problems lies in the “bad” tests. When the problems lie in faulty code (Neely & Stolt 2013 pp.121-128).

Understanding of the CI/CD process. The whole organisation needs to be on the same page about the whole CI and CD process. Initial setup of the system infrastructure needs knowledge and investment of resources. Developers needs to understand the whole deployment pipeline to effectively do their daily work. Developers that lack understanding can diminish the whole CI and CD process. If the feedback of the system is not used, of what use is it? (Ståhl & Bosch 2014 pp. 54–63) The whole organization needs to be on the same page and understand how to use the CI and CD system effectively.

The challenges of adopting successful CI and CD practises relies heavily on management and how the whole process is designed from the get-go as a team. Transparency of the ways of working, practises to adopt and the goal of the changes are important to state clearly to the whole organization. In the following chapter, the tools, designing and implementation of the application will be presented.

3 PLANNING THE APPLICATION

When starting a new project, it is very important to plan each step carefully. When planning one needs to figure out the specific needs of the application before starting to design on how the CI/CD system will be implemented. This chapter will start with a brief walkthrough of the application implemented for this thesis.

The application, which was implemented to be used in this thesis, is a supplement tracking application. The reason for choosing to build a supplement tracking application from a technical standpoint, was that there are many different technical challenges to be considered for an easy to use tracking application. The user needs to be able to input their supplements easily, answer questions about possible benefits and get their data displayed in a simple and easy to understand fashion. Inputs, outputs and data processing is covered, which means that there is a lot to test in an application like this. As the application needs to be easy to access, we decided to go the progressive web application route. A progressive web application is an application that runs in the browser but has many of the same

capabilities as a normal Android or IOS application, such as home screen integration, offline interaction and notification to name a few. The application is being developed as a mobile first Angular application. The application will be hosted in Amazon Simple Storage Service (Amazon S3) and the database used will be Amazon DynamoDB.

3.1 CI and CD tools

There are many different CI and CD tools to choose from. Which CI/CD tool suits a specific project is largely coupled with the frameworks that are in use, skill-set, stakeholders and other factors such as starting costs and ease of implementation. Aversion control system (VCS) and a CI server is always needed in a CI system. Popular VCSs are GIT, SVN and Mercurial. These are usually coupled together with an issue tracker, code review tool and file browser such as GitHub, Bitbucket and Gitlab (Rejström 2016 p.7). There are also many different CI servers. Open source solutions consist of Jenkins, Hudson, Go and Integrity. Commercial solutions consist of Bamboo, UrbanCode, Teamcity and one of the best-known solutions is Travis CI which offers both and enterprise plan and an free plan for open source projects. Another big name is also Gitlab, that is a VCS, CI and CD all in one. There are some newcomers to this space that has gained a lot of traction. Amazon web services (AWS) has created their own solution supporting everything from VCS to continuous delivery. Also, at the time of writing GitHub has announced that they are shipping a complete solution for CI and CD together with Google cloud. Each of these tools offer a similar core product, with all of them having their specific advantages to differ from the competition. Which one you choose for your project is mainly based on your needs, price, ease of use, setup and maintainability.

When choosing a tool, especially for a new project or a start-up one usually wants to keep the initial costs low. Many of the CI/CD provider mentioned above has free plan to get started with. There are often free trials and or restrictions on how many builds you can do before you need to upgrade to a paid plan. All of the CI/CD systems needs to be hosted on premises or in the cloud. Hosted solutions come with less hassle to set up and greater scalability (Pecanac 2016).

To this specific project AWS CodePipeline was chosen. AWS CodePipeline is a fully managed continuous delivery service that helps you automate everything from builds to testing (Amazon Web Services, Inc 2019). AWS CodePipeline is also extensively customizable to suit your specific needs. Figure 2 describes the different tools that AWS CodePipeline offers.

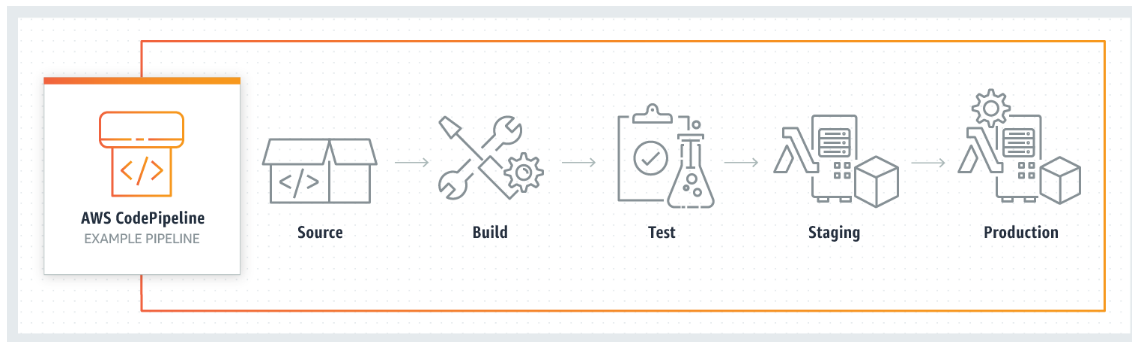

Figure 2 How AWS CodePipeline works. Amazon web services

AWS CodePipeline offers a price point that is hard to match by its competitors. When starting a project, you have to pay for a small, medium or large EC2 Linux or Windows instance 3 GB, 7 GB or 15 GB respectively. For reference the small Linux instance costs 0.005$ per build minutes and first 100 minutes per month of the small Linux instance is covered in the AWS CodeBuild free tier (Amazon Web Services Inc 2019).

Some additional costs may come in to play depending on your implementation such as lambda functions and s3 storage. This all depends on the amount of runtime and storage needed, which is why AWS has great plans for startups or new projects that want to keep initial costs low before the need to scale up. Scaling up is also easy because one can always go and change the EC2 instance in use and all the other factors scale automatically. AWS has also extensive documentation and all the services are centered into one place. In this project we decided to use other AWS services in the whole project as much as possible. This weighed heavily on the decision to have the CI/CD implementation in AWS. With AWS you can deploy the full stack using AWS CloudFormation which is a YAML or JSON based document describing what requirements and access, each piece of the application has. This template increases maintainability and allows one to create a

similar stack from scratch in literally seconds. In chapter 4 the implementation of a Continuous integration and Continuous delivery stack using AWS CodePipeline is described.

3.2 Version Control

Git was chosen to be our Version control system (VCS) and GitHub as the repository hosting service as both developers developing the application have previously worked with Git and GitHub. GitHub offers all the needed tools that is needed from a VCS hosting service. GitHub is also the leading VCS hosting service and is therefore widely used. GitHub also integrates nicely with AWS CodePipeline using webhooks.

3.3 Code Review

Code review is an essential part of successful implementation of CI practises. Thus, it needs to be present in the development workflow. The benefits of code review are the collective ownership of code, peer reviewing and integration with the CI server. Code reviews ensure that only quality working code is merged to master and thus ensures that the build will have a better chance of succeeding. Code reviews also increase transparency when everyone can see what's happening (Humble & Farley 2010). When a developer has completed a new feature, rather than pushing the changes directly to master the developers creates a feature branch. Now the developer pushes this feature branch to GitHub and that push initiates CI to run code checks and tests. When the checks are run the developer requests a review from another developer. The reviewer looks for possible improvements to be done and points out the problems to the developer. These developers discuss possible proposed changes and when approval is given the developer can merge the changes to master. The PR is automatically closed, and the feature branch can be deleted. When the branch is merged it initiates another CI process and or CD process to publish the changes. This model emphasizes continuous merging to master to keep the master branch as up to date as possible. Figure 3 displays the feature branching mechanism in GitHub.

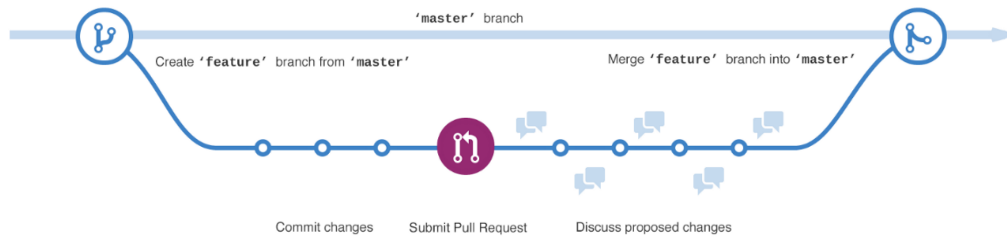

Figure 3 GitHub workflow. GitHub 2017

4 IMPLEMENTATION

In the following chapter the implementation of the release pipeline is presented. Based on the chapter on planning of the application AWS CodeBuild was chosen to be utilized together with GitHub in this project. The continuous integration pipeline is configured, managed and monitored through Amazon web services. CodePipeline also provides easy to view management console, where you can manage all the aspects either through AWS management console, AWS CLI or with building your own CloudFormation template. The following section will go through the general architecture and the different components needed for this implementation. To further investigate the implementation in detail and integration flow model is presented. Lastly the problems and pitfalls are presented in the last section of this chapter.

4.1 Architecture

To this application the different components were decided to be centralized as much as possible to Amazon Web Services AWS. By centralizing all components to one place, we get easier management of services and greater integration between components. The application utilizes AWS computing power in the form of s3 buckets, EC2 Linux instances, CloudFront, Lambdas and DynamoDB.

This project has been setup with the Angular CLI which makes it easy to create an application with TSlint, karma and protractor already in place. The developer can easily test locally their changes against unit tests and end to end tests. The developer has also the ability to run a local server to see changes in real time.

As can be seen in Figure 5 the application has 4 different environments development, staging and production environment. Picture below displays the relation with these environments.

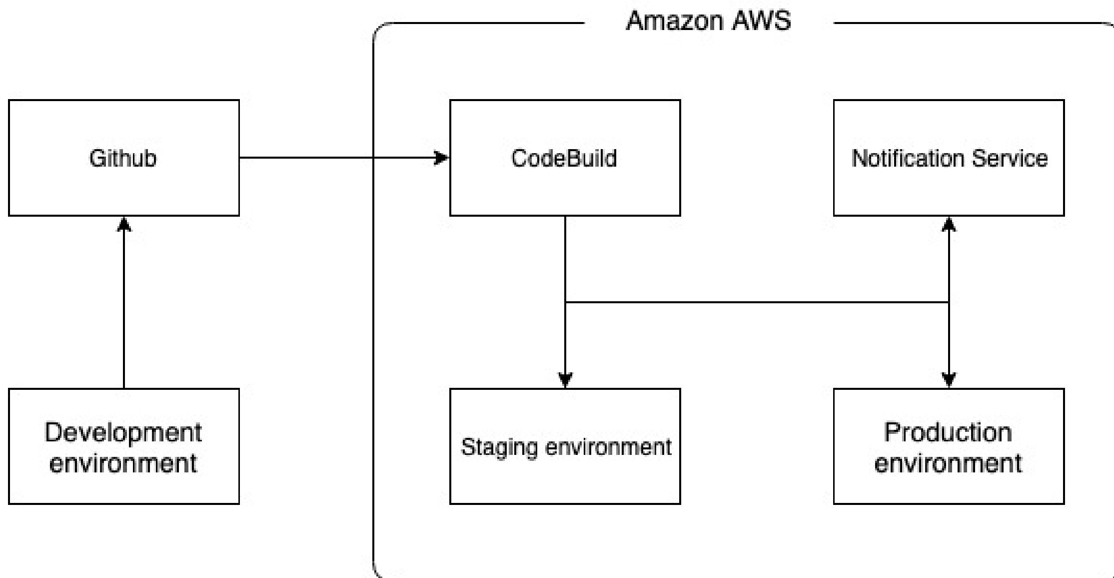

Figure 4 Environment model

Development environment. The local development environment is where all new features and refactorings are developed. All developers have a local copy of the GitHub repository and the repository contains specific rules for linting and local test. All developers must test their changes locally with the most recent changes and tests and lint check is run with every commit.

Staging environment. This is a copy of the production server also hosted on AWS. Staging instance mirrors the production server as closely as possible. This is also where the product will be manually tested by product owners and possible beta testers before publishing changes to the production environment.

Production environment. The live environment accessed by the end customers. Differs from staging only in capacity.
Notification service. From each instance notifications are pushed to discord, for an easier access to information about the Continuous delivery process.

4.2 Development Flow Model

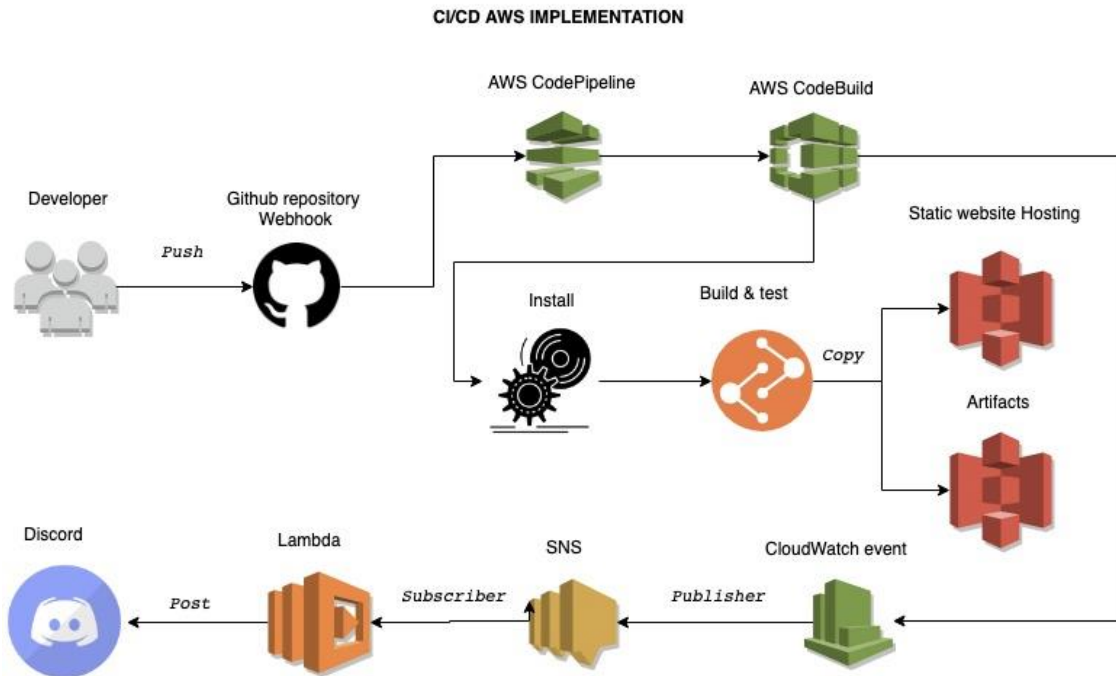

Figure 5 Continuous delivery flow model.

As can be seen in Figure 5 data flows in the following way: Developer checks out code from the GitHub repository. New features are developed and tested locally trough unit- and manual testing. Changes are committed and pushed to GitHub. Developer creates a pull request; CI is triggered and runs the test specified. When the tests have passed, then the pull request is ready for review. A team member reviews the pull request and after their acceptance, the code is merged onto the staging/development Brach. This triggers a new test run against the simulated production environment. When tests pass, the CI server will install all the needed dependencies, build the application and run tests against it. If the build phase is successful then the CI server will copy the build files to the s3 bucket for staging where developers, product owners and beta testers can acceptance test the finished feature. When acceptance tests are done, the new feature will be merged to master, and the CI will run test again before new features goes live on the production server.

4.3 A deep dive into to AWS CI/CD

In this chapter the reader will get a more concrete example on how you could implement a CI/CD stack in AWS with the help of GitHub, CodePipeline, S3, SNS and Lambdas.

As described in the last chapter everything starts from when a developer pushes their new feature or improvement to GitHub. In GitHub you can set up WebHooks that is subscribed to a certain event, such as a pull request or a change in the master branch. When an event is triggered GitHub sends a HTTP POST payload to the defined URL. This payload triggers the right CI build.

When the post event happens, AWS automatically spins up a CodeBuild EC2 Linux instance. This instance is responsible of testing and building the application. The instance is configured through a buildspec YAML file. The buildspec YAML file consists of a collection 7(run-as, install, pre_build, build, post_build, artifacts, cache) different sequences and other configuration settings that can each be configured in the buildspec YAML file to match your need for every phase of the build process. Below is the main build file that builds the application to production after the following sequences as can be seen in Figure 6:

```yaml
version: 0.2
phases:
  install:
    commands:
      - bash -c "$(curl -fsSL https://raw.githubusercontent.com/thii/aws-codebuild-extras/master/install)"

      # Install the Angular CLI
      - npm install -g @angular/cli

      # Upgrade apt
      - apt-get upgrade

      # Update libs
      - apt-get update

      # Install apt-transport-https
      - apt-get install -y apt-transport-https

      # Use apt to install the Chrome dependencies
      - apt-get install -y libxcursor1
      - apt-get install -y libgtk-3-dev
      - apt-get install -y libxss1
      - apt-get install -y libasound2
      - apt-get install -y libnspr4
      - apt-get install -y libnss3
      - apt-get install -y libx11-xcb1
      - apt-get install -y firefox wget
      - export FIREFOX_BIN=/usr/bin/firefox

      # Print out missing libs
      - echo "Missing Libs" || ldd ./node_modules/puppeteer/.local-chromium/linux-549031/chrome-linux/chrome | grep not
      - npm install

  build:
    commands:
      - echo Build started on `date`
      - ng build --prod
      - echo Running tests...
      - echo starting tests...
      - ng test --watch=false

  post_build:
      commands:
      - echo cp to s3...
      - aws s3 cp dist/supplement-app s3://supplement-tracker --recursive
      - echo Build completed on `date`
artifacts:
  files:
    - '**/*'
  discard-paths: yes
  name: build-$(date +%Y-%m-%d)
  base-directory: 'dist/supplement-app'
```

Figure 6 buildspec YAML file describing CodeBuild process

Install. Firstly, we will fetch a repository that will enable us to get better logging information to AWS CodeBuild and thus being able to faster and with more accuracy recognize problems faster. At this point we will start to install different dependencies. The CI instance installs Angular CLI, upgrades and updates Linux packages and installs Chrome and Firefox and the needed dependencies for them.

Build. In this phase the CI instance builds the application to production so we can later access it from the distribution folder. Here the CI server also runs the unit tests defined in the Angular application.

Post_build. Copies the production-built application form the dist folder to the specified s3 bucket which hosts the production application.

*Artifacts.*Saves the build process artifacts in a separate s3 bucket.

The process of setting up a simple build is quite easy at first glance as it's highly abstracted. But when the process is so abstracted it can be hard to know or understand what goes on under the hood, which can make the process hard to understand and debug. The build environment sends information to Codebuild and Amazon CloudWatch for each build step. These logs can be used to build a Simple Notification service. In the next section we will dive deep into how Amazon Simple Notification Service (SNS) works.

4.3.1 Amazon Simple Notification Service (SNS)

Amazon Simple Notification Service (SNS) is a publication and subscription messaging service. It enables you to decouple microservices, distributed systems and serverless applications. They provide high-throughput, push-based and many-to-many messaging. With SNS topic you can deliver messages to many subscriber endpoints for parallel processing with Amazon SQS, AWS Lambda functions and HTTP/S webhooks. Furthermore, you can send notifications directly to end users using mobile push, SMS and email (Amazon Simple Notification Service 2019). Figure 7 describes the different components in AWS Simple Notification Service.

Figure 7 How AWS Simple notification service works. Amazon web services

In this thesis Amazon CloudWatch and Simple Notification Service (SNS) was used together with AWS Lambda function to POST messages to a webhook provided by Discord (A free messaging platform).

As noted in the earlier chapter 4.3, the build environment sends information to Amazon CloudWatch which in turn acts as a publisher that publishes messages to SNS. This is configured through a JSON object as can be seen in Figure 8, that has rules when to publish a message e.g. when install phase is completed or when there is an error in the build process. The message format is configurable through your CloudWatch rule. In this thesis the input transformer option was used to have a simple JSON object key, value pair combination. This way you can write a custom message and inject e.g. the project name where you want it. It is important to give the developers not only the build status, but also in case of failure a simple but understandable message about where the problem is as stated in chapter 2.1.4

SNS topic

Topic* CodeBuildStatus

Configure input

Matched event

Part of the matched event

Constant (JSON text)

Input Transformer

```
{
"completed-phase":"$.detail.completed-phase",
"project-name":"$.detail.project-name",
"completed-phase-status":"$.detail.completed-phase-status"
}
```

```
"Build'<project-name>' has completed the build phase of '<completed-phase>' with a status of '<completed-phase-status>'."
```

Add target*

Figure 8 Publish messages to SNS topic

From here the message goes to the SNS topic that decouples the message from the Lambda function. The lambda function acts as the subscriber to the topic and receives the message as can be seen in Figure 9. The Lambda function gets the message through the event object and with a simple HTTP POST request posts the message to Discord in the right format.

```
const https = require('https');
exports.handler = (event, context, callback) => {

  const postData = {
    username: "AWS build status",
    content: event.Records[0].Sns.Message
  };

  const options = {
    method: 'POST',
    hostname: 'discordapp.com',
    path: '/api/webhooks/553932824911151114/0hrK9pXJLIXFy2RO8vo_JLHACCrG57Lqzsmt4OCDejzQZYW
    headers: {
      "Content-Type": "application/json"
    }
  };

  const req = https.request(options, (res) => {
    let body = '';
    res.setEncoding('utf8');
    res.on('data', (chunk) => body += chunk);
    res.on('end', () => {
      if (res.headers['content-type'] === 'application/json') {
        body = JSON.parse(body);
      }
      callback(null, body);
    });
  });
  req.on('error', callback);
  req.write(JSON.stringify(postData));
  req.end();
};
```

Figure 9 HTTP POST to Discord

4.4 Challenges of implementation

As Amazon web services has a lot of products and a lot of ways to implement similar kinds of systems with a small difference in them, it was hard to know where to start from and how to find the right information to the CI/CD implementation. AWS docs are highly abstracted and only describes the basic information. This was a challenge when large amount of the job included was describing the right information to the right files. Another challenge in the implementation phase was to get headless Chrome and Firefox to run in the build pipelines test suite. The biggest challenges experienced in the implementation of these services was understanding how Amazon works and how the data needs to flow to achieve the wanted result.

5 CONCLUSION

This chapter will go through the benefits and challenges of implementing a CI/CD process. The thesis will be reviewed and discussed critically. Chapter 5.1 will discuss possible future studies to be performed.

Planning and development of a CI/CD system can be time consuming if the start-up or company does not have proficiency or earlier templates to use. Depending on the scope of the application one needs to decide if a CI/CD pipeline is needed. As you have learned this far there are major benefits in favour of implementing a CI/CD pipeline. The decision comes largely down to how many developers, how often the product needs to be updated and how scalable the application needs to be. If the product in mind is a simple website with one developer and low update rate, then it's reasonable not to implement a system like this. But if one expects that the team will be larger in the future. Or that the product needs to be updated or/and worked upon daily, then there is lot of evidence described in chapter 2 that a well-structured CI/CD system will in the long run benefit the startup at hand greatly. A CI/CD system will give transparency to the whole development process. When setting up a system, you need to plan the process thoroughly and thus achieving a more complete and straightforward development process. This will help you avoid being faced with no plan and suddenly having a large group of developers working as they best see fit. Every aspect of an CI/CD system won't be necessary in the beginning but can easily be added once the product and team get larger. When the basis of the development is sound, it's easier to do small adjustments to move the CI/CD process in the right direction. When starting the process early and the code is transparently viewed by peers, tested and kept up to date. There will not be "legacy", untested or unreviewed code.

5.1 Future studies

Future studies could go in on tracking two small companies or startups, one that haven't implemented an CI/ CD pipeline and one that has. This could be an interesting study as it would be possible to track how efficiently a company's or teams internal processes runs and track the difference between a team / company that has implemented a CI/CD system from the beginning and one that has not. In the same study one could track the long-term

costs of CI/CD pipeline and find out which of the startups succeeded better / were more cost efficient in their process

KÄLLOR / REFERENCES

Hukkanen, L., 2015, *Adopting Continuous Integration – A Case Study*

Hukkanen, L., 2015, *Adopting Continuous Integration – A Case Study, p.8*

Amazon Simple Notification Service (SNS), Available: https://aws.amazon.com/sns/, Accessed: 10.04.2019

Ståhl, D., Bosch, J., 2014, *Automated software integration flows in industry: a multiple-case study,* In Companion Proceedings of the 36th International Conference on Software Engineering, pp. 54–63

Miller, A., 2008. A hundred days of continuous integration. In: Agile 2008 Conference, pp. 289–293.

Downs, J., Plimmer, B., Hosking, J., 2012. Ambient Awareness of Build Status in Collocated Software Teams. 2012 34th International Conference on Software Engineering (ICSE), pp. 507–517.

AWS CodeBuild pricing, Available: https://aws.amazon.com/codebuild/pricing/, Accessed: 07.04.2019

Ablett, R., Sharlin, E., Maurer, F., Denzinger, J., Schock, C., 2007. BuildBot: robotic monitoring of agile software development teams. In: 16th IEEE International Symposium on Robot and Human interactive Communication, pp. 931–936.

Debbiche, A., Dien´er, M. and Berntsson Svensson, R., 2014, *Challenges when adopting continuous integration: A case study*. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi M¨annisto¨, Ju¨rgen Mu¨nch, and Mikko Raatikainen, editors, Product-Focused Software Process Improvement, volume 8892 of Lecture Notes in Computer Science, pages 17– 32. Springer International Publishing, ISBN 978-3-319-13834-3.

Olsson, H., Alahyari, H., Bosch, J., 2012, Climbing the "stairway to heaven"; – a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In Software Engineering and Advanced Applications (SEAA), 38th EUROMICRO Conference on, pp. 392–399

Humble, J., Farley, D., 2010, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, first edition, Ch. Anatomy of the Deployment Pipeline

Humble, J., Farley, D., 2010, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, first edition

Humble, J., Farley, D., 2010, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Upper Saddle River, first edition, pp. 60-62

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

Rodriguez, R., Haghighatkhah, A., Lwakatare, L., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T, Kuvaja, P., Verner, J. and Oivo, M. *Continuous deployment of software intensive products and services: A systematic mapping study. Journal of Systems and Software*, 2016.

Fowler, M., 2006, *Continuous Integration,* Available: https://martinfowler.com/articles/continuousIntegration.html Accessed: 28.01.2019

Haverila s., 2016, Impacts of Continuous Delivery in Software Projects, p. 40-60

Rejström, K., 2016 *Implementing Continuous Integration in a Small Company: A Case Study*

Rejström, K., 2016 *Implementing Continuous Integration in a Small Company: A Case Study, p.7*

Development Practices, 2018, Available: https://insights.stackoverflow.com/survey/2018#development-pactises Accessed: 27.2.2019

Beck, K., 2000, *Extreme programming explained: embrace change,* Second, Addison Wesley Longman, Inc, p.16

Conway, Melvin E., 1968, "*How do Committees Invent*

Siroky Dalibor, *How to move to a Continuous Delivery Model (Proven Experience),* Available: *https://www.plutora.com/blog/continuous-delivery-model,* Accessed: 03.03.2019

Goodman, D., Elbaz, M., 2008, "It's not the pants, it's the people in the pants" – learnings from the gap agile transformation. In: Agile 2008 Conference, pp. 112–115.

Poppendieck, M. and Poppendieck, T., 2003, *Lean software development: an agile toolkit*. Addison-Wesley Professional.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, Kern, J., Marick, B., Martin, C., Mallor, S., Shwaber, K., Sutherland, J., *Manifesto for Agile Software Development*, 2019, Available: https://agilemanifesto.org Accessed: 27.2.2019

Booch, G., 1994, *Object-oriented analysis and design with applications*, Second edition, Addison Wesley Longman, Inc., CH 7 , p.273

Downs, J., Hosking, J., Plimmer, B., 2010*, Status communication in agile software teams: a case study*. In: Fifth International Conference on Software Engineering Advances, pp. 82–87

Leppänen, M., Mäkinen, S., Pagels, M,, Eloranta, V., Itkonen, J., Mäntylä, M., and Männistö, T., *The highways and country roads to continuous deployment*., 2015, IEEE Software, pp. 64–72.

Nurdiani, I., Börstler, J. , Fricke, S., 2016, *The impacts of agile and lean practices on project constraints.* A tertiary study, volume: 119, p. 162

Pecanac, V., 2016, *Top 8 Continious Integration Tools*, Available: https://code-maze.com/top-8-continuous-integration-tools/, Accessed: 24.03.2019

The Role of Feedback in Continuous Integration, Continuous Delivery and Agile ALM, 2013, Available: https://www.collab.net/sites/default/files/uploads/CollabNet_eBook_The_Role_of_Feedback.pdf, Accessed: 03.03.2019

Torben, R., 2011, Available: https://www.torbenrick.eu/blog/change-management/12-reasons-why-people-resist-change/ , Accessed: 03.03.2019