LAPPEENRANTA-LAHTI UNIVERSITY OF TECHNOLOGY LUT
School of Engineering Science
Software Engineering
Master's Programme in Software Product Management and Business

Joonas Ryynänen

CREATING A FRAMEWORK FOR PLANNING SOFTWARE DELIVERY PIPELINES

Examiners : Assistant Professor Antti Knutas
Associate Professor Jussi Kasurinen

Supervisors: Assistant Professor Antti Knutas
Associate Professor Jussi Kasurinen

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Tietotekniikan koulutusohjelma
Master's Programme in Software Engineering and Digital Transformation

Joonas Ryynänen

Viitekehyksen luominen automaattisten julkaisuputkien suunnitteluun

Diplomityö

2021

72 sivua, 12 kuvaa, 4 taulukkoa, 3 liitettä

Työn tarkastajat: Apulaisprofessori Antti Knutas
Apulaisprofessori Jussi Kasurinen

Hakusanat: devops, julkaisuputki, jatkuva integraatio, jatkuva julkaisu
Keywords: devops, deployment pipeline, continuous integration, continuous delivery

Automaattisten julkaisuputkien merkitys ohjelmistotuotannossa kasvaa koko ajan; niiden avulla ohjelmiston kehitystyö tehostuu. Vaikka julkaisuputkien tuomat hyödyt on tutkitusti todistettu, niiden käyttö ohjelmistoteollisuudessa on rajallista. Tämän diplomityön tarkoituksena oli luoda viitekehys, joka selkeyttää julkaisuputkien suunnitteluprosessia. Viitekehys luotiin, demonstroitiin ja arvioitiin käyttäen Peffersin ym. (2008) tutkimusmetodologiaa. Kokeilemalla viitekehystä olemassa olevaan ohjelmistotuotteeseen, voitiin todeta sen selventävän julkaisuputkien suunnittelua.

ABSTRACT

Lappeenranta-Lahti University of Technology
School of Engineering Science
Software Engineering
Master's Programme in Software Engineering and Digital Transformation

Joonas Ryynänen

Creating a framework for planning software delivery pipelines

Master’s Thesis

2021

72 pages, 12 figures, 4 tables, 3 appendices

Examiners: Assistant Professor Antti Knutas
Associate Professor Jussi Kasurinen

Keywords: devops, deployment pipeline, continuous integration, continuous delivery

The importance of automatic software delivery pipelines is growing in the software industry; by using them, it is possible to enhance the development work. Even though the benefits of such pipelines are scientifically proven, their usage in the industry is limited. The purpose of this thesis was to create a framework that helps to plan software delivery pipelines. The framework was created, demonstrated, and evaluated using a research methodology by Peffers et al. (2008). By applying the framework to existing software, it turned out to clarify the planning process of the delivery pipelines.

ACKNOWLEDGEMENTS

This thesis was a great way to improve my knowledge regarding DevOps and automated software delivery. I want to thank my supervisors for providing guidance throughout the project.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Delivery / Deployment
DevOps	Development & Operations
DSR	Design Science Research
E2E	End-to-End Testing
HTTP	Hypertext Transfer Protocol
SDP	Software Delivery Pipeline
SFTP	Secure File Transfer Protocol
SPA	Single Page Application
SSH	Secure Shell
VCS	Version Control System

1 INTRODUCTION

Modern software development strives to be as effective as possible; the field is constantly evolving and the requirements of the software become more and more complicated (Sommerville 2015, 18). Many different ways of automation are used in the development process to achieve a smooth and efficient workflow. A development principle called "DevOps" has emerged during the last decade of software - it is a term that is combined from the words "Development" and "Operations", where development means the activity of software programming and operations means the practice of publishing and maintaining software.

DevOps is a term that is explained in various ways - Hüttermann (2012, 4) states that DevOps details different practices that simplify the software delivery process. He points out the conflict that is commonly present between development and operations activities: the development's goal is to produce changes (new features, bugfixes) to the software constantly, while the operations seeks to keep the software as stabile and immutable as possible (Hüttermann 2012, 5–6). Gupta et al. (2017, 75) state that in DevOps the collaboration between the development and operations teams is emphasized. AL-Zahran & Fakieh (2020, 2780) on the other hand see DevOps as a set of cultural and procedural practices that facilitate digital transformation. The concept of DevOps can be understood from multiple different perspectives, the common denominator being the goal to make the software development and deployment processes efficient and simple.

A software delivery pipeline (SDP) which automates the deployment of a software is an example of a process that enhances the development and deployment processes - therefore it is a concrete way of practicing DevOps. In the software industry, these pipelines are often called with terms Continuous Integration, Continuous Delivery and Continuous Deployment (CI, CD). For example Fowler (2006 and 2013) uses these terms when describing SDPs. These kinds of pipelines are useful when software needs to be rapidly developed and deployed; this is the goal of many software businesses, which makes the delivery pipelines an important aspect of a modern development environment.

This thesis focuses on first reviewing the current state of DevOps and then developing a framework that can be used to create an implementation plan for an SDP. A real-world case software solution (BetterWork) is used to illustrate the created framework in practice. The goal of this thesis is the following research objective:

"O: Create a framework for planning an effective software delivery pipeline"

This research objective is relevant in the current software industry. According to DORA & Google (2019), it is very beneficial to adopt DevOps principles (like SDPs) in an organization. They conducted a survey and identified four groups of performers regarding the adoption of DevOps practices. It was found out that the highest performing group was significantly outperforming the lowest performing group. The "elite performers" were able to:

- Deploy code more frequently (multiple deploys per day vs. once per month / year)
- Achieve a faster lead time from commit to deployment (less than one day vs. 1 month to 6 months)
- Restore services quicker (less than one hour vs. one week to one month)
- Reduce the failure rate (between 0-15% vs. between 46-60%)

These observations show that the companies that adopt the DevOps practices among the best in the industry can achieve considerable advantage compared to other companies. Additionally, the gained advantage grows as time goes on.

While the report by DORA & Google (2019) doesn't exactly only focus on the implementation of SDPs, an automated SDP is likely at the heart of the surveyed software companies' DevOps practices. Therefore it can be used as a strong motivation to achieve the research objective. Additionally, based on Erich et al. (2017) it can be stated that DevOps automation, in general, is a high priority in the software industry. This is further

proof of the relevancy of the research objective and the thesis as a whole.

This thesis strives to produce a unique deliverable that extends the existing research. In the literature review, multiple academic sources regarding DevOps and especially automated software delivery are reviewed. In those sources, some visual illustrations regarding SDPs are presented - for example, the “integrate button” by Duvall et al. (2007) and the process of continuous integration by Sommerville (2015, 742). The framework of this thesis is considering software delivery automation on a higher level than the mentioned models - this is proved by comparing the proposed framework of this thesis to the mentioned illustrations.

1.1 Scope of the thesis

The thesis has the following limitations to ensure that a proper level of focus and depth is achieved.

First, this thesis focuses on the web software. This means that the DevOps practices (regarding especially the software delivery process) and technologies are discussed only in the context of web software development. Additionally, the focus is mostly on aspects that are relevant to the case software that is used in this thesis. The case software is closely analyzed regarding its software architecture and current state of software delivery - then the framework will be applied to plan a more complete software delivery process.

The scope of this thesis is also limited from another point of view - the amount of SDP-related tools that are analyzed. This is because SDPs include multiple phases, each of which contains various different tools in the market. Since it is not possible to research and analyze all of the available tools, the focus is on the most relevant tools from the perspective of the case software.

1.2 Goal of the thesis

The goal of this thesis is to develop a framework that can help software engineering practitioners to plan and implement efficient software delivery pipelines.

The following lists the relevant main requirements related to the objective:

“RQ-1: Help (select a method) to build software automatically in an SDP”
“RQ-2: Help (create a process that allows) to test software automatically in an SDP”
“RQ-3: Help (create / set up a process) to deploy software automatically in an SDP”

If the created framework fulfills these requirements, it is possible to reach the original objective.

To verify that the produced framework achieves the main requirements and the objective, it needs to be demonstrated and evaluated in practice. The framework is demonstrated by using it in the context of a case software in chapter 5. The results of the demonstration are then evaluated by comparing the existing and the upgraded software delivery processes in chapter 6.

1.3 Research process

This thesis aims to research the field of DevOps, focusing on the software delivery pipelines. Based on the research, the goal is to create a deliverable artifact that helps software engineering practitioners plan and create an effective software delivery pipeline. To successfully achieve the goal, a suitable research method and process needs to be selected.

Design Science Research (DSR) in the context of information systems is a research method that focuses on designing IT artifacts that are relevant to a certain application domain (Hevner & Chatterjee 2010, 9). Design science has been previously used in many different research fields - however, in the 1990’s the information systems community also recognized its ability to help solve real-world business problems (Hevner & Chatterjee 2010, 9-10). Automated software delivery (or the lack of it) can be seen as a business problem of a software company; if the software delivery is not optimized, the development work will not be as efficient as possible.

Peffers et al. (2008) present a figure that states a process model of design science research methodology (DSRM). These phases have been used when performing the research in this thesis. Before the actual research took place, a research entry point needed to be identified.

Peffers et al. (2008) identify four possible entry points:

- Problem-centered initiation
- Objective-centered solution
- Design & development centered initiation
- Client / context initiated

In this thesis, the research entry point is objective-centered. This is because there is a clear objective that the deliverable is trying to reach. As chapter 1.2 states, there are also three main requirements for the deliverable that are meant to help reach the main research objective.

After the research entry point is identified, the DSR process is carried out according to the phases by Peffers et al (2008). The phases are as follows:

1. Identify Problem & Motivate
2. Define Objectives of a Solution
3. Design & Development
4. Demonstration
5. Evaluation

6. Communication

In the first phase, the research is motivated by identifying and reasoning the usefulness of reaching the research objective. As the research entry point is objective-centered, the high-level objectives are already clear and therefore the second phase of defining objectives is quite straightforward. In the third phase, the design & development of the artifact is performed. An important part of the DSR process is to demonstrate and evaluate the produced artifact (phases 4 and 5) - as mentioned before, this thesis uses case software in order to show how the artifact can be used in practice in the software industry. By using the artifact in a realistic industry setting, it is possible to assess and evaluate it. The final step of the DSR process is to communicate the results - the final chapters of this thesis are dedicated to communicating the main research findings and the conclusions that are drawn from them.

The figure below illustrates the research entry point and the different phases that are carried out during this thesis.

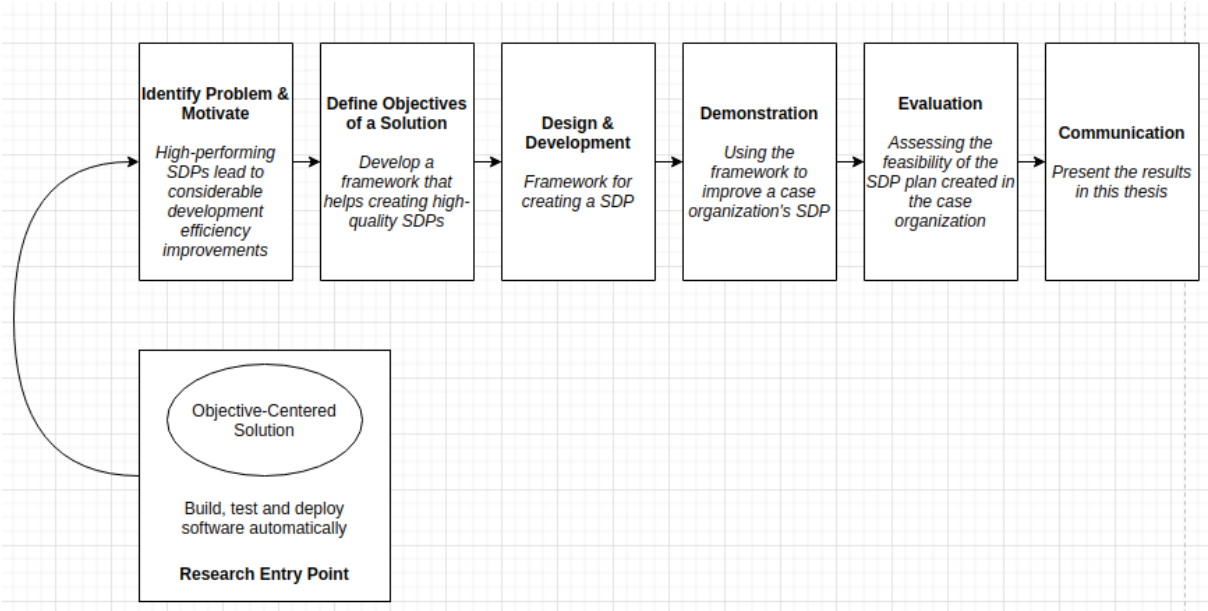


Fig. 1. Research entry point and research phases (based on Peffers et al. 2008)

Even though the phases are in order, it is allowed to reiterate through them in case it is needed. For example, it is possible to iterate back to the design & development phase in

case an issue in the produced artifact is found in the evaluation phase.

Additionally, it is important to note that this thesis conducts a formal experiment in the context of software engineering when evaluating the designed framework. Wohlin et al. (2012) is used as the main source regarding the experimentation phase of this thesis. Their book takes a comprehensive look into the field of experimental software engineering. In that field, the research process is different compared to more traditional fields - this is because in software engineering, the development is the main process, not the production. This is taken into account when planning the experimentation setup of this thesis.

As a final note regarding the experimentation phase, Duvall et al. (2007) identified speed as a key feature of an effective CI solution. SDP has a very similar meaning when comparing it to the definition of CI by Duvall et al. Therefore the evaluation phase of this thesis uses the speed of the SDP as the measurement when conducting the experiment. The experimentation compares the old and the new software delivery process of the case software. As mentioned before, the purpose of the experimentation is to evaluate the proposed SDP framework.

1.4 Structure of the thesis

The structure of the thesis is organized to emphasize the DSR process. First, in the consequent chapter of this thesis, a brief literature review is made to find out the current stage of research regarding DevOps and especially the SDP:s. After that, the research setup is presented in the third chapter. This includes for example discussing the entry point of the research, the problem that is tried to be solved and the requirements for the deliverable of the DSR process.

As this thesis conducts a DSR according to the methodology by Peffers et al. (2008), most of the content is documenting it. In chapter 4, the outcome of the design & development phase is presented. The graphical representation of the deliverable is provided, and the explanation regarding it is given. Chapter 5 demonstrates the usage of the deliverable (the SDP framework) by applying it in the context of a real case software.

The evaluation phase of the DSR process has also a dedicated section in this thesis (chapter 6). It is an important part of the research as it validates the produced artifact in a real software setting. The DSR model also includes a communication phase - the produced model is only communicated through this thesis, and therefore the last chapters act as the communication phase of the research process.

The subsequent chapter includes further discussion regarding the research findings. The discussion focuses on the used research process and the results of the research. Additionally, the feasibility of the produced artifact is assessed critically. Finally, the thesis ends with a concluding chapter that reflects the whole process of conducting the research and writing this thesis.

2 LITERATURE REVIEW

This chapter provides a literature review of DevOps. As this thesis creates a framework in order to plan a software delivery process, the focus of the literature review is on aspects relevant to it. The found literature is used as a basis for creating the deliverable of this thesis.

2.1 DevOps as a concept

Hüttermann (2012) is used as the main reference regarding DevOps from the viewpoint of software developers. He presents many important aspects to be taken into account when planning and implementing a concrete DevOps implementation.

DevOps is not studied very extensively when considering its importance in modern software development. Erich et al. (2014) conducted a study where it was found that not only is there relatively little research on DevOps, but also that the quality of the studies was rather low. Gupta et al. (2017, 76) point out that due to this, there is no uniform definition of DevOps yet present in the research field.

Jabbari et al. (2016) also found out in their systematic mapping study that DevOps lacks a consistent definition. Therefore, from the conducted literature review it can be stated that the field of DevOps lacks research on a fundamental level. The fact that the term "DevOps" itself is not yet fully agreed upon is a sign that more research regarding it needs to be made. However, the current literature still acts as a good starting point regarding the construction of the framework.

2.2 DevOps usage in practice

Erich et al. (2017) studied DevOps usage in practice - their research included six different case companies, which participated in an interview related to their adoption and usage of DevOps practices. They included a summary of DevOps at each case company - one of the categories that are especially interesting in the context of this thesis is automation. This is because the theme of this thesis is related to it (SDPs automate the software delivery

process).

To summarize, each of the case companies found automation to be useful. Especially automation related to the testing and the deployment of the software was generally seen as important (Erich et al. 2017, 7, 10). One of the case companies stated that DevOps automation is “the first subject every person joining the DevOps team should learn about” (Erich et al. 2017, 11).

Even though Erich et al. (2017) also identified the fact that there is no single accepted definition of DevOps in the industry (see chapter 2.1), it is encouraging to notice that automation is seen as a vital part of DevOps practices. This can be used as an argument when motivating the research that is conducted in this thesis.

2.3 DevOps frameworks

In chapter 2.1, it was noted that the research regarding DevOps is still relatively limited. For the purposes of this thesis, it was important to find a way to create a framework that proposes how to plan and create an SDP. Gupta et al. (2017) present a framework that can be used in order to analyze the maturity of a DevOps implementation. The framework is designed for enterprise organizations and their software. Also, the framework is not directly related to the goal of this thesis - however, it is still useful since it presents a really clear procedure and methodology in order to form a framework. Therefore, it can be used as a source when creating the deliverable of this thesis. The following describes the process that was used by Gupta et al. (2017).

First, they selected 18 different DevOps attributes from which ten key attributes were determined using statistical analysis. These ten attributes were then prioritized using Analytic Hierarchy Process (AHP). Finally, a case enterprise company was used in order to analyze their current DevOps implementation from the perspective of the key ten attributes. By using the framework it is possible to establish a utility measure of the implementation - this measure also includes information about the individual attributes and their performances. This can be used to improve the DevOps implementation, and in their article, the utility measure in the case company was increased after the company focused

on enhancing the four most low-performing DevOps attributes.

Because Gupta et al. (2017) present a clear way to form a framework around the context of DevOps, it is used as an inspiration when creating the framework of this thesis. Because the theme of this thesis is different compared to Gupta et al. (2017), their process has to be modified in order to suit the research goals of this thesis.

As it was mentioned, Gupta et al. (2017, 85) identified 18 DevOps attributes and four latent variables where the attributes can be placed. The latent variables of "Automation" and "Source Control" were found to be the two most significant. The following attributes are included in these two latent variables:

Table 1. Attributes of the latent variables found by Gupta et al. (2017)

Automation	**Source Control**
Automated Code Review	Branching Pattern
Automated Testing	Branching Changes
Automated Deployment	Branching Scatter
Automated tools to Monitor	Branching Depth
Infrastructure as Code	Feature Toggle

It is important to note that they were assessing DevOps as a whole, while this thesis focuses on the SDPs, which can be seen as one concrete DevOps practice.

Chapter 2.4 of this thesis states that Humble and Farley (2010) emphasize building, testing, and deploying as the major parts of SDPs. When comparing that statement to the presented latent variables by Gupta et al. (2017), it can be seen that the "automation" variables are more closely related to the context of SDPs compared to "source control" variables.

Automation is at the heart of SDPs - their purpose is to remove manual developer work

from the software building, testing, and deployment processes. However; the “source control” variables that were claimed to be amongst the most important (Gupta et al., 2017) cannot be bypassed completely when creating the artifact of this thesis.

These variables identified by Gupta et al. (2017) are used in the design & deployment phase of this research. The “automation” variables are emphasized more than “source control” variables due to the reasons described above.

Additionally, Duvall et al. (2007) present many relevant concepts regarding continuous integration (CI), which is an essential part of an SDP. In their book, an idea of an “integrate button”, which essentially describes the motive behind CI solutions, is discussed. The integrate button is a concept that emphasizes the need for automating software integration - by having such a button to handle the different stages of CI, the whole process becomes a “nonevent” as is stated. A visual illustration of the integrate button can be seen below.

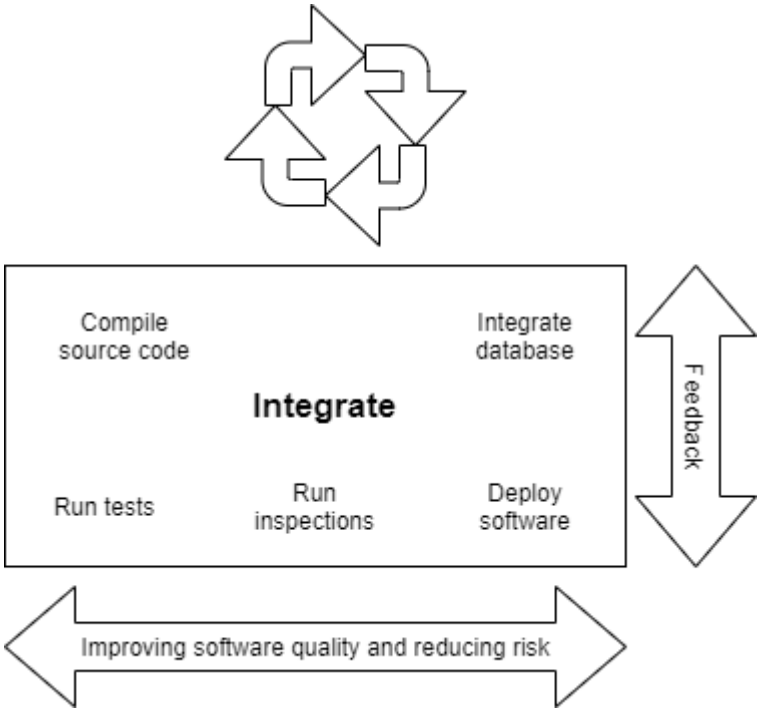


Fig. 2. Concept of integrate button (based on Duvall et al. 2007)

Duvall et al. (2007, 12) also present four necessary features for a CI solution:

1. Access to version control system
2. Automated process for building, testing and deploying
3. A feedback mechanism
4. Automated process for source code integration

According to Duvall et al. (2007, 12), these features make it possible to construct a "bare-bones" CI system.

Duvall et al. (2007, 20) also present an important statement regarding good CI systems: they state that speed is a critical feature. This observation is used later in this thesis when evaluating the results that were obtained by using the produced framework.

In addition, Sommerville (2015) presents a complete overview of the discipline of software engineering in his book "Software Engineering". He also discusses system building and continuous integration, which are relevant in the context of this thesis. According to Sommerville (2015, 743), continuous integration allows to discover and repair problems that arise from the interactions between developers. He also mentions that continuous integration is not always possible.

Since this thesis produces a deliverable in the form of a framework, the visual process model regarding continuous integration by Sommerville (2015, 743) is especially interesting. In the process model, the building and testing of the software are emphasized. The exact phases of the model can be seen in the figure below:

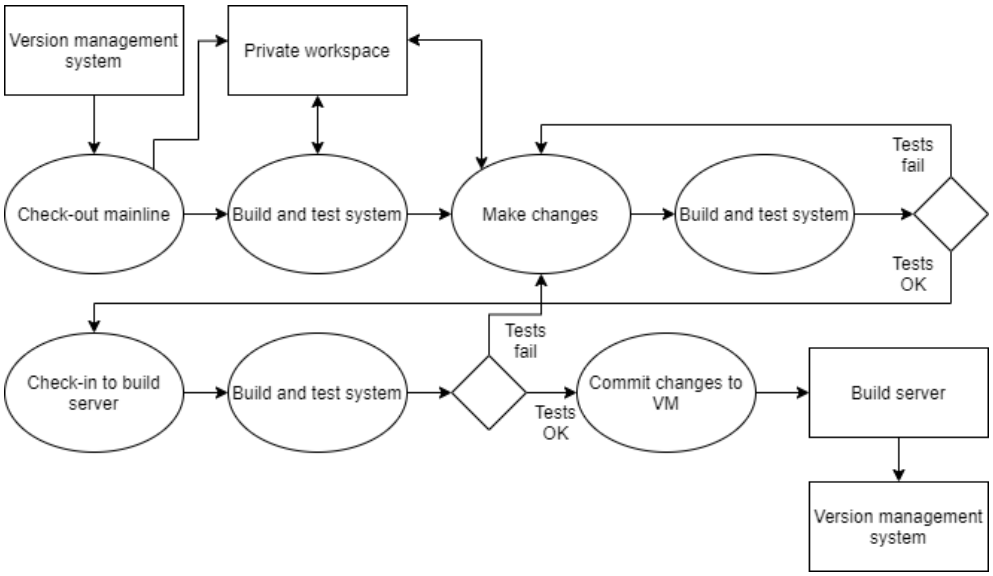


Fig. 3. Process model of continuous integration (based on Sommerville 2015, 743)

Finally, Ståhl & Bosch (2017) present an architecture framework called "Cinders" that is aimed towards describing CI and CD systems. They state that continuous integration is relatively easy to implement for small-scale software systems - however when the software system becomes more complex, it is harder and harder to set up an efficient SDP. Cinders is a rather complicated framework; it has four separate viewpoints (causality, production line, test capabilities, and instances), each of which comes with its optional informational layers. It encompasses two previously created modeling techniques (ASIF and CIViT). In their article, Ståhl & Bosch (2017) detailed twelve requirements for the architecture framework. The Cinders was able to verifiably fulfill eleven of them - one of them was also at least partly fulfilled. The Cinders framework is an example of a framework that is suitable for large and complex software systems.

2.4 Software delivery pipelines

In the software industry, the term "continuous X" is used to describe the automatic nature of performing X, where X can be for example:

- Integration
- Testing

- Delivery
- Deployment
- Development
- Release

Some of these terms are also used as a main category which includes many subcategories. For example, Akela (2016) identified the subcategories of continuous delivery to include continuous development & integration, continuous testing, and continuous release. Since there are a lot of different aspects to be performed in a continuous way, even the term “continuous everything” has emerged in the industry. For example, a commercial whitepaper by Cloudbees (2020) uses this term in order to emphasize this key term of software engineering.

Humble and Farley (2016) present the foundations of a software delivery process: as can be deducted from the title of their book, they have emphasized building, testing, and deploying as the main parts of an SDP. These three stages have traditionally formed the basis for automating the software delivery - because of this, the objectives of this thesis were formed around them.

When creating an SDP, it is essential to be able to identify the correct tools from the market that are compatible with the software development environment. The websites of different tools are used as the main source when researching them because they contain the latest information regarding their features and compatibility.

3 RESEARCH SETUP

In this chapter, the research setup of this thesis is described. First, the research entry point and the main requirements for the framework are presented. After that, a more detailed look into the requirements of the framework is provided. This is done by identifying a problem that relates to the SDPs from the field of software engineering. Existing literature is used to find a problem that the framework solves. The problem (and its solution) is then used as a basis for the requirements gathering. The chapter concludes with a comprehensive list of requirements for the deliverable of the DSR process.

3.1 Research entry point

Peffers et al. (2008) identified four possible entry points that can start the DSR process. In this thesis, the research entry is objective-centered. This is because there is the main objective to be met when creating the SDP planning framework. In order to reach the main objective, three main requirements for the framework were presented. Next, these three requirements are explained in more detail.

3.1.1 Help (select a method) to build software automatically in an SDP

In the context of software engineering, building means the generation of executable software artifacts from the source code of the software. Different software technologies and programming languages have their own tools regarding the build process. However, the high-level process of building is similar in most of them.

In an automated software delivery process, the details regarding the build process are in a minor role. However, it is still useful to understand how the source code transfers into runnable software on a general level. The most important aspect regarding software building in the context of SDPs is the ability to automate the build process; therefore the framework to be developed in this thesis should offer help regarding it.

Software containerization technologies like Docker can be seen as a way to take the building of software to a new level of abstraction. Docker provides a way to “containerize”

both the software and its runtime environment. For example, a Node.js web application can be containerized by using it. The software container holds both the built target software and the Node.js runtime environment. These software containers can be versioned and distributed by using online container registries like Dockerhub. (Docker 2021)

Software containerization is an important aspect of the SDPs since many cloud runtime environments provide integrations to deploy containerized applications. If the target software can be packaged in a software container, the implementation of the SDP is likely more efficient. One example where this is not the case is when the target software can be hosted with a simple runtime environment like a static file hosting service. Then, the containerization process is unnecessary.

3.1.2 Help (create a process that allows) to test software automatically in an SDP

Kasurinen (2013) states that software testing means to make sure that the software being built is working as it is hoped to work. In other words, software testing means the act of verifying that the developed software works as desired. Kasurinen (2013) sees software testing to belong to the field of software development - this means that it should be an integral part of the development of any software product.

Software testing contains many different levels of testing. Kasurinen (2013) identifies multiple testing levels that should be used during development. Additionally, he presents different ways to test the software before the deployment. Unit, integration, and system testing are examples of testing during development, while acceptance and smoke testing are examples of testing before deployment.

As it was discussed in the literature review of this thesis, testing is one of the core functions of an automated delivery process. Therefore an SDP should be able to test the software automatically. Kasurinen (2013) states that the goal of test automation is to take test cases that are done routinely and run them without the need for manual work. The intention is to remove the need for the software developers to use their resources to run the built software tests.

There are many different testing frameworks and tools in the markets for software testing - most of them specialize in some programming language or software development framework. For example, Jest is a unit testing tool for JavaScript-based projects and jUnit for Java-based projects (Jest, JUnit 2021). In the higher levels of testing, the testing tools seem to become more universal: for example, Percy is a visual testing tool that can be used for testing almost all kinds of web applications (Percy 2021).

The reason why software testing is one of the three main SDP stages (build, test, deploy) is that effective testing procedures can save a considerable amount of resources in a software project. Kasurinen (2013) presents that the earlier an error is found in the tested software, the easier it is to fix. While the SDP doesn't necessarily focus on software testing but rather on automating it, it can still help to find errors in the software for example in the cases where some core feature of the software is updated and an error is accidentally produced to the software. If the SDP is run before deploying the software, it can detect the problem before the software is deployed.

It is important to note that since there are multiple different levels and styles of testing, the SDP most likely cannot automate all of the testing processes. Therefore it's important to decide on the most relevant aspects of testing and include them in the SDP - the target software and the available resources should be taken into account when deciding which testing processes are to be included in the SDP.

3.1.3 Help (create / set up a process) to deploy software automatically in an SDP

The third main requirement was to help deploy software automatically in an SDP. Hüttermann (2012, 111) uses the word "automatic releasing" for a similar purpose. He defines the act of releasing as a process that makes the changes of the software accessible to the end-users. In this thesis, deploying software automatically means running the software in the production environment. In most cases, this also leads to the situation where end users can use the software.

There are multiple ways to deploy web-based software. For example, all of the major cloud providers offer several services to run web applications (Amazon, Google, Microsoft

2021). For an SDP to automatically deploy software, it needs to be able to access the production environment. This is also discussed in chapter 4.2.4.

3.2 Requirements of the framework

By taking a look at the existing literature related to SDPs, multiple challenges and problems can be identified. First of all, Duvall et al. (2007, 32-33) list multiple problems that prevent software companies from using SDPs:

- Additional resources are needed to maintain an SDP
- Adding an SDP introduces too much change
- Adding an SDP introduces too many failed builds
- SDP requires additional hardware and software costs

To make SDPs more appealing for the software industry, it would be beneficial to help solve the concerns stated above. Even though the concerns are valid, they don't necessarily always hold true - for example, if an SDP is planned and implemented correctly, the amount of failed builds can be minimized. Also, the costs that an SDP introduces depends largely on the tools that are selected: if an open source tool like Jenkins is used on a local server that a company already owns, the cost of an SDP can be very low compared to the benefits it brings.

Also Sommerville (2015, 743) sees challenges in the implementation of continuous integration (which can be viewed as a part of an SDP). He argues that continuous integration is not always possible due to the following reasons:

- Large software systems may take a long time to build and test
- Differing development and production environments can lead to difficulties when testing the system

While these reasons make the implementation of continuous integration harder and possibly not feasible, they can also be viewed as challenges that need to be solved to start utilizing SDPs. By improving the knowledge regarding the planning of an SDP, it is possible to overcome the presented challenges.

By looking at the concerns presented by Duvall et al. (2007) and Sommerville (2015), it seems like there is a misunderstanding in the industry regarding SDPs - they are seen as difficult, costly, and unreliable solutions that are not always feasible to implement. This observation can be phrased as a problem as follows:

Problem: "The software industry sees SDPs as difficult, costly and unreliable solutions, which are not always feasible to implement."

By finding this problem, it can be argued that the software industry needs a framework that helps to plan and implement SDPs. The solution that this thesis offers to the presented problem is phrased as follows:

Provided Solution: "A framework that helps the software industry to plan their SDPs."

The three main requirements offer a high-level solution to the presented problem - however, to design an effective framework that completes the main objective set for it, a more thorough gathering of requirements was performed. The research objective, the main requirements, and the presented problem with the solution were used as a basis in the requirements engineering process. The following lists the detailed requirements for the framework:

Table 2. The requirements list of the framework

1	The framework must be targeted towards personnel with a high level of expertise
2	The framework must be apply to web-based software

3	The framework must apply to large software systems
4	The framework must increase the knowledge regarding the planning of SDPs
5	The framework must decrease the additional resources needed to maintain SDPs
6	The framework must decrease the level of change and the amount of failed builds
7	The framework must help the software industry to see SDPs as a resource-efficient and a reliable way of improving the development workflow
8	The framework must include a careful assessment of the current software environment regarding for example version control and used technologies
9	The framework must emphasize the three main pipeline stages (build, test, deploy)
10	The framework must guide the user into selecting a suitable CI / CD tool for the SDP

The requirements above deal with the issues found from the research by Duvall et al. (2007) and Sommerville (2015). They also address the target audience, the scope, and the purpose of the framework.

At this point, the first two phases of the DSR process by Peffers et al. (2008) have been performed. The next chapter describes one of the most important phases - the design & development of the framework. The framework is designed based on the requirements list presented above.

4 SOFTWARE DELIVERY PIPELINE PLANNING FRAMEWORK

This chapter describes the framework that was the result of the third phase (Design & Development) of the DSR process by Peffers et al. (2008). First, the graphical representation of the framework is provided and discussed. After that, the different phases of the framework are gone through in more detail. The goal of the framework is to help plan and implement an SDP.

The framework is based on existing knowledge and literature about modern web software systems and SDP solutions. The framework strives to fulfill the requirements that were presented above. Duvall et al. (2007) and Sommerville (2015) were important sources in the requirements gathering - therefore these sources also influenced the produced framework. Finally, information regarding various software technologies (including programming languages, web development frameworks, version control systems, production environments, containerization technologies, and CI / CD tools) was used when designing the framework.

4.1 Graphical representation of the framework

The figure below displays the graphical representation of the framework:

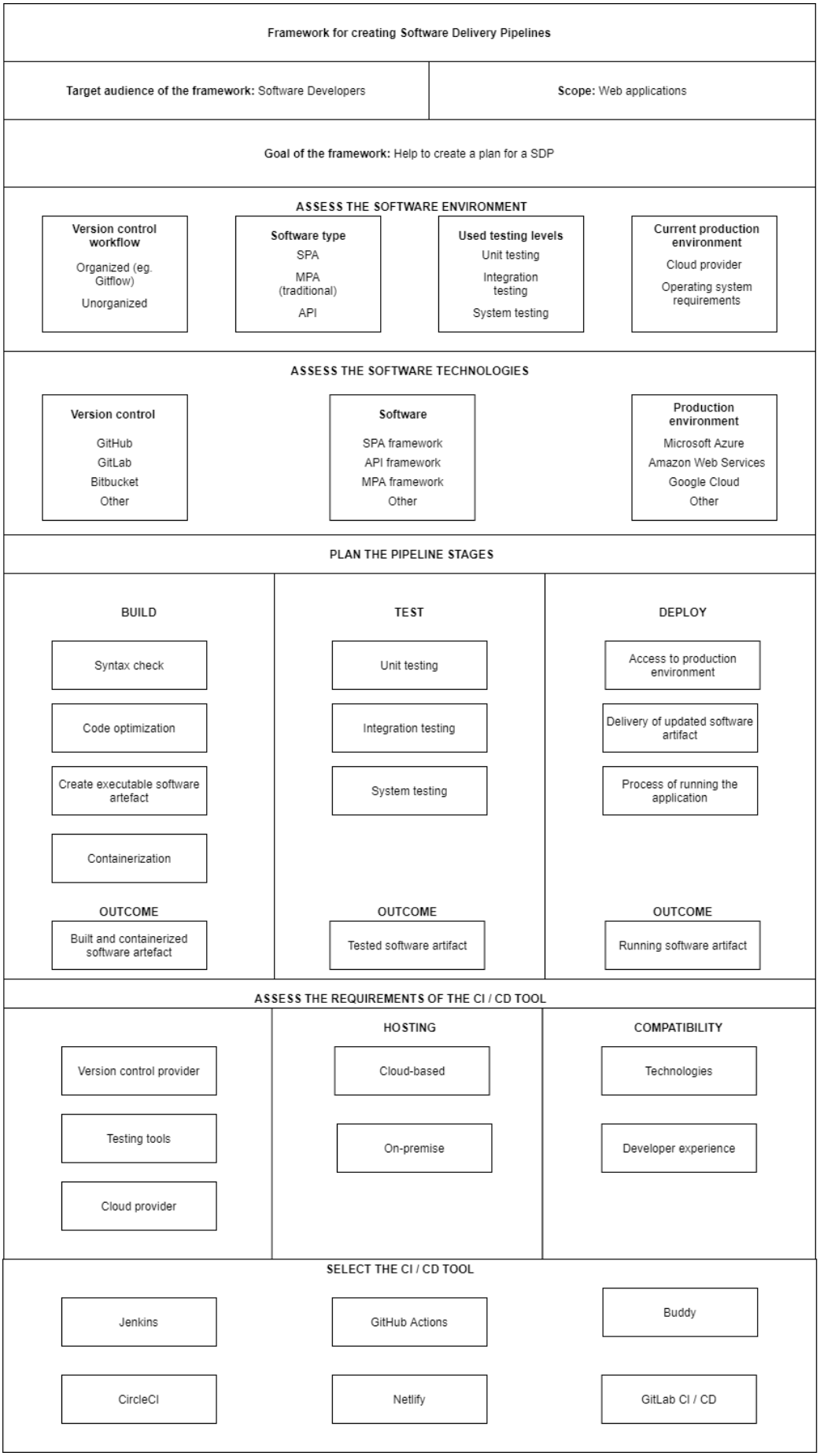


Fig. 4. The graphical representation of the framework

As it can be seen, the framework includes some general metadata (target audience, scope, and the goal) as well as five separate phases. The phases of the framework are as follows:

- Assess the software environment
- Assess the software technologies
- Plan the pipeline stages
- Assess the requirements of the CI / CD tool
- Select the CI / CD tool

By using the framework, it is possible to clarify the process of planning and implementing an SDP for a certain target software. In addition to the presented framework, an empty and fillable template is meant to be used alongside it (refer to the appendix). The idea is that the template is filled with notes that are relevant in the context of the analyzed target software. The original framework brings guidance and inspiration to the filling process - without it the process of filling the template is harder. Next, the phases of the framework are discussed and justified.

4.2 Phases of the framework

This chapter details the five phases of the framework. The phases are meant to be gone through in order - this clarifies the different aspects of planning an SDP efficiently. Three of the phases have to do with assessing the software from different perspectives - in these phases, a list of questions is included in order to help the assessment process. The idea for these questions is taken from the book by Laporte & April (2018). An example of the questions list can be found on page 525 of their book.

4.2.1 Assess the software environment

The first phase of the framework encourages the user to assess the environment of the software. This is a very high-level assessment, and it includes four categories:

- Version control workflow
- Software type
- Used testing levels
- Current production environment

Duvall et al. (2007, 7) state that a version control repository is essential in order to perform automated software delivery. Additionally, the workflow that is used is important - in an ideal situation, the main branch of version control is dedicated to versions that are ready to be deployed automatically. This kind of approach is encouraged for example in a “Gitflow workflow”. (Atlassian 2021)

The type of the software has a big effect on the SDP to be planned - a single page application can be deployed more easily compared to an API software. Therefore the type of the software is assessed in the first phase of the proposed framework.

Testing is one of the three main stages of an SDP pipeline. Kasurinen (2013) identified multiple different testing levels that are being used in software development. The different testing levels have different characteristics, and therefore they are also assessed early in the framework.

Finally, the current production environment of the software is analyzed in the first phase. Helping to deploy the software was one of the main requirements for the framework, and therefore it is important to analyze the current production environment. It is possible that the production environment has to be changed in order to build a working SDP implementation. An example of this situation arises if the SDP is not able to communicate with the current production environment. The communication lines from SDP are

explained in more detail in chapter 4.2.4. Since the change of a production environment can require a lot of resources, it is part of the software environment assessment in the framework.

The assessment should be done by an experienced software developer who is familiar with the mentioned four categories. For example, a lead developer or a full-stack developer is a suitable person to make the assessment. To help the assessment process, the following list of questions can be used:

- Is a structured version control flow like "Feature branch workflow" or "Gitflow" used?
- Is the software placed in a single or multiple version control repository/repositories?
- Does the software consist of multiple components like an API and a client application?
- What is the overall type of software?
- What testing levels are currently included in the testing procedure?
- What testing levels are planned to be included in the future?
- Is the production environment maintained by the developing company or a 3rd party provider?
- Is the production environment in a cloud environment?
- What kind of a runtime environment does the software require?
- Are there any special requirements in order to run the software?

By making this assessment, the knowledge regarding the overall software environment is improved. This knowledge is critical when planning the pipeline stages in the third phase of the framework. For example, the build stage of the pipeline depends heavily on the software type and the test stage on the used testing levels.

4.2.2 Assess the software technologies

The next phase of the framework is focusing on the technologies of the software. In this phase, a list of technologies is enough - no further analysis is needed. The technologies are

listed from the following categories:

- Version control
- Software
- Production environment

As the previous chapter states, version control is essential in order to perform automated software delivery. SDP needs to be able to communicate with the version control system - this is because the SDP triggers when a certain event has occurred in the VCS.

The software technologies also play a major part when the SDP is planned and implemented. In this section, the listing should include the software development and testing technologies. This is because all the three stages of SDP are in close relation with the software itself - different kinds of software are built, tested, and deployed in their own distinct ways.

Finally, the technologies of the production environment are listed. It is important to be aware of the technologies since as mentioned above, the SDP should be able to communicate with the production environment. As chapter 3.1.3 states, cloud platforms like Google Cloud and Microsoft Azure offer a modern way of deploying (web) software. If this phase of the framework shows that the current production environment doesn't utilize cloud services, they should be taken into consideration. This is the case especially if the current production environment needs to be changed anyway due to the mentioned communication issue.

This assessment of software technologies is easier compared to the previous assessment of the whole software environment. It can be done by a developer who is familiar with the certain aspect under assessment. For example, an IT administrator can list the technologies regarding the production environment while a software developer can list the technologies regarding the software. To help the assessment process, the following list of questions can be used:

- What version control provider is used as the remote repository manager?
- What front-end framework is used if any?
- What back-end framework is used if any?
- Which main libraries are used in software development?
- Is software containerization used when deploying software? If so, what?
- If a cloud provider is used, which one? Which cloud services are utilized?

When this assessment is successfully conducted, the exact technologies regarding the software development environment are known. This knowledge helps to for example select a CI / CD tool that is compatible with the technologies. Since there are a lot of integrations between technologies and tools, this assessment also helps to search and utilize them.

4.2.3 Plan the pipeline stages

After the first two phases of the framework, the knowledge regarding the software environment and technologies has improved. In this phase of the framework, the three stages emphasized by Humble and Farley (2016) are tentatively planned. The stages are as follows:

- Build
- Test
- Deploy

The detailed meaning of these stages has been previously explained in this thesis. The purpose of this phase is to list the different operations that are to be executed in each of these stages. For example, the build stage of the pipeline can include the creation of the software artifact as well as containerizing the software.

Each of the pipeline stages has its own suggested outcome, which can be seen from the framework. The outcome of the “build” stage is “built and containerized software artifact”. The outcomes for the “test” and “deploy” stages are “tested software artifact” and “running software artifact”, respectively. If these outcomes are realized in the SDP, it works

logically. Hüttermann (2012, 114-115) describes the steps that may be performed in an SDP; by comparing the steps to the three stages in this framework, it can be seen that they resemble each other. However, the steps by Hüttermann also show that the SDP can perform tasks like testing in multiple phases of the SDP - this is because software testing includes multiple different testing levels and methods as can be deduced from Kasurinen (2013). This also means that the three pipeline stages only act as a general guideline for the order of events in the SDP.

4.2.4 Assess the requirements of the CI / CD tool

In this phase of the framework, the requirements of the CI / CD tool are assessed from three different perspectives:

- Integrations
- Hosting
- Compatibility

The purpose of this assessment is to help gather knowledge in order to select the correct CI / CD tool, which is the final phase of the framework.

Integrations make the implementation of an SDP more efficient - many popular CI / CD tools have direct integrations to version control systems, testing tools, and cloud providers. These integrations remove the need for manual configuration almost completely. Duvall et al. (2007, 253) state that some integrations are essential to consider when selecting tools regarding SDPs.

The hosting of web software can be roughly categorized as cloud-based or on-premises hosting. Since it is necessary for the SDP to be able to communicate with the production environment (see the figure and the explanation below), this is an important consideration before selecting the CI / CD tool.

Finally, the compatibility regarding the CI / CD tool and the software development environment is assessed. This perspective is similar, but more high-level compared to the integrations-perspective that was presented above. In the context of this framework, integrations mean that two tools are able to communicate and perform actions with very little configuration. Compatibility on the other hand suggests that two tools can be used alongside each other - however, manual configuration is needed in order to perform the desired actions. Compatibility is an important aspect regarding SDP: according to Duvall et al. (2007, 253), it is important to check for example whether or not the CI / CD tool supports the version control system of the software environment.

Generally speaking, an SDP:

- Starts when an update of the software is pushed to the VCS
- Ends when the software is successfully deployed to the production environment

Hüttermann (2012, 115) describes SDP as a process that takes the software changes as its input and pipes them through the pipeline until the production environment is reached. This means that the SDP must communicate with the VCS (where the software changes are made) and the production environment (where the updated software is deployed). The figure below illustrates these lines of communication:

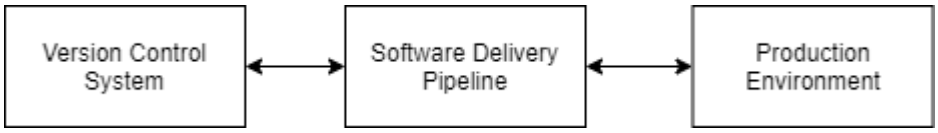


Fig. 5. The communication flow in automated software delivery

As the figure states, the SDP communicates with the version control system and the production environment of the software. Therefore, in this phase of the framework, it is necessary to understand that one of the key requirements for the CI / CD tool to be selected is that the presented communication is feasible to implement.

The framework also includes “Developer experience” in the compatibility section of this

phase. This is to remind the user of the framework that the current knowledge and experience of the software development team also plays a part when assessing the suitable CI / CD tool. For example, if there are two good candidates for the CI / CD tool from the technical viewpoint, the development team's knowledge and opinions should be taken into account when making the decision.

The requirements of the CI / CD tool should be assessed by the same person who assessed the software environment. A broad knowledge of software development is preferred. The following questions can be used to help the assessment process:

- Which integrations are available between the used version control provider / testing tools / cloud provider and the possible CI / CD tool?
- Is the used hosting environment cloud-based or on the company's premises? If cloud-based, how can the CI / CD tool communicate with it?
- How can the CI / CD tools communicate with the used version control provider?
- What CI / CD tools are the software developer familiar with if any?

After this assessment, the requirements regarding the CI / CD tool to be selected are known. If necessary, the requirements can be collected to a list. The gathered requirements help in the process of selecting the most suitable CI / CD tool, which is the next and final phase of the framework.

4.2.5 Select the CI / CD tool

The last phase of the framework is to select the CI / CD technology to be used in the pipeline. The selection should be done by utilizing the knowledge that was gained in the previous phases of the framework. Duvall et al. (2007, 246) state that even though there are multiple ways and tools to implement automated SDPs, the selection of the tool does matter.

There are a wide variety of different tools to implement SDPs for software. Duvall et al. (2007, 229-232) list various tools that were relevant at that time. However, as the field of

software development and DevOps has evolved considerably over the recent years, a lot of new tools have appeared in the markets. The following states a list of popular tools that are compatible with web software at the time of writing this thesis:

- Jenkins
- GitHub Actions
- GitLab CI / CD
- CircleCI
- Buddy

These kinds of popular tools have the largest user base and the best selection of integrations and templates. Therefore it is usually the most efficient to utilize them when implementing the SDP solution. (Jenkins, GitHub, GitLab, CircleCI, Buddy 2021) For example, if the software environment uses GitHub as the VCS and Microsoft Azure as the production environment, it makes a lot of sense to choose GitHub Actions as the CI / CD because of the high level of integration between the tools. Similarly, GitLab as the VCS and GitLab CI / CD as the SDP tool is a sensible choice.

5 DEMONSTRATION OF THE FRAMEWORK

This chapter uses a real-world software solution to demonstrate the produced framework.

5.1 Case software introduction

The case software of this thesis is called “BetterWork” - it is a software solution consisting of the following components:

- Two native mobile applications
- Web-based admin panel
- Web API
- Relational database
- Cloud-based production environment

Due to the presented scope of this thesis, the mobile applications and the relational database are not taken into account at all. The focus will be on analyzing the current state of the admin panel, the API, and the production environment. As the case software is an important part of the research that is conducted in this thesis, its development practices, software architecture, and the current state of DevOps is closely assessed in this chapter.

5.1.1 Software development environment

The case software is a relatively modern software solution - its development started in March 2020 which means that it has been developed for about 1.5 years at the time of writing this thesis. The software is produced by a start-up company, and the development team behind the software is small (two full-stack developers). Even though the case software is rather simple and the development team is small compared to many other development teams, it is a valid test software for this thesis.

5.1.1.1 Version control

The version control of the application is in GitLab, which is a cloud-based version control solution. The different components of the software (for example the web API and the

admin panel) are placed in separate repositories. The version control workflow inside the development team is following roughly the following guidelines:

- A certain branch is the “main” branch which contains the latest source code of the software
- New features and fixes are developed to a new branch, which is eventually merged to the main branch
- If possible, the branches are reviewed by the development team before merging

This workflow resembles the “Git Workflow”, however it isn’t strictly followed by the development team. (GitLab, Atlassian 2021)

5.1.1.2 Software components

The admin panel of the case software is a web-based software solution. It is developed using React.js, which is a popular web application framework. The application is bundled into static files (HTML, JS, CSS, etc.) and it can be served with any common web server software. The admin panel is an important part regarding the DevOps implementation of the case software - it is critical to be able to develop it as efficiently as possible.

The Web API of the case software is a Node.js application that provides functionality to both the admin panel and the mobile application. For example, user authentication is provided to the software by using the Web API. In addition to the admin panel, the Web API is an important part of the improved DevOps solution of the case software.

The case software also includes a PostgreSQL database, which acts as a central database of the software. The database is administered mainly using the PgAdmin GUI administration tool. The database includes

- Two database schemas (one for logging and one for other application data)
- Multiple functions (for adding data)
- Multiple tables that are related to each other in different ways (for storing and associating the data

The database development is not very structured currently. The database is modified by altering the production database with tools like PgAdmin. To recover from unexpected conditions, periodic backups are taken from the database.

Finally, the production environment of the case software utilizes Amazon Web Services (AWS) and specifically its “Elastic Compute Cloud” service (EC2). Essentially, the environment is a Ubuntu-based Linux server that hosts both the admin panel and the Web API. Nginx is used to handle the incoming requests and forward them either to the admin panel or the Web API.

5.1.2 Software architecture and containerization

As mentioned, the case software consists of multiple different software components. To design the improved software delivery pipeline, it is important to understand the architecture of the software.

The figure below describes the high-level architecture of the case software. All of the software components except the mobile applications are running inside an AWS EC2 instance. This decision has been made to simplify the administration of the software. However, it is important to notice that from the viewpoint of SDPs, this runtime environment is suboptimal. The EC2 service can be seen as the most “general” service of AWS - it doesn’t offer considerable integrations with existing SDP tools.

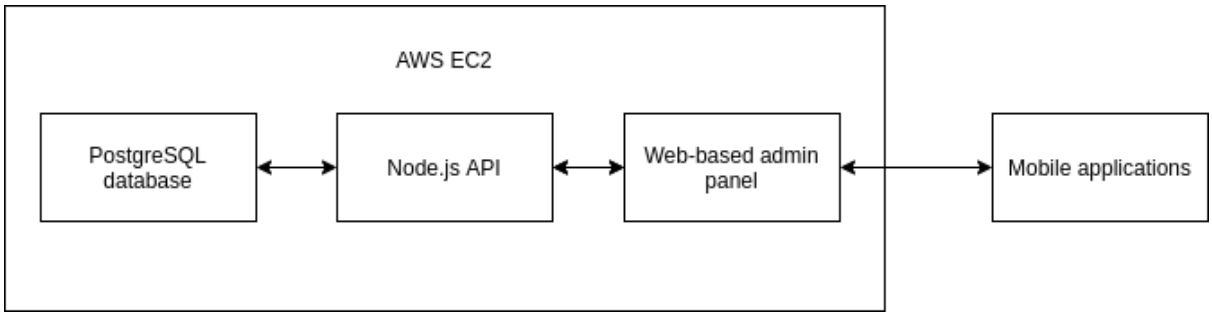


Fig. 6. Case software overall architecture

The runtime environment of the case software utilizes Docker and Docker Compose to run some of the software components. The Node.js API and PostgreSQL database are

packaged into separate containers, which are then run and managed by Docker Compose. The figure below shows the container setup of the software solution.

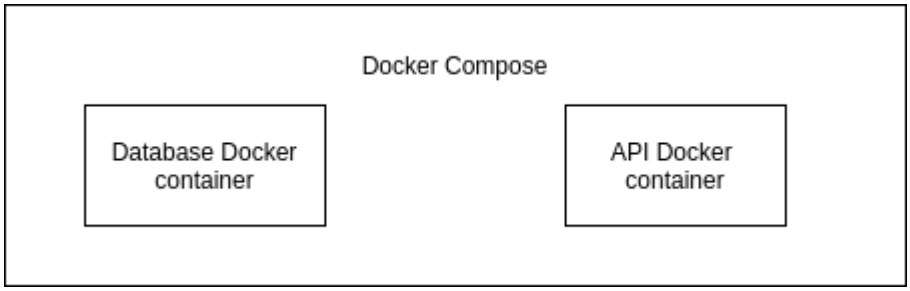


Fig. 7. Container setup of the case software

5.2 Previous software delivery processes

This chapter presents and analyzes the previous software delivery processes that were used before using the framework produced in this thesis. The web-based admin panel and the Node.js API (see previous chapters) are the most important aspects of the delivery process - therefore the focus of this thesis is in them.

Duvall et al. (2007, 3) state that one of the characteristics of continuous integration is that the process of building, testing and deploying software becomes a “nonevent”. By looking at the previous software delivery processes of the case software, it can be argued that they don’t possess this characteristic. Therefore there is a need to improve the software delivery processes of the case software. In chapter 5.3, the framework is used to help plan the improvements.

The following chapters detail the previous delivery processes of the Node.js API and the admin panel.

5.2.1 Node.js API

The previous software delivery process of the Node.js API relied heavily on manual developer labour. When a new version of the API was to be deployed, the relevant source code changes needed to be fetched manually to the EC2 instance. After the instance

contained the newest source code, the Docker container of the API was rebuilt and restarted by the developer. The figure below describes the delivery process as an UML activity diagram:

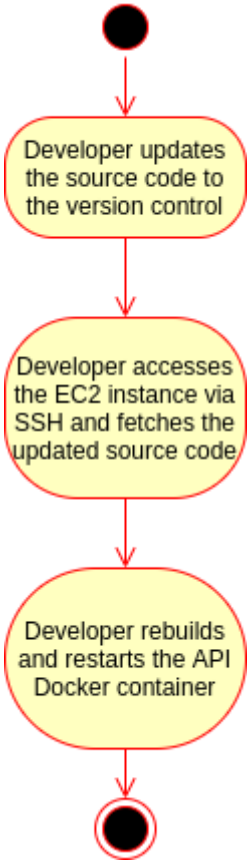


Fig. 8. UML activity diagram of the API delivery process

5.2.2 Admin panel

The delivery process of the admin panel was similar compared to the Node.js API. The biggest difference compared to the delivery of API is the fact that the admin panel isn't running inside a Docker container - this removes the Docker-related actions from the delivery process. However, there is an extra step related to building the software: the admin panel is a React.js-based "single page application", which needs to be converted into static file assets for Nginx to be able to serve the software. This conversion is usually done by the developers before updating the source code to the version control. The figure below describes the delivery process of the admin panel:

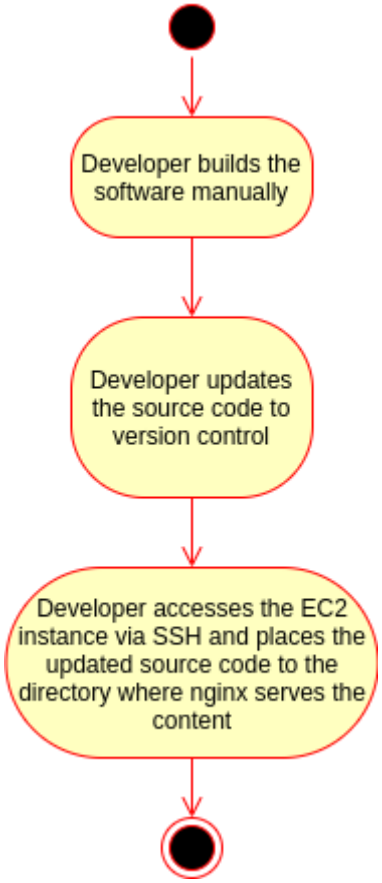


Fig. 9. UML activity diagram of the admin panel delivery process

5.3 Using the framework in the context of the case software

This chapter utilizes the developed framework to reach the objectives that were set for the SDP. The previous chapter stated the reasons why the SDP processes of the admin panel and the API should be improved.

The produced framework was used to help plan the improvements. It is worthwhile to note that the framework was used separately for the two software components. In principle, it would be possible to arrange a single SDP that would deliver both of the software components - however, since the components differ a lot regarding the technologies and the used testing levels, it is more practical to design separate pipelines for them.

The filled framework templates of the API and the admin panel can be found below:

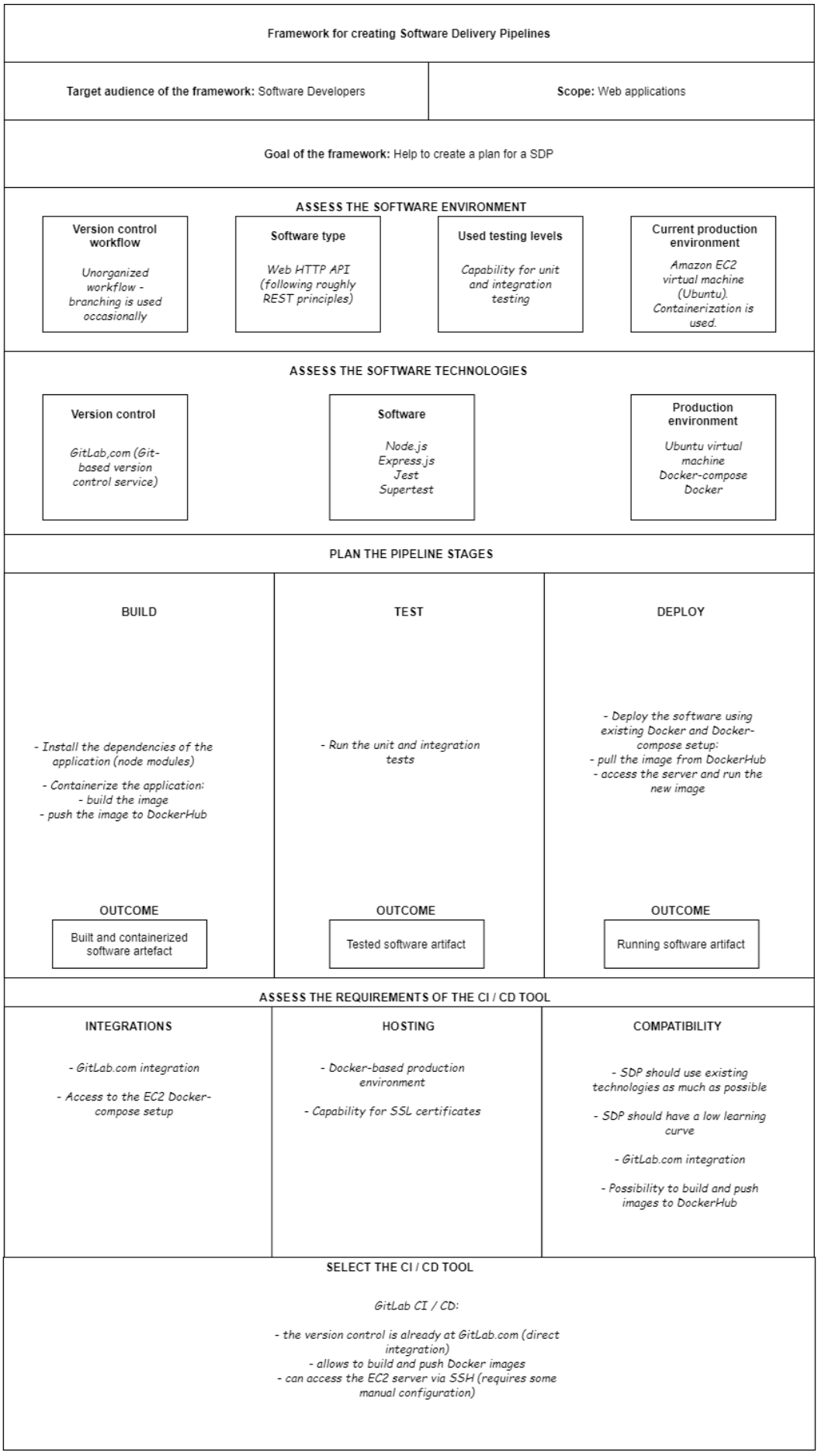


Fig. 10. The filled framework template of the API

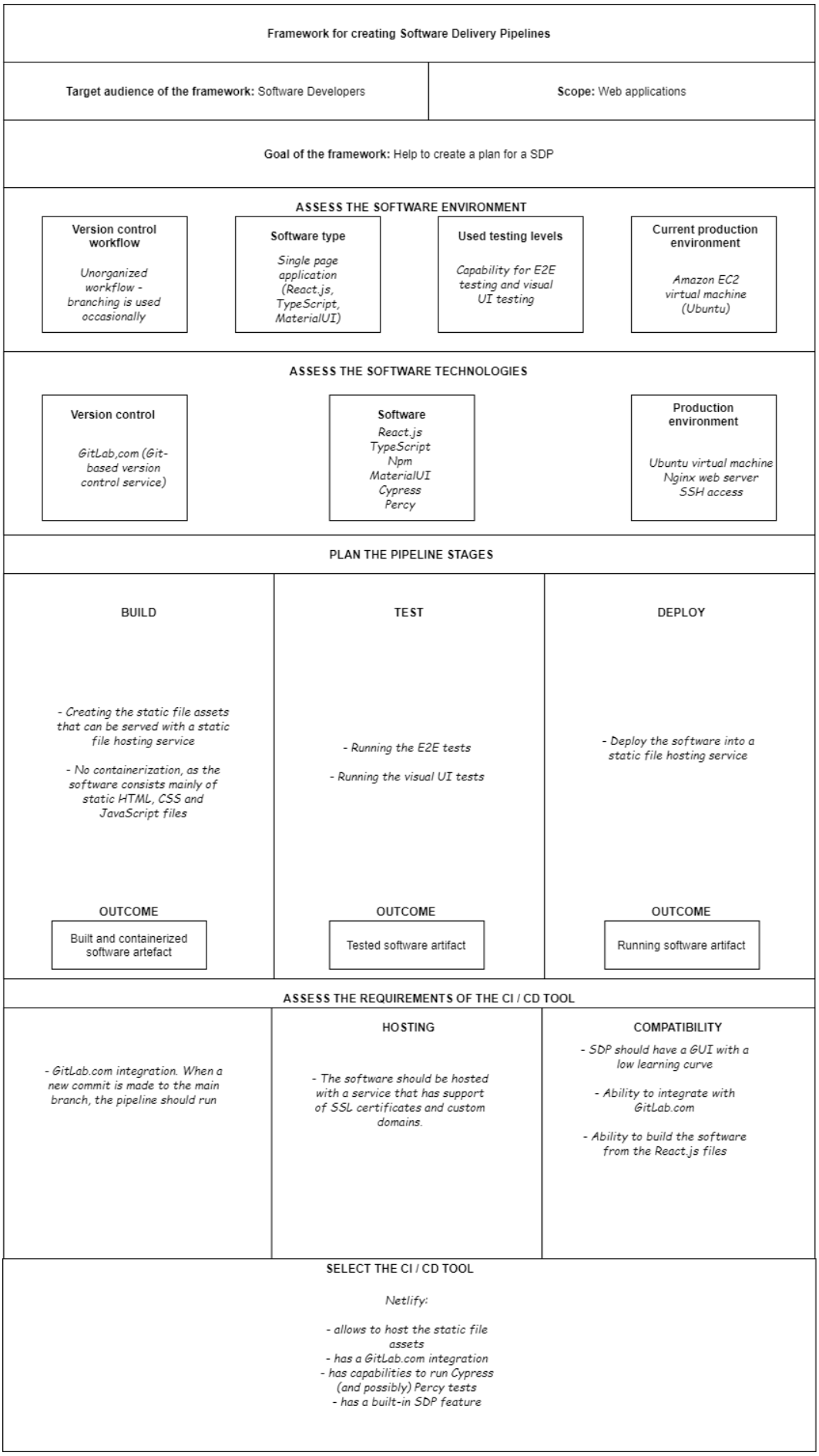


Fig. 11. The filled framework template of the admin panel

By filling the framework template, the knowledge of multiple relevant aspects of the case software was improved. The following chapter discusses how the filled framework templates were used to create plans for the SDP implementations.

5.4 Improved SDPs of the case software

This chapter of the thesis discusses the plans and concrete implementations that were made after applying the framework to the case software. First, the plans regarding the updated delivery pipelines are presented. The framework's effect on creating the plans is also discussed. After that, the concrete SDP implementations that were made based on the framework and the plans are described.

5.4.1 SDP plan for the BetterWork API

As the deployment process of the API relied heavily on manual work by a software developer, the main goal of the SDP was to improve the process in such a way that it deploys the software automatically. Additionally, since the SDP doesn't yet include testing, one of the goals is to emphasize the extendability of the pipeline. In the filled framework template it is stated that the software environment of the API has the capability for automated unit and integration testing. Therefore testing likely is to be included to the SDP in the future.

The usage of software containerization is strongly present in the SDP plan. The old deployment process included manual container management which is to be replaced with an automatic process. This means that the SDP has to build, push and finally pull the container image automatically. An external container registry was used to store the latest container image of the software. The external container registry can be treated as an additional line of communication to the already discussed ones (see chapter 4.2.4).

The detailed SDP plan for the API can be found from the Appendix B of this thesis. In addition to the points mentioned above, it includes information about the main technologies to be used. The plan concludes in a section where an SDP solution that fulfills the criteria is described. If the SDP works according to the "definition of done", the

implementation is considered a success.

There are multiple points in the plan which show the influence of the SDP planning framework. Most importantly, the notes from the phases of “Plan the pipeline stages” and “Select the CI / CD tool” are present in the SDP plan. Software containerization is a major part of the SDP, and in the filled template the high-level deployment process is already stated. Additionally; the selection of the CI / CD tool (GitLab CI / CD) is already present in the framework which helps when starting to create the plan. Also the other phases of the framework have been useful when creating a plan - even though the notes from them might not transfer directly to the SDP plan, they certainly help in improving the overall knowledge required to plan the SDP.

5.4.2 SDP plan for the BetterWork admin panel

The previous software delivery process of the admin panel also relied on manual developer labour. Therefore the main goal of the SDP is to automate the whole process. Additionally, since the application can be built into static file assets, the current production environment (EC2 instance) was seen as too complicated for it. There are a variety of services in the markets which allow the deployment of these kinds of applications with very little configuration and cost. In the previous production environment, an Nginx web server had to be maintained to publish the application. With a 3rd party service, the web server is no longer needed since the hosting is done by the service.

Since the admin panel application is using Cypress E2E testing technology, the pipeline plan contains a phase of the pipeline where the E2E tests are run. Currently E2E testing is the only testing level that needs to be supported; however, the pipeline should be planned in a way that further testing is possible to be performed.

The detailed plan for the admin panel can be found from the Appendix C of this thesis. It contains information about the observations made above - in addition, the used technologies and the “Definition of Done” is provided.

The framework has also had a clear influence on the SDP plan in the case of the admin panel. Once again, the phases of “Plan the pipeline stages” and “Select the CI / CD tool” are most clearly present. In the case of the admin panel, the fact that the software is built into static file assets is the most important note in the framework. This is because it allows the software to be deployed into a static file hosting service. Additionally, the fact that the CI / CD tool is selected before making the plan makes the planning process easier. Other phases of the framework have their influence in the SDP plan as well; for example, in the “Assess the requirements of the CI / CD tool” it is noted that the software should be hosted in a service that has support for custom domains and SSL certificates. Netlify can fulfill this requirement, and thus it can be used as the CI / CD tool as well as the hosting service of the software.

5.4.3 Concrete SDP implementation of the BetterWork API

The SDP plan for the BetterWork API includes six steps (see appendix B). As mentioned before, software container management is clearly present in the steps, and automating it is the hardest task of the implementation.

GitLab CI / CD is used as the technology of the SDP. The pipeline is created by adding a special file called “gitlab-ci.yml” to the root of the project repository. This file is used to describe the stages and the actions to be performed during them. The first task of the implementation was therefore to create this file with some default configuration.

Before the software container image of the API can be built, some additional files need to be added to the project repository. These files included confidential information, and therefore they are not placed in the version control. GitLab CI / CD has a feature called “CI / CD variables” that can be used to store the mentioned files (GitLab 2021). After the files were stored as variables, they were available in the CI / CD pipeline.

After the confidential files were added to the existing software source code, it was time to start with the containerization. In order to use Docker in the SDP, some special configuration options were added to the pipeline configuration file. When Docker was usable in the SDP, it was used to first build the image of the BetterWork API. The image

was built according to a Dockerfile that was created to the repository. The Dockerfile is a file that includes a set of instructions to build customized container images (Docker 2021).

The next step after building the image was to tag it and push it to an external container registry. The pipeline needed to authenticate into the registry to push an image there - the CI / CD variables mentioned earlier were used to store and expose the credentials to the pipeline. In addition to the CI / CD variables that are defined by the user of GitLab, certain predefined variables can be used in the pipeline (GitLab 2021). When pushing the container image of the application to the registry, a predefined variable called "CI_COMMIT_SHORT_SHA" was used to tag the image. Since each commit to the version control has an unique commit hash, the image tag can be used to find the commit where the image was built from.

The final phase of the SDP implementation deploys the updated container image to the production environment of the BetterWork API. As has been mentioned before, an AWS EC2 instance is the production environment of the API. An SSH connection is established to the instance to perform the required tasks. This is achieved by storing the SSH private key as a CI / CD variable on GitLab and then connecting to the instance with it. After the connection has been established, the following operations are performed regarding container management:

- the currently running container image of the API is stopped and removed
- the latest image (that was built earlier during the pipeline) is pulled
- the pulled image is run

Since the production environment is an Ubuntu-based instance, a shell script file can be used to perform the aforementioned operations. A file called "deploy.sh" was therefore made to the project repository. When the pipeline establishes the SSH connection to the instance, the commands of that file are run. This concludes the SDP of the BetterWork API.

5.4.4 Concrete SDP implementation of the BetterWork admin panel

The plan for the SDP of the BetterWork admin panel had three separate steps (see appendix C). The SDP is less complicated compared to the API - for example, there are no tasks related to software containerization.

The most important decision regarding the new SDP was to decide on the 3rd party service that would deploy the static file assets of the application. Multiple options were available, for example Netlify, AWS S3 and Microsoft Static Web Apps (Netlify, Amazon, Microsoft 2021). Netlify was selected as the technology of choice in the SDP implementation.

Netlify offers a direct integration with GitLab.com, which means that it is convenient to configure the repository and the branch that is watched by the SDP. When an update is made to the configured branch, the pipeline will begin automatically. As the plan states, the pipeline should support E2E tests. The SDP process that is executed on Netlify is possible to be extended via “build plugins” (Netlify 2021). There is a build plugin that supports the E2E testing technology of the admin panel, which fulfills the presented requirement.

Netlify is primarily a deployment tool for static web applications. It automatically deploys the application from the configured source branch. The filled framework template also mentioned the need for custom domains. By default, Netlify offers a domain automatically; however, it is possible to be changed with an own domain name.

The API used GitLab CI / CD as the SDP tool; it had a special configuration file, where the functionality of the pipeline was described. With Netlify, a similar configuration file can be created (netlify.toml). In addition to the configuration file, many of the settings can be adjusted directly via the browser-based GUI. The setup is fairly efficient, and the implemented SDP fulfills the “Definition of Done” presented in the plan.

6 EVALUATING THE FRAMEWORK

This chapter evaluates the effectiveness of the produced framework. Evaluation is the fifth phase of the DSR process by Peffers et al (2008). In the previous chapter, the SDP was demonstrated by applying it in the context of case software. The evaluation is done by comparing the old and the new deployment process of the case software.

The quality of the planned and implemented SDP is measured by the velocity of deployment. As it was stated in the literature review, Duvall et al. (2007) identified speed as a key feature of a good CI system. Therefore the elapsed time from a committed change to the successful deployment of the software is used as the measure in the comparison. If the new SDP processes were improved by applying the framework, it can be argued that the framework reached the original research objective that was presented in the introduction of this thesis.

6.1 The old deployment process of the BetterWork API

The old deployment process of BetterWork API included manual work by the software developers. As chapter 5.2.1 states, a certain procedure was used in order to deploy the changes that were made to the source code of the software. It is worthwhile to note that in order to complete the procedure, a certain level of expertise and knowledge is required. The procedure is completed quicker by a developer that is used to it, compared to a developer who is new to the BetterWork software environment.

The old deployment process was performed by a software developer from the development team of BetterWork. The process followed the model provided in figure X. The velocity of the deployment was measured, and the result of 3 minutes and 4 seconds was obtained.

6.2 The improved deployment process of the BetterWork API

The improved process automatized the whole deployment of BetterWork API. The implementation can be considered a real SDP since it can detect a change in the source code of the software and deploy the modified software automatically.

The improved deployment process was triggered by producing a test change to the software of the BetterWork API. This triggered the deployment process automatically on GitLab. The velocity of the deployment was measured, and the improved deployment process was completed in 2 minutes and 53 seconds.

As an additional note, it is important to understand that the improved deployment process performed more actions compared to the old manual process. The biggest addition was related to software containerization and an external container registry - the improved deployment process built and pushed the container image to Dockerhub, and finally pulled it from there to the production environment. In the old deployment process, the container image of the API was built and stored in the production environment rather than in the Dockerhub.

6.3 The old deployment process of the BetterWork admin panel

The previous deployment process of the admin panel was partly manual - the developer needed to build the static file assets and place them in the correct directory on the production environment. The build process can be initiated with a certain command that can be found from the application's main configuration file (package.json). When the static files have been generated, they are moved to the EC2 instance by using a SFTP connection.

This process requires expertise and concentration, which can lead to errors in the deployment. Also, similarly to the API, a secured connection is made to the production environment which means that the developer who executes the deployment needs to have access to the instance.

The deployment process was performed by a software developer who was familiar with the process. The execution followed the steps described in figure X. The velocity of the deployment was measured similarly to the BetterWork API, and the result of 3 minutes and 1 second was obtained.

6.4 The improved deployment process of the BetterWork admin panel

The improved deployment process of the BetterWork admin panel is fully automatic. As did the process of the API, this process also fulfills the criteria of a real SDP. Compared to the old manual process, the improved one also performs E2E testing during the pipeline. The plan stated that E2E testing would be the second step of the pipeline - however, due to technical reasons, the testing is executed after the application is deployed. This means that the application is deployed even in the case where the E2E testing fails. While this is not an optimal situation, it is fairly easy to periodically check (manually) that the tests of the application are passing. Therefore the implemented pipeline can be considered a success.

This improved process was triggered by changing the source code of the admin panel and pushing the change into the version control. Netlify noticed the change and began the deployment automatically. According to Netlify's logs, the deployment process was completed in 3 minutes and 10 seconds.

6.5 Comparing the old and new deployment processes

The previous chapters describe the old and the new deployment processes of the BetterWork API and admin panel. To further illustrate the differences, a figure that presents the processes can be found below. The figure compares the deployment processes and contains the following information:

- steps that are present in the deployment process
- the time that the steps take relatively to each other
- the time that executing the process takes
- amount of commands and clicks in the manual process

In the new (and automatic) deployment process there are certain steps that are marked with a “0” as the relative time. This is because the time that takes to execute them is so small that it isn’t significant compared to some other steps. For example, authenticating into the Dockerhub takes considerably less time than tagging and pushing an image there.

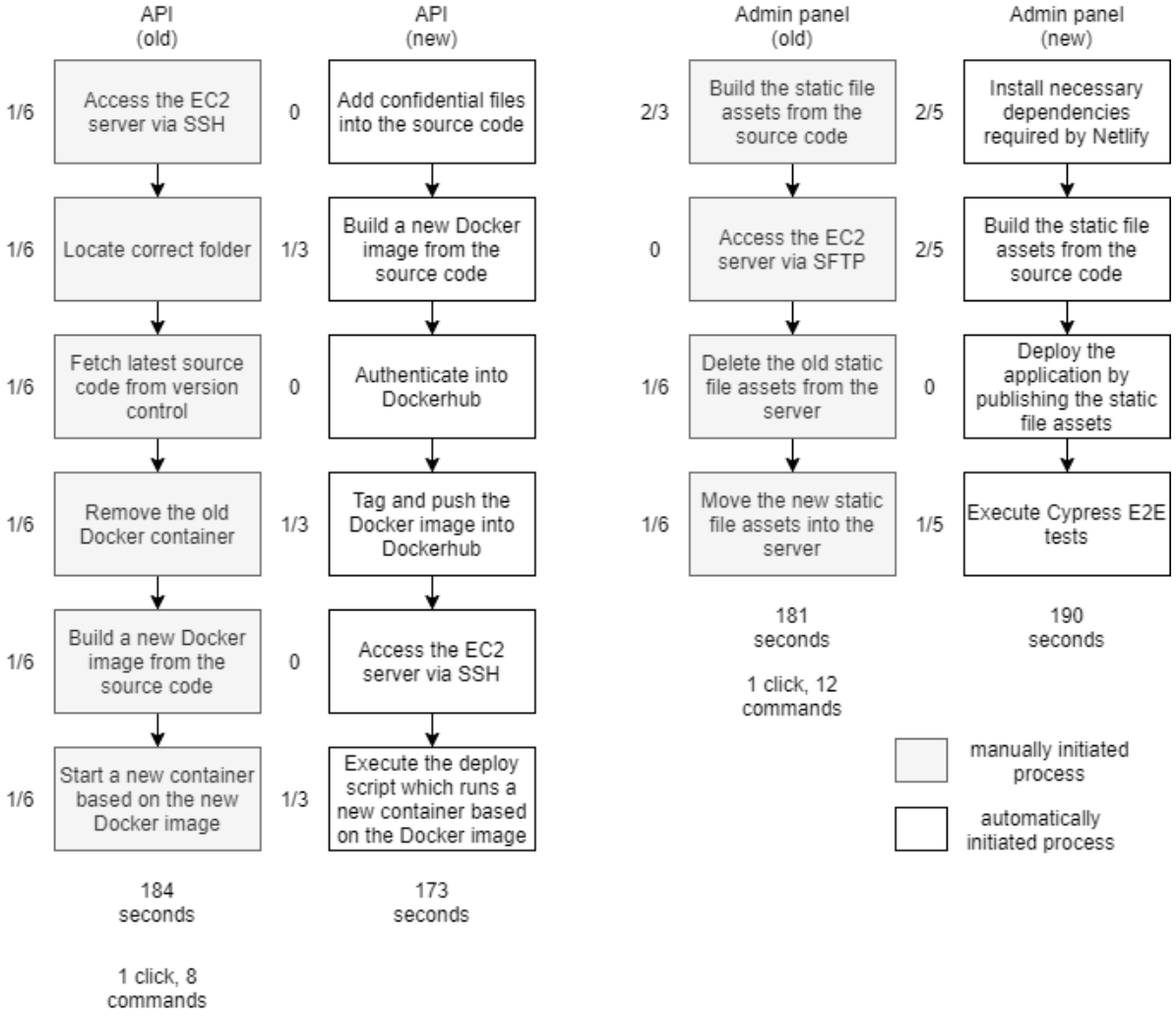


Fig. 12. The old and new deployment processes of the API and the admin panel

7 RESEARCH FINDINGS

The previous chapter describes how the software delivery processes of the BetterWork API and the admin panel were improved based on the knowledge gathered with the SDP planning framework. The table below shows what the velocity of the deployment was in the previous and the improved delivery processes:

Table 3. The velocity of deployment in previous and improved delivery processes

BetterWork API (old)	3 minutes 4 seconds (3:04)
BetterWork API (improved)	2 minutes 53 seconds (2:53)
BetterWork admin panel (old)	3 minutes 1 second (3:01)
BetterWork admin panel (improved)	3 minutes 10 seconds (3:10)

As the table shows, the velocity of the deployment process was improved in the API by automating it. In the admin panel, the velocity of deployment was not improved. One reason for this is the fact that the improved automated process included additional tasks compared to the old process.

By looking at the times it can be stated that the velocity of deployment didn't improve or deteriorate significantly. The automatization of a process doesn't therefore necessarily mean that it is done quicker compared to the situation where a human performs it. If a new version of software isn't deployed very frequently, the automatization isn't necessarily worth the effort. However, most of software companies strive to continuously deploy their application, which means that automating the process would be very efficient.

To illustrate the benefits of automating the delivery process, let's consider a situation where a software company aims to deploy its application twice each weekday. Let's also assume that the velocity of the deployment is similar to the BetterWork API that was used in this thesis. By manually deploying the application, the consumed time would be approximately 2 hours each month. When the process is automated, this time can be used

for other tasks like further development of the software. It is also important to note that the velocity of manual deployment can be significantly bigger when the deployed software is more complex than the used case software. If the velocity of manual deployment is for example 10 minutes, the saved time in the case of deploying twice a day would be approximately 6.5 hours in a month.

It is important to note that the old manual processes also introduce the possibility of human errors. The more difficult and long the manual deployment process is, the more likely the errors are. The table below shows the number of manual clicks and commands that need to be executed in order to deploy the case software components manually:

Table 4. The amount of clicks and commands in the manual deployment processes

	Clicks	Commands
BetterWork API	1	8
BetterWork admin panel	1	12

By looking at the table, the amount of clicks and commands isn't very big. However, the level of expertise has a large effect on the amount - a person who is not accustomed to the deployment process needs more commands and clicks to deploy the software.

As a final note, the main goal of the framework was to help plan an effective software delivery pipeline. The five phases of the framework help the user to analyze the software and its environment from the necessary aspects in order to plan the SDP. While the framework doesn't offer a concrete plan of the SDP as its output, the improved knowledge that is gained through applying it helps the planning process significantly.

8 DISCUSSION

This chapter is dedicated to the discussion related to the conducted research. The research was done according to the DSR methodology by Peffers et al. (2008). First, the produced framework is compared to other existing frameworks and models that were found from the sources of this thesis. Second, the results of the research are analyzed. The DSR methodology included separate phases for demonstration and evaluation of the framework - the analysis is done based on the gathered information from those phases.

In the literature review of this thesis, multiple different academic sources were presented that related to DevOps and especially SDPs. The sources discussed the topic from many perspectives. The most relevant findings from the literature review are these four models and frameworks:

- DevOps maturity level assessment framework by Gupta et al. (2017)
- Integrate button by Duvall et al. (2007)
- The process model of continuous integration by Sommerville (2015)
- Continuous integration and delivery architecture framework “Cinders” by Ståhl & Bosch (2017)

The following compares the SDP planning framework that was produced in this thesis to the presented four models and frameworks that were found in the literature review.

First, Gupta et al. (2017) proposed a framework that software practitioners can use to measure, assess and improve the maturity of a DevOps implementation. They used exploratory and confirmatory factor analysis in order to detect the two most important factors of DevOps - automation and version control. Those factors included five attributes each, and by assessing the quality of those attributes the quality of a DevOps implementation can be measured. Version control and automation are key concepts also in the SDP planning framework of this thesis. These are the main differences between the SDP planning framework and the framework by Gupta et al. (2017):

- SDP planning framework is targeted more towards “normal” scale software

whereas Gupta et al. (2017) focuses on enterprise software

- SDP planning framework has more phases - it also looks at software delivery from a higher level
- Gupta et al. (2017) has a stronger scientific validity due to a more thorough analysis

Gupta et al. (2017) is more suitable for enterprise companies who are looking to analyze their already present SDP, while the SDP planning framework is more suitable for smaller companies who are looking to implement a new SDP solution.

Second, the integrate button by Duvall et al. (2007) visualizes an automated build and delivery of software that can be initiated by a push of a button. The integrate button can be viewed as one way of describing an SDP - the three main stages of build, test, and deploy are present in it. Compared to the SDP planning framework of this thesis, the integrate button is a rather simple model. While the SDP planning framework encourages assessing the software thoroughly, the integrate button simply guides to include certain processes into an automated SDP. The integrate button is useful in situations where the concept of an SDP is wanted to be revised quickly. Even though it is difficult to implement an SDP based on it, it does inform about the purpose of it rather well.

Third, the model of continuous integration by Sommerville (2015) illustrates different steps that make up the continuous integration process. Building and testing the software are clearly present; however, the third major stage of deploying the software is missing. This is because the concept of continuous integration usually doesn’t include deploying the application. Continuous delivery is the process where the deployment of the software is performed.

By comparing the SDP planning framework and the model by Sommerville (2015), multiple differences can be found:

- SDP planning framework addresses the whole process of continuous integration and delivery while Sommerville (2015) focuses on the former

- Sommerville (2015) describes the steps of continuous integration with a higher level of detail; the SDP planning framework offers a more high-level view
- The model by Sommerville (2015) is more opinionated than the SDP planning framework

Sommerville (2015) is a good source of information regarding the steps of continuous integration. The SDP planning framework on the other hand is more useful when the whole SDP process needs to be planned and implemented. It also is less opinionated, leaving more room for case-by-case modifications.

Finally, the architecture framework "Cinders" by Ståhl & Bosch (2017) can be used to describe continuous integration and delivery systems. The framework is rather complex: it encompasses two previously created modeling techniques and offers four separate viewpoints. Compared to the SDP planning framework, using Cinders requires a high level of expertise and resources. Additionally, Cinders is targeted towards describing existing SDP solutions rather than planning and creating new ones. Cinders is useful in the case of large and complex software systems that already have SDPs - therefore it is most suitable for enterprise companies.

In addition to academic sources, frameworks regarding DevOps are being created by various commercial entities. Compared to the SDP planning framework of this thesis, they are usually created for business-related reasons - for example to market some specific DevOps-related solution.

The SDP planning framework of this thesis differs from the four main models and frameworks that were found in the literature review. The most important features of the SDP planning framework are the following:

- it helps to think about the relevant aspects regarding planning SDPs
- it is suitable for almost all kinds of web software
- it can be applied in a relatively short time

Compared to the four presented models, the main novelty value of the SDP planning framework is its ability to bridge the gap between an ideal SDP solution and a concrete plan to implement such an SDP. At this point, multiple sources define how an ideal SDP should work - however, tools to help plan the implementation of such an SDP are lacking. The SDP planning framework is also rather unopinionated, which means that case-by-case modifications are possible and even encouraged. The only limitation is that the framework should be applied only to web software.

8.1 Analyzing the results of the research

The SDP planning framework was applied in the context of the case software (BetterWork). Two software components (the API and the admin panel) were selected from it, and the framework was applied separately for them. By applying the framework, plans to create the SDPs were formed. Additionally, the SDPs were implemented in practice to assess whether the created plans were feasible to implement. The concrete implementations were evaluated, and it was found out that the deployment processes of the API and the admin panel were successfully automated.

Compared to the old and manual deployment processes, the velocity of deployment did not significantly change by implementing the new automatic SDP. In other words, when measuring a single deployment, there isn’t any noticeable time savings. However, it was stated that there will be significant time savings over time, since the manual deployment process obviously has to always be performed by a software developer whereas the automated process can be executed whenever needed without developer labour.

The amount of clicks and commands of the manual processes were also measured. It was rather low, but still the possibility of careless errors is there. One of the main benefits of automating the software delivery is the fact that the SDP works consistently - the steps of the SDP are executed in the same way each time.

In hindsight, the velocity of deployment and the amount of manual clicks and commands during a single deployment process shouldn’t have been used as the only metrics to

determine the effectiveness of an SDP. To emphasize the benefits that are gained by creating an automatic SDP, the focus should have been more on the effects that the automation has on the long term. A quick example was given in this thesis - if a company were to deploy a software like BetterWork API twice each weekday, the automated process could save about two hours each month.

To conclude this chapter, it is important to note that the conducted research also has one significant limitation; the artifact of the DSR process deals with SDPs on a relatively high level. The framework is not able to offer concrete steps to plan and implement the SDP in practice - it rather acts as a general guideline to help create an SDP for web software.

9 CONCLUSION

This thesis produced a framework for planning SDPs - the DSR methodology was used to create it. The framework was applied in the context of the case software (BetterWork): plans and concrete SDP implementations were made to two software components of it. As the outcome of the research, it was noted that the produced framework can help in planning and implementing SDPs. Additionally, regarding the concrete SDP implementations, it was observed that even though the saved time isn't significant on a single deployment operation, it grows over time and becomes significant. This holds true especially for large software systems which are deployed multiple times a day.

The most important implications of this thesis and especially the SDP planning framework come from the fact that the framework can improve knowledge regarding the case software and guide towards creating a plan for a concrete SDP implementation. Even though applying the framework does require software expertise, it is relatively easy to use given that every software company should already have suitable software developers who are developing the software under assessment. The fact that the framework is able to help bridge the gap between an ideal SDP solution and a concrete plan to implement one is beneficial to software companies who strive to improve their software delivery automation.

The original objective of this thesis was to create a framework for planning effective software delivery pipelines. The created framework was designed, presented and evaluated during the thesis. By using real-world case software, the effectiveness of the framework was tested. As a result, it can be stated that the framework helps to create SDPs - therefore the original research objective can be considered as fulfilled.

Four other models and frameworks were found in the literature review, and by comparing them to the SDP planning framework it can be said that the produced framework brings a new perspective into the existing research. The SDP planning framework is rather unopinionated, meaning that it can be applied for a wide variety of web software. It also is relatively simple – this means that experienced software practitioners can use it by utilizing their existing knowledge. Rather than introducing new complex processes and concepts, the framework focuses on clarifying the aspects that affect the SDP planning process.

The produced SDP planning framework is limited to the context of web software, it cannot be properly applied to for example native mobile or desktop applications. Also, the framework's unopinionated nature can be seen as a limitation. In some situations, the framework might not bring enough detail to the SDP planning process. Finally, it is important to note that the framework was tested only for one case software. To further validate the framework, it should be applied to multiple other software.

To address the problem of being too unopinionated, the framework could be extended in the future work. As the goal of the framework is to help plan the implementation of an SDP, it could for example offer a concrete structure for the plan. IEEE (1989) has defined a standard that defines how to construct software quality assurance plans - it could be used as an inspiration for the future work of the framework. By offering a structured template for the SDP plan, the framework could be more clear to use.

REFERENCES

Akela, A., 2016. Assembling the Key Components of Continuous Delivery [WWW Document]. DevOps.com. URL https://devops.com/assembling-components-continuous-delivery/ (accessed 4.29.21).

AL-Zahran, S., Fakieh, B., 2020. How DevOps Practices Support Digital Transformation. IJATCSE 9, 2780–2788. https://doi.org/10.30534/ijatcse/2020/46932020

Amazon, 2021. Amazon Web Services (AWS) - Cloud Computing Services [WWW Document]. Amazon Web Services, Inc. URL https://aws.amazon.com/ (accessed 6.14.21).

Atlassian, n.d. Gitflow Workflow | Atlassian Git Tutorial [WWW Document]. Atlassian. URL https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (accessed 6.18.21).

Ståhl, D., Bosch, J., 2017. Cinders: The continuous integration and delivery architecture framework. Information and Software Technology 83, 76–93. https://doi.org/10.1016/j.infsof.2016.11.006

Buddy, 2021. Buddy: The DevOps Automation Platform [WWW Document]. Buddy: The DevOps Automation Platform. URL https://buddy.works/?utm_source=dark&utm_medium=social&utm_campaign=open-graph-url-facebook (accessed 6.14.21).

CircleCI, 2021. Continuous Integration and Delivery [WWW Document]. CircleCI. URL https://circleci.com/ (accessed 6.14.21).

Cloudbees, 2020. Continuous Everything: Why Software Delivery Management Matters.

Docker, 2021. Empowering App Development for Developers | Docker [WWW Document]. URL https://www.docker.com/ (accessed 6.18.21).

DORA, Google Cloud, 2019. The 2019 Accelerate State of DevOps: Elite performance, productivity, and scaling.

Duvall, P.M., Matyas, S., Duvall, P., Glover, A., 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley.

Erich, F., Amrit, C., Daneva, M., 2014. A Mapping Study on Cooperation between Information System Development and Operations, in: Jedlitschka, A., Kuvaja, P., Kuhrmann, M., Männistö, T., Münch, J., Raatikainen, M. (Eds.), Product-Focused Software Process Improvement. Springer International Publishing, Cham, pp. 277–280.

Erich, F.M.A., Amrit, C., Daneva, M., 2017. A qualitative study of DevOps usage in practice. Journal of Software: Evolution and Process 29, e1885. https://doi.org/10.1002/smr.1885

Fowler, M., n.d. bliki: ContinuousDelivery [WWW Document]. martinfowler.com. URL https://martinfowler.com/bliki/ContinuousDelivery.html (accessed 6.18.21a).

Fowler, M., n.d. Continuous Integration [WWW Document]. martinfowler.com. URL https://martinfowler.com/articles/continuousIntegration.html (accessed 6.18.21b).

GitHub, 2021. Features • GitHub Actions [WWW Document]. GitHub. URL https://github.com/features/actions (accessed 6.14.21).

GitLab, 2021. GitLab CI/CD | GitLab [WWW Document]. URL https://docs.gitlab.com/ee/ci/ (accessed 6.14.21).

Google, 2021. Cloud Computing, Hosting Services, and APIs [WWW Document]. Google Cloud. URL https://cloud.google.com/gcp (accessed 6.14.21).

Gupta, V., Kapur, P.K., Kumar, D., 2017. Modeling and measuring attributes influencing DevOps implementation in an enterprise using structural equation modeling. Information and Software Technology 92, 75–91. https://doi.org/10.1016/j.infsof.2017.07.010

Hevner, A., Chatterjee, S., 2010. Design Research in Information Systems: Theory and Practice, 2010th edition. ed. Springer, New York ; London.

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st edition. ed. Addison-Wesley Professional, Upper Saddle River, NJ.

Hüttermann, M., 2012. DevOps for Developers, 1st ed. edition. ed. Apress, New York.

IEEE, 1989. IEEE 730-1989 - IEEE Standard for Software Quality Assurance Plans.

Jabbari, R., Ali, N., Petersen, K., Tanveer, B., 2016. What is DevOps?: A Systematic Mapping Study on Definitions and Practices. https://doi.org/10.1145/2962695.2962707

Jenkins, 2021. Jenkins [WWW Document]. Jenkins. URL https:/www.jenkins.io/ (accessed 6.14.21).

Jest, 2021. Jest [WWW Document]. URL https://jestjs.io/ (accessed 6.14.21).

JUnit, 2021. JUnit 5 [WWW Document]. URL https://junit.org/junit5/ (accessed 6.14.21).

Kasurinen, J.P., 2013. Ohjelmistotestauksen käsikirja, 1. p. ed. Docendo, Jyväskylä.

Laporte, C.Y., April, A., 2018. Software Quality Assurance.

Microsoft Azure, 2021. Cloud Computing Services | Microsoft Azure [WWW Document]. URL https://azure.microsoft.com/en-us/ (accessed 6.14.21).

Netlify, 2021. Netlify: Develop & deploy the best web experiences in record time [WWW Document]. Netlify. URL https://www.netlify.com/ (accessed 6.29.21).

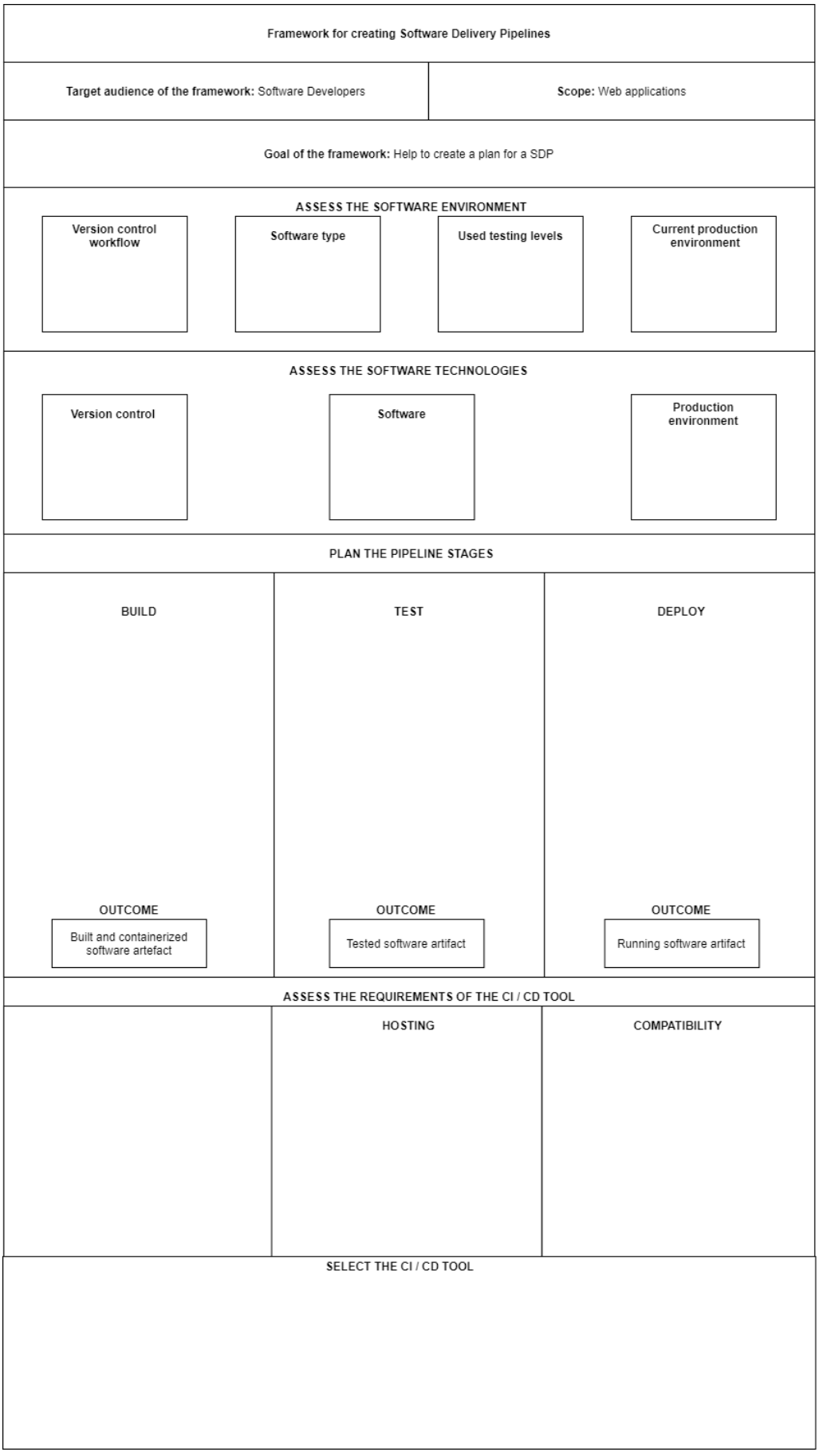
Peffers, K., Tuunanen, T., Gengler, C., Rossi, M., Hui, W., Virtanen, V., Bragge, J., 2006. The design science research process: A model for producing and presenting information systems research. Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST.

Percy, 2021. Percy | Visual testing as a service [WWW Document]. URL https://percy.io/ (accessed 6.14.21).

Sommerville, I., 2015. Software Engineering, 10th edition. ed. Pearson, Boston.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering, 2012th edition. ed. Springer, New York.

APPENDIX 1. The fillable framework template



APPENDIX 2. The SDP Plan for BetterWork API

Improved Software Delivery Pipeline Plan

BetterWork API

Joonas Ryynänen

Description

The main goal of the improved software delivery is to remove manual developer labour from the process. This will make the development of the BetterWork API more effective and feasible. Additionally; the pipeline should be implemented in such a way that it is extendable. For example, unit and integration testing should be possible to add to the SDP without major changes.

Steps of the pipeline

1. Add confidential files (containing environment variables and private keys) to the project repository
2. Build a Docker image from the software
3. Push the Docker image into a container registry
4. Access the production environment
5. Pull the pushed Docker image into the production environment
6. Restart Docker-Compose setup in order for it to use the new Docker image

Technologies

VCS: GitLab.com

CI / CD tool: GitLab CI / CD

Production Environment: Amazon EC2 instance (Docker-Compose)

Definition of Done

The SDP starts automatically, when a software developer introduces a change to the version control repository of BetterWork. The SDP performs the steps detailed above and deploys the software successfully into the production environment.

APPENDIX 3. The SDP Plan for BetterWork admin panel

Improved Software Delivery Pipeline Plan

BetterWork Admin Panel

Joonas Ryynänen

Description

The main goal of the improved software delivery is to remove manual developer labour from the process. This will make the development of the BetterWork admin panel more effective and feasible. One of the main goals is to change the production environment away from the EC2 instance. The admin panel application is a SPA which can be run with a static file hosting service.

Steps of the pipeline

1. Build the application to static file assets
2. Run E2E tests
3. Deploy application with a static file hosting service

Technologies

VCS: GitLab.com

CI / CD tool: Netlify CI / CD

Production Environment: Netlify deployment

Definition of Done

The SDP starts automatically, when an update to the source code of the software is made. The SDP builds the static file assets from the source code files, runs E2E tests and deploys the application into Netlify.