University of Magdeburg

Faculty of Computer Science

Bachelor Thesis

A Semi-Automated Release Management Process for Microservices

Author:

Mohammed Khaled

February 15, 2022

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

M.Sc. Elias Kuiter

M.Sc. Hannah Carolin Pinnecke

Department of Business Computer Science

Khaled, Mohammed:
A Semi-Automated Release Management Process for Microservices
Bachelor Thesis, University of Magdeburg, 2021.

Abstract

The release management process of microservices is an increasingly complex task in many IT companies, especially companies that use a fully manual release process because of human efforts, which are not perfect. The manual process is a time-consuming task and causes many errors because it puts pressure on humans to be correct in all their tasks and at every time. That leads to problems in software quality and can be expensive. However, automating these issues could be avoided, making the process more effective and easier to control and observe. Moreover, it makes the process faster with fewer errors. However, a completely automated process may not be possible when the process must include human activities like manual testing or manual approvals. In this thesis, we provide a semi-automated release management process for microservices to reduce the time spent to perform a release process in a specific IT company we call FinTech. We design an ABA study to compare the current manual release process with the semi-automated one and evaluate our results. Our case study shows that the semi-automated process reduces the time costs compared to the manual release process.

Contents

List of Figures

List of Tables

1. Introduction

Release management process is an essential task for a software company. The primary goal of releasing is to deploy the software in target environments and deliver it to internal clients, external clients and testers [KRH+20]. Previous release management studies have confirmed that release management is a significant challenge due to the large size of release packages, many customers with different environments and needs, and the required service availability requirements [LJ11]. Moreover, in many IT companies, the release process includes manual and repetitive tasks, which are time-consuming, with the possibility of many errors occurring [KRH+20]. Many of these challenges result from the situation that release processes often include a lot of recurring manual tasks, which are time-consuming and error-prone. Introducing solutions that automate the release process as much as possible brings the ability to verify changes, make the process reproducible across a range of environments, and essentially eliminate the opportunity for errors to creep into production [HF10, p. 29]. It also reduces stress and personnel costs while at the same time being more productive and increases the speed and quality of software releases [HF10, p. 29].

However, in the case of releasing microservices, those challenges and issues could happened more often, because we need to release several small software systems that create a system. Microservices are a highly modular and extensible architectural pattern inspired by service-oriented computing [MM+17, p. 196]. This architecture pattern is used to break down the software into individual systems, that can communicate with each other [MM+17, p. 197]. The separation into smaller components opens up the possibility to release only parts of the overall system at a time [DVDE+16]. This allows the reduction of lead times and risk involved in a single release while enabling more frequent delivery of features. Besides all of the benefits that microservices offer, this increases the overall architectural complexity and brings in new challenges in the maintenance of infrastructure and software delivery. As a result, the management and execution of software releases face challenges like increased complexity and demand for more frequent albeit smaller deliveries.

1.1 Motivation

Automation processes have become popular with many IT companies; nevertheless, many have yet to embrace application release process automation. One of those companies is the company that is a partner of this thesis, which we call FinTech. We keep the original name of that company anonymous because of marketing and privacy issues.

However, FinTech is a German startup digitizing the business model of banks and asset managers in Germany. FinTech provides asset and wealth managers with a flexible platform that allows each client to have a completely customized solution. Microservices architecture pattern is one of the most crucial architecture concepts in FinTech. FinTech uses microservices to reduce the complexity of distributed systems and separate it into smaller components, which opens up the possibility to release only parts of the overall system at a time.

Nevertheless, FinTech uses a manual release management process to deliver its microservices into multiple environments. That release process involves works on many systems like a Ticketing System (Jira[2]), a version management application for software projects (GitLab[3]) and others. Configuration and verification of infrastructure and middleware must be done. In some cases, data migration has to be performed. That high workload mentioned above causes high time and personnel cost. Also, bugs in production keep appearing because of human errors in the pre-production steps.

1.2 Goal

This thesis aims to automate parts of the existing manual release management process to make the release process more effective, reduce personnel costs and release issues, and speed up the current process. An effective release process can reduce the number of risk issues [KRH+20].

However, in our case fully automating the releasing process is not possible in our case due to administrative regulations inside FinTech. Those regulations are that someone should be responsible for the process and test it. New releases should be discussed with customers, and release conditions should be defined, e.g., when and when not a new release should be created.

After semi-automating the current release process, is an open sourcFintech can profit from lower personnel costs, deployment errors, and more productivity.

In order to achieve the previous goals, we take the following steps:

- We describe and analyze the current manual release process at FinTech. We discover the issues of the current process and find out which parts of the process we can automate.

[2] https://www.atlassian.com/software/jira
[3] https://about.gitlab.com

- Next, based on the current process analysis, we provide a solution concept for a semi-automated release process. Then, we implement that concept to be able to evaluate the concept.
- Finally, we perform experiments in the form of case studies that test the time costs between the manual and semi-automated release processes. Then we evaluate those studies to determine whether we have achieved our goals.

However, based on this thesis, IT companies that have a similar problem with the manual release process and think of automating that process, even those who do not use microservices, can primarily learn if it is worth the automation of the process; also, adapt the concept to their case.

2. Background

In this chapter, we introduce our central relevant research theme in our thesis, release process, and microservices. We describe the mentioned themes in the following chapters in more detail as needed.

2.1 Release Process Management

Release Process Management (RPM) involves people, systems, infrastructures, middlewares, and activities to plan, build, test and deploy software and hardware releases effectively into production [oGC07]. In RPM, two relative terms, deployment and release, are often used interchangeably to describe the rollout of these updates. The primary difference between those two terms is the business rationale. Deployment does not necessarily mean release [Gfa12]. *Deployment* is a technical term that is used in the domain of the software team, and it means the applying of a collection of changes into an environment [Gfa12]. This means a software is deployed when it is running in an environment. The *deployment environment* is an infrastructure or a system used for a particular purpose, where a software deployment is executed. For example :

- Development (DEV): is used by developers to try and test the changes without affecting the other environments.
- User Acceptance Testing (UAT): is the environment where the user can test and verify the software before it goes live.
- Production (PROD): is known as live. It is the environment where the end-user interacts with the software.

Release is the process by which software is made available to and obtained by its users [vdHHHW97]. The primary goal of releasing is to deploy the software in target environments and deliver it to internal, external clients and testers [KRH+20]. That

means the software is released when it is available to end-users. The releasing is more a business term, while the term deploying is a technical one.

In this thesis, we will use the term *release*, because this *release* is contains the term *deployment*. In other words, each release is a deployment, but not the other way around.

2.1.1 Phases of Release Process Management

RPM is different between software companies and should be customized for each one individually; nevertheless, there are five major phases to plan and deploy releases into a deployment environment.

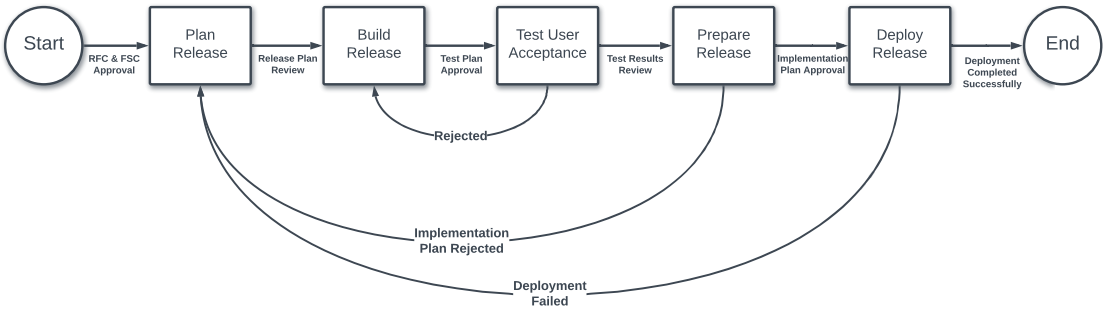


Figure 2.1: Five Major Phases of Release Management Process

Plan Release

In this phase, the release gets structured. That helps the release team to stay on track. This phase includes organizational information, e.g., what and when the changes (features) are planned to be deployed [KRH+20]. Release planning determines proper priorities and assigns features to releases.

However, this phase is sensitive because wrong decisions can result in specialist and technical problems in the end-product. That can lead to unsatisfied customers, cost time and money, and poor quality product [RS05]. Moreover, any problems in this phase will harm the subsequent phases of the RPM.

In many IT companies, the release process inspecily release planning, includes manual and repetitive tasks, which are time-consuming, with the possibility of many errors occurring [KRH+20]. That makes the Release Process (RP) ineffective. An ineffective RPM leads to a lack of control over the delivered changes and their quality, which can cause problems for both companies and customers [KMY05].

Build Release

In this phase, the changes that have been chosen in the previous phase (cf. Plan Release) get executed, built, and submitted to be integrated with the system. The release build is a compiled source code into executable files, packages or programs [Buf05].The build get released or deployed , when it successfully passes the testing phase. This phase requires a tool to get done, like the building tools Gradle or Maven, which varies for each programming language.

However, there is several methods to trigger a new build :

- Manual build trigger: In this method, the new build gets triggered manually whenever the team is ready to create a new build. This method is usually used to trigger a new build for sensitive environments like PROD or UAT, which gives us the control when we need a new build.

- Automated build trigger: In this method, a new build gets triggered automatically every time a new change comes into the version control system. This method is used to build the code for development or local testing environments like DEV.

Test User Acceptance

In this phase, the release build gets tested in an appropriate test environment known as UAT to examine if the changes work like requested. This phase gives the users access to the new changes to test them and give feedback according to specifications before they get deployed into PROD. This phase is vital to identify bugs and fix them before moving to the next phase, to reach a high-quality end-product as possible [SYM17].

Prepare Release

When the release passes the building and testing phases, the team make sure that all the conditions, documents and approvals are ready to deploy the release [KRH+20].

Deploy Release

This is the last phase of the RPM. In this phase, the release build gets deployed into PROD environment. Then it is vital to make sure that the users can use the product ideally. Any bugs or issues in the PROD should immediately get documented and fixed as a hotfix or in the next release [KRH+20].

2.1.2 Challenges of Release Processes

However, previous RP studies have confirmed that Release management (RM) is a significant challenge due to the large size of release packages, many customers with different environments and needs, Moreover, the required service availability requirements [LJ11]. A well-designed RPM is pivotal for software companies, whose primary concern is to deliver their product releases with the best possible functionality, and quality [KMY05]; An ineffective RPM leads to a lack of control over the delivered changes and their quality, which can cause problems for both companies and customers [KMY05].

Different reasons can cause an ineffective RPM, e.g., complexity in version control like microservices, growing volume, frequency of software releases, and manual management process, which are time-consuming and error-prone.

2.1.3 Semi-Automated Release Processes

Introducing solutions that automate the release process as much as possible brings the ability to verify changes, make the process faster, reproducible across a range of environments, and essentially eliminate the opportunity for errors to creep into production [HF10, p. 29] [SS18]. It also reduces stress and personnel costs while at the same time being more productive and increases the speed and quality of software releases [HF10, p. 29].

However, fully automating the releasing process is not possible in our case due to administrative regulations inside FinTech. Someone should be responsible for the process and test it. New releases should be discussed with customers, and release conditions should be defined. For example: When and when not a new release should be created. In this case, a Semi-automated release process can mitigate the release-related problems.

Semi-automated release process is a procedure that is performed through the joint activities of man and machine. [LS11]

2.2 Microservices

Microservices are a highly modular and extensible architectural pattern inspired by service-oriented computing and interact via messages [MM+17]. Service-Oriented Computing is a paradigm for distributed Computing. It harnesses the complexity of distributed systems and marge several software applications [MLM+06].

A *Microservice* is a small and cohesive independent process or application with a single responsibility that can be deployed, scaled, and tested independently. Single responsibility means it does only one thing, and it has a single reason change [Tho15].

The microservice architecture is an application built with microservices. This microservice architecture is used to break down the software into individual systems that can communicate with each other [MM+17].

As an example, an application intended to analyze some data. In this case we assume, we have three microservices, that work independently: Data-Reader , Data-Analyzer and Data-Displayer.

The first is the data-reader microservice that collects data from a data resource and then stores it in a database. However, it should not provide another functionality like analyzing or displaying the data. The second is the data-analyzer that analyzes the stored data and passes the results on to the third microservice (data-displayer) to represent the analyzed data.

However, the separation into smaller components opens up the possibility to release only parts of the overall system at a time. [DVDE+16]. That allows the reduction of lead times and risk involved in a single release while enabling more frequent delivery of features. Besides all of the benefits that microservices offer, this increases the overall architectural complexity and brings new challenges in the maintenance of infrastructure and software delivery. As a result, the management and execution of software releases face challenges like increased complexity and demand for more frequent albeit smaller deliveries, which we discuss in this thesis.

3. Analysis of the Manual Release Process

In this chapter, we analyze and describe the current situation of the manual release process in FinTech. Initial situation analysis aims at investigating all the knowledge related to the project [CRZ10]. It is essential to have enough information about the current situation, to make a comprehensive representation of the process to be analyzed. One of the most important sources of information in our case is the company's employees, as they know the work processes and have experience with data and work equipment. We discuss the situation with the employees to understand how the release process works, besides finding out which steps, systems, and infrastructures are included. Moreover, to determine what steps should get automated. We also perform the current release process ourselves to learn more about the process and discover any problems or exceptional cases not mentioned in the discussion.

Based on that information we collect, we provide an overview of the current release process by describing its workflow and its issues. After that, we illustrate the current workflow using an activity diagram (cf. Figure Figure 3.1).

3.1 Overview

Based on discussions with the employees of FinTech and doing the release process ourselves, we give an overview of the current release process in this section.

In FinTech, each team works on several microservices, and an employee is responsible for releasing these. They release the microservices manually at two stages: User Acceptance Test (UAT) and Production (Prod). This release process involves works on many systems like a Ticketing System (Jira[2]), a version management application for software projects (GitLab[3]) Moreover, the management system of containerized applications (Kubernetes[4]). Configuration and verification of infrastructure and

[2]https://www.atlassian.com/software/jira
[3]https://about.gitlab.com
[4]https://kubernetes.io/

middleware must be done. Each squad collects information about the releases they plan in tables for each squad one table. The tables are located on a page in the software Confluence [5], what we use in FinTech to as Wiki-software . We call that page a Deployment Page. Every table has the information needed about each microservice of the squad to create a new release (cf. Table 3.1). The tables overview each microservice's versions, changes, and related tasks. However, the tables should be updated before starting a new release. Because, based on the information in the tables, can the release planner find out if there are new versions or changes in one of the microservices to start a new release or, in some cases, if there is any problem that must be fixed. Nevertheless, updating the deployment table costs most time of the release process.

Finally, smoke testing, which is being done, to determine if the version is stable or not, must be done whenever the release process is finished to ensure that the new release works. This workflow should be done for each microservice, sometimes several times a day in some cases, because of the need for frequent deployments. That high workload mentioned above causes time and personnel costs which we can reduce. The process is described and modeled with more details in the next Section (cf. Section 3.2).

However, the discussion with the employees confirmed that fully automating the release process is not possible in our case due to administrative regulations inside FinTech. Someone should be responsible for the process and test it. New releases should be discussed with customers, and release conditions should be defined. For example: When and when not a new release should be created. That is why we design a semi-automated release process. The semi-automation process is a procedure that is performed through the joint activities of humans, and machines [LS11]. Besides, we decide to focus in this thesis on automating the deployment table (cf. Section 3.3), because the discussion shows that this part of the process costs the most time, between 30 and 50 minutes related depending on the number of microservices that should be released.

3.2 Process Modeling

In this section, we represent the current manual release process using an activity diagram. Activity diagrams explain the behavior and the networking between two or more elementary actions while processing activity between a user and the system. Also, they represent higher-level processes, such as how the process is working, because activity diagrams are less technical than sequence diagrams [Bel03]. Based on that, the activity diagram helps us explain the process and the networking between an employee and the different systems and between themselves using simple syntax and without going into the details and the technical actions.

The activity diagram, shown in Figure 3.1, describes how an employee plan and prepares a new release in each step, system, or infrastructure. The workflow starts with updating the table, goes to the creation of the release ticket, and ends with giving the ticket to Tech-Ops to start the deployment. In this case, Tech-Ops are the sysadmins responsible for keeping the systems running.

[5] https://www.atlassian.com/de/software/confluence

The diagram (cf. Figure 3.1) includes four swimlanes. Each swimlane represents a system or an infrastructure in the process (Jira, git, deployment table, Kubernetes) (cf. Section 3.1). The workflow starts with updating the deployment table (cf.Table 3.1). After opening the deployment page, where the deployment tables are located, we start working on the squad table. Every row in the table represents a microservice; we update each row (microservice) individually.

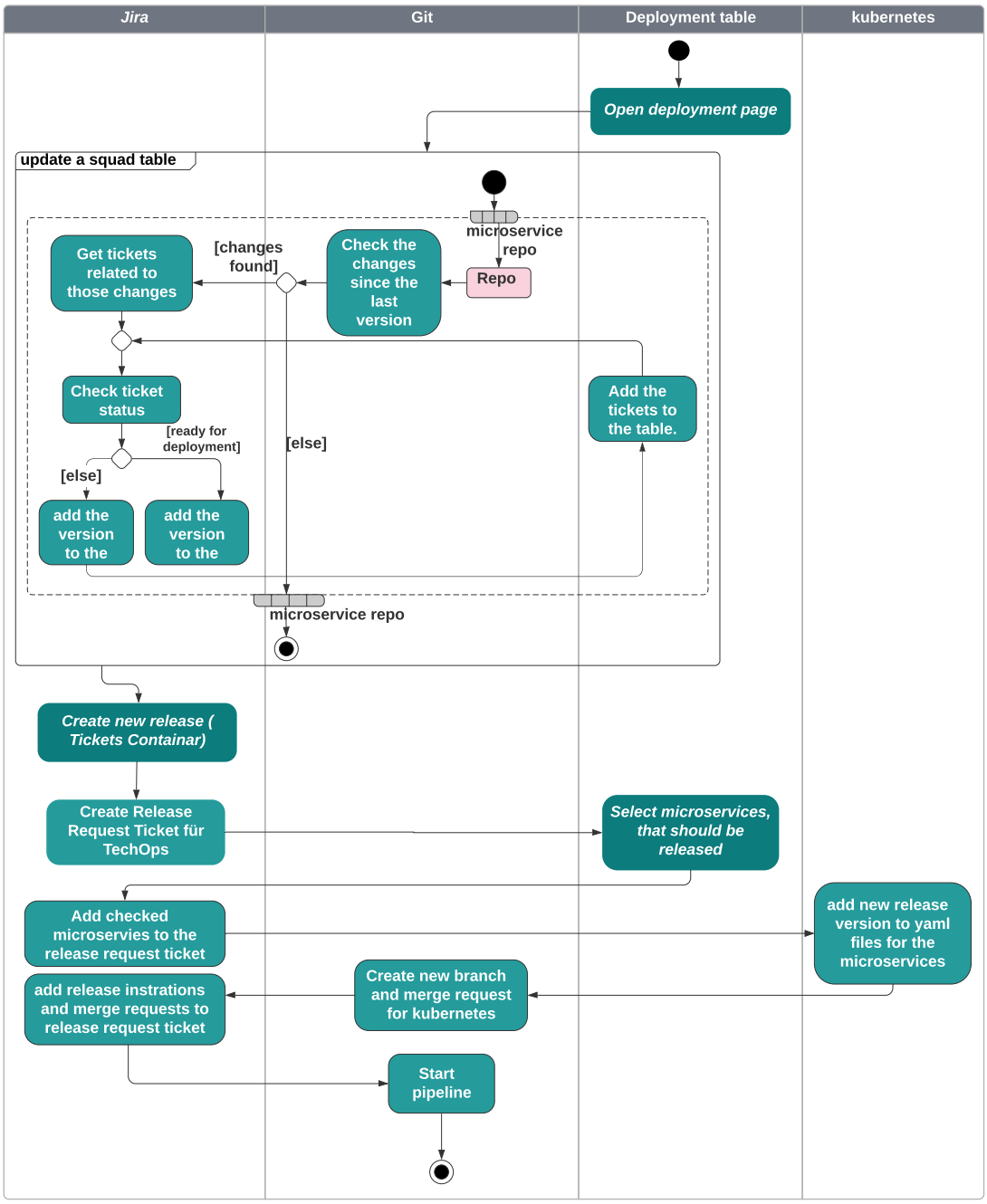


Figure 3.1: Activity diagram description the manual release process

For each microservice, we go to Gitlab and open the microservice repository and check the changes since the last version. If no changes are found, we move to the next microservice.

However, if there are indeed changes, we go to Jira to access the tasks related to those changes. The status of these tasks gets checked. If the ticket is ready for deployment, there is no need to change anything in the row. However, if the task is not ready for deployment, it will be added to the table. Anyhow, in both cases, the version of the microservice will be added to the task. That work should be done for every microservice in the table.

When the updating of the table is finished, a new release in Jira will be created. The release in Jira is a type of container for the tasks that need to be released.

After that, a release request ticket will be created for TechOps and add the microservices that need to be released to it. The next step is to go to Kubernetes, update the version in the config files for the microservices, and create a merge request for these changes in Kubernetes. Besides that, the release instructions and merge requests will be added to the release request ticket we created before a few steps (cf. Section 3.2).

This process finishes with starting the pipelines in GitLab.

3.3 Manual Deployment Table

In this section, we describe in detail the deployment table and its usage in the release process using an abstract view of the table (cf. Table 3.1). Moreover, we illustrate and explain the updating process using a workflow chart (cf. Section 3.3.2).

However, the release process includes some exceptional cases, difficulties, and problems. We present these also in this section and explain how to solve them (cf. Section 3.4).

3.3.1 Overview

As mentioned before (cf. Section 3.1), the employees use the table to get information about the microservices and to document the release process. The Figure (Table 3.1) shows an abstract view of the deployment table updating. The table has four columns, and each row represents one microservice. Three cells of the row should be updated (Prod - UAT - Pending tasks in git history).

Prod and UAT should have the version number of the microservice in the related stage. That gives an overview of the versions on the stages. It helps to compare that versions with newer ones that have not been released yet.

Pending tasks in Git history include every version in Gitlab greater than the version in the stage. Also, every unfinished task relates to that version. This information helps decide if it is possible to start a new release. Because releasing unfinished work or wrong tasks will cause bugs and losing time.

The table generally gives an overview that shows the possibility of creating a release for each microservice, besides the history of old releases.

Table 3.1: Abstract tabular view of the deployment table

Microservice	Pending tasks in git history	Prod	UAT
service1	90.1.0	89.1.0	89.1.0
service2	92.2.0	92.2.0	92.2.0
service3	90.1.0 PLC-0000 Foo [PM TEST] 89.1.0 PLC-0002 Foo [PM TEST]	88.1.0	88.1.0

3.3.2 Manual Updating Process of Deployment Table

To understand how the updating of the Deployment Table works and which systems are included in this job, we model a workflow chart that represents the updating process of the table (cf. Figure 3.2).

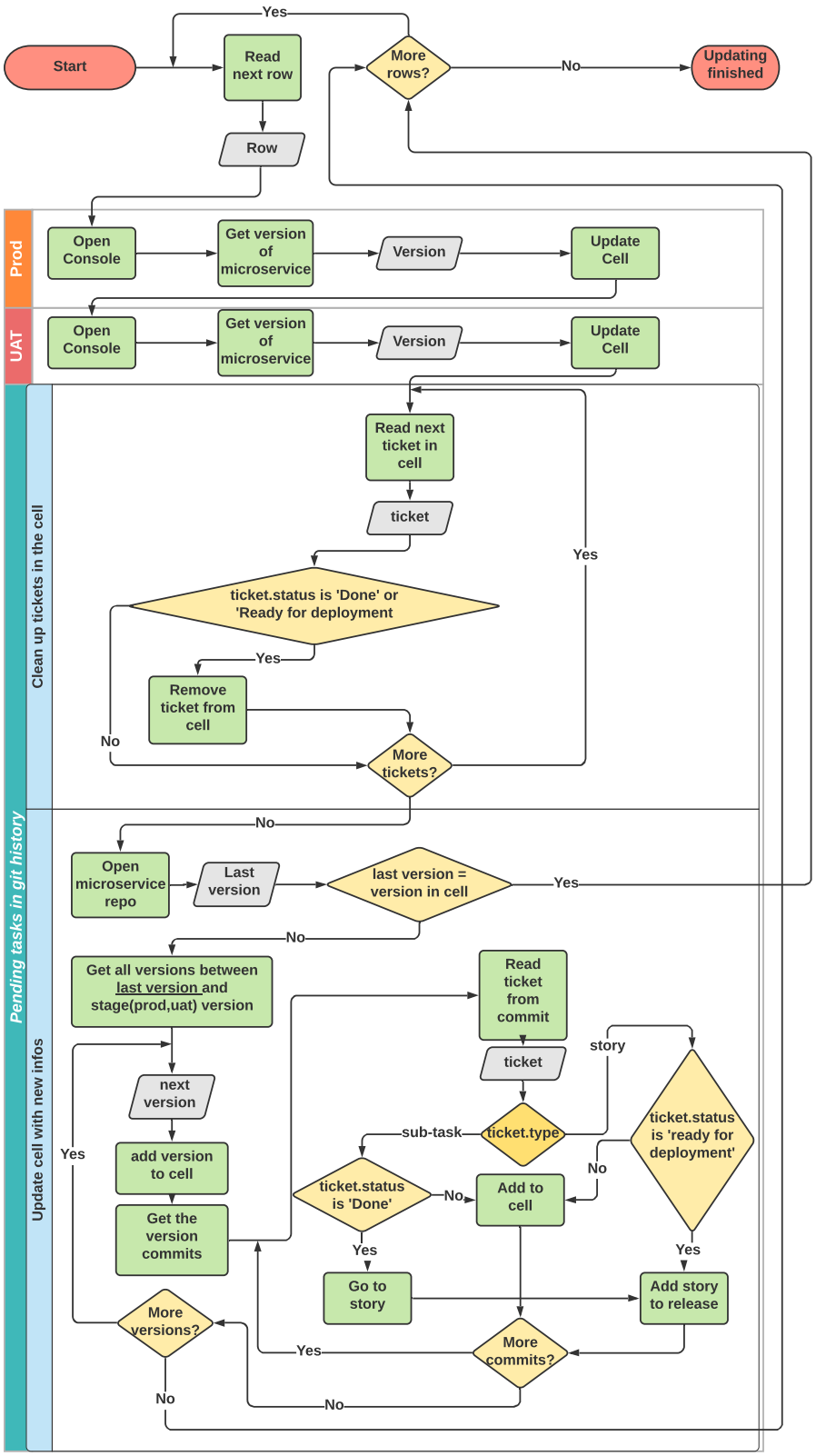
We use swimlanes with the workflow chart to structure the diagram in specific parts. Each swimlane represents a column in the deployment table (Prod, UAT, Pending tasks in git history) (cf. Table 3.1). Also, each row in the table represents a microservice.

The updating starts with updating the columns (Prod and UAT) for each microservice. As mentioned above (cf. Section 3.3.1), Prod and UAT include the version number of the microservice in each of these stages. To update the version number, we use the console and send a request to Kubernetes to get the version number of the microservice; then, we update the cell with the new version. This workflow is valid for both stages Prod and UAT.

Next, the column (Pending tasks in git history) will be updated with the new versions of the microservice and the related tasks. Before starting adding any new information to this column, the column needs to be cleaned up. To clean up the column, we check every ticket in the cell. Every ticket with the status (Done or Ready for deployment) will be removed from the cell.

After cleaning up the cell, we start adding the information about the new versions of the microservice. That starts with opening the repo of the microservice in GitLab. We get the last version from the last tag in the repo and compare it with the current version in the environments. We use tags in GitLab to archive the versions of the microservices. The tag is a text label that can be assigned to a specific revision. If they are equal, then we move to the next row (microservice). Nevertheless, if the versions are not equal, we check every newer version than the current one and add it to the cell. For each new version, we get their commits, and for each commit, we check the status of the related ticket based on the type of the ticket (story or sub-task). If it is a story, we check if the status is ready for deployment; if yes, the story will be added to the release, but if not, we add the story to the cell. However, if the ticket is a sub-task, we check if it is done. If yes, we move to the story of this sub-task and add it to the release, but if it is not done, we add the sub-task to the cell. However, we determine the related tickets using the commit message.

Figure 3.2: Flowchart for updating a deployment table



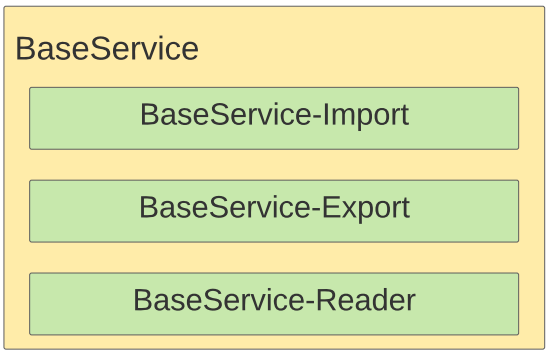
We use the commit message to bind commits to their related Jira tickets by putting the ticket key as a prefix in the message. For example, "PLC-100: project created". This commit message starts with a Jira-Ticket key, which means that the commit has requested changes in that ticket. For each new version, we get their commits, and for each commit, we check the status of the related ticket based on the type of the ticket (story or sub-task). We determine the related tickets based on the commit message. This updating workflow should be done for each commit of the new versions.

3.4 Exceptional Case in Multi-Module Projects

FinTech uses Maven[1] to manage multi-module projects. Maven is a package manager and builds automation tools for Java projects. Maven allows to individually build and deploy multiple modules in a specific order [SVHMB21].

We use an example to explain the case. The Figure 3.3 shows an abstract example of a multi-modules service, where the base service contains three modules. In this case, we have a problem finding out which commit is bound to which module. As

Figure 3.3: Abstract multi-module service



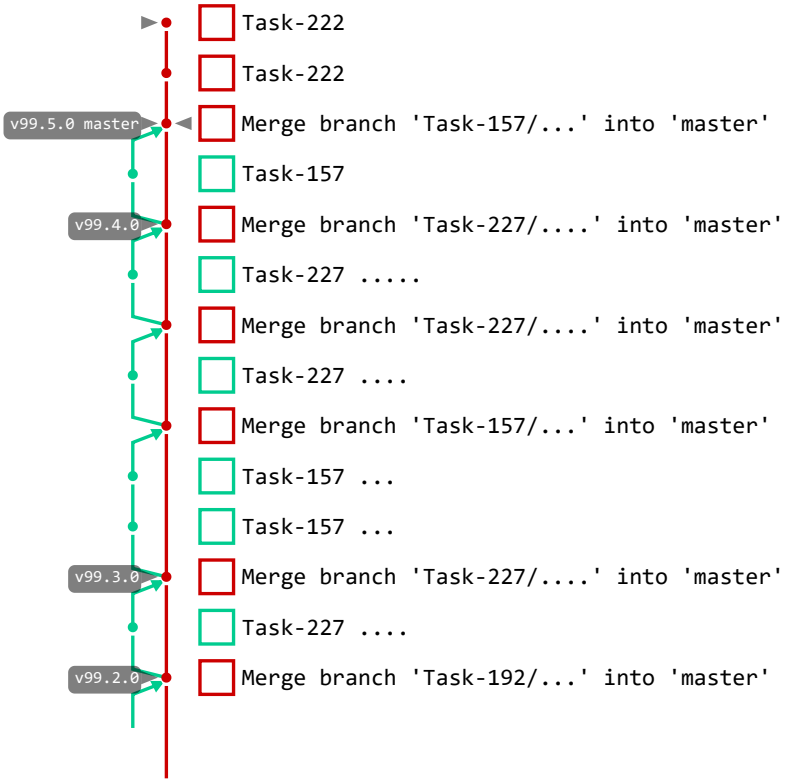
shown in Section 3.3.2, we read the git commits of service to determine the versions of this service and which task is related to this commit. Nevertheless, because we have multiple modules in the service and each module should be individually released and get a version, we have the problem of determining which modules have which commit.

To understand the problem more, we present an abstract Git graph (cf. Figure 3.4) for the previous multi-modules service example (cf. Figure 3.3). In this case, we can not determine which commits or versions (tag) are related to which module. That causes a problem for the updating process of the deployment Table 3.1.

To solve this problem, we can check the task related to the commit and use its label to determine the module. However, the tasks are not always labeled with the module name. We can also check for the pipelines for every commit to find out which modules have been addressed. The safest way to solve this problem is to check the commit changes to determine which service (module) is affected. Nevertheless, this way is relatively slow, especially with commit with significant changes.

[1] https://maven.apache.org

Figure 3.4: Abstract Git graph for multi-module service



4. Automation of the Release Process

In Section 3.2 we described how the current manual release process works and what problems and challenges. This chapter provides a concept for a semi-automated release management process. We design a constantly updated tabular view with several required views and functions.

4.1 Concept

The basic concept solution idea is to have a semi-automated process that merges the work between the employees and the computer. As mentioned before (cf. Section 3.2), we have several steps that the user (employee) should go through to create a new release.

Since updating the deployment page is the most complex step compared to the other steps, and it costs the most time in the current release management process (cf. Chapter 3), we focus in this thesis on the replacing of the existing deployment page (cf. Section 3.3) by a continually updated tabular views with the functions of searching, grouping, and filtering.

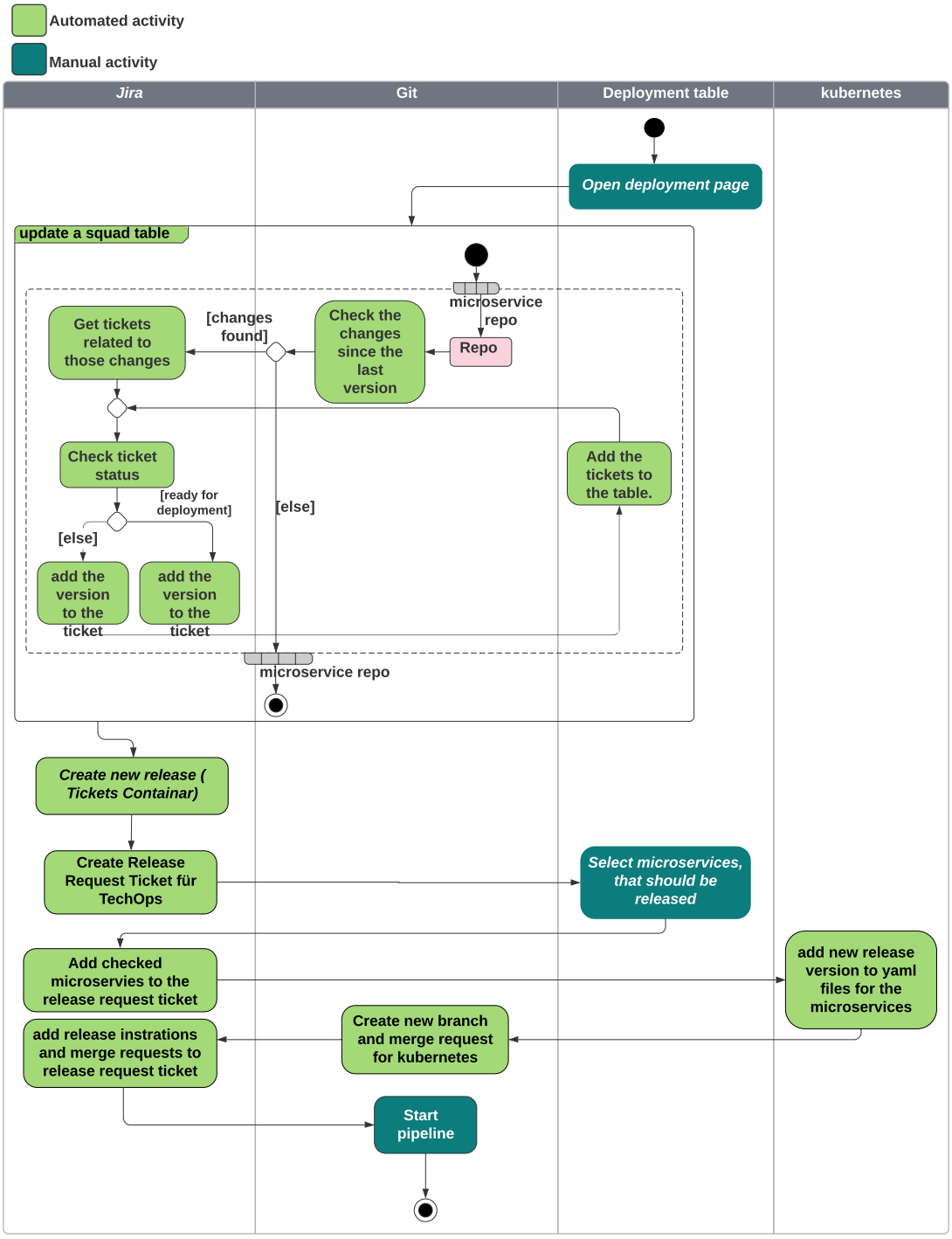
On the other side, selecting microservices that should be released and starting the deployment pipelines are the steps that must stay manually because of the conditions we mentioned before (cf. Section 3.1).

However, the other steps of the process are not part of this thesis. We have mentioned that those steps can be automated in the future (cf. Chapter 7).

The following flowchart (cf. Figure 4.1) represents which steps we can automate; comparing with the current manual process(cf. Figure 3.1). The chart shows that the automation of the specific parts reduces the work that the user should do (manual activities), and also it ends the switching between the different services. Furthermore, the automation reduces the manual work updating the deployment page, which we think can reduce the time costs of the release management process.

As mentioned above, we replace the current deployment page with a continually updated tabular view. We replace the one deployment page that includes multiple

Figure 4.1: Semi-automated release process concept activity diagram



tables for each squad with continually updated tables that all squads can use. Each table has a specific purpose and includes information needed to create a new release. We think the use of those tables stops the need to use other systems or infrastructures that we mentioned above, which we currently use to collect the information (cf. Chapter 3).

However, to solve the exceptional cases and problems we mentioned above (cf. Section 3.4), we need to make some part of the process configurable, which means we need to set configurations manually for our process. We provide more details in the following parts.

In the next part, we provide the tables and the information that should be included to make the table usable instead of the current deployment page and reduce the time costs, which is the primary purpose of this thesis.

4.2 Tabular View

In this section, we describe the tables we mentioned the previous Section 4.1. (Microservices table, commits table und tasks table). We provide the columns each table includes and show why we need each of those tables and how they can be helpful and save time for us. Furthermore, we give an example for each table in different cases.

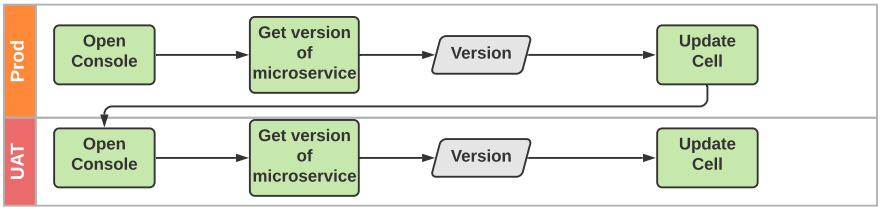
Those three tables are always up to date and give all the information needed to manage the next deployment. The user can customize the tables according to his needs. Using these tables, the user can select safely which new changes are ready for deployment for each microservices.

4.2.1 Released Microservices Table

The released microservices table gives us an overview of the currently deployed microservices, including their version numbers in the different environments and their related squad.

This table replaces the two swimlanes from the manual updating process of a deployment table Figure 3.2, that are responsible for updating the version number in each environment. The Figure 4.2 is a part of the Figure 3.2 and shows the two swimlanes this table replaces. That means that the table replaces the usage of other systems like Kubernetes or the Console to find out if a microservice is already deployed in a specific environment and what version number it has.

Figure 4.2: Replaced swimlanes of the Figure 3.2 by the microservices-table



However, this table consists of two static columns (microservice, squad) and an unspecified number of other columns, based on the count of the deployment environments we have; In our case, we have three environments (STAGE, UAT and PROD).

We describe those five columns and show their usage with an example. The following Table 4.1 represents an abstract microservices table. The table shows us an example of released microservices and their version numbers in each environment. That replaces what the current deployment table (cf. Table 3.1) provides as for the version numbers in several environments. Additionally, the table has a squad column representing what squad is responsible for what microservice.

Each squad finds its microservices based on the squad column by filtering or grouping the table by the squad name. Those filtering and grouping should be saved as templetes for each user to save time the next time we want to create a new release.

For example, users from squad Green want to start a new release; they can create a template that shows only the microservices they need. In this case (MDR-Import and MDR-Export). We think they save time using this template the next time, mainly when the table includes a big count of information that is not relevant to them.

Table 4.1: Abstract microservice table

Microservice	Squad	STAGE	UAT	Prod
MDR-Export	Green	32.0.0	30.0.0	30.0.0
MDR-Import	Green	31.0.0	31.0.0	30.0.0
RT	Blue	44.2.0		

Also, this table gives us a customized overview of the microservices' version numbers that have already been released. This information is essential to determine if a microservice has new changes we have to release by comparing the version numbers with the local ones.

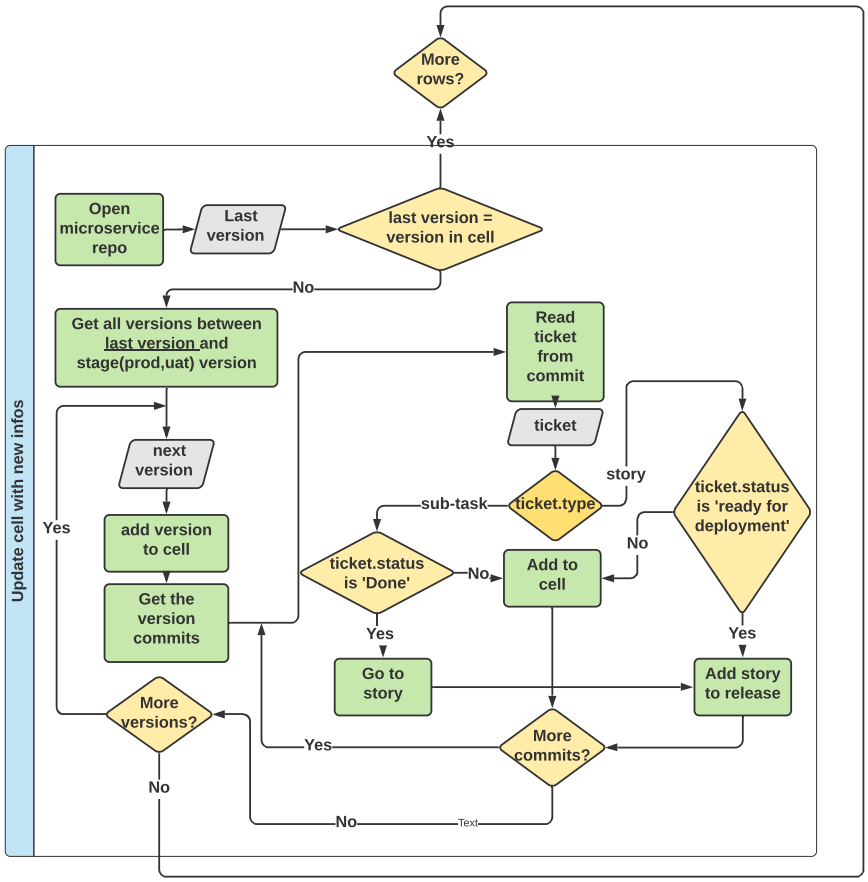
4.2.2 Deployment Table

The deployment table represents the combined information we usually get from Gitlab, Jira, and Kubernetes (cf. Section 3.3.2). This table shows whether a microservice needs to be released, based on information we collect from commits in Gitlab, Tickets in Jira, and the current version number from Kubernetes. We automate the workflow of the lastest swimlane in the current manual updating process of a deployment table Figure 3.2, which is responsible for reading data from GitLab and Jira then adding it to the table.

In the following Figure 4.3, which is the last swimlane of the Figure 3.2, we see the work we replace and reduce by this table, which is continuously updated. That means we do not need to jump between Gitlab's projects and commits, Jira-Tickets or Kubernetes-Console, and redo this workflow for every microservice. We believe that this table reduces the time costs, because of the reduction of the manual work of the users.

However, as mentioned before, GitLab includes projects, and each project contains one or multiple microservices. The commits are related to the project, not to the microservice, which means, in cases of multimodule projects, we need to find out

Figure 4.3: Replaced swimlane of the Figure 3.2 by the commits table



what commit of the project is related to what microservice. We solve this issue by checking the changes in each commit and mapping these changes to the relevant microservices (cf. **??**). To automate this part of the work, we need to define manually what paths of the project repository are relevant to what microservice, then we can compare the changes in the commits with that definition to determine what commits are related to what microservices.

Moreover, because we use the commit message to find out the Jira tasks that are related to that commit (cf. Section 3.3.2), besides, the developer can make mistakes while writing the commit message, we make the relationship between the commits and their tasks overwritable. We describe that issue in more detail in the data model section (cf. Section 4.3).

The following Table 4.2 provides an example of what this table includes and what each column provides.

Suppose that users from squad Blue want to create a new release for RT. They can directly filter by squad or microservice or use a saved filtering template. The tables show that RT is up to date when we compare the latest version number of the latest changes with the current version number in the microservices table (cf. Table 4.1). We also can see that there are no non-released Jira Tickets in STAGE; However, if

they want to create a release for UAT or PROD, they already have the information that there are no releases for this microservice (RT) in those environments.

Another example, suppose that the green squad wants to check if new changes should get released. The table shows that both microservices MDR-Export and MDR-Import have new changes that have not been released yet in PROD. MDR-Export has unready changes with ticket PLC-458, which means that changes of the commit related to this ticket are not safe to be released. However, version 32.0.0 with the ticket PLC-416 is only released on STAGE and needs to be released for UAT and PROD. The same version with the same ticket is not released on any environment for the MDR-Import microservice.

Table 4.2: Abstract automated deployment table

Squad	Microservice	Version	Ticket	Status	Summary	Released On
Green	MDR-Export	NULL	PLC-458	in Progress	update foo	
Green	MDR-Export	32.0.0	PLC-416	Done	make foo	STAGE
Green	MDR-Import	32.0.0	PLC-416	Done	make foo	
Green	MDR-Import	31.0.0	PLC-379	Done	create foo	STAGE - UAT
Blue	RT	44.2.0	PLC-213	Done	fix foo	STAGE
Blue	RT	44.2.0	PLC-219	Done	add foo	STAGE
Blue	RT	44.2.0	PLC-218	Done	remove foo	STAGE

4.3 Data Model

The previous section provides the tables we need in our process and which information they have. In this section, we provide a data model for those tables.

The following Figure 4.4 represents our data model in an entity-relationship diagram. We describe each entity and its attributes in separate sections; Then, we provide an example for each entity.

4.3.1 Project

$$\mathcal{R}(\text{ Project }) = \{\underline{\text{ id }},\ \text{name }\} \tag{4.1}$$

This entity contains information we collect from GitLab about the project. The projects include the *Commits* (changes) in the code we want to release, besides the *Tags* that we use to archive the versions. We need the projects to map it to the *Microservices* because the project includes the commits we want to include in a release process of a microservice. Each project has at least one microservice (cf. Section 4.3.4).

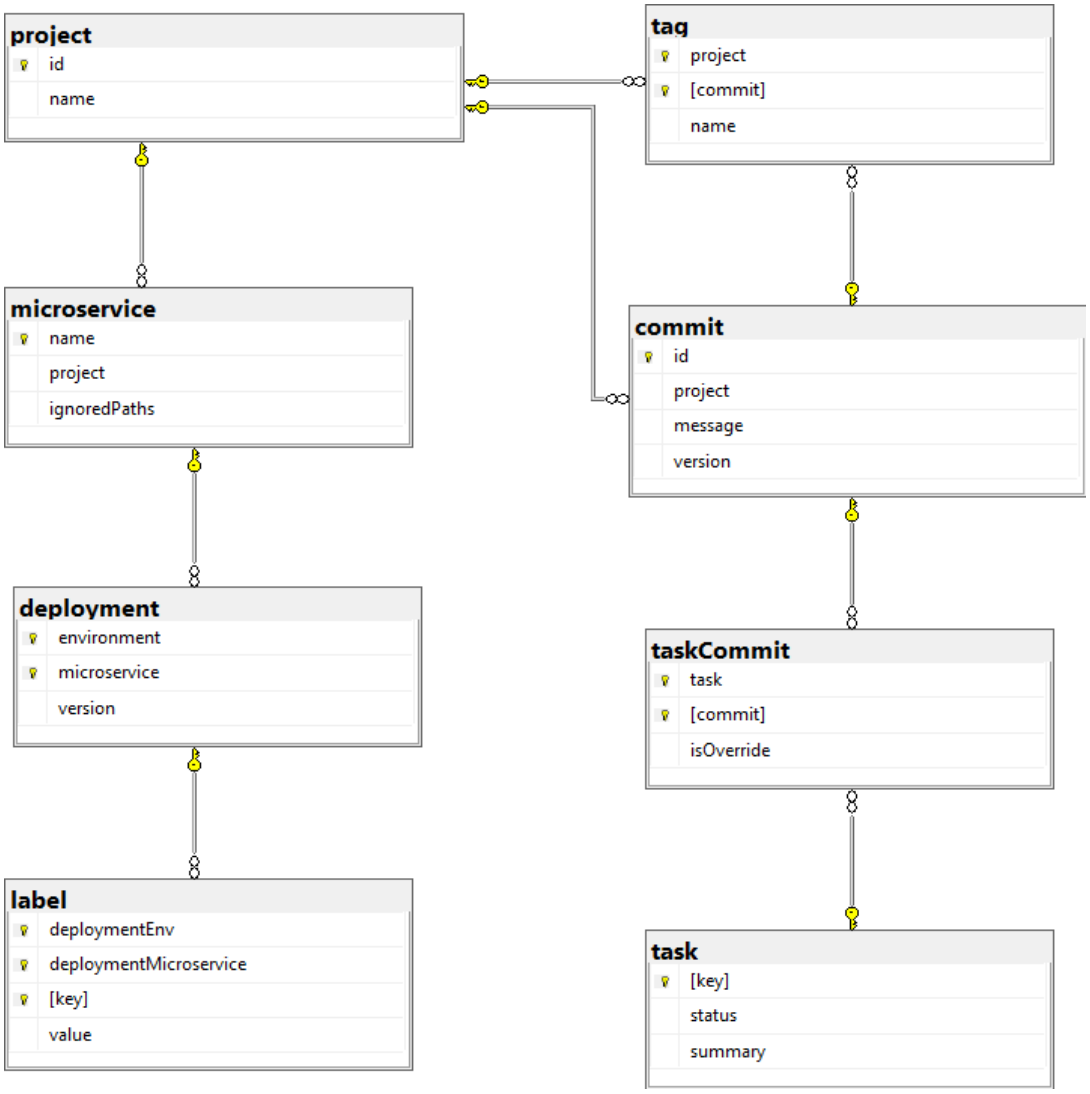


Figure 4.4: Data Model ER Diagram

Attributes

The project entity has two attributes :

- id : Id of project. The Id of Gitlab-Project are unique, so we use it to identify the projects.
- name : Name of project. The name of Gitlab-Project is essential to use to filter by it.

Example

The following table represents an example of projects we showed in the Section 3.3.

Table 4.3: Example of Projects

Id	Name
100	MDR
101	RT

4.3.2 Commit

$$\mathcal{R}(\text{ Commit }) = \{\underline{\text{ id }},\ \text{project } \rightarrow \text{ Project.id },\ \text{createdAt },\ \text{message },\ \text{version }\}thecommitec1asdhasbeentagg \tag{4.2}$$

This entity represent a commit of a *project* inside Gitlab. Each commit is related to one *Project*. A commit can be marked by a *Tag*.

The commits include information about the changes in the project. We can determine what and when a developer changed code into a project using the commits. Based on this information, we can find out what new changes to release and what commits are included in which release.

Attributes

This entity includes following attributes:

- id: Id of the commit.

  An Id of a commit is unique, and we use it to identify the commits.

- project: Project-Entity related to the commit. Each commit is related to one project.

- createdAt: Creation Date of the commit.

  The creation date is essential to discover what commits are created after the last release. That is important to sort the commits and release just the new changes.

- message: Massage of the commit.

  The message of a commit is a description of what changes this commit contains. We use the commit message to bind commits to their related Jira tickets by putting the ticket key as a prefix in the message. Using the mapping between the Jira-Tickets and the commits we can find out if the commit is ready for deployment or not, based on the ticket status.

  However, because the writing of the commit message is a manual job, it can cause problems when the user writes a wrong key or no key in the commit message. That will cause a wrong mapping between our two entities *Tasks* **??** and Commits. We solve this problem by allowing the user to override the map between these two entities. (cf. *Task-Commit Map* **??**).

- version: The version number of the commit. Each commit is related to a release. While each release has a version number, we bind the commit to their versions based on the tags.

  The commits between two tags are related to a specific release with the version number archived in the latest tag. That means that a commit gets its version number based on the incoming tag. (cf. Section 4.3.3)

Example

In this example, there are data about commits of the two projects in the previous example (cf. Table 4.3). The table shows multiple cases we mentioned before. Commit with id tc3afa is in the project with id 100 and has no version yet. However, the commit with id sa2uef belongs to project 101 and has version 44.2.0.

Table 4.4: Example of Commits

id	projectId	createdAt	message	version
au6asf	100	16.10.2021	PLC-458: foo	NULL
tc3afa	100	12.10.2021	PLC-416: foo	32.0.0
bd6ydh	100	11.10.2021	PLC-379: foo	31.0.0
ec1asd	100	10.10.2021	PLC-333: foo	30.0.0
ac1raf	100	09.10.2021	PLC-331: foo	30.0.0
fs3zgl	100	30.09.2021	PLC-164: foo	29.0.0
zr5wsc	100	28.09.2021	PLC-124: foo	29.0.0
gt1ras	101	06.09.2021	PLC-213: foo	44.2.0
sa2uef	101	05.09.2021	PLC-219: foo	44.2.0
xc7bok	101	05.09.2021	PLC-281: foo	44.2.0

4.3.3 Tag

$$\mathcal{R}(\text{ Tag }) = \{ \underline{\text{ project } \rightarrow \text{ Project.id , commit } \rightarrow \text{ Commit.id }}, \text{ name } \} \tag{4.3}$$

This entity is a tag in *project* inside GitLab. A commit can be marked using a tag in the repository. We use the tags to archive release versions in the repository by naming the tags with the version number. Based on the tags, we can find out which commits under what version are released.

Attributes

A Tag has following attributes:

- project: Project of the tag.

  Relationship to the tag's project.

- commit: Commit related to the tag.

  Relationship to the tag's commit.

- name: Name of tag.

  The name of the tag is used to define the version number.

Example

In this example, there are row data for tags of the two projects in the example (cf. Table 4.3).

The table shows that in project 100, the commit with ID (ec1asd) was tagged with version 30.0.0 and the commit with ID (fs3zgl) with version 29.0.0. That means all commits between fs3zgl and ec1asd are in version 30.0.0. Besides, in project 101, the commit gt1ras has been tagged with version 44.2.0, which means all commits between this and the previous tag belongs to version 44.2.0 (cf. Table 4.4).

Table 4.5: Example of Tags

projectId	commitId	name
100	ec1asd	30.0.0
100	fs3zgl	29.0.0
101	gt1ras	44.2.0

4.3.4 Microservice

$$\mathcal{R}(\text{ Microservice }) = \{\underline{\text{name}},\ \text{project} \rightarrow \text{Project.id},\ \text{ignoredPaths}\} \tag{4.4}$$

A *Microservice* is an entity that represents a microservice of a project. Each *project Section 4.3.1* has at least one microservice.

As mentioned before, some projects can be multi-module; in other words, they can have multi microservices, so projects have a (one to many) relationship between projects and microservices. In that case, the project's folders and files are divided between general components relevant to each microservices and those related to a specific microservice. However, even for the project that is a single module, the

related microservice should not necessarily have the same name as the project (cf. Example).

Because there is no constant way to find out the microservices names of a project or if the project is multi-module, the mapping between the projects and the microservices should be possible to override manually when a new microservice gets added to a project, or microservice gets removed or edited.

Furthermore, in the case of the multi-module project, there is no specific way to find out what paths are relevant to what microservice; that is why we have to make it possible for the users to map the microservices manually to the project's paths. In our case, we choose to make all paths relevant to all microservice by default, and the user can ignore the irrelevant paths for each microservices. This process should be done once as long as the project tree has no changes. It is essential to map the microservices to the paths because we need to define what changes (commits) are relevant to what microservice. Based on this information, we can know if there is something new to release in a microservice, and it prevents the creation of unnecessary releases that include no new changes.

Attributes

A microservice has following attributes:

- project: Project of the microservice.
- name: Name of microservice. The name of a microservice is uniqe.
- ignoredPaths: This attribute represents the paths of the project tree that are not relevant for the microservice in the case of multimodule projects.

Example

The following Table 4.6 represents an example of microservices of the project from the previous example (cf. Table 4.3). The example shows that the project MRD with the id 100 is a multimodule project with two microservices: MDR-Import and MDR-Export. For each of these microservices, there are paths defined that we should ignore when we bind the commits in the project to those microservices (cf. CommitsMicroservices).

However, the microservice RT is the only one for project 101, which means that this project is a single module. That leads to having no ignored paths because every change (commit) in the project is relevant for the microservice.

4.3.5 Deployment

$$\mathcal{R}(\text{ Deployment }) = \{\underline{\text{ environment }}, \underline{\text{ microservice } \rightarrow \text{ Microservice.name }}, \text{ version }\} \tag{4.5}$$

A *Deployment* is an entity that describes the current deployment of a microservice on a specific environment. Each deployment has a version number, and each deployment

Table 4.6: Example of microservices

name	projectId	ignoredPaths
MDR-Import	100	/MRD-Export/ /MDR-Export-common/
MDR-Export	100	/MRD-Import/
RT	101	NULL

has multiple labels that include user-customized information about the deployment (cf. Entity: Label).

However, we use the deployments of a microservice to find out what version is currently running on which environment. Based on that information, we can compare the versions on the several environments with each other or with the not-released versions to create a new release on a specific environment.

Attributes

A deployment has following attributes:

- environment: Name of the environment, where the microservice is deployed. There are usually several environments for different purposes like Testing or Production. In our case, we need STAGE, UAT, and PROD environments. On UAT and PROD environments, we deploy the release to the clients. Stage environment is used for Testing or for local purposes (cf. Section 2.1).
- microservice: Microservice related to the deployment. Each microservice can be deployed once in an environment. That is why we use environment and microservice as a primary key in this Entity.
- version: The version number of the release deployed in the environment

Example

The following table (cf. Table 4.7) represents an example of current deployments of the microservices from the last example (cf. Table 4.6).

Based on the table, we can tell that the microservice RT is deployed only on the environment STAGE. On the other side, the two microservices of project MDR (MDR-Export and MDR-Import) are deployed on all the environments.

However, While the project MDR is a multi-module with two microservices (MDR-Export and MDR-Import), we can see that those microservices can have different versions. For example, MDR-Export has a newer release version on STAGE than on PROD and UAT, which means that there are new changes in testing, that should be released later on the other environments UAT and PROD.

On the other side, the microservice MDR-Import has in UAT and STAGE identical versions that are lower than in PROD, which means that the client has access to this release but not the end-user. In this case, the end-user has access to release 30.0.0, and the release 31.0.0 should be later deployed on PROD.

Table 4.7: Example of Deployments

environment	microservice	version
STAGE	RT	44.2.0
PROD	MDR-Export	30.0.0
UAT	MDR-Export	30.0.0
STAGE	MDR-Export	32.0.0
PROD	MDR-Import	30.0.0
UAT	MDR-Import	31.0.0
STAGE	MDR-Import	31.0.0

4.3.6 Label

$$\mathcal{R}(\text{ Label }) = \{\underline{\text{ key }},\ \text{value},\ \underline{\text{deployment} \rightarrow \text{Deployment.environment}, \text{Deployment.microservice}}\} \tag{4.6}$$

A *Label* is an entity that represents a label of deployment of microservice in Kubernetes. Each deployment contains different pairs of keys and values known as labels. Those labels allow us to create a custom map of pairs that add organizational structures in the deployment. We use labels to filter the microservices and the deployments to make the searching easier. For example, we add squad names and microservice names for each deployment.

Attributes

- key: Key of the label. The key is a text that represents the first part of the pair, for example, squad or name.
- value: Value of the label. The key is a text that represents the second part of the pair.
- deployment: Deployment of the label. The deployment where the label is located. As mentioned before, each Deployment can have multiple labels.

Example

The following Table 4.8 is an example of abstract collected data from labels of several deployments in several environments. The table shows different keys and values for the microservices in different environments. For example the microservice (RT) is only a deployment on STAGE and has one label with the key (Squad) and the value (Green).

Table 4.8: Example of Lables

key	value	deployment.env	depoyment.microservice
Squad	Red	STAGE	MDR-Import
namespace	default	STAGE	MDR-Import
Squad	Red	UAT	MDR-Import
namespace	default	UAT	MDR-Import
Squad	Red	PROD	MDR-Import
namespace	default	PROD	MDR-Import
Squad	Blue	STAGE	MDR-Export
namespace	default	STAGE	MDR-Export
Squad	Blue	UAT	MDR-Export
Squad	Blue	PROD	MDR-Export
Squad	Green	STAGE	RT

4.3.7 Task

$$\mathcal{R}(\text{ Task }) = \{ \underline{\text{ key }}, \text{ status }, \text{ summary } \} \tag{4.7}$$

A *Task* is an entity that represents a jira ticket.This entity includes required information about the Jira tickets, like their status or descriptions. Each task is related at least to one *Commit* (cf. Entity: TaskCommit).

Attributes

- key: Key of the task (Jira ticket).

  Every project has a prefix of letters and suffix of numbers, making it unique to use as a primary key.

- status: Status of the task.

  The status represents the lifecycle stage of the ticket. In our concept, we need to define if the ticket is ready for deployment or not. If the ticket is not a sub-task, then it is ready for deployment when the status *Done* or *Ready for deployment*. However, if the task is a sub-task, then it is ready for deployment when its parent (Story) is ready for deployment independently of the task itself has the status Done (cf. Section 3.3.2).

- summary: Title or description of the task.

  This information gives the release manager an overview of what the Developer has done, which makes the discussion about any task easier without opening Jira every time if there is a need to ask about a task.

Example

In the following table (cf. Table 4.9), there is an example of several tasks in different statuses and scenarios.

For example, the task PLC-458 is a sub-task in progress related to a Story PLC-90, which is in progress too. That means that the related commit with id au6asf is not safe to be released (cf. Table 4.10).

Tasks PLC-379 and PLC-333 are both sub-tasks of the story PLC-80; one is in progress, and the other is Done, which makes the story (PLC-81) not ready to be released because of its status.

Table 4.9: Example of Tasks

key	status	summary
PLC-458	in progress	foo
PLC-416	in progress	foo
PLC-379	Done	foo
PLC-333	Done	foo
PLC-331	Done	foo
PLC-164	Done	foo
PLC-124	Done	foo
PLC-213	Done	foo
PLC-219	Done	foo
PLC-218	Done	foo

4.3.8 TaskCommit

$$\mathcal{R}(\text{ TaskCommit }) = \{ \underline{\text{ task } \rightarrow \text{ Task.key }}, \underline{\text{ commit } \rightarrow \text{ Commit.id }}, \text{isOverride } \} \tag{4.8}$$

The *TaskCommit* is an entity that represents the many to many relationship between *Commit* and *Task*. Each *Commit* is related at least to one *Task* and each *Task* is related at least to one *Commit*.

However, the binding between tasks and commits is our method to find out the status of changes in microservices. Based on the status, we can know what changes (commits) are ready for the next release or discover the problems like commits without Jira-Tickets or commits with wrong tickets. In this case, we can override this mapping between the commits and the tasks.

As mentioned before, because the commit's message is manually written, the developer can make mistakes while writing the task's key in the message or forget to write

it. That causes problems in the mapping like commits without tasks or commits mapped to wrong tasks, which leads to bugs in the releases when wrong or unready commits is released. To understand this problem with more details, we give an example in the following example table (cf. Table 4.10). To solve those problems, we make the mapping editable, and we flag the manually edited or added mapping to distinguish between the automatic and manual mapping.

Attribues

This entity include the following attributes:

- task:

  Task of the commit (cf. Section 4.3.7).

- commit:

  Commit related to the task (cf. Section 4.3.2).

- isOverride:

  A Flag shows if the task key is a default one parsed from the commit message or if the task is manually mapped to the commit. The flag avoids confusion when the message includes a different task key than the mapped task because of the manual edit.

Example

In the following Table 4.10 we provide the mapping between commits and tasks we showed in previous examples (cf. Section 4.3.7 Section 4.3.2). Comparing this table with the two other tables (cf. Table 4.9 Table 4.4), we can notice what commit is mapped to what task.

A particular case in this example is the commit (xc7bok), whose message is "PLC-281: foo" but it is mapped to the task "PLC-218" and flagged as overridden. That means that a user has manually overridden this particular commit because the commit message includes a wrong Jira ticket key.

Table 4.10: Example of TaskCommit

commit	task	isOverride
au6asf	PLC-458	false
tc3afa	PLC-416	false
bd6ydh	PLC-379	false
ec1asd	PLC-333	false
ac1raf	PLC-331	false
fs3zgl	PLC-164	false
zr5wsc	PLC-124	false
gt1ras	PLC-213	false
sa2uef	PLC-219	false
xc7bok	PLC-218	true

5. Implementation

The previous chapter represents the concept of the semi-automated release process. We discuss in this chapter the prototypical implementation of that concept (cf. Chapter 4), which is a proof of concept. We show our technologies choices of programming languages, databases, and frameworks. Moreover, we describe all the functional requirements and the problems that challenged us and give the other developers some directions for future works.

5.1 Architecture and Technologies

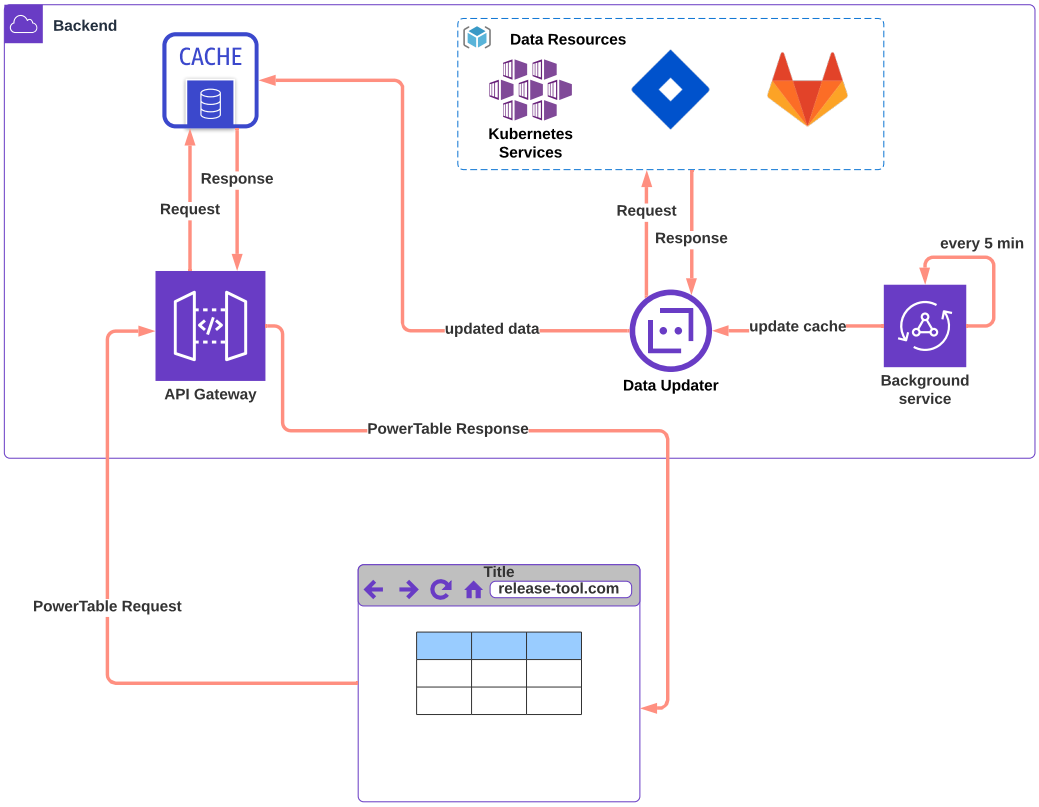
In this section, we discuss what technologies we use to implement the tool and its architecture.

We implement a web application and name it release tool. The tool includes frontend and backend sides. The frontend developers in the company assisted us in developing the frontend part because we have to use self-made frameworks by them. We have to use those frameworks because of company regulations to use the company's technologies, which makes understanding our tool easy for other developers that may continue our work in the future. That is also the main reason for using the technologies we use to implement the tool.

However, the Figure 5.1 represents the how the different parts of the release tool communicate.

First, the frontend side is responsible for showing, filtering, and grouping the tables. We use the framework PowerTable, which was built in-house at Fintech using Javascript and Java. PowerTable binds the frontend with the backend using REST-API. It creates tables with dynamic columns based on the API-Response from the backend. That means we can control the tables that we show in the frontend from the backend, with no need to adjust the frontend every time we have something new. Moreover, PowerTable has filtering and grouping features that we can also control from the backend. We describe the PowerTable with more details in the Section 5.2. Also, in this thesis, we should notice that we are focusing on process automation

Figure 5.1: Communication of Release Tool Parts



and not on the user interface because our application is a prototype that examines the possibility of reducing the time costs of the release process through automation regardless of what UI we have or what technologies we use to build our prototype.

On the other side, the backend includes the REST-API with a database as a cache, and several backend services, that get needed data from different systems. We use Postgresql as an object-relational database management system and Java with Spring-Boot framework to build the REST-API. Spring Boot is an open source framework for creating a web application that runs on Java [WSL+20].

We use the PowerTable framework to send and receive data between the tables and the database. As mentioned above, our selection is related to the technologies that are used in the company.

However, we start with an API gateway between the frontend and other backend services. The API gateway receives requests using PowerTable from the front end and responds with data from a database, which we use as a cache. Using cache in the tool aims to reduce the count of sent requests to the data resources services, reduce the costs and pressure on these services, and make the tool more efficient. That makes the tool more performant because we do not collect data every time we receive a request. We also do not cache in the memory because of the big data size. Using a database as a cache, we do not need to cache the data every time we start the tool. Moreover, using of PowerTable requires database tables. That makes the database a better solution in our case. Moreover, we create for

each table we mentioned in the Section 4.2 a SQL view. We bind those views with PowerTable to show them in the frontend. We discuss those views in the next section (cf. Section 5.2). Also, the cache gets updated using a data updater automatically performed in the background. Besides, the data updater receives data from several services (Kubernetes, Jira, and GitLab) then writes these data into the database.

So, in summary, we build a web application using Java as a programming language and Postgresql as a database in the backend that builds a REST-API and Javascript with PowerTable in the frontend. PowerTable is used in the backend as well. It works as a communication channel between the frontend and backend.

5.2 SQL Views and PowerTable

This section discusses the SQL views and the binding with PowerTable. Besides, we describe how PowerTable works.

Based on the data model in Section 4.3 and the tables we discuss in Section 4.2, we create a SQL view for each table. The Listing 5.1 represents the SQL view for the released microservices table (cf. Section 4.2.1), while the Listing 5.2 is the SQL view for the deployment table (cf. Section 4.2.2).

```
1  CREATE VIEW microservices AS
2  SELECT squad.value squad,
3         m."name" microservice,
4         dProd."version" prod,
5         dUat."version" uat,
6         dStage."version" stage
7  FROM microservice m
8
9  LEFT JOIN project p ON p.id = m.project_id
10
11 LEFT JOIN deployment dProd
12 ON dProd.microservice_name = m."name"
13 AND dProd.environment = 'PROD'
14
15 LEFT JOIN deployment dUat
16 ON dUat.microservice_name = m."name"
17 AND dUat.environment = 'UAT'
18
19 LEFT JOIN deployment dStage
20 ON dStage.microservice_name = m."name"
21 AND dStage.environment = 'STAGE'
22
23 LEFT JOIN "label" squad
24 ON squad.deployment_microservice_name = m."name"
25 AND squad.deployment_environment = dStage.environment
26 AND squad."key" = 'squad'
```

Listing 5.1: Microservices View

However, we use PowerTable with REST-API to send and receive data between the frontend and backend. Each table requires an endpoint to send the PowerTable information with the SQL view data to the frontend and a PowerTable configuration.

We discuss only the endpoint and the PowerTable configuration for the microservice view because the process is analogous to the other table.

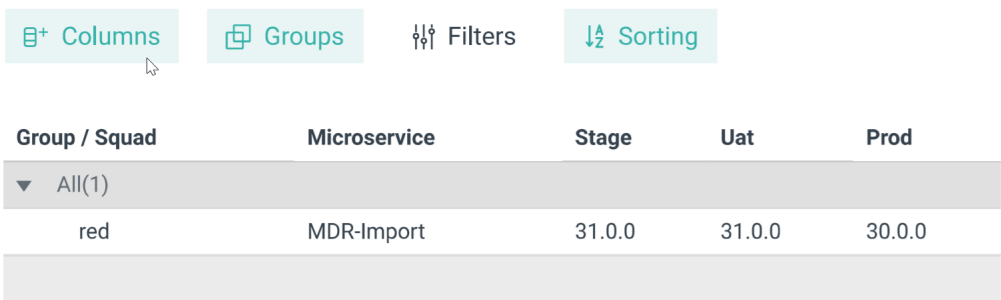
The following Listing 5.3 shows the configuration of the PowerTable for the microservice view. We define a PowerTable dependency using Bean, what Spring framework offers, to inject it then through the constructor in the microservice controller (cf. Listing 5.4). The PowerTable requires an entity class, an EntityManager, and a FieldsSchema to read the data from the database, map it to the entity class, and then send that data to the frontend through an endpoint. The class MicroservicesViewEntity represents entity class for the microservice view with identical columns.

The Listing 5.4 shows the microservice RestController, which includes an endpoint method for reading data using PowerTable and sending it to the frontend. We inject the PowerTable in the constructor of the RestController and then in the endpoint method (getData) we call the method (groupData), which pulls and maps the data from the database and returns a JSON response. At this point, the REST-API process ends.

However, in the Listing 5.5 and Listing 5.6 we give an example of the REST-API request and its response that the PowerTable use to send and receive the data of microservices view. The frontend side of PowerTable sends a request to an endpoint, then using the receives a response, it builds the PowerTable UI. The following Figure 5.2 shows how the PowerTable UI looks like with the received data we mentioned above (cf. Listing 5.6). The PowerTable UI includes the table and the filtering, sorting, and grouping features.

In summary, PowerTable requires database tables to work. That is why we create SQL views. PowerTables read the data from those views through an endpoint and show them the frontend using PowerTable UI.

Figure 5.2: Example of a PowerTable UI



5.3 Data Updater

The previous section discusses how PowerTable, REST-API, and database work. This section shows how we collect the data from several infrastructures and systems and write them in the database. As we mentioned in Section 5.1, we collect data

from Jira, Gitlab, and Kubernetes using their REST-API and save cache them in our database. Besides, we update the cache regularly using a background service that reruns the data updater every specific time.

However, to avoid redundant data processing, e.g., receiving and storing the same or old data when we perform an update, we try to compare the data we cache with the new data and cache only the new ones. The shows how we update just the new commits of a project by noticing the creating time of the commits we cache and load only the commits from that time. We use similar ways to update the data for other entities. Also, we run the updating methods for several entities parallelly to speed up the updating process.

5.4 Endpoints

This section discusses the endpoints the API gateway includes and explains why we need them and how they work.

As mentioned before, PowerTable uses endpoints to send and receive data between backend and frontend, which mean we need for each table an endpoint. In our case we need two endpoints for microservices table and deployment table. Moreover, other features like overriding the mapping between tasks and commits and adding the ignored paths of a project in multimodule projects need endpoints to work. We create an endpoint for each of those features, which can be used through the frontend.

5.5 Summary

This chapter outlines the methods and technologies we use to implement the concept we discuss in Chapter 4. We also discuss the tool's architecture and give code examples of it. We use Java, Postgresql, and a self-build framework called PowerTable, which bind the backend and frontend using REST-API.

However, the tool has two sides: the front and backend. They communicate using a REST-API. The company's frontend team assisted us in building the front end while we built the back end. The implementation of the web application, what we call Release-Tool, is the method we use to compare the current release process with our semi-automated one. Using the tool, we can evaluate our concept to determine if and what time costs we save by this concept.

```sql
CREATE VIEW deployment_view AS
 SELECT squad.value squad,
        cm.microservice_name microservice,
        c.version,
        t."key" task_key,
        t.status task_status,
        t.summary task_summary,
        commit_task.override task_overridden,
        Concat(sprod.environment, '␣', dprod.environment, '␣', uprod.environment) AS released_on
 FROM task t

 /*Check if the mapping between tasks and commits has been overridden.*/
 LEFT JOIN
   (SELECT ct.commit_id,
           ct.task_key,
           ct.override,
           co.override_count
    FROM commit_task ct
    JOIN
      (SELECT ct.commit_id,
              Count(CASE
                        WHEN override THEN 1
                    END) override_count
       FROM commit_task ct
       GROUP BY ct.commit_id) co ON ct.commit_id = co.commit_id
    WHERE ((co.override_count > 0
            AND ct.override)
           OR (co.override_count = 0
               AND NOT ct.override)) ) commit_task ON t."key" = commit_task.task_key

 LEFT JOIN commit_microservice cm
 ON cm.commit_id = commit_task.commit_id

 LEFT JOIN "commit" c ON c.id = cm.commit_id

 LEFT JOIN project p ON p.id = c.project_id

 LEFT JOIN deployment dprod ON dprod.microservice_name = cm.microservice_name
 AND dprod."version" = c."version"
 AND dprod.environment = 'PROD'

 LEFT JOIN deployment uprod ON uprod.microservice_name = cm.microservice_name
 AND uprod."version" = c."version"
 AND uprod.environment = 'UAT'

 LEFT JOIN deployment sprod ON sprod.microservice_name = cm.microservice_name
 AND sprod."version" = c."version"
 AND sprod.environment = 'STAGE'

 LEFT JOIN "label" squad ON squad.deployment_microservice_name = cm.microservice_name
 AND squad.deployment_environment = 'STAGE'
 AND squad."key" = 'squad'
```

Listing 5.2: Deployment View

```
/*The EntityManager API is used to create and remove persistent
    entity instances,
to find entities by their primary key, and to query over entities.*/
@Autowired
private EntityManager entityManager;

@Bean
public PowerTable<MicroservicesViewEntity>
    microservicesViewPowerTable() {
    return new PowerTable<>(MicroservicesViewEntity.class,
        entityManager, getMicroservicesSchema());
}

private FieldsSchema getMicroservicesSchema() {
    return new FieldsSchemaBuilder()
        .field("squad", "Squad", FieldType.TEXT)
        .field("microservice", "Microservice", FieldType.TEXT)
        .field("stage", "Stage", FieldType.TEXT)
        .field("uat", "Uat", FieldType.TEXT)
        .field("prod", "Prod", FieldType.TEXT)
        .build();
}

@Entity
@Table(name = "microservices_view")
public class MicroservicesViewEntity {

    private String squad;
    private String microservice;
    private String uat;
    private String prod;
    private String stage;
}
```

Listing 5.3: Microservices View

```java
@RestController
@RequestMapping("/microservices")
public class MicroservicesController {

    private final PowerTableRepository <MicroservicesViewEntity>
        powerTableRepository;

    public MicroservicesController(PowerTableRepository <
        MicroservicesViewEntity> powerTableRepository) {
        this.powerTableRepository = powerTableRepository;
    }

    @PostMapping(path = "getData")
    public HttpResponse  getData(@RequestBody final GroupRequest
        request) {
        var body = powerTableRepository.groupData(request);
        return body;
    }

}
```

Listing 5.4: Microservices View

```json
{
    "columns": [
        {
            "alias": "squad"
        },
        {
            "alias": "microservice"
        },
        {
            "alias": "STAGE"
        }
        ,{
            "alias": "UAT"
        }
        ,{
            "alias": "PROD"
        }
    ],
    "where": "squad=='RED'"
}
```

Listing 5.5: Example of a PowerTable request to get data from microservice view

```
{
    "data": [
        {
            "squad": "red",
            "microservice": "MDR-Import",
            "stage": "31.0.0",
            "uat": "31.0.0",
            "prod": "30.0.0"
        }
    ],
    "page": {
        "size": 1,
        "number": 0,
        "hasNextPage": false
    }
}
```

Listing 5.6: Example of a PowerTable response of data from microservice view

```
/*Save commits of a project */
private final CommitsSaver commitsSaver;
/* Read and write commits entities in the database */
private final CommitRepository commitRepository;
/* Read data from Gitlab using its REST-API*/
private final GitInteraction gitInteraction;

private void updateCommitsForProject(ProjectEntity projectEntity) {
        List<GitCommitDto> gitCommits;
        Instant dateToLoadFrom = getDateToLoadFrom(projectEntity,
            commitRepository);
        if (dateToLoadFrom == Instant.MIN) {
            gitCommits = gitInteraction.loadCommits(projectEntity.
                getId());
        } else {
            gitCommits = gitInteraction.loadCommits(projectEntity.
                getId(), dateToLoadFrom);
        }
        commitsSaver.save(projectEntity, gitCommits);
 }

 private static Instant getDateToLoadFrom(@NonNull ProjectEntity
     projectEntity,
                                          @NonNull CommitRepository
                                              commitRepository) {
    Optional<CommitEntity> lastCommitEntity =
            commitRepository.findByProjectOrderByCreatedAtDesc(
                projectEntity.getId()).stream().findFirst();

    if (lastCommitEntity.isPresent()) {
        return lastCommitEntity.get().getCreatedAt();
    }

    return Instant.MIN;
 }
```

Listing 5.7: Code of Updating the commits for a project

6. Evaluation

In Chapter 4, we proposed a concept for a semi-automated release management process that can exchange the current manual process, which we discussed in Chapter 3, to reduce the time costs of conducting a release process. In this chapter, we examine whether our concept reduces the time we spend on conducting a release process compared to the current manual one. We describe the experimental setup for this evaluation. After that, we empirically evaluate a small case study, which we operate interactively with our release tool prototype.

6.1 Study Design

In this section, we define concrete research questions to evaluate our process. Then we develop a case study to answer those questions and provide reasons for choosing that specific case study.

6.1.1 Research Question

We want to address the following research question with our evaluation:

Does automating parts of the manual release management process reduce the time cost of conducting the release process of microservices, and if so, by how much?

By answering this question, we evaluate the efficiency of our concept compared to the current manual process. We answer this question by measuring the time a user needs to finish a release process using our release tool prototype compared to the current manual process (cf. Section 6.1.2).

6.1.2 Experimental Setup

For our evaluation, we are challenged by a small number of participants because, at FinTech, there is a small specific group of employees who can progress the release process. That can have effects on the reliability of our evaluation because it increases the margin of error and reduces the power of the study, especially in small true effect

size cases [Hac08]. In our case, we do not know the actual effect size of our concept on the process; that is why we design a study that reduces the effect of the small sample size.

We develop an extended version of A/B-Test, which is a test method to evaluate two variants of a system or process [KL17]. The A/B-Test and its developed extensions A/B/n-Test are simple controlled experiments with multiple phases in which the participant offered one of two or more variants of a system: Baseline (A), intervention (B) and any number of additional variations (n) [You14] [KHS07]. We design an A/B/A study. The A/B/A-Test has one phase more than an original A/B-Test, which is the withdrawal phase. The three phases of our study are designed as follows:

A is the baseline phase. The baseline is the current unmodified situation, which is, in our case, the current manual release process.

B is the intervention phase. At this point, the intervention occurs in the current situation to examine this intervention's changes. In our case, the participants use the semi-automated release process.

A is the withdrawal phase. We withdraw the intervention and move back to the first situation in this phase. This phase is essential in our case because phase designs could have particular problems with internal validity. After all, time inevitably passes during the study, and many things change over time, e.g., the manual release process can have some changes over time, like more or fewer steps. We avoid that issue using withdrawal the phase because it is supporting evidence that an observed effect is due to the intervention and not the changes mentioned above [DFT12].

However, it is hard to compare two similar things by working with human participants because there are differences among individuals and among observation occasions that affect the measurements [HdPB09]. Suppose one of the participants has a migraine that gets worse over time when she/he starts working at one of the release processes. If this person has to start with the same process every time, this will affect their speed and concentration. On the other hand, if a healthy participant does the same release process every time, the same successive jobs are likely to be more accessible and faster over time, which in both cases will bring undesired effects on the measurements of a phase. These problems might be avoided by choosing the intervention and withdrawal points at random for each participant. Using randomization of the intervention points allows avoiding bias and demonstrating effectiveness and removes a threat to internal validity [SSQM08].

Hence, we have four participants; each one releases the same microservices twice a week based on their squad. They keep their usual work of releasing microservices into several environments but they alternate between the release tool and the current manual release process. We use a stopwatch to measure the spent time to finish a release process. We start collecting the time that each participant spent to finish a release, we call that (the *time cost*), and the processed number of microservices while perforing the release processes for each participant. It is crucial to collect the

number of the processed microservices each time because each squad has a different number of microservices, that they maintain. The number of processed microservices could change. That could produce invalid results, e.g., comparing the time costs of releasing 20 microservices with ten microservices. To avoid that, we calculate the average time costs of one microservice each time. However, we will randomly transition to using our prototype release tool after that (intervention phase). Also determined at random, as mentioned above, they go back to the manual release process until the end of the study (withdrawal phase).

Our hypotheses are

H0 The use of the semi-automated process will not reduce the time cost of the process.

H1 The use of the semi-automated process will reduce the time costs of the process.

We collect 14 observations for each participant over seven weeks, based on two releases weekly. However, we believe it is essential to have the same minimum number of conducting each phase to avoid domination because we move randomly between the phases, which means that one of the phases can be conducted only once, leading to an invalid experiment. Hence, we decided to observe each phase at least three times because we believe that 3 is a fair number to have enough space for the randomization, based on the 14 observations total. Based on that, we decide that at least three observations are needed in each baseline and withdrawal phase where the participant uses the manual release process, and at least three are needed for the release tool. That means we can only move to intervention phase when we measured the baseline phase at least three times; besides, we move to the withdrawal phase only after we measure phase intervention three times, and we must also measure the withdrawal phase at least 3 times.

Hence, we have 21 possible pairs of transition points to move to the intervention phase and then to the withdrawal phase for each participant. The possible transition points (TB, TA), where TB is the transition point from phase A to B and TA from B to A, are as follows:

(4,7) (4,8) (4,9) (4,10) (4,11) (4,12) (5,8) (5,9) (5,10) (5,11) (5,12) (6,9) (6,10) (6,11) (6,12) (7,10) (7,11) (7,12) (8,11) (8,12) (9,12)

We select randomly from those transition points for each participant. The following Figure 6.1 represents the result of our randomized selection of the transition points for the four participants.

The result of random selection is as follows:

The intervention phase starts for participant 1 after collecting four observations for the baseline phase, then the withdrawal phase starts at transition point 9, which gives us six observations for this phase and four for intervention. For participants 2 and 3, the intervention phase starts on point 4 after performing phase A three times but ends on different points; participant 2 starts the withdrawal phase at point 10, while participant 3 at point 12, as participant 4, who starts intervention phases at point 7.

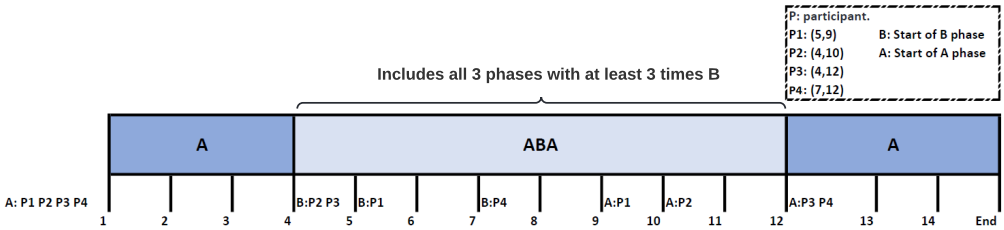


Figure 6.1: The Transition Points between of Phases B and A for each participant

However, we collect our observations using Excel datasheets after the transition points selection. Our datasheets include four columns: phase, time cost, processed microservices, and time cost per microservice. We collect this data based on the needs discussed in this section above.

Then, when we finish the collecting, we calculate the time cost pro microservice mean of the B phase ($\bar{x}_B$) and the same mean of both A phases ($\bar{x}_A$) and then calculate the difference between those two results ($\bar{x}_A - \bar{x}_B$). We use the mean score and not the median because we have a low number of outliers values in our data: Outliers are individual values deviating from the significant distribution of the data set [BKNS99]. We use the interquartile range rule (IQR) to determine any outliers in the data sets of each participant. The interquartile range is the range within which the middle half of the data points lie. That means, it is the distance between the two quartiles Q3 AND Q1; IQR= Q3-Q1 [WI05].

We perform that calculation for each participant separately and then for all participants together. Because we have human participants, that leads to potential variation between the times each participant spends to finish a release process because of the physical or psychological trait differences in humans [Fis19].

However, we use that calculation ($\bar{x}_A - \bar{x}_B$) as the test metric for our study. Our alternative hypothesis is that the intervention using the release tool will reduce the time spent to finish a release process, and the value of that calculation result will decrease if our alternative hypothesis is correct; otherwise, the null hypothesis will be valid.

6.2 Results

In this section, we present and discuss the results of our experiment and answer our research question (cf. Section 6.1.1).

Hence, we investigate whether our concept saves the time costs of performing a release process. To this end, we analyze the collected data for each participant as follows:

First, we use the Excel datasheets of each participant to design an diagram that overviews the data sets of the participants. The following Section 6.2 represents the datasheet of participant 1. Analog to this datasheet we provide in Section A.1 the datasheets with row data of all four participants. This data sheet divided 14 observations into four baseline observations, four intervention observations, and six

	Phase	time (min)	Processed Microservices	Time per Microservice (min)
1	A	40:00	28	01:26
2	A	12:02	7	01:43
3	A	45:00	31	01:27
4	A	09:30	6	01:35
5	B	34:10	28	01:13
6	B	32:22	29	01:07
7	B	10:44	10	01:04
8	B	34:10	31	01:06
9	A	21:22	15	01:25
10	A	14:10	9	01:34
11	A	45:08	30	01:30
12	A	14:10	10	01:25
13	A	36:12	25	01:27
14	A	42:19	28	01:31

Table 6.1: Datasheet of Participant 1

withdrawal observations. For each observation, we collect the spent time to finish the release process with the number of relevant microservices and then we normalize the time by calculating the arithmetic mean of the time costs per microservice according to the following formula:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The arithmetic mean is calculated as the sum of all values in the dataset divided by the number of values [Jac94].

The following Figure 6.2 represents the datasets results of each participant as an XY graph. The X-axis represents the 14 observations points, where each participant performed a release process, while Y-axis represents the spend time in each process. We use the letter P to represent a participant. Each participant has a different color and number to identify them. The points in the graph represent time spent score a participant in a specific observation in a specific phase, while the lines between the points represent the time changing between the observations. Hence, we observe that phase B's time costs per microservices are lower than on both A phases in each participant.

We use the interquartile range rule to identify the outliers of our data for each participant in both the B and A phases. An often used rule is that a value is an outlier if it is more than $Q_3 + 1,5 \cdot \text{IQR}$ and lower than $Q_1 - 1,5 \cdot \text{IQR}$ [HVdV08]. The following Figure 6.3 represents our results of performing the IRQ rule for each participant and all together as one experiment. The results show that we have no outliers in any participant, but because participant 2 spend more time on phase A, that causes few outliers when we sum the data from all participants to use them as one experiment. As result for that, we can use the mean score as a test metric to determine if using the release tool reduces the time costs of the release process.

We calculate the difference between the means $\bar{x}_A - \bar{x}_B$ and its percentage for each participant. The following Figure 6.4 represents the results our calculation. Based

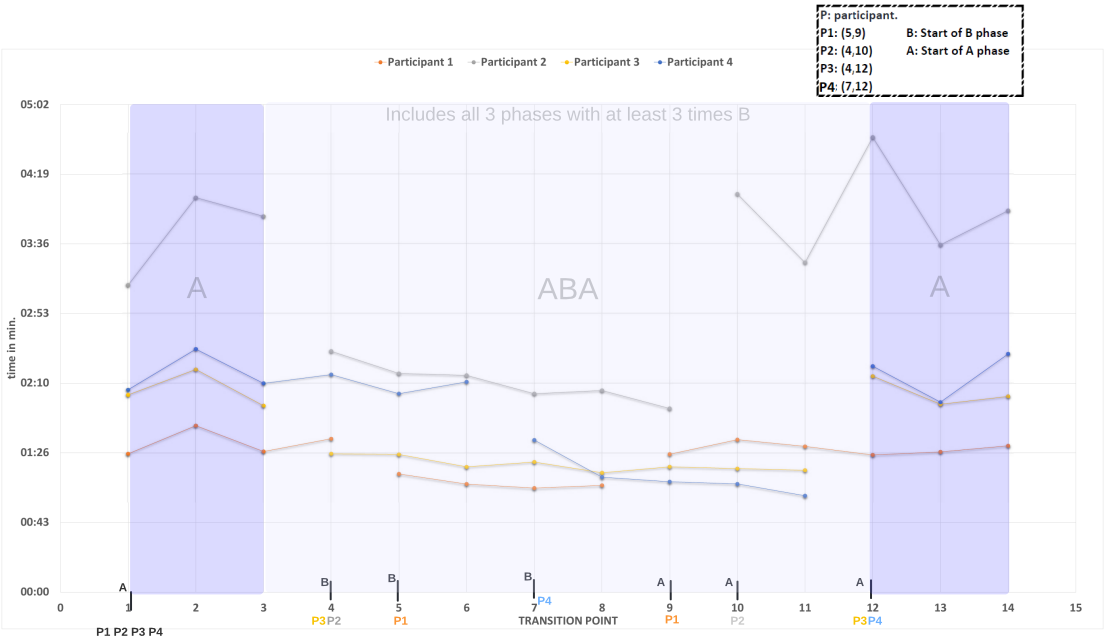


Figure 6.2: Diagram Representation of the Datasets of all Participants

Participants	A	B	Means	IQR A	IQR B	1,5·IQR (A)	1,5·IQR (B)	Outliners A	Outliners B
P1	01:25, 01:34, 01:30, 01:25, 01:34, 01:30, 01:25, 01:27, 01:31 01:25, 01:27, 01:31	01:13, 01:07, 01:04 01:06	Xa=01:33 , Xb=01:08 Xa-Xb= 00:25	Q1=01:26 Q3=01:34 IQR=00:07	Q1=01:06 Q3=01:09 IQR=00:03	Q1−1,5·IQR= 01:15 Q3+1,5·IQR = 01:45	Q1−1,5·IQR = 01:01 Q3+1,5·IQR= 01:13	-	-
P2	03:10, 04:05, 03:53, 04:07, 03:24, 04:42 03:35, 03:57	02:29, 02:15, 02:14, 02:03, 02:05, 01:54	Xa=03:52 , Xb=02:10 Xa-Xb= 01:41	Q1=03:33 Q3=04:05 IQR=00:33	Q1=02:05 Q3=02:15 IQR=00:10	Q1−1,5·IQR= 02:44 Q3+1,5·IQR = 04:54	Q1−1,5·IQR= 01:50 Q3+1,5·IQR= 02:31	-	-
P3	02:02, 02:18, 01:56, 02:14, 01:56, 02:01	01:26, 01:25, 01:18, 01:21, 01:14, 01:18, 01:16, 01:15	Xa=02:05 , Xb=01:19 Xa-Xb= 00:45	Q1=01:26 Q3=01:34 IQR=00:07	Q1=01:26 Q3=01:34 IQR=00:07	Q1−1,5·IQR = 01:38 Q3+1,5·IQR = 02:31	Q1−1,5·IQR = 01:08 Q3+1,5·IQR = 01:30	-	-
P4	02:05, 02:31, 02:09, 02:15, 02:03, 02:10, 02:20, 01:58, 02:28	01:34, 01:11, 01:08, 01:07, 01:00	Xa=02:13, Xb=01:12 Xa-Xb= 01:01	Q1=01:58 Q3=02:11 IQR=00:13	Q1=01:16 Q3=01:22 IQR=00:06	Q1−1,5·IQR= 01:44 Q3+1,5·IQR = 02:42	Q1−1,5·IQR = 01:01 Q3+1,5·IQR= 01:17	-	-
All	01:25, 01:34, 01:30, 01:25, 01:27, 01:31, 03:10, 04:05, 03:53, 02:02, 02:18, 01:56, 02:05, 02:31, 02:09, 02:15, 02:03, 02:10, 01:25, 01:34, 01:30, 01:25, 01:27, 01:31, 01:25, 01:34, 01:30, 01:25, 01:27, 01:31 04:07, 03:24, 04:42 03:35, 03:57, 02:14, 01:56, 02:01, 02:20, 01:58, 02:28	01:13, 01:07, 01:04 01:06, 02:29, 02:15, 02:14,02:03, 02:05, 01:54, 01:26, 01:25, 01:18, 01:21, 01:14, 01:18, 01:16, 01:15 01:34, 01:11, 01:08, 01:07, 01:00	Xa=02:23, Xb=01:29 Xa-Xb=00:54	Q1=01:35 Q3=02:31 IQR=00:56	Q1=01:10 Q3=01:44 IQR=00:34	Q1−1,5·IQR = 00:11 Q3+1,5·IQR = 03:54	Q1−1,5·IQR = 00:18 Q3+1,5·IQR = 02:35	04:05 04:07 04:42 03:57	-

Figure 6.3: Results Table of Performing the IQR Rule

on these results, we observe that all participants spent less time finishing the release process while they used the release tool. Nevertheless, the reduced time is variable, based on the participant, e.g., participant 1 saved over 26% time costs through using the release tool, while participant 2 could save over 43%, despite the higher spent time in each phase.

At the end, the results answer our research question as follows: The semi-automated release process reduces the time cost of conducting the release process of microservices in different ratios, based on the user. In our study, the ratio ranges between 27% and 45%, while on average among all participants, the ratio is over 37%.

Participants	Baseline (A)	Intervention (B)	Withdrawal (A)	Means	Percentage of Saved Time
P1	01:25, 01:34, 01:30 01:25, 01:27, 01:31	01:13, 01:07, 01:04 01:06	01:25, 01:34, 01:30, 01:25, 01:27, 01:31	Xa=01:33 , Xb=01:08 Xa-Xb= 00:25	26,86%
P2	03:10, 04:05, 03:53	02:29, 02:15, 02:14, 02:03, 02:05, 01:54	04:07, 03:24, 04:42 03:35, 03:57	Xa=03:52 , Xb=02:10 Xa-Xb= 01:41	43,80%
P3	02:02, 02:18, 01:56	01:26, 01:25, 01:18, 01:21, 01:14, 01:18, 01:16, 01:15	02:14, 01:56, 02:01	Xa=02:05 , Xb=01:19 Xa-Xb= 00:45	36,50%
P4	02:05, 02:31, 02:09, 02:15, 02:03, 02:10	01:34, 01:11, 01:08, 01:07, 01:00	02:20, 01:58, 02:28	Xa=02:13, Xb=01:12 Xa-Xb= 01:01	45,90%
All	01:25, 01:34, 01:30, 01:25, 01:27, 01:31, 03:10, 04:05, 03:53, 02:02, 02:18, 01:56, 02:05, 02:31, 02:09, 02:15, 02:03, 02:10, 01:25, 01:34, 01:30, 01:25, 01:27, 01:31"	01:13, 01:07, 01:04 01:06, 02:29, 02:15, 02:14,02:03, 02:05, 01:54, 01:26, 01:25, 01:18, 01:21, 01:14, 01:18, 01:16, 01:15 01:34, 01:11, 01:08, 01:07, 01:00	01:25, 01:34, 01:30, 01:25, 01:27, 01:31 04:07, 03:24, 04:42 03:35, 03:57, 02:14, 01:56, 02:01, 02:20, 01:58, 02:28	Xa=02:23, Xb=01:29 Xa-Xb=00:54	37,90%

Figure 6.4: Results Table of Performing the Means Difference Calculating

However, in the future, the results we have in this current study can get better or worse in different timelines and different companies. As we have a small number of participants, who tested a specific case of the release management process, a broader analysis can uncover more study cases.

7. Related Work

In this thesis, we developed a concept of a semi-automated release process for microservices to reduce the time costs of performing the release process. We are not aware of other works that specifically address the same theme in the same conditions. However, previous studies have introduced several research models for the release management process and others for managing multimodule projects and microservices. In this chapter, we provide selected studies and how they relate to our work.

A Previous study worked on analyzing and identifying the challenges and problems of a manual release process and providing a semi-automated process to improve the release management [KRH+20]. This study intersects in several points with our thesis, especially in the idea of automating only parts of the release process. The case study from Antti and Marko [LJ11] contribute to presenting the challenges and problems of the release management process in the IT Infrastructure Library (ITIL) framework. Their study included a high release distribution rate and poor traceability between the incident and release records, which we do not have in our thesis. However, we discuss as in that study the challenges and problems of a release process and provide a solution concept. Old paper from the year 1996 [vdHHHW97] discussed the issues in software release management and provided an initial set of requirements for a software release management tool. Paper about automated deployment [EBF+17] provide automatically executable models for deploying applications in could. This paper discusses the automation of the pipeline, while we discuss the management part of the process before arriving at the pipeline point. Another paper [Che17] discusses the automation for the pipelines, but with the concept of Continuous Delivery (CD). In the domain of microservices, the study [Spe19] presents the Multi-Project issue management like microservices and provides a prototype of a multi-project coding issue management system. However, to the best of our knowledge, there is no scientific paper about the release management process of microservices.

8. Conclusion

The release process is an essential task in IT companies. Many IT companies perform the release process with manual and repetitive tasks, which leads to more error possibilities and time costs [KRH+20]. Automation of the release process can reduce those issues, which increases the productivity and the quality of the software [HF10, p. 29].

At FinTech, the release process needs human efforts, so full automation is impossible. So, semi-automating is a better solution in our case. Hence, we design and implement a semi-automated release management process to reduce the time costs of the release process. We built a web application called Release-Tool as a prototype of our concept to evaluate our concept and compare it with the current manual process.

After implementing the tool, we started to evaluate our concept. We were challenged with a small number of participants because there is in FinTech a small specific group of employees who can progress the release process. We used the ABA test, which is an extended version of AB-test, to compare the current manual release process with our concept to solve that problem.

However, we find out that release time and cost have been reduced while the participants used our release tool, which means the semi-automated release process is more effective and saves personnel time, which usually saves personnel costs.

In the future, we can automate the entire process, which may ensure more efficient releases without human involvement.

A. Appendix

A.1 Datasheets of ABA Study

	Phase	time (min)	Processed Microservices	Time per Microservice (min)
1	A	40:00	28	01:26
2	A	12:02	7	01:43
3	A	45:00	31	01:27
4	A	09:30	6	01:35
5	B	34:10	28	01:13
6	B	32:22	29	01:07
7	B	10:44	10	01:04
8	B	34:10	31	01:06
9	A	21:22	15	01:25
10	A	14:10	9	01:34
11	A	45:08	30	01:30
12	A	14:10	10	01:25
13	A	36:12	25	01:27
14	A	42:19	28	01:31

Table A.1: Datasheet of Participant 1

	Phase	time (min)	Processed Microservices	Time per Microservice (min)
1	A	09:31	3,00	03:10
2	A	20:23	5,00	04:05
3	A	15:32	4,00	03:53
4	B	07:28	3,00	02:29
5	B	04:31	2,00	02:15
6	B	06:43	3,00	02:14
7	B	02:03	1,00	02:03
8	B	04:10	2,00	02:05
9	B	09:29	5,00	01:54
10	A	20:34	5,00	04:07
11	A	10:13	3,00	03:24
12	A	04:42	1,00	04:42
13	A	14:21	4,00	03:35
14	A	19:43	5,00	03:57

Table A.2: Datasheet of Participant 2

	Phase	time (min)	Processed Microservices	Time per Microservice (min)
1	A	18:20	9,00	02:02
2	A	06:54	3,00	02:18
3	A	21:11	11,00	01:56
4	B	15:43	11,00	01:26
5	B	05:41	4,00	01:25
6	B	11:38	9,00	01:18
7	B	06:43	5,00	01:21
8	B	13:33	11,00	01:14
9	B	03:53	3,00	01:18
10	B	02:33	2,00	01:16
11	B	05:02	4,00	01:15
12	A	24:32	11,00	02:14
13	A	09:42	5,00	01:56
14	A	18:12	9,00	02:01

Table A.3: Datasheet of Participant 3

	Phase	time (min)	Processed Microservices	Time per Microservice (min)
1	A	35:32	17,00	02:05
2	A	07:32	3,00	02:31
3	A	32:21	15,00	02:09
4	A	24:43	11,00	02:15
5	A	08:12	4,00	02:03
6	A	28:14	13,00	02:10
7	B	07:51	5,00	01:34
8	B	20:10	17,00	01:11
9	B	12:32	11,00	01:08
10	B	02:14	2,00	01:07
11	B	13:55	14,00	01:00
12	A	21:00	9,00	02:20
13	A	33:21	17,00	01:58
14	A	07:23	3,00	02:28

Table A.4: Datasheet of Participant 4

Bibliography

[Bel03] Donald Bell. Uml basics: An introduction to the unified modeling language. *The Rational Edge*, 2003. (cited on Page 10)

[BKNS99] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Optics-of: Identifying local outliers. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 262–270. Springer, 1999. (cited on Page 48)

[Buf05] Jim Buffenbarger. A large-scale fault-tolerant distributed software-build process. In *British Computer Society Configuration Management Specialist Group Conference*. Citeseer, 2005. (cited on Page 6)

[Che17] Lianping Chen. Continuous delivery: overcoming adoption challenges. *Journal of Systems and Software*, 128:72–86, 2017. (cited on Page 53)

[CRZ10] D. Cavallucci, F. Rousselot, and C. Zanni. Initial situation analysis through problem graph. *CIRP Journal of Manufacturing Science and Technology*, 2(4):310–317, 2010. Competitive Design: CIRP Design Conference, Cranfield, United Kingdom. (cited on Page 9)

[DFT12] Pat Dugard, Portia File, and Jonathan Todman. *Single-case and small-n experimental designs: A practical guide to randomization tests*. Routledge, 2012. (cited on Page 46)

[DVDE+16] Shahir Daya, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo Martins, et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016. (cited on Page 1 and 8)

[EBF+17] Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Declarative vs. imperative: two modeling patterns for the automated deployment of applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, pages 22–27. Xpert Publishing Services (XPS), 2017. (cited on Page 53)

[Fis19] Ronald A Fisher. The causes of human variability. *The Eugenics Review*, 10(4):213, 1919. (cited on Page 48)

[Gfa12] Peter Gfader. Use scrum+ continuous delivery to build the righ thing. *Scrum. org*, 2012. (cited on Page 5)

[Hac08] Allan Hackshaw. Small studies: strengths and limitations, 2008. (cited on Page 46)

[HdPB09] Gaël P Hammer, Jean-Baptist du Prel, and Maria Blettner. Vermeidung verzerrter ergebnisse in beobachtungsstudien. *Deutsches Ärzteblatt*, 106(41):664–668, 2009. (cited on Page 46)

[HF10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010. (cited on Page 1, 8, and 55)

[HVdV08] Mia Hubert and Stephan Van der Veeken. Outlier detection for skewed data. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 22(3-4):235–246, 2008. (cited on Page 49)

[Jac94] Harold R Jacobs. *Mathematics: A human endeavor*. Macmillan, 1994. (cited on Page 49)

[KHS07] Ron Kohavi, Randal M Henne, and Dan Sommerfield. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 959–967, 2007. (cited on Page 46)

[KL17] Ron Kohavi and Roger Longbotham. Online controlled experiments and a/b testing. *Encyclopedia of machine learning and data mining*, 7(8):922–929, 2017. (cited on Page 46)

[KMY05] Mira Kajko-Mattsson and Fan Yulong. Outlining a model of a release management process. *Journal of Integrated Design and Process Science*, 9(4):13–25, 2005. (cited on Page 6 and 7)

[KRH+20] Umme Ayman Koana, Shadikur Rahman, Fakhrul Hasan, Aras M Ismael, and Karmand Hussein. c. *management*, 29(8):4942–4949, 2020. (cited on Page 1, 2, 5, 6, 7, 53, and 55)

[LJ11] Antti Lahtela and Marko Jäntti. Challenges and problems in release management process: A case study. In *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pages 10–13. IEEE, 2011. (cited on Page 1, 7, and 53)

[LS11] Marcel Langer and Dirk Söffker. Human guidance and supervision of a manufacturing system for semi-automated production. In *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, pages 1–6, 2011. (cited on Page 8 and 10)

[MLM+06] C Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service

oriented architecture 1.0. *OASIS standard*, 12(S 18), 2006. (cited on Page 8)

[MM+17] Manuel Mazzara, Bertrand Meyer, et al. *Present and ulterior software engineering*. Springer, 2017. (cited on Page 1 and 8)

[oGC07] Great Britain. Office of Government Commerce. *Service operation*. IT infrastructure library. Stationery Office, 2007. (cited on Page 5)

[RS05] Günther Ruhe and Moshood Omolade Saliu. The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005. (cited on Page 6)

[Spe19] Sandro Speth. Issue management for multi-project, multi-team microservice architectures. Master's thesis, 2019. (cited on Page 53)

[SS18] P Shanmugasundaram and B Sarojini. An overview on release and deployment management strategy. *International Journal of Pure and Applied Mathematics*, 118(9):5–12, 2018. (cited on Page 8)

[SSQM08] Antonio Solanas, Vicenta Sierra, Vicenç Quera, and Rumen Manolov. Random assignment of intervention points in two-phase single-case designs: Data-division-specific distributions. *Psychological reports*, 103(2):499–515, 2008. (cited on Page 46)

[SVHMB21] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):1–44, 2021. (cited on Page 15)

[SYM17] Sherolwendy Anak Sualim, Noraniah Mohd Yassin, and Radziah Mohamad. Comparative evaluation of automated user acceptance testing tool for web based application. *International Journal of Software Engineering and Technology*, 2(2), 2017. (cited on Page 7)

[Tho15] Johannes Thoenes. Microservices. *IEEE Software*, 32(1):116–116, 2015. (cited on Page 8)

[vdHHHW97] André van der Hoek, Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. Software release management. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, page 159–175, Berlin, Heidelberg, 1997. Springer-Verlag. (cited on Page 5 and 53)

[WI05] Dewey Lonzo Whaley III. *The interquartile range: Theory and estimation*. PhD thesis, East Tennessee State University, 2005. (cited on Page 48)

[WSL+20] Phillip Webb, Dave Syer, Josh Long, Stéphane Nicoll, Rob Winch, Andy Wilkinson, Marcel Overdijk, Christian Dupuis, Sébastien Deleuze, Michael Simons, et al. Spring boot reference documentation. *Retrieved June*, 22, 2020. (cited on Page 36)

[You14] Scott WH Young. Improving library user experience with a/b testing: Principles and process. *Weave: Journal of Library User Experience*, 1(1), 2014. (cited on Page 46)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 15 February 2022