Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Detection of Unintended Configuration Changes in Continuous Deployment Pipelines

Johannes Günthör

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Thomas Düllmann, M.Sc., Christian Endres, M.Sc.
Commenced:	2018-04-23
Completed:	2018-10-23

Abstract

Once an Amazon Web Services employee took numerous servers offline that should have stayed online. The resulting US-East outage originated from a single integer that was incorrectly inserted in a configuration file. The configuration file defined the number of servers that should be up and running. The change was legal, but the number of servers that where taken offline was to high. We took the idea of such simple errors and applied them to continuous deployment pipelines. Continuous deployment pipelines are the next evolutionary step in continuous build pipelines. The Amazon Cloud is a highly automated environment. Pipelines are similar to the Amazon cloud a highly automated environment. Small errors can have potential catastrophic outcomes. We interviewed two experts in two differently sized companies which are using continuous integration and continuous delivery pipelines. Based on the information that was provided by both experts about the state of current continuous pipelines, we derived influence factors that are problematic in continuous deployment pipelines. The discovered influence factors already exist in currently used continuous integration/delivery pipelines were they do pose less significant threats than in continuous deployment pipelines. The enhanced automated deployment process of continuous deployment pipelines are making these factors problematic. We developed classification and improvement methods for each of the discovered influence factors. These methods can be used to strengthen a pipeline against unintended configuration changes.

Kurzfassung

Vor einiger Zeit schaltete ein Mitarbeiter bei Amazon Web Services einige Server aus, die jedoch hätten eingeschaltet bleiben sollen. Der resultierende Systemausfall der Zone US-East hatte ihren Ursprung in einem einzigen falsch eingegebenen Zahlenwert in einer Konfigurationsdatei. Die Konfigurationsdatei definierte die Anzahl der Server die angeschaltet waren. Die änderung war beabsichtigt, aber die Anzahl der abgeschalteten Server war zu hoch. Wir haben diesen Vorfall von Fehlern in automatisierten Umgebungen genommen und haben das Konzept auf Continuous Deployment Pipelines angewandt. Continuous Deployment Pipelines sind der nächste Schritt in der Evolution von Continuous Pipelines. Die Amazon Cloud ist eine hochautomatisiertes Umgebung. Pipelines sind ähnlich wie die Amazon Cloud ein ebenfalls hoch automatisiertes Umfeld. Selbst kleine Fehler einen katastrophalen Ausgang nehmen können. Wir haben zwei Experten in unterschiedlich großen Firmen interviewt. Beide Experten arbeiten mit den evolutionären Vorgängern von Continuous Deployment Pipelines, nämlich Continuous Integration und Continuous Delivery Pipelines. Basierend auf den Informationen, die uns die beiden beide Experten über den aktuellen Stand von Continuous Pipelines gaben, habe wir Einflussfaktoren abgeleitet die problematisch im Umfeld von Continuous Deployment Pipelines sind. Probleme, die in momentan existierenden Continuous Delivery Pipelines heute noch eine untergeordnete Rolle spiele, werden jedoch bei hochautomatisierten Continuous Deployment Pipelines problematischer. Die entdeckten Einflussfaktoren sind Probleme die bereits in momentan existierenden Continuous Integration/Delivery Pipelines existieren, in welchen sie jedoch nur eine untergeordnete Rolle spielen. Die erhöhte Automatisierung in Continuos Deployment Pipelines machen diese Einflussfaktoren problematischer. Wir haben sowohl Klassifikationsmöglichkeiten als auch Verbesserungsmöglichkeiten für jeden der entdeckten Einflussfaktor entwickelt. Diese Methoden können zur Stabilisierung von Pipelines genutzt werden, um sie gegen unbeabsichtigte Konfigurationsänderungen zu schützen.

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Research Goal and Questions . . . 2
1.3 Thesis Structure . . . 2

2 Foundations and Related Work 5
2.1 Motivation . . . 5
2.1.1 Amazon Web Services (AWS) . . . 6
2.1.2 Facebook . . . 6
2.2 Unintended Configuration Changes . . . 7
2.3 Continuous Software Engineering Techniques . . . 8
2.3.1 Continuous Integration . . . 8
2.3.2 Continuous Delivery . . . 9
2.3.3 Continuous Deployment . . . 9
2.4 State of the Art Pipelines . . . 9
2.4.1 Manual Deployment . . . 10
2.4.2 Continuous Deployment Pipelines . . . 11
2.4.3 Pipeline Components . . . 13
2.5 Pipeline Editors . . . 15
2.5.1 DevOps . . . 16
2.5.2 Editors according to literature . . . 16
2.6 Related Work . . . 17
2.6.1 ConfErr . . . 17
2.6.2 JobConfigHistory . . . 18
2.6.3 Infrastructure as Code . . . 19
2.6.4 Canary Analysis . . . 19
2.7 Foundational conclusion . . . 20

3 Case Study 23
3.1 Case study Introduction . . . 23

3.2 Designing the Case Study . . . 23
3.2.1 Defining the Case . . . 24
3.2.2 Binding the Case . . . 25
3.2.3 Protocol Intentions . . . 25
3.3 Question Selection . . . 27
3.4 Question Protocol . . . 29
3.4.1 Introduction and General Information . . . 29
3.4.2 Implementation of Methods and Techniques Proposed by Literature 30
3.4.3 Verification or Disproval of Suspected Influence Factors . . . 32
3.4.4 Discovery of Additional Influence Factors, Outages and Priority . 36
3.4.5 Discovery of Implemented Prevention Techniques . . . 37
3.5 Planned Data Collection Methodology . . . 38
3.6 Protocol and Data Collection Improvements . . . 39
3.7 Ethical Considerations . . . 41
3.8 Contributing Companies . . . 42
3.8.1 Company Alpha . . . 42
3.8.2 Company Beta . . . 43
3.9 Case Study Execution at Company Alpha . . . 43
3.10 Answers of Company Alpha . . . 44
3.10.1 Introduction of the Thesis, the Interviewer and the Interviewee . 44
3.10.2 Implementation of Methods and Techniques Proposed by Literature 45
3.10.3 Verification or Disproval of Suspected Influence Factors . . . 48
3.10.4 Discovery of Additional Influence Factors, Outages and Priority . 52
3.10.5 Discovery of Implemented Prevention Techniques . . . 54
3.10.6 Case Study Summary at Company Alpha . . . 54
3.11 Observed Incident at Company Alpha . . . 55
3.12 Case Study Execution at Company Beta . . . 57
3.13 Answers of Company Beta . . . 57
3.13.1 Intro and General Information . . . 58
3.13.2 Implementation of Methods and Techniques Proposed by Literature 59
3.13.3 Verification or Disproval of Suspected Influence Factors . . . 60
3.13.4 Discovery of Additional influence factors, outages and priority . . 63
3.13.5 Case Study Summary at Company Beta . . . 65
3.14 Evaluation . . . 65

4 Influences and Classification 67
4.1 Data Analysis Technique . . . 67
4.2 Qualitative Data Analysis . . . 68
4.3 Identified Influence Factors . . . 68
4.3.1 Conditions & Systematic Testing . . . 69
4.3.2 Number of Configuration Interfaces . . . 70

4.3.3 Insertion Slips . . . 70
4.3.4 Pipeline Security . . . 70
4.3.5 Engaging Pipeline Logic . . . 71
4.4 Reporting the Case Study . . . 72
4.5 Case Study Validity . . . 73
4.5.1 Conclusion Validity . . . 73
4.5.2 Internal Validity . . . 74
4.5.3 Construct Validity . . . 76
4.5.4 External Validity . . . 78
4.5.5 Validity conclusion . . . 79
4.6 Classifying Identified Influence Factors . . . 80
4.6.1 Classifying Slip Origins & Configuration Interfaces . . . 80
4.6.2 Classifying Conditions and Testing Requirements . . . 82
4.6.3 Classifying Security . . . 83
4.6.4 Classification of the Engagement of Pipeline Logic . . . 83

5 Measures to Improve Pipelines 85
5.1 Improving Pipeline Testing by Conditional Execution . . . 85
5.1.1 Testing Environment . . . 85
5.1.2 Canary Analysis . . . 86
5.2 Safeguarding Pipelines Against Unintended Changes in Configuration Interfaces . . . 88
5.2.1 Implementation UICPT . . . 89
5.3 Improving the Security of a Continuous Deployment Pipeline . . . 90
5.3.1 Jenkinsfile . . . 91
5.4 Reducing the Number of Slips while Inserting Pipeline Parameters . . . 92
5.4.1 IDE & Compiler . . . 92
5.4.2 Predefined Parameterization . . . 93
5.5 Improving the Approach of Pipeline Logic . . . 94
5.5.1 Testing Environment . . . 94
5.5.2 Logic Containments & Affecting Live Environment . . . 94
5.5.3 Graphical Pipeline Builder . . . 94

6 Evaluation, Conclusion and Future Work 99
6.1 Evaluation of the Identified Influence Factors . . . 99
6.1.1 Evaluational Setting . . . 100
6.1.2 Evaluation Execution . . . 100
6.1.3 Conditions & Testing . . . 100
6.1.4 Number of Configuration Interfaces . . . 101
6.1.5 Insertion Slips . . . 101
6.1.6 Pipeline Security . . . 101

6.1.7 Engaging Pipeline Logic 102
6.2 Discussion of Results 102
6.3 Conclusion 103
6.4 Future Work 104

Bibliography 105

List of Figures

2.1 Schematic representation of including continuous techniques 10
2.2 General structure of continuous deployment pipelines according to [Che15; GRH15b; LSKM15; SBZ17; URS+17] 13
2.3 Basic Canary Release Process 20

3.1 Tree representation of fault classes according to Avizienis et al. [ALRL04] 28
3.2 Development Pipeline at Company Alpha 46
3.3 Release Pipeline at Company Alpha 46
3.4 Deployment Pipeline at Company Alpha 47
3.5 Deployment Control Flow 1 56
3.6 Deployment Control Flow 2 56

4.1 Classification System Slips & Interfaces based on Chapter 3, [OWA18] and [GR92] 82

5.1 Activity Diagram Unintended Interface Change Prevention Tool (UICPT) 87
5.2 Activity Diagram Unintended Interface Change Prevention Tool Implementation (UICPT) 89
5.3 EGit Checked out Main Project With Jenkinsfile Inside Submodule 91
5.4 Both Individual Repositories 91
5.5 Jenkins Snippet Generator 95
5.6 SharePoint Query Builder 95
5.7 Example of coding in a Game Engine 96

Chapter 1

Introduction

1.1 Motivation

Since Fowler's popular article about continuous integration [FF06] further advancements of automatically integrating software have been made [SBZ17]. Continuous build pipelines are resulting in a higher IT performance and are reducing the pain of the deployment process [Fos+17]. A continuous build pipeline should therefore be part of every software project. Build pipelines are continuously evolving toll chain which can now also do automated deployment to production [SDG+16]. The new techniques improved the speed of software development but they also pose new challenges [LMP+15] like being more error prone to configuration mistakes. Since the entire deployment step is automated with continuous deployment pipelines [SBZ17], it is easy to shutdown or change the live environment by just changing a few values in the configuration of a pipeline. We have seen in other highly automated environments, such as the Amazon cloud, that small unintended configuration changes can result in catastrophic outcomes like the outage of an entire Availability zone [AWS17].

The problem of faulty configuration values from human interfaces is an accepted existing problem [Woo04]. It is however not yet perceived as a common problem in build pipelines. The reason for this is that most pipelines are not continuous deployment pipelines. This is partially because it is not purposeful to implement a continuous deployment pipeline in projects where the production environment is not a ubiquity (e.g., firmware for hardware) [SBZ17]. The pitfalls of continuous deployment pipelines are therefore still subject to research. We inspected two pipelines that are at the current state of continuous integration and continuous delivery pipelines. In our research we formulated a number of potential influence factors that could potentially alter the behavior of pipelines by unintentionally changing the configuration of a pipeline. We used information as well as approaches from other environments that are already

engaging the problem of faulty inputs and unintended configuration changes in software to apply them on continuous deployment pipelines (e.g., Conferr [KUC08]). We then interviewed two experts in different companies that work on a regular basis with continuous integration/delivery pipelines. The experts were interviewed to verify or disprove the suspected influence factors.

We used the gathered information about the current state of the art in build pipelines to derive identified influence factors that are potential threats in continuous deployment pipelines.

For each of the identified influence factors we built a classification system to specify whether a custom pipeline is exposed to the identified influence factor.We also proposed a number of possible enhancements that can be implemented to harden a pipeline and potentially prevent unintended configuration changes in the improved build pipeline.

In our evaluation we present the results to a third party expert that is also working with build pipelines on a regular basis. He verified the results of the case study regarding validity in his current project.

1.2 Research Goal and Questions

The first goal of this thesis is to verify whether the existence of unintended configuration changes poses a threat to continuous deployment pipelines. In other words, "are there unintended configuration changes in build pipelines?". We also investigated the frequency of such configuration changes. Do these configuration changes effect on the development or production environment. In our case study we used the following three questions to analyze how to improve pipelines in the future in order to avoid unintended configuration changes.

RQ.1 What are influence factors that can potentially result in unintended configuration changes?

RQ.2 How to know if a specific pipeline is affected by an influence factor?

RQ.3 How would a solution to the discovered influence factors look like to strengthen a pipeline against unintended configuration changes?

1.3 Thesis Structure

This thesis is structured as follows.

Chapter 2 - Foundations and Related Work This chapter gives an overview about fundamentals of the definition of unintended configuration changes, the state of the art in build pipelines, and how other environments already deal with similar problems.

Chapter 3 - Case Study Based on the research prior to this chapter we discovered a set of possible factors that can potentially influence the behavior of pipelines in an unintended way. Following up on the suspected influence factors is a case study which is the main part of this thesis. We designed and conducted interviews in two different companies. Guiding through the interview are the suspected influence factors which we try to confirm or disprove throughout the interviews.

Chapter 4 - Influences and Classification We took the gathered data from both interviews in the conducted case study to identified which of the potential influence factors in fact pose a risk on continuous deployment pipelines. The identified influence factors are identified based on the findings in the case study in Chapter 3. We also validated the case study and its results based on validity concerns that are associated with case studies. The chapter ends with the development of a classification system for the identified influence factors. Based on this classification system, each custom pipeline can be evaluated whether it is potentially affected by one of the influence factors.

Chapter 5 - Improving Continuous Pipelines We propose improvements for pipelines so that each of the discovered influence factors may prevented. Some of these improvements are organizational structure while others are just methods or implemented tools that can be downloaded and plugged into the pipeline. All improvement approaches strive for hardening a pipeline against the corresponding identified influence factor.

Chapter 6 - Evaluation This chapter concludes the thesis. We interviewed an expert about our results and the effectiveness from his point of view. The expert is independent from the first two experts that we interviewed. We also discussed the overall result of this thesis and what we would like to see in future work about unintended configuration changes in continuous deployment pipelines.

Chapter 2

Foundations and Related Work

Part of the software development process is the usage of other software tools [Wol16]. To cope with increased size and different requirements of today's large-scale software projects development processes have been continuously improved. Developing software today is a streamlined process that is widely established by using software engineering techniques like continuous integration [SBZ17].

This chapter will give a comprehensive overview of techniques and tools that are part of modern deployment pipelines. The foundational basis provided in this chapter provides the motivation on which the thesis is built. We present two outages that occurred in large automated systems to demonstrate the risks posed by modern software systems. Next we present a definition of unintended configuration changes as well as the state of the art of continuous pipeline techniques and their components. The section about pipeline editors shows collected information about people who commonly change the behavior of a pipeline. In the context of deployment pipeline editors, the closely emerging term 'DevOps' [ZBC16] [SDG+16] will be described. Finally, we investigate related tools and techniques that cover unintended changes in pipelines, tools and approaches in other environments.

2.1 Motivation

Errors in a software system can have catastrophic outcomes for the affiliated company and are therefore worth preventing. This thesis is about the detection of configuration changes in a deployment pipeline that were not desired by the maintainers of the pipeline. An outage that occurs, no matter the cause, can have multiple consequences. Users of the service are usually really upset about the unavailability of the service (e.g.

[1]). What follows is not just fixing the problem but also a public statement why the service was not working properly. The following subsections are going to summarize some large outages were the root cause was connected to an unintended configuration change in a large automated system. We use these examples to show what unintended configuration changes in large automated systems can cause. We further investigate if such configuration changes are applicable to deployment pipelines which are similarly large automated environments. We investigate the question if similar errors and outages are reasonably caused by deployment pipelines and how they do occur.

2.1.1 Amazon Web Services (AWS)

Amazon Web Services (AWS) is a secure cloud services platform, offering compute power, database storage, content delivery and other functionality [AWS18]. On 28.02.2017 an outage on AWS US east occurred. While trying to improve a slow system, a small number of servers should be taken offline by a trained member of a maintenance team. The selected person of the debugging team closely followed the recommended procedure for this. Unfortunately, he inserted a wrong value that defined how many servers should be shut down. This resulted in the outage of availability zone 'US East N. Virginia' from 9.37 AM to 1.18 PM [AWS17].

The exact severity of such an outage is hard to estimate. Many factors like perception of users are hard to measure. The measurable aspect that is minimal consequence of this outage is the Service Level Agreement (SLA) violation by AWS. AWS advertises their cloud storage with availability zones that range from fault tolerant (less than 45 minutes outages per month) to high availability (less than 5 minutes outage per month). This outage lasted 3 hours and 41 minutes and therefore violates the SLA agreement with every contractor in this zone. Depending on the rented service and the belonging SLA, the next billing cycle gets reduced by 10%, 30% [AWS16a], or even 100% [AWS16b]. The exact severity of such an outages is hard to measure, but it is obvious that such outages results in significant losses.

2.1.2 Facebook

Facebook is a social network which connects users by allowing them to share content like texts, pictures and videos. On 23.09.2010 a large outage happened. The public apology tried to explain what happened. The fallback configuration value that was stored in a

[1] https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919

database was changed. A system for validating configuration values that is used in each client signaled an error on the new value. The result was that all clients made a call to a database which resulted in a feedback loop. The database was not meant to handle such large numbers of requests and had to be restarted. As a result Facebook's website was not accessible for two and a half hours [Joh10].

This shows how catastrophic small changes in such automated environments can be. A small changed value that was faulty and probably not connected to most of Facebooks infrastructure caused catastrophic damage. More interestingly is that the outage was caused by a system that should help to identify incorrect configuration values. It therefore seems obvious that Facebook anticipates problems with faulty configuration values or had problems with this in the past. This is however just speculation since such problems are usually kept a secret to prevent bad publicity and are only made public if an explanation is needed for an outage that was visible to the common user. Besides damage done to the public opinion (the comments below the apology paint a clear picture) there was most likely a financial drawback. The cost on how much money was lost was not made public and can therefore only be estimated. At the time, Facebook had roughly 500 million users [Sta17b] and the global revenue in the relevant quarter was 467 Million U.S. dollars [Sta17a]. The outage occurred at 11.30 a.m. PST [ODe10] (7.30 p.m. UTC) and took between 2.5 hours [Joh10] and 3.5 hours [ODe10] to resolve. It is clearly visible that the outage occurred during the day when a lot of users where online and tried to use Facebook. In this estimation it is assumed that every hour results in the same amount of revenue for Facebook. This is clearly not an exact representation since there are clearly less people using Facebook at 05.00 a.m. than at 11.00 a.m. As a result, the calculated revenue loss is probably lower than the real value. The hereby estimated loss was calculated with the formula shown below. The formula calculates to a lost revenue of roughly 0.63 million U.S. dollars.

$$\frac{QuarterRevenue}{QuarterDays * HoursPerDay} = RevenuePerHour$$

2.2 Unintended Configuration Changes

This thesis is about the detection of unintended configuration changes in continuous deployment pipelines. A pipeline that is set up once has a specific purpose of delivering or deploying software.

Pipelines can and have to be changed during their lifetime. The result being that changes to the configuration of a pipeline sometimes are resulting in an undesired

behavior of the pipeline. The pipeline is not behaving in accordance to the needs of the pipeline user. Transitioning from desired to undesired behavior, by modifying the software of a pipeline, is called an unintended configuration change.

Detecting these unintended configuration changes is the first step in reverting, preventing or undoing changes so that the behavior of a pipeline matches the needs of the users of the pipeline.

2.3 Continuous Software Engineering Techniques

Continuous software engineering techniques are state of the art in developing software [FF06]. Depending on how much automation is used in the development process is, the continuous technique is named differently. Each of today's existing techniques are extending the previous one (See Figure 2.1). The three existing techniques are called continuous integration, continuous delivery and continuous deployment. All these three existing techniques are explained in the following Sections.

2.3.1 Continuous Integration

Continuous integration (CI) is the process of frequently compiling a new build (at least once per day), based on the latest code commit of every developer [Lei17a]. The benefit being that there is no merging of different parts before the release of a project. Before CI was an established software development technique, the last step before deploying a project was the integration. Integrating software was hard to estimate since it was unknown how many new bugs would appear by merging different code packages[FF06]. CI constantly merges changes fully automated into a new build on a build server and runs all available automated test cases. As a result, each bug directly appears as a failed build and can be fixed while the code is still in fresh memory of the developer. An additional benefit is that the build server is a fresh machine. If the last build was working on a build server, it will most likely work on client machines and make the final integration a non-event [FF06]. CI therefore works also as a anti pattern to a problem that was quite common during software development. In some projects the code and build would only work on developer machines and not in neutral environments [Lei17a]. In a CI environment bugs and faulty behaviors like this would be discovered within minutes after the commit of such faulty code.

2.3.2 Continuous Delivery

Continuous delivery can be seen as an extension to CI. Enhancing the CI pipeline to always be able to deliver the current build to the client [SBZ17]. This requires not just unit tests but also being able run in a production like environment. The result of a continuous delivery pipeline should therefore be an image that can then be manually copied to the production environment [SBZ17].

2.3.3 Continuous Deployment

Continuous deployment (CD) contains the most automated steps in the continuous software development process [SBZ17]. Sections 2.3.1 and 2.3.2 describe how this definition can be broken down into different levels of automation. What differentiates CD from continuous delivery is the automatic deployment into the production environment [SBZ17]. The final build that is generated by a continuous deployment pipeline is automatically published and can be used by the clients that are using the product [Che15].

Automation generally reduces bugs\time and as a result the cost of development [Che15]. However companies may decide not to automate every single step in a development process. Sometimes, tests are expensive (e.g., long-term hardware resilience) or not possible to fully automate (e.g., user acceptance tests, exploratory tests [MSB17] [Wol16]). The core of CD remains CI that can be set up in every development process (even one man projects) to ensure that future growth of the project will kept the now state of the art continuous deployment technique. Converting old projects and long living noncontinuous pipelines to CD pipelines is costly [FF06]. The benefits are clear but as of now there is no single tool that can be set up. Usually, multiple tools need to be plugged in similar to Lego blocks to build a complete CD environment. CD tool representatives that are part of a modern CD pipeline and that are popping up most commonly are Jenkins, Docker, Github, Subversion, Microsoft Azure and AWS [BHR+15]. A hierarchic representation of continuous techniques can be seen in Figure 2.1

2.4 State of the Art Pipelines

State of the art software development infrastructures mostly contains at least a continuous integration pipeline [FF06]. As described in Section 2.3, continuous pipelines have started with continuous integration [FF06] and have progressed since then. Not just integrating but also being able to always deliver or even always deploy to production

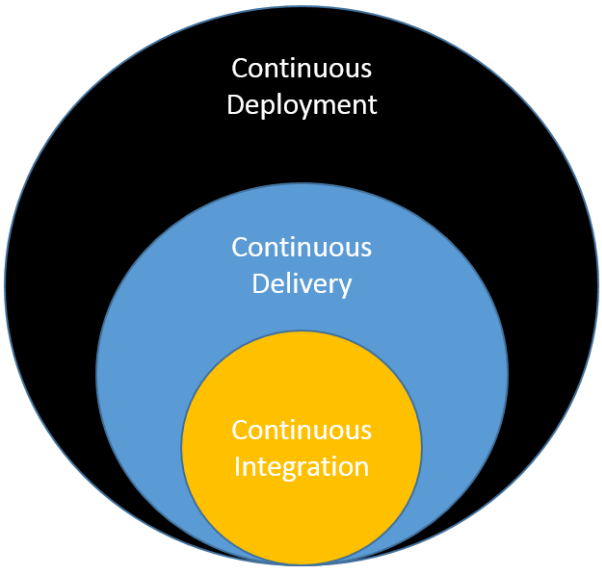

Figure 2.1: Schematic representation of including continuous techniques

[SBZ17]. These now further automated environments significantly reduced the cost of the deployment of software that stood at the end of each release cycle [Che15].

Since complete deployment pipelines are a quite young technique and not yet well established the challenges that are rising are not completely known. There is circumstantial proof that large companies that adopted continuous deployment already have prevention techniques that cope with configuration changes in deployment pipelines [Joh10] but there was no literature found that directly deals with the topic of unintended configuration change detection in pipelines. However, both of the problems described in Sections 2.1.1 and 2.1.2 indicate that pipelines in large organizations deal with this problem and it is therefore worth investigating. Continuous deployment is a extension of continuous integration\continuous delivery[SBZ17], we describe the different techniques to later on understand what the differences are between the influences that we found in continuous integration\delivery and continuous deployment. The differences are important because they have to be considered when drawing conclusions about continuous deployment.

2.4.1 Manual Deployment

Manual deployment is not a defined term. It is merely the sequence of steps that a developer has to take to change his written code into an executable product for the user. Since the beginning of software development this process was done manually

by developers. This manual process was very prone to errors [SDG+16; Wol16] and was quite costly [Che15]. In the early days of the 21st century, continuous integration was proposed as an alternative to manual deployment [FF06]. Since then, continuous deployment techniques have become an imperative in developing software. It has been adopted by global players like Netflix, Facebook, Amazon, Google, Microsoft and many more. High frequency of change is a common practice which is almost impossible to achieve while working with a manual deployment [Ber14a; FFB13]. Since manual deployment is not longer the state of the art in developing software, its flaws and benefits will not be part of this thesis.

2.4.2 Continuous Deployment Pipelines

Using CI in a project is powerful but would be a huge task if it would be done manually after every commit. Global web-based service providers like Netflix, Amazon, and Facebook produce new builds of parts of the entire system every day [Ber14a; FFB13; GRH15b]. The automation in a CI environment is therefore mandatory. Most commonly, Jenkins is used [Wol16] to realize the core of such an environment. Environments where the deployed image is running (e.g., Amazon Web Services) and source code management (SCM) systems (e.g., Git) are also part of a CI/CD Pipeline. Since Jenkins is the widest spread tool it will also be focus in this thesis. Nonetheless, it is important to note that there are useful alternatives. CruiseControl and Travis CI are two CI viable tools that pose an alternative [Han16]. They both bring their own benefits and drawbacks . Instead of Git, Subversion is a useful SCM [Ott09] tool and Azure is an alternative platform to AWS [Col17].

The idea of Pipelines is to organize and automate different tasks that need to be executed to create a new build version. These tasks get streamlined so that each task is automatically performed in a sequenced work flow. Most tasks start with a simple checkout from the SCM and could contain anything from compiling, starting new instances of a docker container, a container which is light weight version of linux that improves the portability of applications [Mer14]. The task then continuous by running unit tests or finishing the pipeline with the final deployment of the newly build image. Jenkins uses plug-ins to realize the extendability of its pipeline to the user. The control flow can be defined by using a Jenkins file, a file in which the commands for Jenkins are defined [Sma11], to set any number of steps in an CI/CD process [Jen17].

Continuous Deployment Pipelines - Common description in Literature

Deployment pipelines are not always the same. They have a standardized skeleton of tools that are used but the final setup looks always different. They emerge from the process of transitioning towards a continuous deployment environment. As a result, each pipeline is built by a set of different tools [URS+17].

Nonetheless, the structure of such a pipeline is often very similar. The general structure of continuous delivery and continuous deployment pipelines is shown in Figure 2.2. It shows four basic stages that every pipeline has. The first stage is a commit that is done to a Source Control Management system (SCM). If a change is made the CI server produces a new application image based on the newest code that is currently stored in the SCM. The frequency of a new build can be customized, but it is recommended that a new build is compiled at least once per day [Lei17a]. The new image then runs through existing tests to ensure that the new code base is working properly. The final step is the deployment of the new build to production [Che15; GRH15b; LSKM15; SBZ17; URS+17]. The difference that each of the examined pipelines seems to have is the testing step. Continuous delivery pipelines are not completely automated [SBZ17] and can therefore contain manual testing. This is one of the reasons why not every company transitioned towards a completely automated deployment pipeline [LMP+15]. Legacy code, user acceptance, exploratory, and similar tests are hard to automate and are therefore still being done manually. It is possible to try and further automate these by implementing A/B tests that are implicitly done by users [You16]. It however is not possible for every environment to fully automate the pipeline and will therefore probably never be a complete continuous deployment pipeline. Pipelines also have a different amount of tests set up at different stages of the pipeline (e.g., performance tests). This is useful since automation requires a large amount of testing [GRH15a]. These tests are however not displayed in Figure 2.2 because only a rough generalization is displayed. A common pipeline almost always gets triggered with the commit of a developer. The commit will then be used to create a new build that contains the old code and the last commit. If the building of the new image is complete it will be tested. If all tests are successful, the new image will be published to the client(s). The time line of these steps varies for each pipeline between just five minutes and possibly years [LMP+15]. Leppänen et al. [LMP+15] states that experienced users that build/use deployment pipelines have mentioned that the higher the automation the lower the time to release a new build to the client(s).

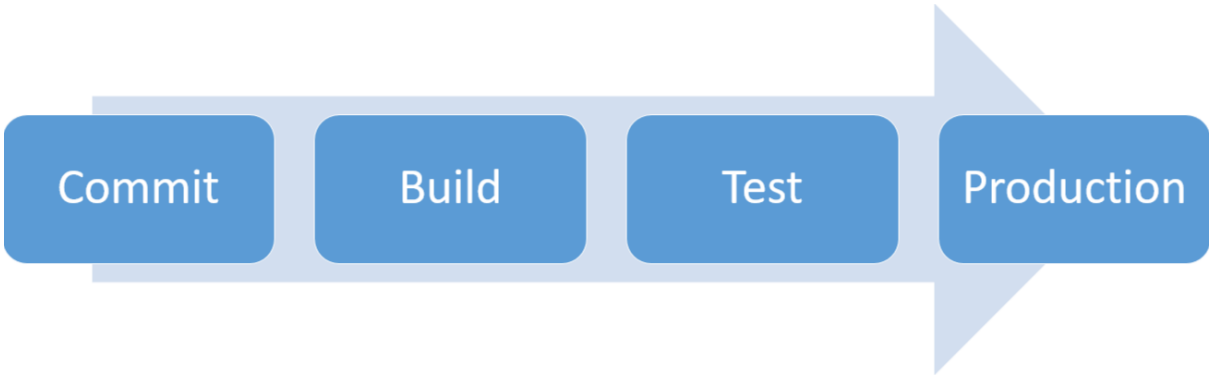

Figure 2.2: General structure of continuous deployment pipelines according to [Che15; GRH15b; LSKM15; SBZ17; URS+17]

2.4.3 Pipeline Components

This section shows what tools are commonly used in a deployment pipeline. Listing specific technologies would probably be a never ending task. Shahin, Babar, and Zhu [SBZ17] summarized amongst other things the tools that were found in their examined environments and can be looked at if specific technology is considered to be used somewhere. The enumeration that Shahin, Babar, and Zhu [SBZ17] made includes not only core functionality but extensions that may or may not be considered useful depending on the product. This difference of possible tool sets was already mentioned in Section 2.4.2. Pipelines are not a defined set of tools but an individual set of components that differ for each organization or even for different software products. Some core tools and their functionality that are mandatory in all deployment pipelines are listed below:

Source Control Management

By today's standards almost every organization should use a source control management (SCM) system. Most sources researching or discussing deployment pipelines were implicitly assuming, that a SCM is used [Che15; FFB13; SBZ17; Spi05]. It is suggested that an SCM is used even in sample projects to ensure a restorable history. Other benefits are that everybody involved in a project is working on the same code base and developers in large projects do not step on each others toes that much [Spi05]. As a result, a SCM reduces the integration time on developer machines. The most commonly used SCMs used in deployment pipelines are Git and Subversion [Ott09; SBZ17]. As a side note Spinellis [Spi05] states that before SCMs became a standard, the transitioning towards an environment that uses a SCM system required the building of a culture that embraces this new technology. The transitioning from no SCM to using SCMs by default has

quite a few similarities to the transitioning from a normal to a continuous development practice.

Continuous Integration Server

Section 2.3.1 already described CI. Building a CI pipeline requires tools. In Fowler's article from 2006, he suggests to use CruiseControl [FF06]. This however seems to be a tool that had already reached its peak and is no longer a recommended tool to use [Han16]. Whether Git or Subersion is the most dominating tool does not seem to be clear. Most sources suggest [Wol16] [Ace17] [SBZ17] that Jenkins has gained the most attention because of its large community, open source license, and availability of plug-ins. Other scientific works from the past three years that have amongst other things worked with CI servers always mention Jenkins as their example [URS+17] [GRH15b] [BHR+15]. This leads to the subjective perception that Jenkins actually is the most commonly used CI server that exists today. A study investigating continuous integration in open-source projects however discovered that Travis is the service that is overwhelmingly used [HTH+16]. This deviation from the general opinion that Jenkins is the largest representative for CI tooling can have multiple reasons. It is possible that, for example, the time needed for projects to switch towards a different possibly better technology is longer than the time that has passed since Jenkins showed up. Another reason could be that Jenkins is open source while Travis sometimes has a monthly cost between 69 USD and 489 USD [Tra18] which may disqualify it from being popular amongst researchers. When building or talking about a CI pipeline, the tool, that is used or should be considered to build such a pipeline, is most likely Jenkins. It has the most support amongst researchers and is the perceived state of the art when it comes to CI tools.

Testing

No matter what kind of continuous practice have been adopted (integration, delivery, or deployment) the automation of tests is highly recommended [FF06]. This is because the side effect of automation is that less humans oversee the resulting code image which makes it easier for mistakes to slip through. Automated tests are mostly unit test that ensure that code that was written and has responsible unit test(s) is working correctly. Reality shows that nowadays acceptance [GRH15a] and exploratory [Wol16] [LMP+15] tests are still done by human beings that are part of the development team. Automating these kinds of tests has, for most companies, not been adopted yet. As long as these tests show results companies do not consider to think about transitioning towards a more continuous model [LMP+15]. It is possible to automate for example

user acceptance tests by realizing an automated A/B deployment model [KDWH16]. This automation lets users make decisions (mostly without their knowledge) on the advantage and disadvantages of new features and designs. The implicit outsourcing of quality assurance to customers is the recommended way when in comes to continuous practices because it can be automated but it may come at a cost. As a downside, resources drawn from customers are quite significant and since continuous deployment is largely adapted by web companies the public opinion can suffer from such a practice. An example where outsourcing the QA team was publicly tried is the youtube.com heroes program [You16] that was negatively perceived by customers for the extra effort that they would have to put in for testing unfinished features.

Deployment Server

A deployment server is nowadays usually build using an external machine in which only a limited amount of people should have access. The deployment server ensures that code that a developer built on his machine is also working on any other machine. The deployment server should therefore not be configurable by the developers [Lei17a].

Further functionality

The above described pipeline components are building the skeleton of most pipelines. Section 2.3.3 already mentioned that no pipelines, aside from the skeleton, are the same. Enhancing pipelines to fit the need of an application or organization is absolutely reasonable and recommended. Possible enhancements that can be made are for example adding additional testing tools/steps or performance monitors to further improve the code quality of the product that uses the pipeline. Another interesting enhancement that was found during research was a deployment management system [SDG+16]. According to Savor et al. [SDG+16] such a management system was built in-house at both companies they investigated. It helped the operators in their task to release a new build to the public. The exact functionality is however not further explained.

2.5 Pipeline Editors

This section aims to answer the question who has access to edit a pipeline, who commonly changes the pipeline and who is maybe forced to do so. Similar to the project-specific tool sets of pipelines as described in Section 2.3.3, the access rights are most likely also project-specific. The listed sources describe who should have access to a pipeline

according to literature and how the access rights are handled at the examined environment. The term DevOps that has gained increasing popularity in the last couple of years [KDWH16] is also described because of its close relation to this topic.

2.5.1 DevOps

DevOps is one of these practices that is continuously becoming more common. According to a study from 2017 more the 25% of the questioned people now work as a part of a DevOps team [Fos+17]. The problem with DevOps is that it is a word that has a different definition for everyone who describes DevOps. The term was coined at a conference in Belgium in 2009 and is the combination of the words Development and Operations [Wol16]. A scientific approach to describe DevOps has been tried in the past.

> "DevOps is a development methodology aimed at bridging the gap between Development (Dev) and Operations, emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices."[JAPT16]

According to this most general definition, the two departments development and operations are now working closer together. The boundaries between developers and operators become fuzzy. The assumption that both, development and operations, have access to the same pipeline is not far fetched. The worst case thinkable is that an operator if forced to change a pipeline that was exclusively built and maintained by the development department. Nonetheless nobody that talks about DevOps will have this general definition in mind. Because this general definition is not applicable to the examined environments, the case study will determine how DevOps is implemented at the examined environments.

2.5.2 Editors according to literature

Information about what tools to use in a pipeline is common knowledge in any scientific work about deployment pipelines [KDWH16; SBZ17; Wol16]. The information who should have access rights to a deployment pipeline are almost never mentioned. It is possible that everyone writing about deployment pipelines acknowledges the different structures of organizations and pipelines, and is therefore reluctant to propose a specific setting of access rights. Multiple sources have mentioned that malicious intrusions are concerns that exist [BHR+15; HF10; KDWH16]. None of these sources however discuss how access rights should be assigned to different roles. That frustrated employees can

manipulate the digital infrastructure is not far fetched. An example would be that a twitter employee deleted a popular political account because he was disagreeing with the contents that were posted there [Hea17]. This is a case where even subcontractors had the same access rights as real employees and therefore the same permissions. When talking about safety, Humble and Farley [HF10] revealed that some companies only allow senior personnel to access the environment in which the compilation and assembly happens. We are now assuming that these companies would now transition towards a continuous deployment pipeline. The guideline about access only to senior personnel for the final step would consist the logical result would be that the entire pipeline is only accessed by senior personnel.

2.6 Related Work

Part of the research on this topic includes the state of the art in unintended change prevention techniques. In other words how is the topic of this thesis already handled in scientific approaches and real deployment pipelines. During our research we did however find only a limited amount. A tool that evaluates errors of human interface configuration was found and is described in Section 2.6.1. Additionally the outage at Facebook that is described in Section 2.1.2 mentions that a configuration validation tool is in use [Joh10]. Facebook however is not willing to share such internal processes and the outage message is a rare peak of the internal work flow.

That (unintended) change detection in deployment pipelines is not handled on a higher level could have different reasons. It would be conceivable that since deployment pipelines are still a young technique that is not that well established yet what the downside of this new technique is. Another reason might be that since pipelines are custom build that validation must also be self-build to satisfy the need of the project in which a pipeline is used. We point out that the tools and methods, that are described in the following, are just a related topic to this thesis. They do not explicitly engage unintended configuration changes in continuous deployment pipelines. We show not just tools and techniques that we researched prior to the thesis, but also additional important tools and techniques that we discovered during the writing of this thesis.

2.6.1 ConfErr

While researching human interface related outages, a tool named ConfErr was found [KUC08]. The paper that describes the tool describes why the tool is important, for what it can be used and the effectiveness of the implementation. To justify the realization of

such a tool the problem that human interaction causes errors in software configuration is shown. According to Keller, Upadhyaya, and Candea [KUC08] approximately 30% of software failures are caused by human errors. It is further mentioned that approximately 50% of human related errors are based on the configuration. These error rates exist in either simple Internet services for regular human beings as well as high risk environments such as nuclear reactors. Considering that people that are involved with such configurations range from inexperienced to well trained would suggest that errors made to an interface are a human problem that shows up independent from the environment. This would lead to the assumption that similar numbers can be estimated for configuration errors made to pipelines. The tool that is described takes configuration files of software and manipulates them. This happens based on different mistakes that were identified as possible outcomes of distracted, inattentive, unexperienced or untrained humans. The manipulated files are then inserted instead of the original files. The tool then tests the faulty configuration files in startup and functional tests for the specific software. As an output reports are generated that show how resilient the given software is to the inserted faulty configuration files. According to the paper the tool was build to enhance testing in the development process since human interface safety is not considered a worthy test candidate.

This thesis however focuses on the unintended detection of pipeline changes. Assessing the resilience of interfaces would therefore be a helpful tool to prevent unintended changes. Tools that are often used in pipelines (e.g., Jenkins) cloud be evaluated on their weaknesses in human interfaces. This weaknesses would then influence the evaluation of the later developed classification. This however would require access to the alleged open accessible source code that can be found [2]. Access to the source code and documentation has been repeatedly requested. The inclusion of ConfErr will therefore depend on the availability of source code or binary [KUC08].

2.6.2 JobConfigHistory

Much like ConfErr this tool is not directly build for change detection in deployment pipelines. The tool however implicitly states with its existence that getting an overview of the configuration and history of a pipeline is functionality that is demanded. The fact that it is used in literature [Wol16] and the number of downloads [BFF+18] show the real applicability. JobConfigHistory is a Jenkins plug-in that is available in the Jenkins plug-in market place or through the Jenkins wiki page [3] . This plug-in permanently stores each old Jenkins configuration file. Is the currently active configuration file changed

[2] http://conferr.epfl.ch/
[3] https://wiki.jenkins.io/display/JENKINS/JobConfigHistory+Plugin

the plug-in collects the now outdated and saves it. Should a rollback be required it is possible to simply select a older configuration file and restore it. It is also possible to compare two configuration files and visualize the difference that exists between those two. The visualization is done similarly to the solving of merge conflicts in SCMs. The simplicity and easy access result in a useful enhancement of a pipeline. However good the enhancement is, without modification it is a plug-in that helps recovering and not detecting changes in a deployment pipeline.

2.6.3 Infrastructure as Code

The idea behind infrastructure as code is that the configuration of a service is now done by coded commands. This idea was most likely founded by Mark Burgess with his program CFEngine in the early 1990's [Joh17]. With the rising of DevOps practices the concept gained increasing popularity [Mor17]. Because configuring a service is now done by code inside a file, it is now possible to apply "normal" software development techniques. It is possible to store those configuration files inside an SCM, do automatic tests to a configuration and let machines and services now be automatically configured [Mor17]. Tools that are commonly used to realize infrastructure as code are for example Puppet and Chef [KDWH16]. The benefits of having Infrastructure as Code are obvious. Creating a new service with the same configuration as currently used is now a trivial task. Human error is eliminated by executing an automated task reliable and repetitive if necessary. With the increasing popularity of cloud computing such configuration tasks have to be done more frequent and the invested time into configuring a virtual machine significantly decreases by using infrastructure as code [Mor17].

2.6.4 Canary Analysis

Canary analysis is similar to A/B testing, a method in which two instances of the same (Canary Analysis)/similar(A/B testing) software are accessible by clients/customers. However, the aim in Canary analysis are not the users. The target is a newly deployed service. Canary analysis tries to detect problems with a newly deployed version that was not existing in the old one [Mar15a]. The canary release process is not replacing old software in a cloud environment. Instead a new cluster is started. Small amounts of production traffic are distributed towards the newly deployed software on the so called 'canary'. In case of errors or performance issues of the new system a rollback is easy to do [Blo18]. The old version of the software is still running and handles most of the production traffic anyways. A cooperation between Google and Netflix published an open source automated version of this canary analysis release process. The software was

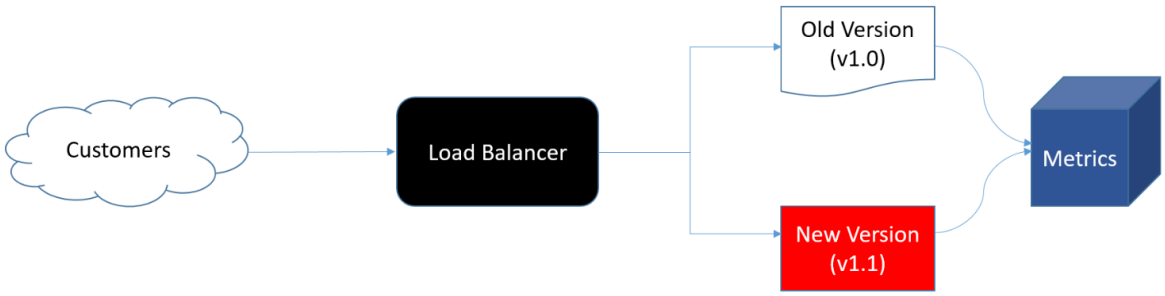

Figure 2.3: Basic Canary Release Process

published on the 2018-05-10, is called Kayenta and can be found in Github [4]. The visual representation of how canary analysis is supposed to work can be seen in 2.3.

2.7 Foundational conclusion

This chapter shows basic information about the topic of this thesis. Key questions like what technologies are used to build a pipeline, who configures the behavior and what are existing change prevention techniques have been answered. The answers to these questions function as a solid foundation on the continuing thesis. It also shows that continuous deployment pipelines are the fastest and most beneficial way to deploy software to a production environment.

A problem however is that continuous deployment is seen with reluctance since the final image is not overlooked by testers before it is deployed to production. The fear that errors are showing up in production is not neglectable. To ensure that continuous deployment works as intended, automated tests are required to be executed before every deploy. Since automated tests are also part of continuous delivery and continuous integration (the continuous techniques are part of continuous deployment), the abstinence of automated testing in continuous deployment is not considered to be the biggest problem. Each predecessor already contains automated tests.

A problem that is however rising up with continuous deployment is the correct functionality of the deployment pipeline. If a pipeline is not working correctly in continuous integration or continuous delivery the result of an faulty image is not considered to be a big problem. This is because before deploying an image to production is overlooked by humans. Ann error that originates from a faulty delivery process would be picked up. In a continuous deployment pipeline however, errors that originate from the deployment

[4]https://github.com/spinnaker/kayenta

process would not be picked up until they show up in production. It is therefore worth to detect unintended configuration changes in continuous deployment pipelines to prevent errors (e.g., a changed load threshold that should have been tested for) that originates from the now fast deployment method of continuously deploying software.

Chapter 3

Case Study

3.1 Case study Introduction

We conducted case studies in two companies that operate in different industry sectors. Both companies operate on an international level and employ more than 10000 employees. One of which already made the transition to a delivery pipeline based deployment. The other company is still in between continuous integration and continuous delivery. These case studies are the core knowledge source for the current state of the art of pipelines and how certain principles that are proposed by the literature (e.g., DevOps) are implemented.

The main goal of these case studies is to verify or refute influences that can change the behavior of a deployment pipeline in an unintended way. In context of suspected influence factors (see Section 3.4), new or potential known factors may be discovered. The fact that two companies with different pipelines are observed is also beneficial. While company Alpha employs hundreds of software developers company Beta only employs approximately twenty people that deal with software development. From these people only one is responsible for the pipeline. This contrast can help to identify what internal structures and settings which are beneficial for a deployment pipeline.

3.2 Designing the Case Study

To ensure that the case study is of high quality, we designed this case study based on the approach of Runeson and Höst [RH08]. The case study was then adjusted by including the guidelines from Baxter and Jack [BJ08]. Runeson and Höst [RH08] provide step

by step explanations for the design, execution, and interpretation of a case study. The structure of the case study follows these steps closely.

3.2.1 Defining the Case

Runeson and Höst [RH08] defined six basic elements that should be part of each case study. These six elements are displayed here.

- The objective of this case study is exploratory [BJ08]. The task of the case study is the verification or disproving of the potential influence factors on the configuration of deployment pipelines. Also discovering additional influence factors, that are not yet known is part of the objective of this case study.
- The case that is the subject of this study is the detection of unintended configuration changes in continuous deployment pipelines. The fact that delivery and deployment pipelines are closely related (they are only differentiated by the final deployment step Section 2.3) makes it possible to study either of them and draw conclusions that apply to both.
- The theory on which the case study is based upon was stated in the motivation in Chapter 2. The idea is that even a small misconfiguration in a large automated environment can change its behavior. This way, it may be possibly to change the corresponding environment which is seen by customers. Ultimately resulting in undesired behavior that originates from unintended configuration change of the deployment pipeline. Such configuration mistakes pose a threat to a service provider that makes these mistakes worth detecting.
- The interview questions are written and justified in Section 3.4. Each of these questions that should be answered in the case study serve a specific purpose of verifying or disproving a suspected influence factor. A specific description why a question is asked in the interview is explained below the question itself.
- The method that was chosen to conduct the case study is categorized as direct (See. Runeson and Höst [RH08]). An interview with a maintainer/developer of a pipeline was conducted to answer the research questions. An important subject that is proposed by Runeson and Höst [RH08] is the triangulation to compensate for statistical anomalies. For data triangulation more than one pipeline is looked at. For observer triangulation, each interview of the pipeline that will be examined will be conducted with a different maintainer/developer. Unfortunately using different methodological approaches is not feasible. Indirect and independent approaches could not be applied due to restrictions by the contributing companies.

Lastly, using a different viewpoint is also not possible. The topic of this thesis is exactly the viewpoint that is taken to conduct the case study.

- The selection strategy which pipelines to examine is partially set by the contributing companies. Company Beta only uses a similar set of pipelines in a single development team. Company Alpha however has access to multiple different pipelines because each development team builds its own pipeline. The observed pipeline at company alpha is part of a project which tries to implement as many agile development techniques as possible. When possible, the units of analysis are defined as a comparison of standard pipelines. For the same reason, the previously mentioned data triangulation is done. We preferred to get an overview of standard pipelines where typical problems exist. Extreme cases that deviate extremely from a typical pipeline are less helpful because a generalization on problems and solutions is not reliable. Previously done research in Section 2.3 suggests that implementations of pipelines are mostly different and therefore defining a "typical pipeline" is hard.

3.2.2 Binding the Case

Baxter and Jack [BJ08] suggest that it is useful to clarify what is explicitly out of scope of the case study. It is stated that this is a common pitfall that is associated with case studies in general. The reason for unintended configuration changes are not limited to software, only. Changes that are made to the underlying hardware layer may be discovered as influence factors during the case study. It is however not the desired target to collect and later on resolve such problems. Unoptimized organizational structure is also out of scope whereas human errors are in scope. It is important to note that unintended configuration change refers to the pipeline and not the software that is processed by the pipeline. This thesis is about detecting undesired configurations of a pipeline. The objective is to ensures that there are no additional bugs that are showing up just because the pipeline was not correctly configured.

3.2.3 Protocol Intentions

The protocol covers all questions that are asked in the interview [RH08]. The interview is designed to cover all topics that should be part of a case study. For each of the examined pipelines the protocol has the same five main goals that are tracked. These five main goals can be seen in Section 3.2.3.

The first goal is collecting background information about the interviewed partner. The goal is not the knowledge-profit but the setting of a relaxed atmosphere for the relevant

interview questions. The maintainer and the interviewer get to know the technological understanding and current mindset of deployment pipelines and processes. Such an interview introduction is done to match the chosen structured glass time model. The glass time model was suggested by Runeson and Höst [RH08] and is explained in Section 3.5.

The second goal is to discover how much common ground there is between literature and the examined pipeline. Are all the changes that are proposed by literature and researchers a target that the company is aiming for? If not, then it would be possible that live environments and researchers are heading in different directions that lead to different problems. The idea for these questions originates from Section 2.4.3 where it is explained that researchers are expecting Jenkins to be state of the art in integrating continuously. The only survey related to this topic however said that Travis CI is used in most environments. Researching about this topic leads to a number of suspected influence factors that can change the behavior of a pipeline in an unintended way. In the interview, each of those suspected influence factors will be investigated to see if they actually influence the behavior of a deployment pipeline in an unintended way and if they can potentially be harmful.

The fourth goal that is investigated is information that is originating from the investigated company. How severe have outages been in the past and what do the developers/maintainers of the pipeline think about potential threats to changes of a deployment pipeline. The idea is that according to Bertram [Ber14b] the developers of a service know the service best. The best problem assessment and solution is therefore preferably produced by a developer of the environment.

The last goal that is tracked by the set of questions is the discovery of existing prevention techniques. In Section 2.6, we already mentioned that there are existing techniques that are kept behind closed doors by cooperations that build those in house. The intention is to see if solutions are also applied by a company that is less involved with IT. For each of these goals, a number of questions is collected in Section 3.4. The section also explains why a question was asked and what the knowledge-profit is that can be gained from the question. A rough overview of the structure for the case study is displayed below.

- Background information about the interview expert
- Are proposals done by the literature a desired/implemented target (e.g., Continuous Deployment, Security, technologie assumptions (Git, AWS, Jenkins))
- Verify or disprove suspected potential influence factors (e.g., Only-Humans, DevOps, change rate)
- Discover additional influence factors and outages

- Discover implemented prevention techniques

3.3 Question Selection

Some of the following questions are a direct result from researching the topic of this thesis. A example would be a slip (e.g., Character substitution [KUC08]) in a human configuration interface. Other questions are introduced because it is reasonable that such problems exist (e.g., a faulty DevOps implementation).

The research of Avizienis et al. [ALRL04] served as a starting point to identify potential sources for errors in pipelines. To apply the research of Avizienis et al. [ALRL04], pipelines are treated as regular software. Considering how pipelines are usually built (described in Section 2.4) this comparison can be done without restraint. According to this source, failures of software are undesired behaviors. The definition of this thesis in Section 2.2 states that an unintended change results in an undesired behavior. Unintended configuration changes in pipelines can be viewed as software failures since the pipelines fails to result in the desired behavior. Failures are the result of errors that are in turn caused by faults. The logical step to prevent an unintended configuration change (failure) would be the prevention of the root cause, the fault.

Faults can originate from three different sources. These are development, physical and interaction faults. These faults can be further specified in eight different elementary fault classes. This hierarchy can be seen in Figure 3.1 [ALRL04]. We considered the fact that physical faults are always related to hardware. We already discussed in Section 3.2.2 that hardware faults are out of scope of this case study and therefore not considered furthermore. The relevant dimension for our case study in which unintended configuration changes could appear are always rooted in software. Other threat possibilities are malicious intents, deliberate faults through harmful decision making, and finally faults that are made either by accident or by incompetence. Malicious intents always refer to a security problem which explains the questions that are about security (e.g., Section 3.4.2). The problem of deliberately making a harmful decision to the configuration of a pipeline is the next shown fault source. To solve this problem, it would be required to prevent authenticated access of authorized developers. Since there is always an access point to change a configuration, it is impossible to pose a preventative solution to this problem. If there is an agreement for a bad decision it will be implemented, no matter the technological enhancements. The remaining two fault sources are accidental and incompetence. These show either during development of a software (pipeline) or during operation of a pipeline. Both of these are considered a major source of potential damage because they do not only fit the problems suggested by research (e.g., slip according to Keller, Upadhyaya, and Candea [KUC08]) but they also

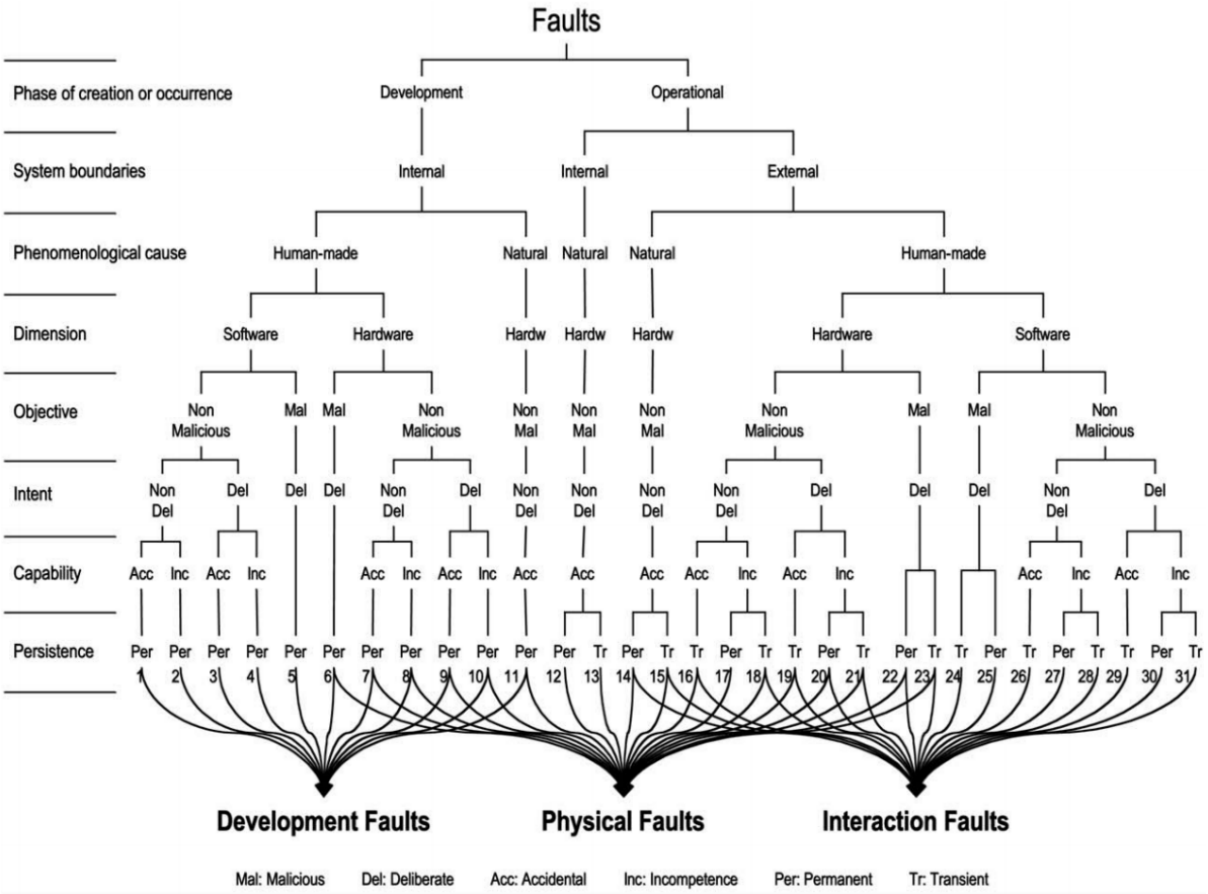

Figure 3.1: Tree representation of fault classes according to Avizienis et al. [ALRL04]

match logical conclusions. Most of the questions that do not originate from researched suggestions focus on these two fault sources.

An important detail in the used taxonomy is that all potential faults that are not physical (hardware related) are always categorized as human made. The result is that human beings are a common factor when talking about faults that are unintended configuration changes. The question: 'How many times do unintended configuration changes originate from humans/machines?' tries to directly verify this suspicion. Depending on the outcome, the human factor will be a significant factor when classifying and solving the verified influence factors.

3.4 Question Protocol

The following questions investigate our suspected influence factors that were asked to the experts that develop or maintain a pipeline. Each expert at each company was asked this block of questions. Below each question is a short explanation what the question should achieve in finding an influence factor for the pipeline. These explanations are only displayed in this thesis and have not be shown to the experts. Also, the suspicions have not been shown to the experts because it would have transformed them into suggestive questions undermining the validity of the study and its results.

3.4.1 Introduction and General Information

The primary focus is not the knowledge-profit but starting the interview on a relaxed manner. The questions are easy to answer and not necessarily restricted to the questions that are stated below. These questions are asked so that trust can be built between both interview partners [RH08].

Introduction of the thesis, the interviewer and the interviewee

This part was added after the execution of the first case study at Company Beta. The idea is to present the topic of this thesis to the interviewed expert. In addition of giving the expert an idea what the thesis is about, both people that are part of the interview gain a common ground of knowledge. Such a common ground is important because like it was explained in Section 2.3 each pipeline and the resulting deployment process look different.

How long has the expert been working with the pipeline?

A short amount of time could mean that importance and impact knowledge is missing. As a result, decisions that are made because of the lack of knowledge can change the pipeline in an undesired way.

How did the expert encounter the pipeline for the first time?

A self thought individual that has only self thought knowledge is probably only in possession of the knowledge of parts over the pipeline that are previously touched by the individual. Changes on different parts of the pipeline are therefore most likely knew and have a high potential to be faulty.

What kind of continuous pipeline is used for the deployment of software (Continuous - Integration, Delivery or Deployment)

Most companies are not using a complete continuous deployment pipeline. This is either because they are still building towards a more continuous development technique or because the final steps of implementing a continuous deployment pipeline are considered too dangerous [LMP+15]. This question should clarify where the examined pipelines are currently at. This helps interpreting the data that has been gathered from this interview. This is a question that was not in the original set of questions. It was added after the first case study was executed.

How does the entire deployment process usually look like?

This question helps the expert/maintainer that is interviewed and the interviewer to get some common ground. This is a question that was not in the original set of questions. It is an extension that was made because it was a huge benefit to the case study at company Beta. The question helped the interview partners to get to common ground on the deployment process. Further questions could sometimes refer back to the overall process. Getting some common ground in the context of deployment pipelines is especially useful because all pipelines are custom build and look differently in even the same companies like it was described in Section 2.3. Furthermore not just the pipelines but also the deployment process is mostly different.

3.4.2 Implementation of Methods and Techniques Proposed by Literature

This first set of Questions is trying to discover how much literature approaches deviate from real environments. The idea is to see what ideas that are proposed by the literature are also implemented in reality. If there are certain ideas that are not being implemented the result could be that there are different problems rising up. The result would be that the suspected influence factors are probably deviating from the real influence factors.

Git,AWS,Jenkins: Are these the used tools and services?

If there are other services used like for example a cloud service provider that has a low availability [GR92] class (or servers in the "basement") that could mean that outages occur from the unavailability of the hardware resources. Also, if there is a less commonly used continuous integration server used that would mean that the implementation of prevention techniques could be harder. A possible example would be plug-ins like the presented JobConfigHistory in Section 2.6.2.

Is continuous deployment the desired target?

According to literature, a continuous deployment pipeline is the best solution for deploying software [LMP+15; SDG+16] because increased automation reduces the time between releases. During research, information indicates that some companies only transition towards a delivery pipeline [LMP+15]. If this would be the case in the examined environment, that would mean that implemented solutions could work different (slower/not immediately) since changes are not directly pushed to clients. The solutions can therefore simpler or less reliable.

How serious is the security in a deployment pipeline taken?

Having an insecure password would make it easy for unauthorized personnel to change the behavior of a pipeline. The literature proposes safe and different passwords for each login [URS+17]. The idea that insufficient security can also unintentionally change the behavior of a pipeline originates from Bass et al. [BHR+15] where it was discovered that malicious attacks are a concern that companies have about their deployment pipelines. Additionally, our experience shows that passwords are loosely handled, especially in "protected" environments. The last time that we worked in a larger project the actual name and password for the integration server was throughout the entire project "admin" (Name) and "jenkins" (password). Important to mention is that the project was conducted in a state of the art research facility.

DevOps - definiton, how and why is implementation a desired target?

It was already stated in subsection 2.5.1 that DevOps is just a vague definition to somehow bridge the gap between the departments development and operations. The interesting question is how the transition is working. Since it is not clearly defined what DevOps is doing exactly, any number of dangerous scenarios are imaginable.

3.4.3 Verification or Disproval of Suspected Influence Factors

This set of questions is probably the most important one. Entities that can potentially be harmful to the configuration of a deployment pipeline have been identified through research about this topic. Each of the discovered factors will be discussed with an expert of a pipeline in the examined environment. This will help to identify threads to the configuration of a deployment pipeline.

Are Slips while Inserting Considered a Threat?

Configuration interfaces do not vary from application to application in a significant matter. Each application will have different settings but the concept generally stays the same. Knowing this, it is obvious that numbers and approaches from other interfaces can be taken into account. Yin et al. [YMZ+11] discovered that most faulty behaviors are originating from incorrect configuration values and that the most common reason for a misconfiguration is a slip (e.g., case alteration, typo) while inserting such values. Since this is also the same reason that was the cause for one of the motivational examples that was described in subsection 2.1.1, it is suspected that this is one of the most important influence factors considering unintended changes in deployment pipelines.

DevOps - What are the remaining differences between development and operation and how is each department trained?

DevOps was already mentioned in the last set of questions. It is also mentioned here because it is suspected that a "bad" implementation could possibly mean that operations and development now have access to the working environment of each other. As a result, it is thinkable that a untrained operations employee is forced to alter a pipeline with a lack of knowledge about it. The clear distinction between tasks that remain in development and in operation is a useful information.

Who has access to a pipeline and its configuration?

This question is the continued train of thought from the previous question. Maybe it is not just DevOps but possibly imaginable that other untrained personnel could possibly access the pipeline and, since they are untrained, accidentally change the behavior of the pipeline without any intention of doing so. Such personnel could include management or an IT department. The unlikeliness of discovering a real threat here is considered to be low but it is nonetheless a possibility. This question was clarified by renaming it

from "Who has access to a Deployment Pipeline" to "Who has access to a Pipeline and its Configuration" after the first case study was conducted.

How many different pipelines exist and what is the difference between those?

A conceivable problem is the change of a pipeline that looks much like another pipeline but it is not the same except a single string (e.g., name of the pipeline). Such a problem would maybe not be detected immediately. For example, the number of active servers is changed for the wrong pipeline. This would then only be visible if the actual load exceeds the number of possible load and the service becomes unavailable for users. The reversed effect (e.g., a larger number is wrongfully inserted in the wrong pipeline) could lead to a higher number of active servers that are not needed for a service. The pipeline that should originally be changed would still carry the old value. The idea for this is originated from personal experience that is humorously summarized in the following tweet[1]:

> puts your life into perspective when someone is shooting a car into space, you're spending an hour looking at a production site wondering why your local changes aren't being reflected

Sometimes human beings make simple mistakes without noticing that they are doing something wrong. With the addition that a configuration change in a deployment pipeline would not be reflected immediately anyway the mistake would then just stay in there without being noticed.

Are there executing conditions in a pipeline?

This question is designed to discover if a pipeline has a "code coverage" of 100% or not. If yes, that means that all misbehaviors that can be generated by a pipeline are inserted during operation. If not, it is possible that the original pipeline contained errors that are dormant until a certain functionality of a pipeline is accessed [ALRL04]. In this case pipeline unit test to prove that a pipeline is working would be a possible solution approach.

[1] https://twitter.com/iamdevloper/status/961230773658427392

How many configuration interfaces impact the behavior of a pipeline?

Having a large amount of tools that contribute to the pipeline can have a dangerous side effect of loosing the overview of the pipeline. Each additional tool brings its own configuration interface which is then inevitably part of the pipeline. Much like a large and old software product has sometimes code fragments of which the use is questionable, to everyone working with the code, this is also a conceivable effect for a pipeline. This knowledge distribution sometimes results in changes that should not be made because there are dependencies on the previous setting [PSV01]. The result being that changes to entities with dependencies may result in undesired behavior of the pipeline.

How many configuration values impact the behavior of a pipeline and how dangerous are single values?

This question is similar to the previous question. A large number of values can also lead to a lost overview of a pipeline [PSV01]. In addition, this question can provide information to discover what potential dangerous impact single values can have on the pipeline. If for example interface values are only a threat in combination with other values, this has to be considered when choosing a prevention technique later on.

How often does a pipeline change?

The change frequency is suspected to be one of the major possible threats. If a pipeline is set up once and never changed it is less likely to contain a configuration error then a pipeline that is permanently changed. It is also important to notice that according to Keller, Upadhyaya, and Candea [KUC08] using an interface over a longer time results in tasks moving from high-level cognitive levels to low ones. Basically, over time the same/similar task gets easier. If an interface is used multiple times, the mistakes made with such an interface that are based on required knowledge go down and simple mistakes like typos go up in their frequency of appearance. This means that the change frequency has a direct correlation with the errors that are made with such an interface [KUC08].

How much time is assigned to the task of changing the configuration of a pipeline?

If changing the pipeline configuration is not seen as a real task that has to be scheduled but rather a thing that has to change "on the fly" it is more likely that a hasty change contains mistakes (e.g., slips).

How familiar is a pipeline maintainer to the tools that are used in the pipeline and how often do the maintainers change?

This question is related to the previous one. If the maintainer of a pipeline is familiar with the tool he then knows what the tool is doing and simple mistakes (e.g., substitutions, case alterations) are more likely to be the dominant error source [KUC08]. The other case would be that the maintainer does not know much about the used tools. In this case it is more likely that changes to the pipeline are severe because the experience gathered with the tool are by trial and error or a similar approach. The result of this question could be that if a maintainer frequently changes, that different error prevention techniques are promising effective results [KUC08].

How many people change a pipeline on a regular basis and how closely do they work together?

If multiple people are maintaining a pipeline, it would be possible that unusual constructs that stand on thin ice are easily changed by another authorized maintainer that does not understand how the pipeline in its current form work. The number of people that work together on a pipeline is therefore a potential influence factor.

How many times do unintended configuration changes originate from humans/machines?

A hypothesis that came up during research is that configuration changes always have to originate from human beings working with a pipeline [ALRL04]. A once set up pipeline will always deliver the same result. A correctly configured pipeline is then only behaving imperfectly if a unintended configuration was made. This has to come from a human being. As a result, the most dangerous (or only) influence factor for unintended changes in deployment pipelines would be human error.

How does testing the pipeline work?

Is testing specifically for a pipeline already done (e.g., unit tests for a deployment pipeline). Since pipelines seem to evolve into custom software projects, this could be an approach that could help prevent unintended changes. If not, how is the company already ensuring a well functioning deployment process.

3.4.4 Discovery of Additional Influence Factors, Outages and Priority

The following questions are designed to let the questioned maintainer talk more freely about the topic of unintended configuration changes in his/her deployment pipeline. This approach is taken because it is a similar one that is taken by Netflix.

"The developers know their service best" [Mar15b]

A maintainer of a pipeline may know where the weak spots are that can possibly bring down the entire construction similar to a normal developer of a software product/service. This helps with identifying influence factors that exist but were overlooked during research. In addition to the personal views of the maintainer, this set of questions also investigates outages that originally motivated this thesis. How severe and often do outages related to pipeline configuration happen? Also how often is such an outage visible to the public? In the motivational Section 2.1 it was suspected that pipeline-related outages are not made public because it is an internal procedure that companies do not share. Especially when the topic is about mistakes that were made. These questions helps to verify or disprove this suspicion.

Do known outages exist in regard to unintended configuration changes in the deployment pipeline?

Question to discover the potential impact of the discovered implemented prevention techniques that this thesis engages. This helps to potentially justify the existence of further research about this topic beyond this thesis.

How often are pipeline connected outages visible to clients of the environment produced by the pipeline?

A hypothesis that was was made in Section 2.1 was that pipelines are an internal process that companies are not willing to share. Especially in the scenario that something goes wrong the negative public relations prohibits a company from publishing mistakes. Outages of deployment pipelines are therefore only published if the result was visible to users of a service. This question is again helping to discover the influence that unintended changes in deployment pipelines have over real environments.

What kind of software is processed by the pipeline?

With the question about the final result of a pipeline, the severity of an outage can be identified. Is the resulting image of a pipeline a service that is used internally the potential danger that can result from an outage in such an environment is most likely less dramatic than the failure of the greeting web page. This question can possibly help to answer the question how much benefit would be gained from a prevention system in what kind of pipelines.

Severity of an outage?

This question is asked to see how important a well functioning pipeline is to the company in which the case study is conducted. The solution that enhances a pipeline to protect it against unintended configuration changes depends on the priority that the pipeline has. If a pipeline is vital to the service that a company provides the solution can be more expensive than a solution in a pipeline that is only responsible for a feature/service that is "nice to have".

Recovery time from an outage connected to unintended changes in a deployment pipeline?

This question is an addition to the previous two. Depending on how long the mean time to repair (MTTR) of an outage is, that was the result of an configuration error, it can be assessed how much a prevention technique is desired and how much work is it worth to spend on a prevention technique. It is also worth looking into the recovery time since configuration errors are according to Yin et al. [YMZ+11] sometimes hard to spot. Verifying this would continue to strengthen the justification of this thesis.

3.4.5 Discovery of Implemented Prevention Techniques

In both motivational examples in chapter 2 it was mentioned that there are internal change prevention techniques running in the mentioned services. This lead to the assumption that unintended configuration change is a problem that is handled by each company privately. This set of questions is focusing on learning about prevention techniques that have possibly been already implemented in the examined environment.

What extends the pipeline to prohibit unintended configuration changes?

Question to see what is already dealt with and what was considered a potential problem in the pipeline that has already been handled. Everything that does not serve a functionality that is required to build, test and deploy code can be an answer to this question. Possible examples would be additional tools to monitor the pipeline, custom in house software or just simply (Jenkins) plug-ins.

Lack of automation prohibiting deployment pipelines?

If so what processes are considered not to be automated and why. It is thinkable that because some things (e.g., exploratory tests [Wol16]) are considered impossible to be automated, a continuous delivery pipeline itself is considered to be a prevention technique in contrast to a deployment pipeline.

3.5 Planned Data Collection Methodology

The methodology on how to collect data in a case study is shown in the used guidelines for this case study [RH08]. Data is gathered at two different companies that are contributing to this thesis and are described in Section 3.8.1 and 3.8.2.

At company Alpha multiple pipelines will be examined to have different sources of data from which conclusions can be triangulated.

Triangulating data from multiple data sources enhances the credibility of the case study [BJ08]. The chosen data collection technique is classified as direct or at a first degree level. The used technique is called interview. The interview is conducted in one of the proposed interview models. The questions that are displayed in Section 3.4 are ordered according to the time glass model.

As the time glass model proposes, the interview is starting with loose conversation that then narrows down to the important information. If the important information sufficiently covered, the interview questions are less important and the interview is becoming more loose conversation again.

The interview starts with the introduction of the topic and some easy to answer questions about the interviewed expert. The questions then narrow down to specify suspicions that came up during research of the topic. After all specific questions have been answered, the interview is becoming less structured. Near the end of the interview, the questions start to become more loose conversation what the expert personally thinks about the

topic and how he would engage it. The last part of the interview would then be a short summary what the findings were to prevent misunderstandings. The interview is therefore only structured in the middle of the interview while at the beginning and close to the end it is unstructured.

Runeson and Höst [RH08] propose that the audio of the interview is recorded. This recommendation is followed. Details are therefore easier to catch since the interview can be listened to multiple times. It is also recommended that the transition from interview to text is done by the person who conducted the interview. This recommendation is also accommodated. The last recommendation that proposes that the subjects that are interviewed are based on differences instead of similarities. This is only partially followed. Different people are getting interviewed but all of these need to have a background of being people that deal on technical basis with a pipeline. Should the examining of pipelines and environment reveal that both, development and operations, are still different departments that both have access to the configuration of their pipelines, then at least one interview will be conducted with an expert from each of those departments.

3.6 Protocol and Data Collection Improvements

Runeson and Höst [RH08] specifically mention that the case study protocol has to be a "continuously changed document that is updated". After conducting the first case study at company Beta, the results have been gathered and can be seen in Section 3.12. Based on this first case study, the procedure of collecting data and questions that are asked in the interview have been adjusted. The following sections summarize how the protocol changed after the first case study. The updated protocol was already shown in Section3.4.

Introduction of the thesis and the interviewer

This question was added to build some common ground between interviewed expert and the interviewer. Like mentioned in Section 2.3 each pipeline and the resulting deployment process looks different at each company. The first set of questions is anyway not built to gain information but rather to build trust between the two people conducting the interview.

What kind of continuous pipeline is used for the deployment of software (continuous - integration, delivery or deployment)?

This question was added to explicitly clarify what kind of pipeline is examined. The ideal state of continuous deployment that would be the best pipeline to look at is currently not state of the art. As a result the examined pipelines are mostly not ideal. The design of this case study in Section 3.2 acknowledges this problem and refers to possibly not ideal pipelines to collect information about how to approach unintended configuration changes in continuous deployment pipelines.

Introduction and general information

This first set of questions was originally called "Expert Background". Because of the extensions made the first set of questions no longer contains only information about the expert. The name was therefore changed in the question protocol to clarify what the section is about.

How does the entire deployment process usually look like?

This questions was added for the same reason that question "Introduction of the thesis and the interviewer" was added. Getting some common ground on the deployment process is important because each deployment process looks different. This is the mostly unstructured and it lets the expert talk freely. It also helps to get both (expert and interviewer) on the same page about where the deployment process and the corresponding pipeline are currently at.

How many configuration interfaces impact the behavior of the pipeline?

The previous question was vaguely formulated. The intentions however do not have changed. The ideal answer to this question on would be a specific number. It is however not important what the number is. More importantly is if the interviewed expert has still an overview of the pipeline and could know/count all the interfaces that can influence the behavior of a pipeline. If this is not the case then a lack of over site is probably existing.

How many configuration values impact the behavior of a pipeline and how dangerous are single values?

This question was, like the previous one, vaguely formulated. The question was changed to specify what the information is that should be gathered. The intention however stays the same. A large pipeline can, much like custom software, result in a knowledge distribution [PSV01]. The result being that changes are made that should be made because changes result in unintended side effects.

How much time is assigned to the task of changing the configuration of a pipeline? & how often does a pipeline change?

Both questions stayed the same. The change that was made is that "How often does a pipeline change?" is now placed before "How much Time is assigned to the Task of Changing the Configuration of a Pipeline?". This is because the new order benefits are more natural interview style where less time is required to introduce a question.

How does testing the pipeline work?

The previous question was about testing the code inside the pipeline. This however deviated to far from the intention of this thesis and the case study. It was mentioned by Baxter and Jack [BJ08] and refereed to in Section 3.2.2 that a common problem by conducting case studies is that the study is not precisely about what the intention of the study was. This question was therefore changed to gather information about testing the pipeline. The question about testing the pipeline that was already existing is therefore eliminated since it is now a duplicate.

What kind of software is processed by the pipeline?

This question was rephrased to specify what information should be gathered. The previous question (what is the target of the pipeline) could be easily misinterpreted.

3.7 Ethical Considerations

Ethical considerations are mainly taken into account because the guidelines propose them [RH08]. In fact both companies that are contributing to this thesis do not want

to be named for legal and competitive reasons. The data that is displayed to give a brief overview of the context of the company in 3.8.2 is approved by a manager of the company. The data that is displayed from company Alpha is also approved. The name of maintainers/developers that are interviewed are not displayed because naming those does not serve a purpose from which information can be gathered to improve the case study.

3.8 Contributing Companies

Two companies are contributing to this thesis. Both companies are presented in Sections 3.8.1 and 3.8.2. The main difference between those is that company Alpha, as an insurance provider, is heavily dependent on customer contact (business to consumer). This is provided by amongst other things web services that are deployed through delivery/deployment pipelines. Company Beta is developing and building hardware for industrial purposes only (business to business). Software is merely a byproduct that runs those products. Both of these companies operate in completely different environments with different sizes and importance of their software products.

3.8.1 Company Alpha

For legal reasons this company does not want to be named in this thesis. The company is operating on a global scale with more than 100.000 employees. The examined local office complex employs more than 1000 employees from which more than 100 people develop in-house software tools. These numbers are broadly summarized to preserve the anonymity of the company. The company is a large insurance provider and as such the legal obligations are playing an important roll in providing information. The company is heavily dependent on businesses to consumer contact. Such is realized by human interaction and a digital appearance on the Internet. The here fore used web pages and the underlying architecture (e.g., webs servers) is at some point running through a delivery pipeline. Besides web applications, in-house software that is required for in house tasks and server back end services are also running through delivery pipelines. It is estimated (exact numbers are not known by the company) that roughly 200 Jenkins instances are used. Each of those instances is used by a team which can build any needed number of pipelines. The exact number of existing pipelines is therefore also not known.

3.8.2 Company Beta

For legal reasons this company does not want to be named in this thesis either. The company is operating on a global scale with more than 10.000 employees. The location in which the case study was conducted employs more than 500 people. The team that uses the examined pipeline has the size of ten people from which one of those is responsible for the pipeline. He is also the leader of the team that tests the software. The company is developing and building hardware products that are used for industrial purposes only. Customers are only other industrial companies. Business to consumer (single human being) relationships are non-existent. The products that are developed and built are for example pumps that can then be installed in ships, pools or similar environments where pumps are needed. The software that is developed is embedded firmware that controls these hardware machines. The hardware that is developed has lifespans of about thirty years. During the lifetime of a product the software is continuously extended.

3.9 Case Study Execution at Company Alpha

This case study was conducted on 2018-06-22 at company Alpha. Due to the size and age of this company, there are a lot of legacy systems, processes, and existing approaches. The case study was conducted by interviewing a team member of about 20 people which are working on a single project. The interviewee is one of three people in the project which are mainly working with the pipelines of the project. The project realizes interaction capabilities with large business customers. The project is a development of a system that is already in place. The new implementation that is done right now is replacing the old version as soon as it is able to serve clients with base functionalities. Despite the legacy culture, processes, and systems, the company is trying to keep up with the latest technology. New projects like the one observed by this case study are trying to use modern approaches like DevOps, Cloud Computing and Pipelines. The interview took close to two hours and was conducted about two months after the first case study at company B. Since it was the second case study, it featured the already updated question protocol that can be seen in Section 3.4. This time the introduction to the topic of this thesis was a more structured approach which resulted in the desired effect. The interviewer and interviewee build a relaxed atmosphere over the first couple of questions. As a result the more important questions that followed up on the introductory part where approached with an open mind and no defensive stance of the interviewee. Similar to the first case study at company Beta the introduction also served as a short period where both interview partners could gather a common ground of knowledge about the topic and the structure of the project.

3.10 Answers of Company Alpha

The following section contains the answers to the question protocol which was designed before the the case studies where conducted. The protocol that is used in this case study is however already refined by the feedback and learned lessons from the first case study at company Beta. Conclusions and results from both case studies are not drawn here. Conclusions are drawn in Chapter 4. The following sections only contain the answers as "raw data" from the conducted interview.The interpretation of gathered data is done in Chapter 4 where both case studies are included in the analysis.

3.10.1 Introduction of the Thesis, the Interviewer and the Interviewee

How long has the expert been working with the pipeline?

The expert has been working with the pipeline for about one and a half year.

How did the Expert Encounter the Pipeline for the First Time?

The expert was introduced to pipelines in the project. A colleague that was resigning built the previous pipeline and gave a short introduction to pipelines in general (about half a day). After that the pipeline functionality was a learning by doing task.

What kind of continuous pipeline is used for the deployment of software (continuous - integration, delivery or deployment)

The current state would be described as a delivery pipeline. Is it at any point necessary to deploy code to production it is possible to do this in about 15 minutes if everybody who needs to confirm releases is available. This delivery process of manual release confirms is still existing in the company. These processes of signing of new production software is still existing, even in this more modern approach of developing software agile. The company is trying to be more agile in its new projects but cant let go of the last additional overview of software.

How does the entire deployment process usually look like?

Each developer develops code in his own branch. When a feature completed the branch gets merged into the master branch of the project. The master branch is then built on a development VM instance in a manner of continuous integration. If the development instance is considered worth releasing, the code that was used to build the development instance is then taken to build the project on a new VM instance where product owners have to look over the functionality. Once the product owner satisfied with the new/changed functionality of the software,the same code is used to build on an additional server instance which does an integration test. This is not necessarily a sequential procedure. The building, testing, and deployment of code to the product owner VM and the integration test VM can be done in parallel. This integration test instance is as close to the production environment as possible. These steps are all happening automatically but need a manual trigger to start because of required confirmations from humans. These steps form a single pipeline that is called development pipeline where a graphic representation can be seen in Figure 3.2. If all these steps considered to be okay the deployment pipeline is triggered manually. The deployment pipeline stores the new image as well as the used code and test results in specific revision tools. It also requests the last approval from a manager. These steps form a release pipeline which is graphically represented in Figure 3.3. After the last approval is given the deployment release pipeline gets manually triggered. This last pipeline pushes the previously stored image to the production environment. This last pipeline is shown in Figure 3.4.

3.10.2 Implementation of Methods and Techniques Proposed by Literature

Git, AWS, Jenkins: Are these the used Tools?

These are exactly the tools that are used. Additionally a cloud platform for Docker container called Open Shift is used together with some in-house tools for storing code, test results and images for a long time. Static code analysis is done by SonarQube.

Is continuous deployment the desired target?

Before the current project that the expert is working on was started, it was decided on a management level that new projects are developed with agile development techniques. The current state is a Continuous Delivery Pipeline in which several steps require

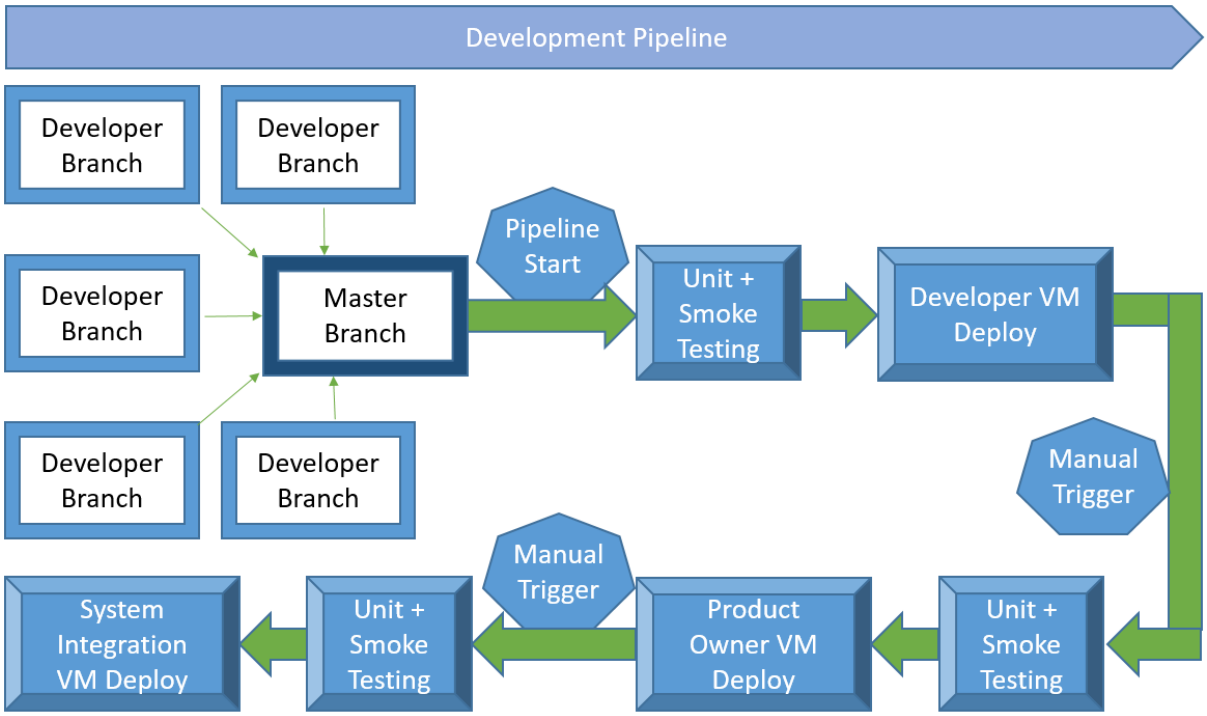

Figure 3.2: Development Pipeline at Company Alpha

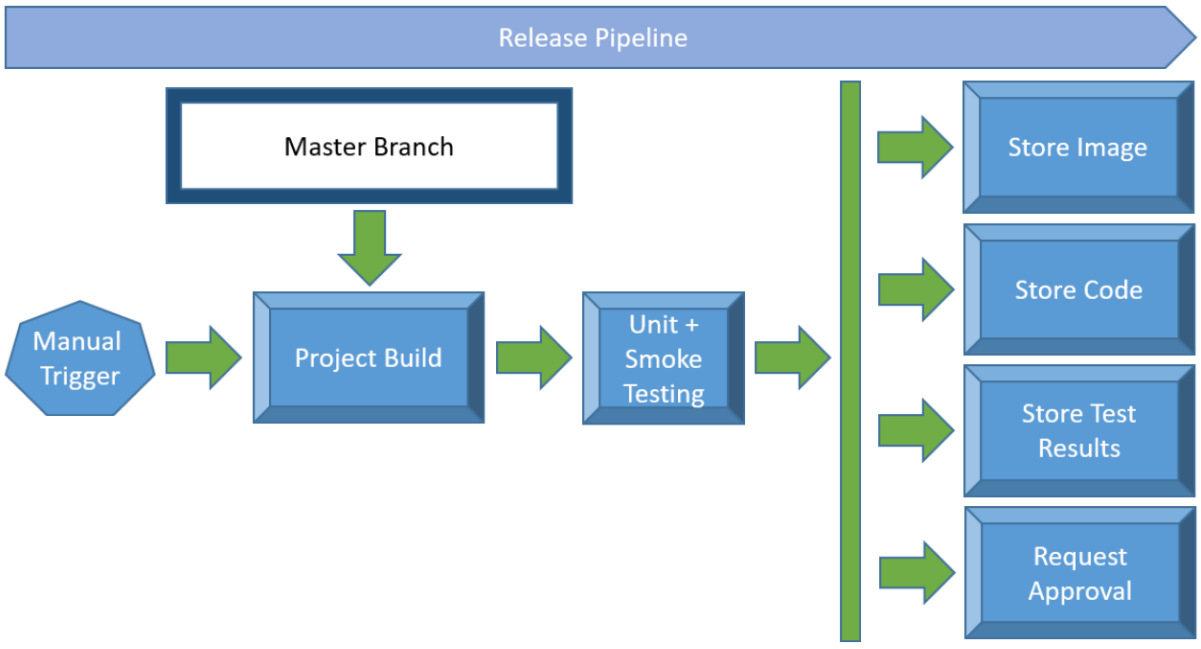

Figure 3.3: Release Pipeline at Company Alpha

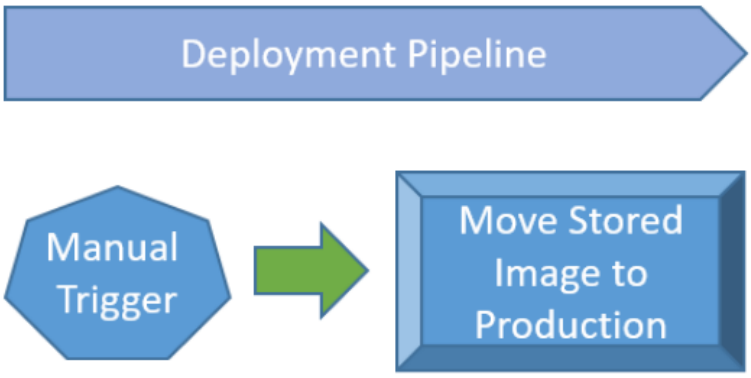

Figure 3.4: Deployment Pipeline at Company Alpha

human admission to allow the continuing of the release process. Are all required people accessible, the process of deploying new software to production takes about 15 minutes. The steps of human oversight however would not be possible inside a continuous deployment pipeline. On paper the target of continuous deployment pipelines was explicitly stated and is still existing. The team that develops the software would also prefer to use a continuous deployment approach instead of a continuous delivery approach. The expert however doubts that the cultural change, that would be required on a management level to transition towards a continuous deployment pipeline, is coming.

How serious is the security in a deployment pipeline Taken?

Multiple security steps are implemented. The Jenkins instance is only accessible through the web UI. The server on which the Jenkins instance is running is hosted by a different department in the company. This department is centralized and deals only with providing other departments with Jenkins instances. Is a new instance required the Jenkins department hosts a new one (automatically) and sends the web address to the team which requires the new Jenkins instance. The newly hosted Jenkins instance is only accessible while the person requesting access is working from inside the intranet of

the company. To gain access to the intranet, a company account has to exist. This company account has to be requested and has password several checks that prohibit simple passwords for the company account. Jenkins accounts for developers are also made. These Jenkins accounts do not have administrative rights. Settings like the global configuration file are already preconfigured by the Jenkins department. The communication between Jenkins and the test/production environments is encrypted and maintained by the centralized Jenkins department. Since the pipeline is stored as a Jenkins file inside the project source code for which it is required, the SCM is also secured in a specific manner. The company uses its own GitHub which is also only accessible through the intranet of the company. Additionally a two factor authentication is set for the internal GitHub. Each developer has access right to the test environments. The production environment is however also secured with another account (Name and Password). This account is only for the production environment. The account authentications are know to only two specially chosen developers.

DevOps - Definition, how and why is implementation a desired target

Similar to the described case in Section 3.10.2 it was decided that agile methods are used. Therefore the usage of DevOps is existing on paper. It is however mainly used because DevOps is named in the context of agile software development. One person is placed inside the developing team which is only doing operational tasks. The team of developers however define them selfs all as DevOps by the following Definition:

Everybody needs to be able to develop the application as well as having the capability to run the application or keep the application running in the production environment.

3.10.3 Verification or Disproval of Suspected Influence Factors

Are slips while inserting considered a threat?

Slips are existing but not considered a Threat. Building or modifying a Pipeline happens based on Trail and Error in a testing environment. During this time slips do happen but the modification and testing is happening on a different branch which gets merged back into the master when the new modifications/pipeline are working on the temporary branch. Live deploys happen in a canary testing manner which is described in Section 2.6.4.

DevOps - What are the remaining differences between development and operation and how is each department trained?

The development team and the one operator are sitting in the same location attending the same meetings and discussing the progress of the software together. The remaining differences are that the operator is only doing operational tasks while the developers are able to operate the application but stick mostly to developing the application.

Who has access to a pipeline and its configuration?

Since the pipelines are configured in Jenkins files which are stored inside the projects repository, it is possible for each developer that can commit to a repository, to change the pipeline and its behavior. The administrative settings of the used Jenkins instance are accessible by three people in the developing team. These three people deal more than the average developer with pipelines. Those pipeline administrators can for example add new users to the Jenkins instance. The administrative users however have also only access to the web UI and can't connect to the underling virtual machine that runs the Jenkins server. Access to this VM is only given to the Jenkins department that was already mentioned in Section 3.10.2. The expert added the comment "even we administrators have relatively few access rights. In the end the Jenkins department decides what runs on those virtual machines".

How many different pipelines exist and what is the difference between those?

The development pipeline is one pipeline which is responsible for each step and can be seen in Figure 3.2. The displayed stages are a conditional executions depending on where the pipeline is at. To simplify here is an example: The pipeline is triggered and choosable parameter are set depending on what is required. The pipeline is building and deploying on the corresponding platform (Developer VM, Product Owner VM or System integration VM). Conditional code is executed that corresponds to the current task. Is the pipeline triggered again with different parameters the same pipeline is executed with other conditional code to execute the currently parameterized pipeline to deploy to the selected test environment. The other pipelines that are used are to prepare the deployment process (Figure 3.3) and to do the actual deployment (Figure 3.4) in production. The team uses these only three pipelines to deploy code. One big pipeline for each branch/task with different conditions and two pipelines that are used to deploy the code to production.

Are there executing conditions in a pipeline?

Yes. Since there is only one pipeline for multiple different branches the pipeline is no longer just a couple of lines long. The pipeline consists of a shared library, ten different classes, about 900 lines of code and is responsible for each branch in each of the three testing environments (seen in Figure 3.2). It is no longer a sequential procedure of steps. The pipeline has conditions, loops, exceptions and everything that is used in a regular (complex) software system. Side effects have turned up in the past, but strict separation of functions and communication between team members could resolve those. Library functions (e.g. downloadFilesFromRepository) are heavily parameterized so that they can be used by everyone who is using the pipeline.

How many configuration interfaces impact the behavior of the pipeline?

The tools that can potentially be integrated in a pipeline are all managed centralized. These tools (e.g. Jenkins, SonarQube, in house custom Repositories) are therefore already configured when they are requested. With the exception of the Jenkins administrative team members, there are no configuration interfaces that can be interacted with. The pipelines them selfs, which are configured through Jenkins file can be changed. To run a pipeline the branch and a test environment has to be selected. this is done with two drop down menus which displays the name of the branch and the test environment as strings. The answer to this question is therefore only two interfaces from which one is regularly used and the other is only accessible by a limited amount of people.

How many configuration values impact the behavior of a pipeline and how dangerous are those?

The regularly used configuration interface has three different test environments as seen in Figure 3.2. The number of branches that can be deployed is summed up to eleven. The drop down menu that defines the test environment is hard coded. The second menu which displays the different branches is generated procedurally, depending on the branches existing within the repository. The regularly used interface has therefore only 14 values which are deploying code to testing environments. A wrongful selection deploys only the wrong code to testing, which can be fixed by running the pipeline again with the right configuration values.

How often does a pipeline change?

On average the pipeline changes every two weeks. The trigger for a change event is sometimes externally. An example would be that a used pipeline tool is changed or added. The other, much more frequent, reason for change is the self optimizing demand that the team members have by them selfs. The members browse news and new features that can used to improve the pipeline. They are not yet completely satisfied with the state that the pipeline is in. An example of such an optimization, that has still to come, is that if a new build is triggered and an old build is still building, the old building gets stopped and only the new one gets build.

How much time is assigned to the task of changing the configuration of a pipeline?

Big centralized changes like updates or the required usage of an in house pipeline tool get scheduled like any feature of the developed software. These tasks are large and can take a lot of time. The last tool that had to be used in the pipeline (because of company policy) took the interviewed expert 3 weeks to implement correctly into the pipeline. Other features that are just nice to have are tracked as minor tasks or not at all.

How familiar is a pipeline maintainer to the tools that are used in the pipeline and how often do the maintainers change?

Besides Jenkins plug-ins the tools that can be used in the pipeline are tools that are managed centralized. Additional tooling is not possible since access to the virtual machines is only partially existent to maintainers of the pipeline. The maintainers are therefore not able to change/configure pipeline tools. All development team members are considered to be pipeline maintainers. Therefore the maintainers change every time the development team grows or shrinks in the size of people.

How many people change a pipeline on a regular basis and how closely do they work together?

There are a core of three people that mainly deal with pipelines. These three work closely together by communication about pipelines and changes on a regular basis. A scheduled task/story of changing a pipeline is however only worked on by one person at a time. Every other member of the development team should have the capability to change the used pipelines. Are all three core developers unavailable (e.g. sick, ill, vacation, conference, ...) other people of the team are supposed to change the pipeline.

These other people are then forced if it is necessary to make a change to one of the pipelines. If changes are made to pipelines, those get reviewed with two other people (mostly the core pipeline maintainers) to present the made changes.

How many times do unintended configuration changes originate from humans/machines?

On source for unintended configuration changes are technical changes like updates. Besides updates, at the beginning of the project there where some problems in which changes where not tested for each feasible case. As a result the pipelines sometimes did not work in any given case.

How does testing the pipeline work?

Testing only works on a trail and error basis. Testing frameworks for pipelines are according to the expert just now showing up. The expert states, that at this time, he knows of only one framework which is not developed by a person in private. Because of current deadlines the application of a testing framework is fading into the background. The application of a testing framework has a high priority inside the team, but current deadlines are more important. An IDE for pipelines is also not used/existing yet. The testing of the pipeline is considered to be one of the biggest weakness of the current project.

3.10.4 Discovery of Additional Influence Factors, Outages and Priority

Do known outages exist in regard to unintended configuration changes in the deployment pipeline?

No. If changes where made that where not sufficiently tested it was occurring at the start of the project, that a pipeline could fail sometimes. Unintended changes however did not occur. Mostly because pipelines only change by changing the configuration file. Input parameters are given by drop down menus. Configuration interfaces to tools are blocked or not accessible at all.

How often are pipeline connected outages visible to clients of the environment produced by the pipeline?

Production environment outages produced by a pipeline are not existing. Even if there would be a problem with the pipeline the user/client would not notice. This is because of so called blue-green-deployment. Blue-green deployment (or canary release, canary testing) is explicitly described in Section 2.6.4.

Target of the pipeline?

The software that is developed is a business to customer website in which specific customers can inform, enroll and managed in specific offers of the company. The entire project runs through pipelines. Front end and back end are both running through the same pipeline.

Severity of an outage?

The project is still in its starting phase. This means that currently only one hundred internal users use some basic functions of the system. Currently an outage would not be dramatic at all. If the system is one day in production, the development team is responsible for the software during the day. During night times there are other maintain departments which would, in case of an outage, handle the situation temporarily by opening error sites or similar quick fixes. There are no people of the development team on call during the night.

Recovery time from an outage connected to unintended changes in a deployment pipeline?

Since there are no outages in connection to unintended configuration changes, there are no recovery times yet. The fixing of a bug in production usually takes about half a day to a day.

3.10.5 Discovery of Implemented Prevention Techniques

What extends the pipeline to prohibit unintended configuration changes?

There are no specific extensions to prohibit unintended configuration changes. Pipelines are parameterized by drop down menus for different settings. This certainly helps to prevent simple errors but it was made for convenience and not to prevent errors. The so called green-blue testing ensures that no faulty software is deployed to the major public. It was however implemented to ensure quality of the software that runs through a pipeline and not test the pipeline in any way. It has the positive side effect that if a pipeline crashed during the building process, the production environment is still running and unaffected from the change.

Lack of automation prohibiting deployment pipelines?

The Expert states that there are no Technological boundaries to transition towards a Continuous Deployment Pipeline. Modifying the current pipelines so that the deployment process is done continuously would simply require the removal of all manual triggers and the merging of the three current pipelines that can be seen in previously mentioned Figures 3.2, 3.3, 3.4 to be merged into one. This process of merging the pipelines would take approximately two days at most.

3.10.6 Case Study Summary at Company Alpha

The company is using a state of the art delivery pipeline. The company distributes a lot of resources towards building, maintaining and tooling used by the pipelines. The centralized management of pipelines tools reduces the workload on developers that are building and modifying pipelines. It also prevents production of errors in pipelines since the used tools are managed by a centralized team. All used tools are state of the art and reflect current technology. The observed team is constantly looking into new technologies and practices to improve the state of the current pipeline. The company has some cultural processes that struggle with adapting new agile techniques like the fast deployment that comes with continuous delivery/deployment. These cultural processes don't seem to change in the near future. They seem to be the major roadblock in transitioning towards continuous deployment methods. Unintended configuration changes are explicitly not considered a problem. The amount of resources (e.g., development time, congresses, technological extensions) distributed towards pipelining seems to be enough to ensure that the deployment process, which is realized with continuous delivery pipelines, is not failing. The technological extension of blue-green deployment entirely prevents

the undesired effect of deployment pipelines on the live environment, no matter what problems within a pipeline exist. This includes unintended configuration changes. In the end it is important to note that the case study was done on a single project in the company. Not all projects in the company are as advanced as the observed project is. The interviewed expert stated that even in other new projects that are developed with agile development techniques in mind, agile processes are not adapted.

3.11 Observed Incident at Company Alpha

Not during the interview but during the observation time at the company, an incident in context of unintended configuration changes was observed at company Alpha. Most notably in a different project which was not part of the case study.

A demonstration of how the deployment process works was scheduled at eight o'clock in the morning. The software was successfully built by a delivery pipeline. All tests of the delivery pipeline accepted the current build. A Jenkins pipeline that purely deploys this specific service to production was already set up because the service was deployed in the past. This deployment pipeline was manually triggered. At some point during this deployment process, the live environment is not accessible for users until the deployment was successful. Since this specific deployment pipeline is only used for the actual deployment step, it only gets used if a new deploy to production is planned. During the observed deployment process an error of the used Jenkinsfile was responsible for an only partially successful deployment. Due to a change that had been made to the deployment pipeline a block of (groovy) script code was moved from inside a stage into a new stage. The moved script was responsible for using a plug-in which is used to verify the validity of deployed code. Due to the move of the code from one stage to another, a parameterized value was no longer in scope. As a result the value was considered null and the plug-in read null instead of the needed integer. The plug-in crashed. The final result was that the pipeline deployed the built image to production but could not use the validation plug-in due to its crash. The described error was found later on. The incident was not taken as a serious problem because until this point the crashed validation plug-in is a new functionality which is not mandatory to use. The inaccessibility of the service due to deploying a new version is also accepted as a cost of deployment. The inaccessibility took about one minute until the new image was deployed. The deployment process is split in 8 stages. These stages are all sequential executed but some are containing executing conditions. An abstracted version of the control flow diagram of deployment pipeline is displayed in Figure 3.5

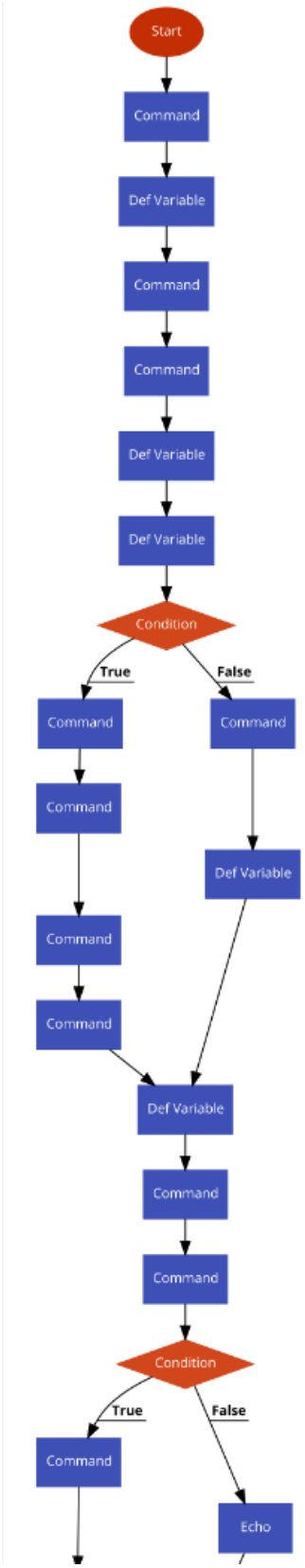

Figure 3.5: Deployment Control Flow 1

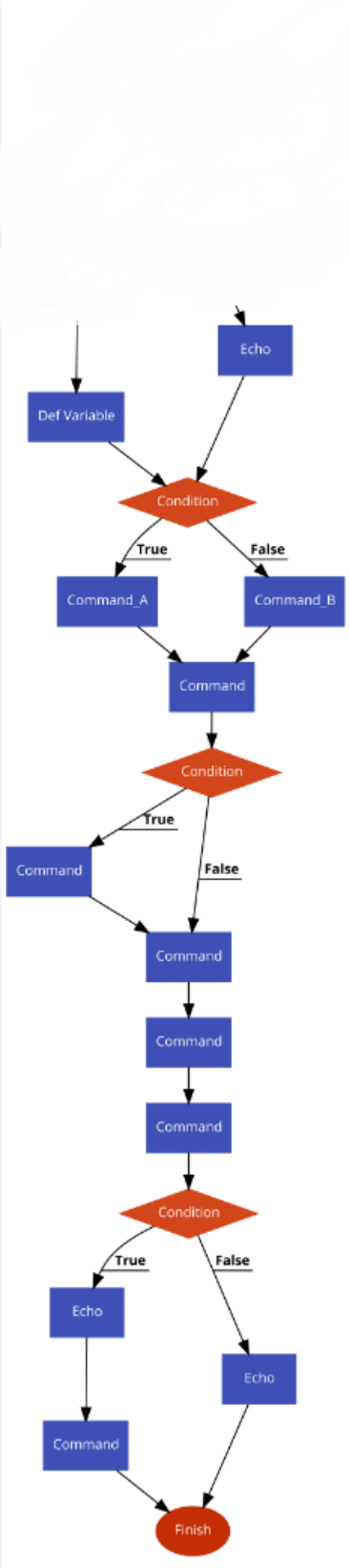

Figure 3.6: Deployment Control Flow 2

3.12 Case Study Execution at Company Beta

This case study was conducted on 12th April 2018 at the company where the deployment pipelines are used.

The aim of this case study was to verify or disprove the suspected influence factors that can potentially alter the behavior of a deployment pipeline in an unintended way. The planned sequence on how the case study was planned to be executed can be seen in Section 3.2.3. A deviation from the planned sequence was done at the beginning. The first two questions were designed to build a relaxed and secure atmosphere on which the important part of the interview could be conducted. This part was highly extended. It featured not only some background information about the expert and how long he has been working with the deployment of software. It also contained general information about the history and development of the deployment process at the company. This helped the researcher and the interviewed expert to get on common ground when talking about the deployment process and the implemented pipelines. This extended intro was the adapted for further case studies.

An additional positive side effect was that since the expert was talking so freely about the topic that he was extremely motivated to provide information about the implemented deployment processes and the usage of pipelines. This extension however was unstructured and did not contain information specifically about unintended changes in continuous deployment pipelines.

One important addition to the set of questions should be mentioned here. Examined pipelines that are part of the a deployment process are not necessarily continuous deployment pipelines. A question were the expert specifies to what extend the deployment pipelines contains continuous techniques was added and asked at the beginning of the interview. The interview was done without significant interruption and took 2 hours and 27 minutes to complete. Like planned the interview was recorded and the recording was used to gather the information that is displayed in the following section.

3.13 Answers of Company Beta

The following section contains the answers to the questions that were designed previous to the case study. Conclusions are not drawn in this section because the recommended data triangulation requires more than one source of information. Conclusions will be drawn later on in Chapter 4 after all case studies were conducted. For qualitative data interpretation [RH08] all answers from conducted case studies will be gathered to see if there are reoccurring influence factors that have been detected.

3.13.1 Intro and General Information

How long has the expert been working with the pipeline?

The Expert has been working with continuous integration for six years. Since the beginning the company has used Jenkins to realize a continuous integration pipeline. The expert is the only one working with the pipeline and he build it from ground up by himself.

How did the expert encounter the pipeline the first time?

Before a Jenkins pipeline was build, automated test scripts that had to be started manually were used. At some point the team manager heard about continuous integration and a Jenkins pipeline was build. The pipeline was extended over the years. In the beginning it was a tool that was used on the side. Because the time effort for automated testing with Jenkins grew over time, the expert is now head of a testing team that overlooks the automation of the pipeline. Tasks for building or changing a pipeline now get scheduled in the used time management tool.

What kind of continuous pipeline is used for the deployment of software (continuous integration, delivery or deployment)?

The used pipeline is a continuous integration pipeline. Building a complete package (multiple software tools and documents) that can then be deployed to a customer is still done manually. The process of building such a complete package took about 3 weeks before continuous integration was implemented. Now days the delivery process takes about three days.

How does the entire deployment process usually look like?

During development a stable trunk is maintained in the repository. Developers branch the repository and then develop a feature. At any point during the development a developer can start the automated tests on his branch. After a Developer is finished building his feature he can then merge the branch into the trunk. The trunk does a nightly build where all existing automated tests are executed. The time that all unit test need sums up to about one hour. Unit tests are executed in a hardware duplicate of a shipped product. Is it required to release a new version of the build software a chain of manual events is worked on. A set of release tests are triggered manually to the

current trunk. About 3 hours later the release tests are finished. Almost always during those release tests at least one bug appears and has to be fixed. Are all bugs removed a final image of the software is made. Documents about new features are then updated (partially automated). A package of the image and the updated documents is composed manually. This is then tested by manually simulating customer behavior. Is this final test also successful people in the hardware production department get notified (manually) that a new software is available. The software gets then send by external hardware to the production department were it can be used. This process of releasing new software is done mostly by hand and takes about 3 days to finish.

3.13.2 Implementation of Methods and Techniques Proposed by Literature

Git, AWS, Jenkins: Are these the used tools?

Used tools are Jenkins as an continuous integration server and Subversion as as a source control management system. Jenkins was the first continuous integration tool that was used at the company. There was no need to replace it since it is satisfying the needs of the company. Subversion is used for similar reasons. Since the source control management system was integrated into the development team it was satisfying the need of the users of the system and there was no need to transition towards another system. In addition to the missing need of using another source control management system (e.g. GIT), the company has still some legacy code in an older source control management system called CVS (Concurrent Versions System). Management and team members agree that using a third source control management system would not be beneficial. Even the idea of replacing a working tool with a new one, that has then be learned, is rejected by the members of the team.

Is continuous deployment the desired target?

Automating the process of delivering software (building a package of software and documentation that is then send to production) is an ideal target that the company wants to be eventually. It is however a task that is not scheduled yet. No timetable exists when this should be done. Building a complete deployment pipeline is not a desired target. It is not seen as useful because the time spans in which the company operates are quite large (e.g. 30 years of product life cycle of a hardware part). Using a couple of days to deploy new code is not seen as a problem. The fear of having bugs in the

system after automated tests are past is to immense. This is especially because fixing a bug would require hardware callbacks that are costly.

How serious is the security in a deployment pipeline taken?

Security was taken into consideration by building the pipeline. The continuous integration server on which the pipeline is build is physically standing in a secure environment. Access rights are coupled to the windows active directory. The team that uses the pipeline has only limited administration rights on the server. The server is maintained by the operations department. Each developer that has to write code, that then uses the pipeline, is granted limited access to the continuous integration server.

DevOps - Definition, how and why is implementation a desired target?

By DevOps the interviewed experts understands "the connecting between development and everything that has to do with IT". Development defines the writing of code and the programming of (in his example) hardware. The term operations can, again according to the interviewed expert, be extended to IT-operations. This means operations provides services like for example the installation of an compiler, an Integrated development environment or an source control management system. The definition of DevOps would be the connection of both departments that handle these to services. Implementing DevOps is not an desired target. To quote the broadest definition of what DevOps means is " bridging the gap between development and operations" [JAPT16]. According to this definition, the operators of the software are more likely customers that use the bought hardware products that runs the firmware that is running inside the hardware.

3.13.3 Verification or Disproval of Suspected Influence Factors

Are slips while inserting considered a thread?

Slips are explicitly considered a thread. Previous to this interview, there were multiple incidences were a faulty pipeline was the result of a configuration change. The configuration change was not made properly and resulted in an undesired behavior of the pipeline. According to the definition in Chapter 2 Section 2.2 the pipeline contained an unintended configuration change. The result was that the next nightly build failed because of the wrongful configuration and not because of the code that uses the pipeline. Since this was a reoccurring problem the company transitioned from manually changing a pipeline to configuring it with script files. Those are then stored in the repository.

The company effectively uses code as infrastructure (see Section 2.6.3) to undo/fix changes that are caused by unintended configuration changes. Using scripts to configure a pipeline also reduced the number of times when a slip caused an error and increased the speed of configuring a pipeline. It however did not completely eradicate them.

DevOps - What are the remaining differences between development and operation and how is each department trained?

Referring to the answer of "DevOps - How and Why is Implementation a Desired Target?". DevOps is not implemented and therefore this question has no answer (it was however part of the interview).

Who has access to a pipeline and its configuration?

Since code is stored in the repository, everyone who has access to the repository could also change the behavior of the pipeline by modifying the configuration. It is however the case that each person who has access to the repository also has access to the continuous integration server. An exception is made to the installation directory of Jenkins and the administration rights to the continuous integration server. To those only the IT department and the team leader have the ability to make changes after inserting an additional password.

How many different pipelines exist and what is the difference between those?

There exist about 60 pipelines. For a single product there are about five pipelines that are only differentiated by a name. Besides a different name these pipelines are almost exactly the same. It is not excluded to mistakenly change the wrong pipeline.

Are there executing conditions in a pipeline?

No. There are no conditions existing in a pipeline. If a pipeline fails at a certain point because of a failure it is stopped. However if a pipeline is well functioning and all manual pipeline tests are successful the procedure of a pipeline is always exactly the same.

How many configuration interfaces impact the behavior of the pipeline?

The answer to this is not clear. The expert counted a lot of possibilities were a configuration interface/file/value could change the behavior of a pipeline. So many however that the expert could not come up with a specific number because it is not clear how many configuration interfaces are existing in additional tools that are part of the existing pipelines. The final answer to this question were "a lot".

How many configuration values impact the behavior of a pipeline and how dangerous are single values?

No specific answer. Because the previous question could not be answered definitively this could also not be answered. Specifically dangerous and non dangerous values exist but most of the configuration values could, if an unintended or intendedly harmful change was made, alter the behavior of the pipeline so that it would not work correctly anymore.

How often does a pipeline change?

After the creation of a pipeline it changes frequently until all required tasks are implemented. This takes about a week to set up a new well functioning pipeline. After this initial phase it is not usual that a pipeline changes.

How much time is assigned to the task of changing the configuration of a pipeline?

In the past building and maintaining Jenkins pipelines was done as a side activity by developing software. Since the automation of integration and possibly delivering continuously takes more and more time, it is now possible to explicitly schedule tasks to change/modify a pipeline. The time gets tracked and it is compared to the past more time available to change a pipeline.

How familiar is a pipeline maintainer to the tools that are used in the pipeline and how often do the maintainers change?

All pipelines are build and maintained by the interviewed expert. Also all additional tools that are used are explored and plugged in by the expert. The maintainer of the pipeline is since the beginning of continuous integration the same person. The interviewed expert

was the only one in the company that dealt with continuous integration and pipeline building.

How many people change a pipeline on a regular basis and how closely do they work together?

Only the interviewed expert.

How many times do unintended configuration changes originate from humans/machines?

The interviewed expert explicitly answered this question with "rather Humans Errors".

How does testing the pipeline work?

Simple syntax checker for paths are in place. Besides that, each pipeline has to be tested manually before it is deployed the the first time. Ideas like having an duplicated Jenkins server where tests are done first have been thrown around but are not implemented. The interviewed expert mentioned that it would be helpful to have a duplicated continuous integration server to preempt errors. It was however questionable if the spent effort would provide a significant payout.

3.13.4 Discovery of Additional influence factors, outages and priority

Do known outages exist in regard to unintended configuration changes in the deployment pipeline?

These outages of the continuous integration server exist. The most common problem for an outage is a misconfiguration by inserting a slip (accidental wrong input e.g. character subsumption). This was also explained in question 3.13.3. Slips are still a problem but they have been reduced by using script files to build the Jenkins configuration. Other existing outages are rare. There was a single incident where the Jenkins plug-in for subversion was not working properly after an update of Jenkins. The result being that the plug-in failed to do a checkout. Jenkins could then not build a project. This meant that a nightly build would fail.

How often are pipeline connected outages visible to clients of the environment produced by the pipeline?

A client does not see any of those incidents/outages. This is because the company only uses a continuous integration technique. The deployment is still done manually and has to go through a longer process.

What kind of software is processed by the pipeline?

Software that is processed by the pipeline is exclusively firmware for the developed hardware products.

Severity of an outage?

Outages would be a major inconvenience for the developers. Despite the pipeline being at a level of continuous integration. New code that is build during a not working pipeline could not be tested automatically. Doing the tests manually would be theoretically possible but not practically. The work Jenkins does automatically takes about 20 minutes compared the expected manual time effort of 3+ hours. As a result code that is developed during an outage does not get tested and it is insecurity about new code is growing among developers.

Recovery time from an outage connected to unintended changes a deployment pipeline?

Fixing slips is estimated to take about 10 minutes after the discovery that such an slip exist. If there is an larger problem such as a not working Jenkins installation this takes about two days because the reset of such an installation is done by another department.

What extends the pipeline to prohibit unintended configuration changes?

There are no specific prevention techniques in place to prevent the outage of a pipeline. Infrastructure as code is used to undo problematic changes as fast as possible.

Testing the pipeline is done to which extend?

This question was scraped due to the similarity to the previous question 3.13.3.

Lack of automation prohibiting deployment pipelines?

Automation is not prohibiting continuous deployment pipelines. The fear of having only unit tested software in production, the mentality of developing new products and lack of a need for building a automated deployment pipeline are the main problems. Since product life cycles are such long time spans the need to deploy new software faster, a process that takes currently about two days, is not seen as a desirable target.

3.13.5 Case Study Summary at Company Beta

The company is processing towards more automated deployment techniques. It is however explicitly said that the final step of continuous deployment is not a desired target. A well functioning pipeline is convenient and improves the development process but outages or errors of the pipeline are already rare. These rare outages do not take long to resolve. The outages of a pipeline are not seen by clients or even other depending departments in the company. A not working pipeline is a inconvenience during development. The usage of infrastructure as code was enough to satisfy the needs of the company in regard to the configuration of a pipeline. The interviewed expert has a far-reaching knowledge of the pipeline. He build it and nobody else is changing it. The human component is therefore largely eliminated to be a factor in non-deliberate faults. Security is also taken into account when designing the pipelines. Usual security measures are taken to prevent malicious changes towards the pipeline (e.g. passwords, rights, ...).

3.14 Evaluation

The evaluation of the case studies is, similar to the planning, done by the recommended procedure from Runeson and Höst [RH08]. The Evaluation however summarizes the results on suspected and discovered influence factors to later on classify and automatically discover/prevent these factors. This is done in Chapter 4.

Chapter 4

Influences and Classification

In this chapter we take the raw data that resulted from the interviews in Chapter 3 and evaluate the results. Based on this evaluation, we build classification systems to classify each of the identified influence factors. The first part of this chapter contains the evaluation of both previously described interviews. In Chapter 3, the case studies were designed, conducted and the raw data was displayed in Sections 3.9 and 3.12. This raw data is analyzed here. Similar to the design, the analysis is also based upon Runeson and Höst [RH08]. The analysis is structured by showing the results of those questions supporting and confirming the assumed influence factors. These suspected influence factors are then classified as identified influence factors in Section 4.3.

Identified influence factors are the foundation for the second part of this chapter: Developing classification systems. The classification systems are capable of taking information about some of the identified influence factors in deployment pipelines and classifying those. These classification systems which are described in Section 4.6 can be used to evaluate and improve any deployment pipeline against unintended configuration changes.

4.1 Data Analysis Technique

Hypothesis generation [RH08] has already been done during research and design in Section 3.2.1. The main hypothesis was that even small configuration changes in automated environments (e.g., pipelines) can have catastrophic outcomes. The hypothesis served as a prerequisite for formulating questions (seen in Section 3.4) and conducting the case study. The corresponding analysis technique for direct case studies is a qualitative data analysis [RH08]. For this exact reason the chosen data analysis technique was qualitative analysis. To confirm the suspected influence factors, the

selected qualitative data analysis technique was hypothesis confirmation. This technique, like stated by the name, tries to confirm or deny the original hypothesis by techniques like data triangulation [RH08] which were used to conduct this case study. The data collection method already covers data triangulation. The conclusions are the identified influence factors that can be seen in listing 4.3. These identified influence factors form the body of knowledge that poses the results of this case study.

4.2 Qualitative Data Analysis

In this Section we show the results that are derived from both raw data sets in Sections 3.9 and 3.12. We only discuss results that we classified as identified influence factors. These factors are shown and are justified on what basis the factor is considered to be an identified influence factor. For the justification, the raw data that was gathered during the interviews is used as reference. Negative results from the case study that or results where the suspicion was not confirmed are ignored. The selected level of formalism is therefore called "editing approach" [RH08]. The results shown during the qualitative analysis are based upon the findings of the case study .

4.3 Identified Influence Factors

Out of all our suspected influence factors we derived five identified influence factors from the raw data set in Sections 3.9 and 3.12. The first two influence factors we identified are the absence of systematic tests and the growing number of conditions that exist in pipelines. We discuss the identification of these influence factors in Section 4.3.1. According to our research, pipelines sometimes contain unknown numbers of configuration interfaces that are affecting the pipeline. We identified these unknown number of configuration interfaces as our second influence factor which we discuss in Section 4.3.2. The third influence factor that we identified in Section 4.3.3 is the faulty insertion of slips while configuring the settings of a pipeline. As our fourth influence factor, we identified the security of pipelines in Section 4.3.4. We discovered that security is a matter that is already handled in some way or another, but there are unseen security problems existing in pipelines. Our last identified influence factor is the current system, how people approach pipeline code. We discuss this last identified influence factor in Section 4.3.5.

4.3.1 Conditions & Systematic Testing

In Sections 3.10.3 and 3.10.3 we discovered that all the examined pipelines are only tested on a trial and error basis.

We also discovered that company Alpha has a set of pipelines which are no longer just simple script files. These pipelines are serving multiple purposes (e.g., different deployment targets, different testing branches). These pipelines are part of one single project which contains logic and commands such as loops, conditions, exceptions, and shared libraries. It was discovered in Section 3.10.3 that instead of giving each branch/sub-team its own pipeline, the different pipelines are implemented using a single scripted file but are logically separated by conditional statements. We consider this to be a threat towards scripted software in general. However, expanding a single sequential script towards complex software containing logic and assumptions could result in dormant faults [ALRL04] that range from malicious intrusions [URS+17] to outdated legacy code. Unintentionally changing a path through a pipeline while another path through the pipeline is changed is likely. Software that is processed by a pipeline is tested in multiple ways. The two conducted case studies revealed multiple testing methods like unit-, smoke-, usability- and integration tests (see Sections 3.10.1 and 3.13.1 and Figure 3.2). According to Myers, Sandler, and Badgett [MSB11] testing software automatically is important. The system itself deploying this software does however not get tested automatically. Information from Section 3.10.3 suggests that more sophisticated pipelines are turning slowly into complex software that is more than just a couple lines of script code.

In the evlauated non-continuous deployment pipelines, a bug is currently not considered to be a big problem (see Section 3.13.3). The result of the used continuous techniques is never bound to a production environment. The deployment pipelines are working on manual triggers and not automatically. If a bug is discovered in such a non-continuous deployment pipeline, a developer is always there to fix it, since the process was triggered by a developer (like it was described in Section 3.11). We highly recommend that bugs are considered to be a bigger problem in continuous deployment practices. We have seen in Section 3.11 that pipelines that are connected to the production environment, can have outages in which the production environment is affected by changes in the pipeline. We therefore consider this influence factor as dangerous because testing is usually the part of software that gets mostly neglected [MSB11]. Additionally, automatically testing pipelines is more relevant once a continuous deployment is practiced. Both interviewed experts stated that, at their current state, the outage of a pipeline is not dramatic at all (see Sections 3.10.4, 3.13.3 and 3.13.4).

4.3.2 Number of Configuration Interfaces

At company Beta, where there are only a limited amount of resources distributed towards pipelines, we discovered slight evidence in Sections 3.13.3 and 3.13.3 that the number of interfaces and values that are affecting a pipeline are getting out of hand. We propose a solution to this problem in Section 5.2. However, to confirm this problem of loosing an overview of all the possibilities that a pipeline can change, it would require a larger dataset then the data set that the two case studies provide. A similar problem in company Alpha could not be found. The centralization of services (described in Section 3.10.3) is resulting in a clear overview of what is accessible and what not.

4.3.3 Insertion Slips

This was suspected to be one of the most influential factors during our research. Company Beta with their continuous integration pipeline did confirm in Sections 3.13.3 to 3.13.4 that insertion slips are an influencing pipeline behavior. Company Alpha however explicitly denied that slips are considered to be a problem. The usage of specific departments for pipeline tools and the building of drop down menus for required String parameters has prevented all input errors (see Section 3.10.3). We would however conclude that this was only made possible because of the amount of resources that were distributed towards building a stable pipeline environment for a large number of projects and development teams. The insertion of slips is a problem which has been successfully handled by company Alpha but not company Beta. Nonetheless it is an influence factor on pipelines. Company Alpha has already handled it in their continuous delivery pipelines while company Beta has still sometimes problems with insertion slips. The problems that company Beta still has are maybe acceptable in a continuous integration pipeline but the problems that their expert described in Section 3.13.3 would be dangerous if they would exist in a continuous deployment pipeline.

4.3.4 Pipeline Security

The security of pipelines is an untypical point in this list of identified influence factors. Most importantly, in both visited companies there is a sense of security at some level. Several security measures are taken by both companies (see Sections 3.10.2 and 3.13.2) to prevent unauthorized access towards their pipelines and the corresponding environments. Also, Bass et al. [BHR+15] indicate that companies be concerned about pipeline security in general. Pipeline security is therefore an identified influence factor which is already addressed by both companies in which the case studies were conducted. How

far the security has to be taken to sufficiently prevent malicious change attempts is a different topic which we briefly address in the improvement development to security threats to pipelines (Section 5.3).

An interesting observation that was made during the interviews is that Jenkins files are currently stored inside the regular repository in which each developer that develops the project has access. According to the experts, both companies made the conscious decision of giving each developer access rights to change the integration/delivery pipeline of the projects in which they are developing. This seems to be a doubtful decision because in both observed projects there are limited amount of people which are considered to be the pipeline experts (See Sections 3.10.3 and 3.13.3). These experts are the ones who regularly change/modify/extend or build new pipelines. Additionally, both companies have enabled the Jenkins authentication with different accounts for each developer. Since Jenkins mainly holds the pipelines, it seems that at some point the security of the pipeline had a higher value. Security was then neglected for the benefits of infrastructure as code and the obstacle of having files inside a repository which are only used by a limited amount of people using the repository. Since security is already taken into consideration by companies that are just using continuous integration/delivery, it seems even more important in a more automated environment such as continuous deployment.

4.3.5 Engaging Pipeline Logic

In both case studies conducted, a common factor when talking about how the experts learned about pipelines was found. In both cases the experts started without any knowledge about pipelines (see Sections 3.10.1 and 3.13.1). Learning how to build and modify pipelines was a task that was learned during the development of the actually used continuous integration/delivery pipelines. According to von Leitner [Lei17b] this is already a bad approach.

Additionally, both experts had the benefit of starting on a green field. The used groovy scripts were new to them but they started with empty files and not with hundreds of lines of legacy groovy scripts from previous pipeline maintainers. While this is not ideal but acceptable in continuous integration/delivery pipelines, it poses a problem in continuous deployment pipelines. If there are development mistakes, errors and uninterpretable code put into a pipeline, this simply means that, at the current state, new commits are not able to be tested automatically. In continuous delivery pipelines this would prohibit the delivery in this exact moment. Continuous deployment pipelines however are also handling the deployment to the production environment [SBZ17]. At some point such a pipeline has to take the old code from the production environment and replace it with

the new code. If an error is made during this stage the production environment would stay offline until the a new deploy is made (see Section 3.11).

Connecting untrained maintainers and a production environment is from our perspective dangerous and has to be addressed at some point during the transition towards continuous deployment pipelines. Especially, if we consider the developments that are made by current pipeline maintainers. All of those advancements have to be learned retrospectively by new maintainers. Is the process of engaging pipelines in continuous deployment pipelines the same as the process in engaging continuous integration/delivery pipelines, we predict that serious unintended changes are going to be made. We therefore identified the lack of pipeline training as an identified influence factor.

To prevent misunderstandings, it has to be said that both experts build them selfs new pipeline dummies to test their changes before copy and pasting them into the actual pipelines. On the one hand this is better than actually code testing in production pipelines, but it would require a real testing environment in continuous deployment pipelines, since the deployment process is significantly more error prone due to the production activity of the server/software. This current state of building a dummy pipeline in Jenkins would be no longer sufficient in continuous deployment pipelines. This relates to Section 4.3.1 where we already mentioned the lack of testing in pipelines in general.

4.4 Reporting the Case Study

following the guidance that was given by Runeson and Höst [RH08], we give a short summary about the report of this case study. Runeson and Höst [RH08] refer to Yin [Yin03] when reporting a case study. Up to this point, the entire thesis is the report of the conducted case study. Each task that is required for a linear-analytical approach was processed. The linear-analytical reporting style is according to Runeson and Höst [RH08] "the most accepted structure" for case study reports. The five stages of linear-analytical reporting are shown in listing 4.4. Each point also contains the information of where it was addressed.

- The problem that this thesis is engaging was shown in Chapter 2 Section 2.1
- Related work was covered in Chapter 2 Section 2.6
- Methods on how to engage unintended configuration changes where discussed in Chapter 3. The chapter explains in great detail how designing, building and conducting the case studies was done.
- Analysis of the data collected in the case studies was done in Section 4.2.

- The conclusion of a case study on hypothesis confirmation would be a boolean response. To enhance this conclusion the identified influence factors that are resulting out of the data analysis are the concluded dangerous entities that have to be handled by the classification system (seen in Section 4.6) and the following improvement approaches.

4.5 Case Study Validity

The following sections and chapters will focus on improving pipelines based on the identified influence factors. The question however remains how valid the discovered results are. We took Wohlin et al. [WRH+12] into account to check the validity of the conducted case study. This Section will go through the provided validity checklist. The checklist mentions four main points of validity which are split into multiple subcategories. We hereby discuss each given sub category and its relevance in context to our conducted case study.

4.5.1 Conclusion Validity

This part of validity is trying to figure out if the case study has the ability to draw correct conclusions from our discovered results. In other words are the conclusions that we made in Section 4.3 valid conclusions between the treatment and the outcome of the conducted case study [WRH+12].

- The low statistical power that the case study is providing is rooted in the fact that the resources of this thesis are limited. A higher statistical power with the given resources would only be able if the chosen data analysis would be quantitative instead of qualitative. Since this is an exploratory case study, the proposed data analysis technique is the qualitative data analysis [RH08] which in turn makes it almost impossible to get a high statistical power from exploratory case studies with the given amount of resources. We therefore consider the low statistical power a threat to validity in our case study.
- Violated assumptions of statistical tests is a possible threat that we can ignore in the validity concern. Due to the fact that no statistical approaches were taken, it is impossible to get faulty conclusions from such tests.
- Fishing for specific conclusions is a threat to conclusion validity that has a high impact in this thesis. The questions (see Section 3.4) that formed the interview all where designed to discover a specific possible threat/error source. We tried to

counteract the fishing problematic by first building a set of raw data that contains the results from both case studies (Seen in Sections 3.9 and 3.12). The raw data sets are the results from both interviews. They contain the results of both interviews without any from of interpretation or conclusion drawing. Drawing conclusions was only started after both data sets were finished. Our countermeasures however do not completely cover the "fishing" threat. We therefore consider fishing for conclusions a valid threat to validity.

- The error rate for a qualitative analysis is non existent since a researcher was conducting the interviews. Each occurred misunderstanding was resolved during the interview. Errors that originate from an insufficient formulated question in the dataset are therefore not existing and can be ignored.
- The reliability of measures is highly problematic. The measured data is collected from human beings which makes it, to a certain degree, subjective data. We tried to counteract possible lies due to insecurity or fear. We designed the case study in a way in which we tried to build an environment and a relationship (See Section 3.2.3) with the interviewed expert so that fears are non existent. It is however impossible to be 100% sure whether an expert was lying or not. In our evaluation of the case studies we talked to another expert to verify whether our measured data and conclusions are accurate. We do however consider the reliability of measures a threat to validity.
- Reliability of treatment implementation is no threat to validity. The treatment (here the interview) was conducted by the same interviewer with the same set of questions (with exception from protocol improvements in Section 3.6 that were proposed by [RH08]) in the same time frame without any pressure. The setting was as similar as possible the same for both interviews.
- Random irrelevancies in the experimental setting are irrelevant in our case. Random irrelevancies do not change the data that we collected and can therefore be dismissed. This is because we did not have an experimental setting.
- Heterogeneity of subjects was not given. Both interviewed experts were chosen based on their experience with with the topic. They both have a comparable knowledge about their pipelines. Since the number of subjects is only two, it is not likely to contain an error either.

4.5.2 Internal Validity

According to Wohlin et al. [WRH+12] we have to differentiate between single- and multi group threats when talking about internal validity of a case study. We conducted two

interviews in different companies. Our case study was therefore conducted in multiple groups. Calling two groups "multiple groups" is for our impression of a valid case study not sufficient. We therefore evaluated our case study based on both, single- and multi group threats to internal validity. Internal validity is trying to verify whether there are influences which alter the results of the case study while the researcher is not knowing about these influences [WRH+12].

- Wohlin et al. [WRH+12] states that the history of the interviewed experts may alter the results gathered from the interview. It may be impossible to exclude that there are events from which we had no knowledge about, but we did our best to gather information in an environment that is as normal as possible. Both interviews were conducted on a normal workday (which was neither Monday nor Friday). The interviews took place shortly before/after lunch and had sufficient time scheduled to conduct them.
- During the design of our case study we explicitly included the process of maturation. We described in Section 3.4.1 that the first set of questions is not necessarily asked to gain information, but bring both interviewer and interviewee on a equal base of understanding. We additionally mention that in both interviews the experts were interested and did not loose interest over time. Maturation is however not considered to be a problem. There are no tasks that can be learned faster or better. The maturation time that is taken at the beginning of the case study is simply to get both interviewee and interviewer on the same page.
- Testing or more accurately including the learning curve in repetition is not a problem since we did not conduct any tests.
- Instrumentation is not considered a problem. We did not hand out anything to the interviewed expert. There are no instruments used besides a recording device and the question protocol. Both were only used by the interviewer and not the interviewed expert.
- Statistical regression is only relevant if the observed subjects (here which pipelines were used to gather information from) are classified in previous statistical studies or experiments. This was not the case and we therefore dismiss this as a possible threat to validity.
- Selection is relevant since pipelines that are observed had to be selected. We already described that while defining the case in Section 3.2.1 we investigated regular pipelines. The raw data of the case studies in Sections 3.9 and 3.12 verify this. We do not think that the selection of using regular pipelines is posing a problem for the validity of our pipelines.

- Mortality is only relevant if there are dropouts from the interviews. We did not have any dropouts and can discard mortality as a problematic validity threat.
- Ambiguity about direction of causal events is not relevant in our case study. We did not investigate what the cause for the discovered problems is. We did just collect the data about possible influence factors and whether they confirm or dispute our suspicion.
- Interactions with selection is according to [WRH+12] the different results that can be gained from different people since they have a different learning speed. The maturity is different. Since we did not give any tasks to the interviewed experts we can disregard this threat to validity.
- Diffusion or imitation of treatments is not considered a problem. Both experts that were interviewed work in different companies and do not know each other. They do not know the other person and can therefore not be influenced by them.
- Compensatory equalization of treatments is irrelevant because we did not give any compensations to experts that we interviewed.
- Compensatory rivalry is also irrelevant since the people contributing to the interviews do not know each other.
- Resentful demoralization is also irrelevant since the people contributing to the interviews do not know each other.

4.5.3 Construct Validity

Construct validity is questioning whether the results or the concepts can be generalized. These threats to validity can originate from the design or social factors [WRH+12]. The design refers to the building of the case study while social threats are concerned with different behaviors that subjects of the case study may offer during its execution.

- Inadequate preoperational explication of constructs is concerned with insufficient description of wanted results. We however did an exploratory case study. We explored what the current state of pipelines is and did not want to get specific results from the case study. We therefore do not consider this threat to validity as a relevant factor.
- Mono-operation bias refers to the usage of single objects that are part of an experiment. We used two different pipelines as investigated objects. More pipelines would be even better but we used more than one, which is eliminating single

occurrences. The fact that each pipeline was only observed through an interview of one expert is more relevant. We however can not exclude the fact that misunderstandings or false information was given.

- Mono-method bias is an even larger threat to validity than the previously mentioned 'Mono-operation bias'. In both inspections of a pipeline we only conducted interviews. It is even suggested by Runeson and Höst [RH08] that interviews are the preferred method for exploratory case studies. The problem of having only a single method is that possible suspected influence factors that could have been discovered through e.g., statistical analysis were not discovered with our setup of interviews.
- Confounding constructs and levels of constructs is concerned with different lengths of experiences an expert has with a pipeline. Although there are different levels of experiences in the conducted interviews, we included this concern in the design of the case study in Section 3.4.1. We discovered that both experts are in a state in which they still have to work on something, but they both were not completely new to the topic. We consider their experiences to be on a comparable level and do not think that this validity threat is given in our case study.
- Interaction of different treatments is discarded as a validity threat because only one single interview was conducted with each expert. They did not participate in any similar interview prior to our conducted one.
- Interaction of testing and treatment is discarded as a problem. We did only gather information on a single point in time. During our interviews the pipelines did not change. Even if the experts are now more aware of the problems that may have shown during the interviews, it would not alter the results of the single interview that we had with the expert.
- Restricted generalizeability across constructs is concerned with having positive effects on a measured results but has negative impacts on other not measured facts. Since our raw data in Sections 3.9 and 3.12 does not compare or measure specific data points we do not see this problem in our data gathering. We explicitly recognize this problem in our solution development in Section 5.4 where we discovered that the influence factor of slips can be reduced by adding additional complexity to the pipeline itself, but this does not change the validity of the collected data or the resulting identified influence factors.
- Hypothesis guessing by the expert would maybe change the information that is given by him. During the design of the case study in Section 3.4 we already covered the fact that behind most questions is a hypothesis that we try to confirm. We also mentioned the fact that we actively hide the hypothesis to the expert to prevent any change in behavior. Due to the fact that we had this threat to validity

already in mind while designing the case study we do not think that it poses a problem in our case study.

- Evaluation apprehension is concerning human behavior. Humans try to look good while being evaluated. The concern is that this 'looking better behavior' is changing the results. For this exact reason we designed our first block of questions. Like explained in Section 3.4.1 the first set of questions is not asked to gain critical information but rather to set up a secure environment. In a secure environment it is less likely for a human to show-off about their work.
- Experimenter expectancies is concerned with the prejudices that the interviewer might have. Unfortunately this thesis is only conducted by on person which does not make it possible to have different people doing the interviews. We tried to counteract the prejudices that the interviewer might have by starting with a raw data collection (Seen in Sections 3.9 and 3.12). In this raw data section we did not do anything in regard to interpreting the data. We had our attempts to prevent prejudices in this case study, but due to the fact that only one person is working on this thesis it is impossible to know for certain if there where no subconscious prejudices altering the behavior of the interviewer during the interview.

4.5.4 External Validity

External validity is concerning all factors that may prevent the results of this case study to be generalized to other pipelines [WRH+12].

- Interaction of selection and treatment is approaching the topic of selecting the target group. Ideally, each group that exist in reality is represented equally. One of the pipelines that we observed is existing in a rather small company, while the other pipeline is existing in a large company (See Sections 3.8.1 and 3.8.2). In both cases we took regular pipelines as our observatory target. This selection could be expanded by other groups such as medium sized companies or one man projects. We however do not think that the selection that we have is posing a validity problem in regard to the selected group.
- Interaction of setting and treatment is targeting the validity of the setup. We did not use any tools in the interview and therefore can ignore this threat to validity.
- Interaction of history and treatment is similar to the validity concern of 'History'. We took a usual workday which is neither Monday nor Friday to conduct our interview. There was more time scheduled to conduct the interviews than needed. We therefore disregard this threat to validity.

4.5.5 Validity conclusion

This Section discussed the checklist of threads to validity that is provided by Wohlin et al. [WRH+12]. We now discuss the validity problems and what this means for conclusions that can be drawn from the thesis. We further discuss the relevance of the upcoming chapters that pose solutions in regard to the discovered identified influence factors. To summarize this Section we list all the threats to validity that pose a threat to our case study here.

- Low statistical power
- Fishing
- Reliability of measures
- Mono-operation bias
- Mono-method bias
- Experimenter expectancies

The most important validity threat is the low statistical power. It is similar to the validity threat of mono-operation bias. The two cases at different companies that we observed is already above average number of cases that a thesis usually contains. To gain a higher statistical power it would be necessary to either increase the number of resources distributed to the topic (e.g., people, timeframe) or transition from an qualitative approach to an quantitative approach. Doing a quantitative approach on an exploratory case study would however contradict the guidelines from Runeson and Höst [RH08] on which this case study was largely based. During our research we found information on a qualitative case study in which 15 data points (different companies where questioned) in Leppänen et al. [LMP+15]. We can use this as a point of reference and assume that approximately 15 companies would be a sufficient statistical power for a qualitative case study.

The validity threats of fishing and experimenter expectancies are closely related. Both are aiming at the interviewer that is conducting the interview. To ensure that the validity of the case study is not compromised we would use multiple different experts to conduct the interviews. Like already explained this can't be done due to the fact that only one person is working on this thesis.

How reliable the measured data is is questionable. To ensure that interview answers are reliable there would have been some sort of control value (e.g., asking experts working on the same pipeline separately the same questions). We did our best to build a safe

environment so that the interviewed expert did not have to fear anything and would give reliable answers in the first place.

Mono-method bias is concerned about the fact that we used only one technique to conduct the case study. Due to the low statistical power it is however questionable if it would be useful to use another method for the second interview. Especially including the fact that the question protocol for the interview was improved after the first interview was conducted.

To summarize we can see that all validity threats are originating from conclusion and construct. Internal and external validity seem to be unproblematic. All the discovered threats to validity could be resolved by using more people, companies and time.

4.6 Classifying Identified Influence Factors

The previously identified influence factors pose a potential threat to continuous deployment pipelines. The reader of this thesis might now ask at what point action has to be taken to reduce or prevent the risk of such an influence factor. We therefore provide a method to classify the identified influence factors. These are however just rough guidelines. In the end, the decision on how dangerous an identified influence factor is, and how much resources are distributed towards protecting a pipeline against such an influence factor, has to be done by pipeline maintainers themselves. We already mentioned in Section 4.3.1 that there is evidence that pipelines are slowly transforming into more complex software projects. The measures that are here proposed are only proposals to ensure a well working pipeline.

4.6.1 Classifying Slip Origins & Configuration Interfaces

Slips and the number of configuration interfaces where both discovered as identified influence factors. As it turns out, these are both closely related. A slip can only be made in a configuration interface. Configuration interfaces can also contain slips while they are changed. We discovered two relevant factors for slips and configuration interfaces.

The first is the number of times a value inside an interface is changed or a slip can occur only if a change is made. This is the only time a change occurs. It is therefore the only time in which a fault can be generated. In case of pipeline outages it is possible that the production environment is effected like we described in Section 2.1. We discovered during our observation in Section 3.11 that this is in fact possible. The worst case

scenario of pipeline outages is that they are affecting the availability of the production environment.

The second factor that plays a role is therefore the availability of the pipeline. These two factors (change frequency and availability-loss aka severity) are closely related to the OWASP Risk Rating Methodology [OWA18]. We therefore based our classification system of slips and interfaces upon this system. OWASP [OWA18] however evaluates the overall risk severity based upon the abstract likelihood and impact. Since this classification system should be a guidance for readers, we specify these. Impact of a pipeline outage is classified into the three categories instead of the provided abstract low medium and high. The three categories are:

- Effects not noticed by users
- Effects noticed by users but still working
- Fatal System/Subsytem outage

The likelihood that an error occurs is the change frequency. We use the availability zones that described the availability of an application in Gray and Reuter [GR92]. The availability zones can also be seen in 4.6.1

- High Availability - 99.999% Available - 5 minutes outage per year allowed
- Available - 99.99% Available - 50 minutes outage per year allowed
- Well Manged - 99.9% Available - 9 hours outage per year allowed
- Managed 99% Available - 90 hours outage per year allowed
- Unmanaged 90% Available - 900 hours outage per year allowed

The availability is however impacted by the MTTR (Mean Time To Repair) of a pipeline and not just the occurrence of an outage. Since outages originate from a change to the configuration we assume, like OWASP [OWA18] proposes, the worst case. The worst case being that every unintended change is resulting in a fatal outage. The resulting calculation for the likelihood of OWASP is shown in 4.6.1

$$ChangesPerYear * MTTRofPipelineInHours < AllowedOutagePerYearInHours$$

If we apply values to the specified cornerstones of the OWASP Risk rating methodology we get the result which is shown in Figure 4.1. Important to mention is that the interface identification can be improved and made more fine grained by considering each value inside an interface as a here described interface. This has the benefit of being more precise since interfaces are just an accumulation of configuration values. On the other hand this drastically increases the effort spent to classify the pipeline.

Fatal System/Subsytem Outage		Low	Medium	High	Critical	Critical
Effects noticed by Users (But Functioning)		None	Low	Medium	High	Critical
Effects invisible to Users		None	None	Low	Medium	High
In Case of Faulty Input		ChangesPerYear * MTTRofPipeline-InHours < 5 min	ChangesPerYear * MTTRofPipeline-InHours < 50 min	ChangesPerYear * MTTRofPipeline-InHours < 9 hours	ChangesPerYear * MTTRofPipelie-InHours < 87 hours	ChangesPerYear * MTTRofPipeline-InHours < 876 hours
	CPY*MTTR < X					

Figure 4.1: Classification System Slips & Interfaces based on Chapter 3, [OWA18] and [GR92]

Using the Classification System for Slips & Interfaces

In this Section we described how the classification system should be used. We describe the usage by an example for which we take the data that has been gathered in the case study at company Alpha. In this example we evaluate the classification of an slip in company Alpha. The expert mentioned in Section 3.10.4 that pipeline errors are not visible to clients. The other needed information is the mean time to repair of the pipeline and the changes per year. It was mentioned in Section 3.10.3 that the pipeline changes every two weeks on average. This would result in a change frequency of 26 changes per year. The mean time to repair , which was described in Section 3.10.4, is about six hours. To use the classification system we multiply the mean time to repair with the number of changes per year and get 156h. We now plug all the gathered information into the classification system in Figure 4.1. The selected row and column cross in a "High" risk cell. This matches the reality that was described in Section 3.10.3 were it was described that the pipeline has to be tested and it is on the list of high priority changes.

4.6.2 Classifying Conditions and Testing Requirements

According to Humble and Farley [HF10] and Myers, Sandler, and Badgett [MSB11] testing software is important. The more test cases exist for a software project the more likely it is to uncover a bug inside the software [MLBK02]. We previously discovered in Section 3.10.3 that some pipelines evolve from single sequential scripts into complex programs. We also discovered in Section 3.13.3 that testing on a trial and error basis can be successful in singular sequential code flows. This classification should now tell

at what point it would be necessary to implement (automated) tests for a pipeline. We did however not find any information related to the topic. In an ideal world there are tests for every software. Literature always assumes that software is automatically tested. The decision on when to implement tests to a project (in this case the pipeline) has to be done by each individual maintainer. We do not have reliable information at what point trial and error testing is no longer sufficient and automated testing has to be implemented. Our suggestion is however to implement the first automated test as soon as a single if-condition is existing. The first condition is transforming simple sequential code that is tested by trial and error, into code in which dormant faults (see Avizienis et al. [ALRL04]) can exist. The outage that we observed in Section 3.11 was caused by a misconfiguration in a pipeline which had a fairly simple control flow (see Figure 3.5). The control flow itself was not the problem, but it shows that bugs can appear even in simple pipelines.

4.6.3 Classifying Security

Research prior to the case study already suggested security may be a problem. This was confirmed during the case studies (see Section 4.3.4). The security aspects in pipelines are however not too different from other digital environments. How to classify security in software projects is a topic which other sources (e.g. Allen et al. [ABE+08], Ullah et al. [URS+17]) investigate in a detailed manner. Security can be improved by adding additional security layers. We discuss in Section 5.3, how pipeline security can be improved, especially by limiting unrestricted access to pipelines by developers. How much is done and how many resources are distributed towards the security is a decision every team has to do on a subjective basis.

4.6.4 Classification of the Engagement of Pipeline Logic

Much like the previous two identified influence factors, we could not develop a hard data classification table on how to classify or rank this influence factor. We however made a list of requirements that have to be fulfilled. If all questions to the list that are seen in Section 4.6.4 are answered with "yes" we would propose to consider the pipeline a problem in context of this influence factor. We propose that the approach on which new pipeline maintainers/developers engage the pipeline is significantly different from the approach that the observed companies made. The manner in which new experts are currently trained was discovered in Sections 3.10.1 and 3.13.1 and is not an acceptable approach in pipelines that meet all the specified criteria. Additionally, we would like to mention that this influence factor is the only one which is not of technical origin but of human capabilities.

- Is there a medium to large amount of legacy pipeline code snippets?
- Is the pipeline containing any conditions, loops, exceptions?
- Is there a practical way to test the pipeline and its behavior without affecting production?
- Is the discussed pipeline affecting the live environment? Deployment pipeline or continuous deployment pipeline?

Chapter 5

Measures to Improve Pipelines

In Chapter 4, we took all suspected influence factors from Chapter 3 and identified the ones that are actually posing problems. In this chapter, we propose techniques and ideas that detect and/or compensate for changes in a pipeline. Importantly to note is that no matter what improvement is used in a pipeline, it is only going to improve the safety or security. It is possible that even when an improvement method is implemented that there are still problems with the pipelines safety/security unrelated to the problem that we built improvements for. Additionally, we mentioned in Section 3.2.2 that we do explicitly ignore certain factors (e.g., Network, Hardware) that are necessarily also affecting pipeline behavior. Each measure contains the information whether it is a preventive measure or a measure to improve recovery. Each improvement also explains how it differentiates unintended and intended changes to pipelines. This differentiation is important because an abstract validation of a specific programs correctness is not possible due to the halting problem [Sch01].

5.1 Improving Pipeline Testing by Conditional Execution

We concluded in Section 4.3.1 that pipelines are still tested manually by trial and error. We hereby propose two solutions on how testing for pipelines could be improved.

5.1.1 Testing Environment

The first solution was already considered by the interviewed expert at company Beta, as was described in Section 3.13.3. Providing an additional testing environment for software that runs through a pipeline, to see if a pipeline produces the required results. The pipeline for this environment has to be the same that is used to deploy to production.

Algorithmus 5.1 Generic Code Example for an Testing Environment

```
procedure PIPELINEFLOW
    generic pipeline code (e.g. compiling)
    generic pipeline code (e.g. testing)
    DEPLOY(Target (e.g. IP-Address): testEnvironment)
    DEPLOY(Target (e.g. IP-Address): ProdEnvironment)
end procedure
```

The different pipeline environment target should be set with a simple boolean input (e.g., check box, same method) to differentiate between production and the testing environment or doing it sequentially. This testing environment is used to discover if the pipeline itself is producing the required results. It is important to use the same pipeline or respectively the same code to deploy to test and production for this process to ensure reliable results. This additional testing environment would serve as a preventive system to improve a pipeline against unintended changes. It can be automated to the point in which it can be set as the step inside a pipeline itself (see Algorithm 5.1) before the deployment process to production is started. The differentiation between unintended and intended changes are only then automatically detected if the pipeline is crashing during its execution. It is additionally possible to block further deployment to production if the pipeline has changed and it has to be turned off by implementing a custom Jenkins plug in. The production blockade would then be resolved by an authorized personnel signing off manually.

5.1.2 Canary Analysis

The second option would be the implementation of canary analysis mentioned in Section 2.6.4. Canary analysis was originally designed to ensure that new software systems that are deployed to production are only seen by a small amount of customers that grow over time. A rollback can be done in seconds since the old software version is still running. As the expert at company Alpha described in Section 3.10.4, they implemented a similar method under the name blue-green deployment. It has the desired effects of preventing software errors to show in production to a larger audience. A positive side effect is that if during the deployment process, or more precisely during the pipeline execution, an error occurs in the wrong moment, the production environment will remain unaffected for most customers. This method would also be defined as a preventive system. Small amounts of customers would be sent to the unavailable software that was produced through the broken pipeline. The bulk of users would not see any changes. Depending on the effects of the wrongful configuration of the pipeline the canary analysis tool

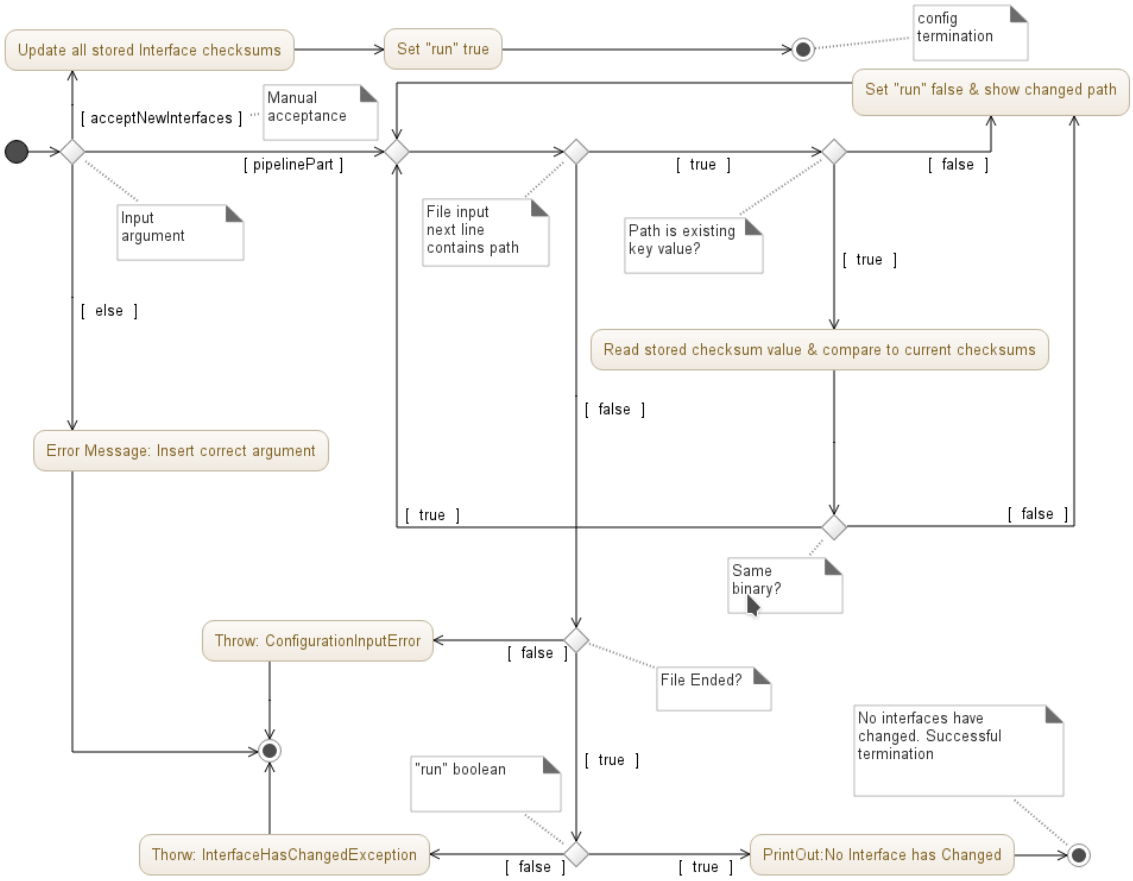

Figure 5.1: Activity Diagram Unintended Interface Change Prevention Tool (UICPT)

(e.g., Kayenta [1]) would see changed behavior in the live software that is used by a small amount of people. The software can decide on custom set violated thresholds that the new software is not correct. The reasons could be simple bugs in the software or other reasons like an insufficient test that was not executed because of a wrongful configured pipeline. The differentiation between intended and unintended changes is happening partially automated depending on the set thresholds (e.g., Smoke tests see Section Figure 3.2).

[1] https://github.com/spinnaker/kayenta

5.2 Safeguarding Pipelines Against Unintended Changes in Configuration Interfaces

This idea behind safeguarding the pipeline is trying to improve the situation (described in Section 4.3.2) about the number of interfaces that are affecting a pipeline. Depending on the size of the pipeline, a large amount of tools is used. Each of those tools has to be configured in one way or another. A configuration is made through a graphical user interface (GUI) or written text inside a configuration file (e.g., DSL). We identified the number of interfaces as an influence factor in Section 4.3.2.

We propose a tool that stores a checksum, hash value or similar information for each interface configuration file. The tool would be plugged into the pipeline. Each time the pipeline is executed the tool checks whether the current checksum has changed. In case of a changed interface, the tool prevents the pipeline from progressing. The tool has then to be executed manually to update the current checksum/hash value. This has the benefit of preventing changes in pipelines that should either not be made or that happen unintentionally (e.g., an update that resets the configuration file).

The benefit of using checksums is that the tool is not using information about environment, programming language or anything else. It is abstract and could be used in any pipeline. Changes to a pipeline tool can result in any wrong behavior, from not executing correct to a broken functionality that would otherwise not registered because the pipeline is automated. Another benefit being that once a tool is plugged into the pipeline, nobody has to remember it about because this tool will remind you if something changed and has to be looked at. The relevant configuration files however have to be inserted manually into the prevention tool.

The idea originates from Bass et al. [BHR+15] where a binary checksum is proposed to see whether a malicious intrusion on a pipeline was done. The proposed internal work flow of such a tool is visualized as an activity diagram in Figure 5.1. In this described form the tool would only function as a preventive tool. Depending on the chosen data storing method it would be possible to extend it towards a tool configuration repository in which changes of all configuration files can be looked up. The differentiation between intended and unintended changes is made by the person who is looking up the changed interface due to the stopping of the pipeline. The downside of this approach is that a manual step is introduced after updating any configuration interface that is part of the pipeline. To some extent automation gets reduced to ensure a safer pipeline. Changes from authorized personal are however not picked up with this prevention technique. The tool also adds more work because each new pipeline tools configuration has to be selected by this tool.

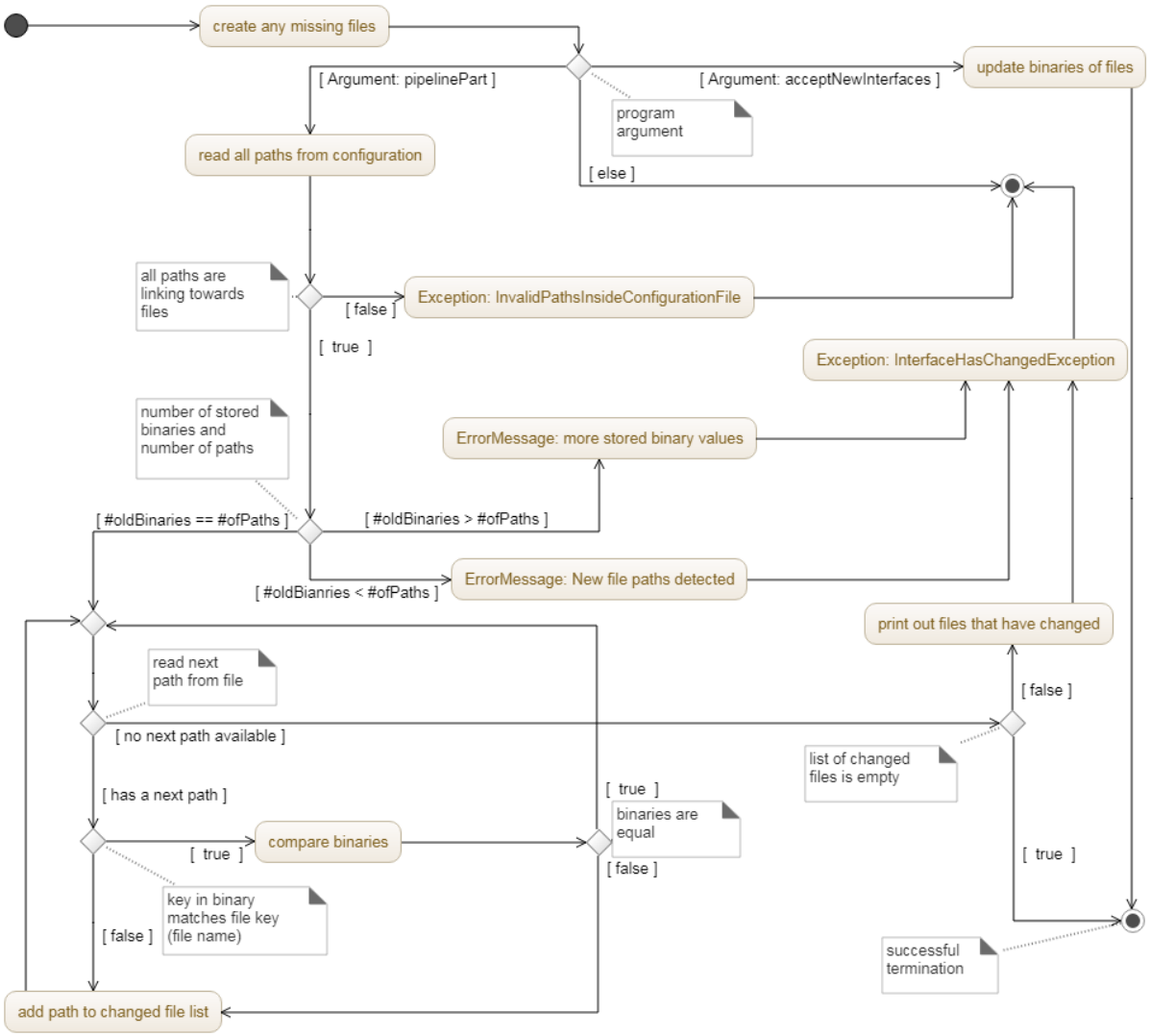

Figure 5.2: Activity Diagram Unintended Interface Change Prevention Tool Implementation (UICPT)

5.2.1 Implementation UICPT

Based on the description in Section 5.2 and Figure 5.1, we implemented a prototype that offers the described functionality. The implementation is capable of storing the binary (complete binary instead of hash/checksum) of any given number of files. Each file is given by a path in a file which only contains paths to interface configuration files. Since this is a prototype, we ignored networking. The tool has two possible run configurations. The first run configuration is the pipeline part in which it is validating if the given number of paths and stored binaries is the same. After this first step of validations it is checking whether each stored interface binary is matching the binary

of the given path. In case that any of the validations are not correct the tool is trowing an exception and is stopping further pipeline executions. The tool is displaying all files that have changed in contrast to the stored ones so that human maintainers can validate whether the change to the configuration was intended or not. Due to the fact that the pipeline is stopped, we propose that the tool is inserted at the beginning of a pipeline before the pipeline could change anything. The second run configuration is updating the stored checksums. This run configuration is one which should be executed manually after a pipeline maintainer added a new configuration file to the tool or a change was detected. After updating the pipeline binaries, the pipeline run configuration executes without throwing exceptions. Should a configuration file which is linked in the tools configuration file change, the tool will start to throw exceptions again until it is updated. This happens not only if a configuration file is changed but also if one is deleted/added. The final version of the code flow that we explained here is visualized in Figure 5.2. It is published as a public repository on Github. The location to the prototype is https://github.com/GJohannes/Unintended-Interface-Change-Prevention-Tool.

5.3 Improving the Security of a Continuous Deployment Pipeline

One possible source of unintended behavior to a deployment pipeline could be a malicious attack. Bass et al. [BHR+15] discovered that concerns about such attacks in deployment pipelines are concerns that already existed prior to this thesis. Changes that are made with bad intentions are an equal damage source to accidental changes. The origin of a faulty production state is usually not cared for. What matters is simply the fact that the live environment is in an undesirable state. However, detecting a malicious attack in a deployment pipeline is equally hard compared to every other digital context. Additionally, just detecting this is not the recommended approach since when a malicious attack was made it is usually too late to fix anything and it mostly is obvious that an attack happened. It is wise to prevent malicious intrusions by taking the usual security measures that every user of digital services should take. This includes the use of appropriate authentication methods like long and complex passwords, adequate encryption (e.g., TLS 1.2) and the role allocation that typical deployment pipeline tools offer (e.g., AWS, Github). We highly recommend to consolidate other sources like for example Ullah et al. [URS+17] that cover this topic as a core feature about their work. Two pipeline-specific measurements that we discovered, are presented in the following:

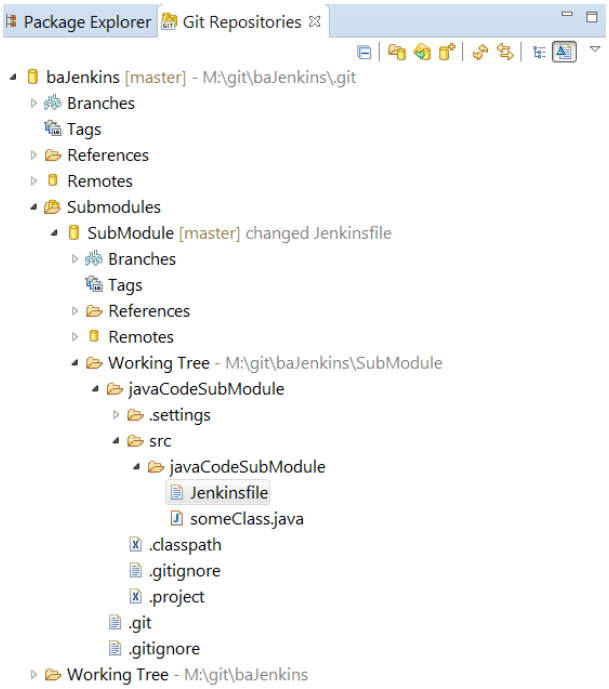

Figure 5.3: EGit Checked out Main Project With Jenkinsfile Inside Submodule

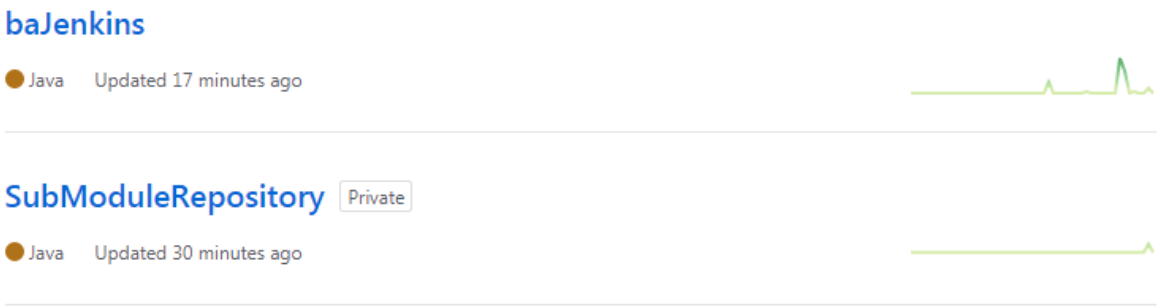

Figure 5.4: Both Individual Repositories

5.3.1 Jenkinsfile

In Section 4.3.4 we described in detail that there seems to be a problem with access rights to the Jenkinsfile. The problem is not just existing for Jenkinsfiles but for the entire Infrastructure as Code concept. In Git, which is one of the most popular SCM tools today, there are only four different permission levels [Git18]. None of which are able to differentiate between contributors who already have write permissions.

A Jenkinsfile is mostly maintained by only a subset of people (see Section 3.10.3). Git currently does not provide out of the box solutions for different authentication levels of different files inside a single repository. This would however be the required solution that

we would propose from a security point of view. People who do not work on a project (e.g., pipeline script) should not have access to it. We found a possible workaround to this problem by using submodules in Git (see Figures 5.3 and 5.4). By adding another repository as a submodule which only contains the Jenkinsfile, it is possible to assign write rights to only a couple of people who edit the pipeline while everyone can still have read access. We vigorously point out that this is only a workaround. Submodules are not designed to work as access right restrictions. Updating them to the latest commit in the parent repository is not as easy as it could be. The path to the Jenkinsfile inside the Jenkins pipeline configuration would then be adapted. An Example how the path would then look like in accordance to the prototype set up in Figures 5.3 and 5.4 would look like this: (MainRepositoryFolder)/SubModule/javaCodeSubModule/src/javaCodeSubModule/(LocationOfJenkinsfile). This method of using submodules would improve security by restricting access, but it would increase the complexity of the procedure to an extent that we suspect it is likely to generate unintended faults by using it. Aside from this workaround we did not find any working alternative to restrict write permissions inside a Git repository.

5.4 Reducing the Number of Slips while Inserting Pipeline Parameters

Research suggested and the case study at company Beta confirmed that slips are potential sources of errors. This section is discussing possible solutions that could reduce the number of slips that are made in context of the configuration of pipelines.

5.4.1 IDE & Compiler

Jenkins files are based upon groovy scripts. Script based programming languages are easier to reuse and result in higher programmer productivity, but the strength of the resulting code is vastly decreased [Ous98]. Especially continuous deployment pipelines are an important part of the development process which would benefit from a strongly typed language. The first step that can improve the typing of a language is the use of an IDE. Scripting languages are more flexible as compiled languages. An Integrated Development Environment (IDE) for scripted languages can never have the same strength as an IDE for a compiled language. An example that shows this would be the type safety that is only resolved during runtime in scripted languages. An IDE for a compiled language can display type violations while this can't be done for a scripted language. It is however possible to implement support like local syntax validation,

highlighting and other basic functions for scripted languages. The IDE support for these basic functions was made in form of plug-ins for Eclipse called 'Jenkins Editor' [2] and INTELIJ called 'Jenkins Job DSL' [3]. The Jenkins Editor was created 2017-09-28 and has since the gotten an average of 800 over downloads per month via the eclipse market place. It offers some of the described basic functionality such as local syntax validation. The next step, which is far more complex, would be the use of an additional compiler, that may not be necessary but could resolve simple syntax errors before the script is executed the first time. Due to the nature of scripting languages and the way Jenkins groovy works this may be hard/impossible to implement. We do not have knowledge of a project that is currently working on this. An IDE or a compiler are working as error prevention techniques which do not validate changes based on the context of the change. Both IDE and compiler are rather for syntax checking and reducing the complexity of the written code due to coloring of variables.

5.4.2 Predefined Parameterization

If there are parameters that should be inserted into the pipeline to run it, we suggest that this is done via selective predefined input methods (e.g. drop-down menus, check boxes, etc.) to prevent the errors that can occur from slips. We propose that if there are any values that have to be inserted by typing, that these values are rather inserted via one of the previously described predefined input methods. We discovered in Sections 3.10.3 and 3.10.3 that company Alpha uses this method for the parameterized string values that define which branch and which tests should be executed. We discovered in Section 3.10.3 that company Alpha does not see slips as a significant problem, mainly because of this parameterization. This parameterization is one of the sources for the increased complexity of pipelines. This is one of the reasons that lead to the assumption that pipelines are transforming from simple scrips into complex software. This method of predefined parameterization would work as an error preventing tool.

Predefined parameterization is a preventative technique which tries to prevent errors from occurring by reducing the number of possible inputs. A differentiation between intended and unintended slips is useless since a slip is always unintended. Preventing the slip is always preventing the undesired behavior.

[2] http://marketplace.eclipse.org/content/jenkins-editor?mpc=true&mpc_state=
[3] https://github.com/jenkinsci/job-dsl-plugin

5.5 Improving the Approach of Pipeline Logic

We mentioned in Section 4.3.5 that we could not provide exact data on how to classify this influence factor. Improving the engagement of pipeline logic is similarly an organizational problem. We are not even sure that it is a problem since we did not encounter any expert in our two case studies that had to deal with all the factors that were listed in Section 4.6.4. We would however propose to minimize this problem by tackling these factors wherever it is possible.

5.5.1 Testing Environment

It is important to provide a separate test environment in which changes do not effect the production in any way. In such a separate testing environment, maintainers can test the changes that where made to the pipeline. This is even more important once a pipeline maintainer/developer engages new custom groovy script. Custom made software projects (e.g., plug-ins) that are developed in-house are more likely to contain errors than large and popular open source projects that are accessible to the public. It is therefore important to test pipeline behavior instead of just reading the documentation about it.

5.5.2 Logic Containments & Affecting Live Environment

These two points are important to notice when trying to rate the stability of the pipeline. It is however not always possible to improve upon this influence factor. In a continuous deployment pipeline it is the desired effect that the production environment is affected by the pipeline. It is possible to try and reduce logic to have a pipeline which is only executed in a sequential flow. Depending on the project and environment this is not always possible. An improvement to these two classification factors is therefore not always possible.

5.5.3 Graphical Pipeline Builder

Jenkins pipelines are currently built by writing (Jenkins) groovy code. Some functions are hidden behind specific syntax commands that have to be learned or looked up (e.g. 'git url : 'https://github.com/GJohannes/baJenkins'' to do a pull request from a repository.) This process is simplified by Jenkins own snippet generator which is seen in Figure 5.5. We know that such a snippet generator is a useful and simple tool for building

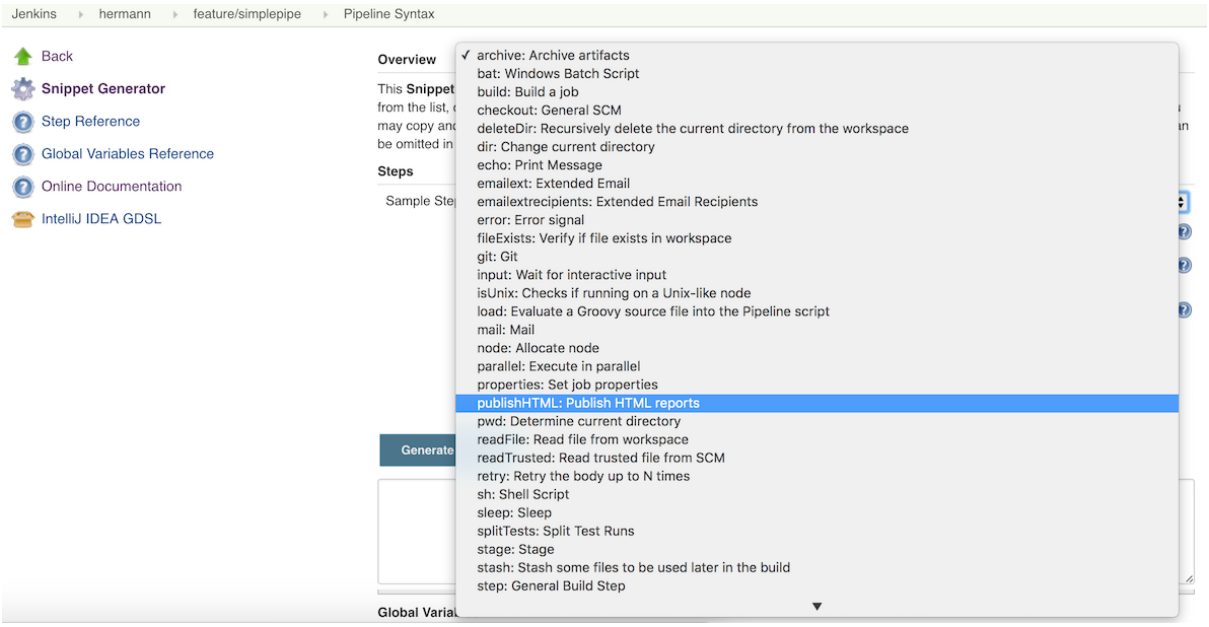

Figure 5.5: Jenkins Snippet Generator

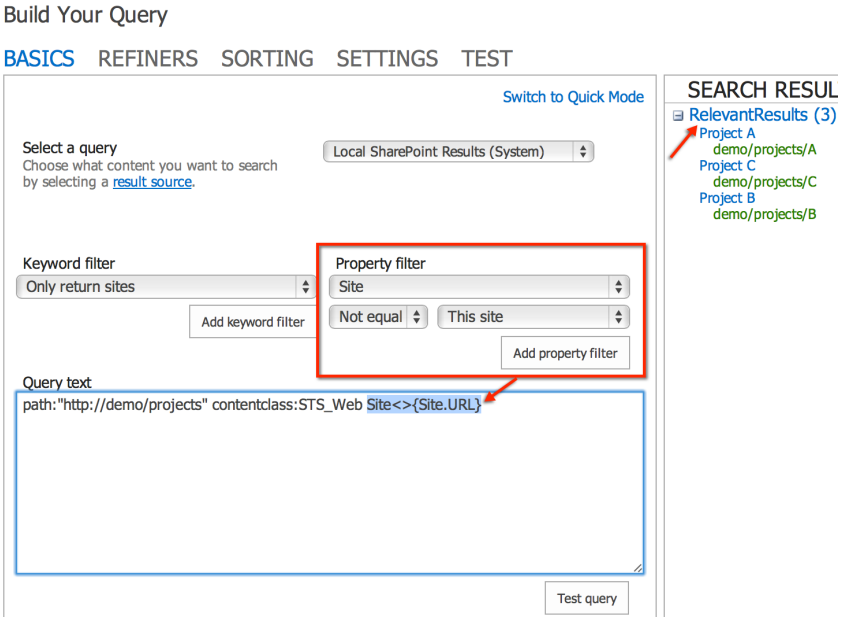

Figure 5.6: SharePoint Query Builder

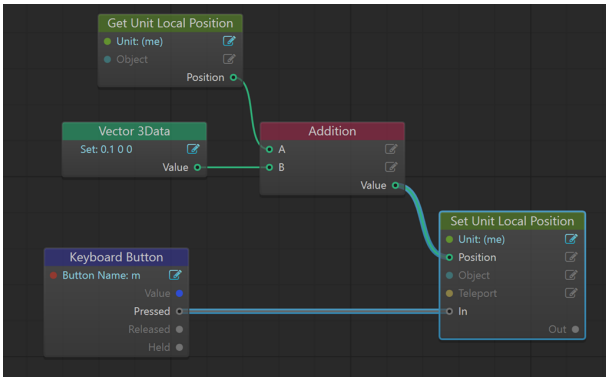

Figure 5.7: Example of coding in a Game Engine

DSL languages because we have seen similar approaches by for example Microsoft that developed a content management system called SharePoint (see Figure 5.6).

However, we would propose to go a step further. We have discovered how complex pipelines are looking and think that such a snippet generator is no longer sufficient, especially since the snippet generator is only helping by building new pipeline parts. Pipelines are large packs that are changing during their time in existence (see Section 3.10.3). Pipelines are used to 'glue' different functionalities together but their current representation exclusively as code are violating two of Shneiderman [Shn10] golden rules of interface design. Rule number two of universal usability and rule number five for error prevention do not comply with the current state. Even simple mistakes like character swaps in a command like "stage" or "setp" are hard to recognize before actually running the pipeline if there is no syntax checker in place. According to [Shn10] rule number five is preventing errors in human computer interfaces as much as possible.

We propose a pipeline builder in which pipeline blocks (e.g., currently existing snippets) can be used via drag and drop to build a the pipeline code. Should the visual representation not satisfy the need of the user it would still be possible to manually adjust the resulting code. Additional functionality like conditions, loops and exceptions could also be built by predefined GUI blocks like it is used in game development engines (see Figure 5.7) or Scratch[4]. Should new custom functionality be implemented they would be built as custom blocks which can then be dragged into the pipeline. The entire process would be simpler and less error prone because the GUI can have additional information like what each block requires as a precondition. It would also be easier to read into existing pipelines because a flow of steps is visually easier represented. Slips like the in Section 3.11 observed incident that had the root cause of an out of scope variable would be non-existent. A graphical pipeline builder would reduce the complexity of pipelines.

[4] https://scratch.mit.edu/

It tries to prevent errors by having the human changing the pipeline make less errors in the first place by reducing the short term memory load of editors [Shn10].

Chapter 6

Evaluation, Conclusion and Future Work

This chapter summarizes the work done in this thesis. The first part in Chapter 6 is an evaluation of the identified influence factors in Section 4.3. We presented them to another third party expert who is independent from the two previously interviewed experts. We used the feedback that was given by the expert to evaluate the results that we concluded from the conducted interviews in Chapter 3. We presented the identified influence factors and took the feedback on whether the expert could confirm them or not. In Section 6.3 we show our conclusion of the case study. We discuss the original motivation in relation to the results that we discovered. We conclude the thesis in Section 6.4 in which we display the future work that we would suggest to be done to reduce the number of unintended configuration changes in continuous deployment pipelines.

6.1 Evaluation of the Identified Influence Factors

The main results of the case study are the identified influence factors. We tried to verify our findings by presenting them to another 3rd party expert. The third party is a pipeline expert at company Alpha. He is a consultant hired by the same company as the first expert that was interviewed at company Alpha. He is working on a different project in a similar environment than the expert that we interviewed in Section 3.9. The interviewed expert is therefore independent from the previous expert that was interviewed in the conducted case study in Chapter 3. We presented the identified influence factors to the expert. The results of this evaluation interview are shown in the following section.

6.1.1 Evaluational Setting

We took what we learned from the first interview in Section 3.6 and started the evaluation interview with an introductory part. During this introduction a relaxed atmosphere was the desired target. After introducing the expert to the topic we presented all our identified influence factors.

6.1.2 Evaluation Execution

The interviewed expert is an external employee that is hired by company Alpha to teach new developers how to develop software. Part of his job is the introduction of pipelines. Amongst other things he helped to build previous continuous delivery pipelines for other projects. The evaluation interview took place on a Monday afternoon. It took about half an hour.

6.1.3 Conditions & Testing

The basis on which we identified this influence factor was discussed in Section 4.3.1. We discovered that pipelines at company Alpha are complex software that is only tested on a trial and error basis. These pipelines include complex logic like conditions, loops, and exceptions. The expert acknowledged the fact that pipelines are growing in size and complexity. He said that the logic that is included in pipelines is however necessary to access the different environments (see Figure 3.2) while using a single pipeline. The problem of having the code base untested was already acknowledged by the expert that was originally interviewed in the case study. The external expert confirmed that they are looking for a testing framework to test the groovy scrips. He did however not see the testing framework as important as the expert that was originally interviewed. He thought that that the different environments which are used for different testing purposes (see Figure 3.2) are also serving as a testing environment for the pipeline. The result being that the used testing environments are affected by a misconfigured pipeline before production can be reached. The production environment should therefore never be affected by a misconfigured pipeline. His view of this influence factor was that the benefits of having logic to differentiate between pipelines outweigh the downsides of the otherwise used copy and paste for building multiple pipelines for similar environments.

6.1.4 Number of Configuration Interfaces

In Section 4.3.2, we discussed why this influence factor is considered to be an identified influence factor. We concluded that some pipelines are at a size from which it is hard to determine what the number of configuration interfaces is, from which the pipeline behavior can be altered. The expert could not confirm this influence factor in his current project at company Alpha. The number of tools that are used in the project in company Alpha are limited by the management. Wherever it is possible the configuration of a tool is put into the Git repository. We mention here that this influence factor was considered to be identified on the basis of the interview at company Beta. We already discussed in Section 4.3.2 that company Alpha seems to successfully have handled this influence factor while company Beta is still struggling with it.

6.1.5 Insertion Slips

The basis on which we consider insertion slips to be an identified influence factor was discussed in Section 4.3.3. Similar to the previous identified influence factor, this identified influence factor was identified based on information from company Beta. The expert however confirmed that insertion slips exist in their current environment at company Alpha. He mentioned a case in which a manual version number input for similar branches had to be made in the pipeline. Some branches have similar names. In one case a wrong name was selected and a wrong version number for the corresponding branch was deployed. We already suspected such a behavior in our original question protocol in Section 3.4.3. The expert and his team made attempts to include predefined parameterization wherever it is possible. A last human effort has to be made in every case since the deployment process is still manually triggered. This last attempt seems to have always exposed to insertion slips.

6.1.6 Pipeline Security

We identified this influence factor in Section 4.3.4. Both companies which are contributing information to this thesis are in a state in which each developer has write access to the project repository. The repository contains not just application code but also the configuration of the continuous pipeline. The authentication that was set in Jenkins for each developer, in which not every developer had write access to the pipeline, was bypassed. The expert told us that he currently does not consider this state to be a problem. In their current state of continuous delivery it is only possible to trigger a deploy by starting a deployment process manually in the graphical user interface of

Jenkins. In case a transition towards continuous deployment would ever be made the expert would not consider the absence of authentication a security problem. He would rather classify the absence of authentication a process change in which it is made clear that every developer has the capability to push to production.

6.1.7 Engaging Pipeline Logic

This last influence factor was identified in Section 4.3.5. We discovered that there is no systematic approach on how new developers start to engage the already existing groovy script that forms the pipeline. The production environment is possibly affected by the pipeline which is modified on a trial and error basis. The expert acknowledged that they do not have an systematic approach on how to train new pipeline maintainers. They all start modifying code on the "production" pipeline code which is later on reviewed before it is pushed to the master branch. Each "code path" for each environment (e.g., testing, production. see Figure 3.2) has the same code base with the differentiation of branches by predefined input parameters. Should a faulty change be made to a pipeline, it would crash in a testing environment and not the production environment. Additionally, the blue green deployment that is used by company Alpha (See Sections 2.6.4 and 3.10.4) is preventing even a pipeline error in the production environment. They currently do not consider the training of new pipeline maintainers to be a problem.

6.2 Discussion of Results

During our research in Section 2.4.2 we already discovered that the final implementation of pipelines is always different. The fact that pipelines are always different results in different influence factors in different pipelines. The last interviewed expert confirmed some of our identified influence factors (e.g., Section 6.1.5) but also denied that some of our identified influence factors are a problem form his point of view (e.g., Section 6.1.6). As a matter of fact there are is definitely room for improvement to prevent unintended configuration changes in pipelines. Each interview revealed at least some problems in the current state of the pipeline. We would therefore conclude that the evaluation interview indicates that influence factors exist, but they are different in each pipeline. Our case study has shown on what basis we identified our influence factors. It is feasible that there are others that we simply did not come across during our research and case study.

Each pipeline that was discussed as a part of the case study (see Chapter 3) was not a continuous deployment pipeline, but in a previous evolutionary state (continuous

integration/delivery). Each pipeline however had influence factors that pose a problem even in the current state. Each and every pipeline should be looked at and examined for influence factors if a transition towards continuous deployment would ever be made.

6.3 Conclusion

The aim of this thesis was to verify or disprove the main hypothesis: Are there unintended configuration changes possibly altering the behavior of continuous deployment pipelines? The hypothesis was supported by the three main research questions that that aimed at discovering and improving the state of pipelines in the context of unintended configuration changes.

The first and most important research question (RQ.1) was to verify whether there are unintended configuration changes in continuous integration/delivery pipelines affecting the development process. RQ.1 was the main reason for the case study which we conducted in Chapter 3. As a result of RQ.1 we derived five influence factors from which unintended configuration changes can potentially originate.

RQ.2 was designed to classify pipelines on their potential to be affected by the identified influence factors. We designed classification systems for each identified influence factor. The classification systems pose a possibility to evaluate custom pipelines on their weaknesses for unintended configuration changes. The classification systems can be seen in Section 4.6.

RQ.3 was the last research question. In RQ.3 we proposed ideas for enhancement measures to prevent the identified influence factors from occurring in continuous deployment pipelines. The improvements that we came up with are presented in Chapter 5.

As our result we conclude that unintended configuration changes are in fact affecting pipeline behavior even in continuous integration/delivery pipelines. The case study however revealed that the closer a company gets to continuous deployment the more off problems are engaged. In our case study we have seen that company Beta is further away from continuous deployment than company Alpha. Company Alpha has a more sophisticated setup which contains less of our suspected/identified influence factors. The identified influence factors that we discovered pose problems that are of rare occurrence. If a company is transitioning their pipeline towards a continuous deployment pipeline, we recommend to consider the identified influence factors that this thesis discovered in Section 4.3. The identified influence factors are the sources of rather rare problems that can potentially occur, but which can result in the worst case scenarios in an outage of the complete production environment. In one sentence the prevention of the identified

influence factor helps to prevent rare pipeline errors in which the worst cases are the outage of the production environment.

6.4 Future Work

For future work in context of unintended configuration changes in continuous deployment pipelines we would like to see the following topics. The first topic that should be addressed is the validity of the case study of this thesis. Our methods are scientifically accurate, but with a statistical power of just two data sources, the case study should be improved in further research. A case study which is based on the results of this case study to remove all validity concerns that we had with our case study (See Section 4.5.5) is necessary before further approaches are taken. We would propose a quantitative data analysis [RH08] to verify our identified influence factors from Section 4.3. The second topic is an evaluation of both the developed classification systems and the proposed enhancements that we developed in Section 4.6 and Chapter 5. The classification systems were only partially evaluated based on the case study that we conducted to build the classification systems. The enhancements that we came up with are sometimes simple to build (e.g., number of configuration interface prevention Section 5.2) and sometimes projects of their own (e.g., a graphical pipeline builder Section 5.5.3).

Appendix

Bibliography

[ABE+08] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, N. R. Mead. *Software security engineering*. Pearson India, 2008 (cit. on p. 83).

[Ace17] J. Acetozi. "Continuous Integration Server." In: *Pro Java Clustering and Scalability*. Springer, 2017, pp. 139–139 (cit. on p. 14).

[ALRL04] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33 (cit. on pp. 27, 28, 33, 35, 69, 83).

[AWS16a] AWS. *Amazon EC2 Service Level Agreement*. https://aws.amazon.com/ec2/sla/. 2016. (Visited on 01/17/2018) (cit. on p. 6).

[AWS16b] AWS. *Amazon Route 53 Service Level Agreement*. https://aws.amazon.com/route53/sla/?nc1=h_ls. 2016. (Visited on 01/17/2018) (cit. on p. 6).

[AWS17] A. C. AWS. *Unintended server shutdown*. 2017. URL: https://aws.amazon.com/de/message/41926/ (visited on 12/18/2017) (cit. on pp. 1, 6).

[AWS18] AWS. *What is AWS*. 2018. URL: https://aws.amazon.com/what-is-aws/?nc1=h_ls (visited on 09/24/2018) (cit. on p. 6).

[Ber14a] C. Bertram. *Injecting Failure at Netflix*. http://www.youtube.com/watch?v=ioXV28GtXeo&t=8m35s. 2014. (Visited on 01/12/2018) (cit. on p. 11).

[Ber14b] C. Bertram. *Injecting Failure at Netflix*. http://www.youtube.com/watch?v=ioXV28GtXeo&t=2m02s. 2014. (Visited on 01/12/2018) (cit. on p. 26).

[BFF+18] S. Brausch, M. Friedenhagen, J. A. Fuerbacher, K. Stutz, Y. Boev. *JobConfigHistory*. https://wiki.jenkins.io/display/JENKINS/JobConfigHistory+Plugin. 2018 (cit. on p. 18).

[BHR+15] L. Bass, R. Holz, P. Rimba, A. B. Tran, L. Zhu. “Securing a Deployment Pipeline.” In: *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. May 2015, pp. 4–7 (cit. on pp. 9, 14, 16, 31, 70, 88, 90).

[BJ08] P. Baxter, S. Jack. “Qualitative case study methodology: Study design and implementation for novice researchers.” In: *The qualitative report* 13.4 (2008), pp. 544–559 (cit. on pp. 23–25, 38, 41).

[Blo18] N. T. Blog. http://www.youtube.com/watch?v=RCc4IrUXr38&t=0m33s. 2018. (Visited on 07/06/2018) (cit. on p. 19).

[Che15] L. Chen. “Continuous Delivery: Huge Benefits, but Challenges Too.” In: *IEEE Software* 32.2 (Mar. 2015), pp. 50–54 (cit. on pp. 9–13).

[Col17] C. Coles. “AWS vs Azure vs Google Cloud Market Share 2017.” In: *Skyhigh[online].[cit. 2017-04-30]. Dostupné z: https://www. skyhighnetworks. com/cloud-security-blog/microsoft-azure-closes-iaas-adoption-gap-with-amazon-aws* (2017) (cit. on p. 11).

[FF06] M. Fowler, M. Foemmel. “Continuous integration.” In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006) (cit. on pp. 1, 8, 9, 11, 14).

[FFB13] D. G. Feitelson, E. Frachtenberg, K. L. Beck. “Development and Deployment at Facebook.” In: *IEEE Internet Computing* 17.4 (July 2013), pp. 8–17. ISSN: 1089-7801 (cit. on pp. 11, 13).

[Fos+17] N. Fosgren et al. “State of DevOps Report.” In: *Puppet and DevOps Research & Assessment. Retrieved June* 10 (2017), p. 2017 (cit. on pp. 1, 16).

[Git18] GitHub. *Repository permission levels for an organization*. 2018. URL: https://help.github.com/articles/repository-permission-levels-for-an-organization/ (visited on 07/18/2018) (cit. on p. 91).

[GR92] J. Gray, A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on pp. 31, 81, 82).

[GRH15a] J. Gmeiner, R. Ramler, J. Haslinger. “Automated testing in the continuous delivery pipeline: A case study of an online company.” In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Apr. 2015, pp. 1–6 (cit. on pp. 12, 14).

[GRH15b] J. Gmeiner, R. Ramler, J. Haslinger. “Automated testing in the continuous delivery pipeline: A case study of an online company.” In: *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE. 2015, pp. 1–6 (cit. on pp. 11–14).

[Han16] T. Hanna. *Comparing CI servers: Jenkins vs. CruiseControl vs. Travis*. 2016. URL: https://jaxenter.com/comparing-vi-servers-jenkins-vs-cruise-control-vs-travis-125426.html (visited on 02/06/2018) (cit. on pp. 11, 14).

[Hea17] A. Heath. "Here's how a low-level Twitter employee was able to deactivate Donald Trump's account." In: *businessinsider* (Nov. 2017) (cit. on p. 17).

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010 (cit. on pp. 16, 17, 82).

[HTH+16] M. Hilton, T. Tunnell, K. Huang, D. Marinov, D. Dig. "Usage, Costs, and Benefits of Continuous Integration in Open-source Projects." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 2016, pp. 426–437 (cit. on p. 14).

[JAPT16] R. Jabbari, N. bin Ali, K. Petersen, B. Tanveer. "What is DevOps?: A Systematic Mapping Study on Definitions and Practices." In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: ACM, 2016, 12:1–12:11 (cit. on pp. 16, 60).

[Jen17] Jenkins. 2017. URL: https://jenkins.io/doc/book/pipeline/ (visited on 01/09/2018) (cit. on p. 11).

[Joh10] R. Johnson. *More Details on Today's Outage*. 2010. URL: https://www.facebook.com/notes/facebook-engineering/more-details-on-todaysoutage/431441338919 (visited on 01/10/2018) (cit. on pp. 7, 10, 17).

[Joh17] S. Johann. "Kief morris on infrastructure as code." In: *IEEE Software* 34.1 (2017), pp. 117–120 (cit. on p. 19).

[KDWH16] G. Kim, P. Debois, J. Willis, J. Humble. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution, 2016 (cit. on pp. 15, 16, 19).

[KUC08] L. Keller, P. Upadhyaya, G. Candea. "ConfErr: A tool for assessing resilience to human configuration errors." In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE. 2008, pp. 157–166 (cit. on pp. 2, 17, 18, 27, 34, 35).

[Lei17a] F. von Leitner. *PatternBuildserver*. 2017. URL: http://www.youtube.com/watch?v=E0_Y53ci9cw&t=9m17s (visited on 01/09/2018) (cit. on pp. 8, 12, 15).

[Lei17b] F. von Leitner. *PatternBuildserver*. 2017. URL: https://www.youtube.com/watch?v=E0_Y53ci9cw&t=30m35s (visited on 08/17/2018) (cit. on p. 71).

[LMP+15] M. Leppänen, S. Mäkinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mäntylä, T. Männistö. "The highways and country roads to continuous deployment." In: *IEEE Software* 32.2 (Mar. 2015), pp. 64–72. ISSN: 0740-7459. DOI: 10.1109/MS.2015.50 (cit. on pp. 1, 12, 14, 30, 31, 79).

[LSKM15] T. Lehtonen, S. Suonsyrjä, T. Kilamo, T. Mikkonen. "Defining metrics for continuous delivery and deployment pipeline." In: *SPLST*. 2015, pp. 16–30 (cit. on pp. 12, 13).

[Mar15a] D. March. http://www.youtube.com/watch?v=RCc4IrUXr38&t=40m20s. 2015. (Visited on 07/06/2018) (cit. on p. 19).

[Mar15b] D. March. http://www.youtube.com/watch?v=RCc4IrUXr38&t=0m33s. 2015. (Visited on 01/30/2018) (cit. on p. 36).

[Mer14] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on p. 11).

[MLBK02] Y. K. Malaiya, M. N. Li, J. M. Bieman, R. Karcich. "Software reliability growth with test coverage." In: *IEEE Transactions on Reliability* 51.4 (Dec. 2002), pp. 420–426. ISSN: 0018-9529. DOI: 10.1109/TR.2002.804489 (cit. on p. 82).

[Mor17] K. Morris. *SE-Radio Episode 268: Kief Morris on Infrastructure as Code*. 2017. URL: http://hwcdn.libsyn.com/p/4/b/9/4b93ba3f492a7c87/SE-Radio-Episode-268-Kief-Morris-on-Infrastructure-as-Code.mp3?c_id=12747187&expiration=1525428168&hwt=b9d781b6baf5387d296fe541377e6481 (visited on 05/04/2018) (cit. on p. 19).

[MSB11] G. J. Myers, C. Sandler, T. Badgett. *The art of software testing*. John Wiley & Sons, 2011 (cit. on pp. 69, 82).

[MSB17] T. Mårtensson, D. Ståhl, J. Bosch. "Exploratory Testing of Large-Scale Systems – Testing in the Continuous Integration and Delivery Pipeline." In: *Product-Focused Software Process Improvement*. Ed. by M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, D. Winkler. Cham: Springer International Publishing, 2017, pp. 368–384. ISBN: 978-3-319-69926-4 (cit. on p. 9).

[ODe10] J. O'Dell. *Facebook Downtime Explained*. http://mashable.com/2010/09/23/facebook-downtime-explained/. 2010. (Visited on 01/22/2018) (cit. on p. 7).

[Ott09] S. Otte. "Version control systems." In: *Computer Systems and Telematics* (2009) (cit. on pp. 11, 13).

[Ous98] J. K. Ousterhout. “Scripting: higher level programming for the 21st Century.” In: *Computer* 31.3 (Mar. 1998), pp. 23–30 (cit. on p. 92).

[OWA18] OWASP. *Risk Rating Methodology*. 2018. URL: https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology (visited on 07/05/2018) (cit. on pp. 81, 82).

[PSV01] D. E. Perry, H. P. Siy, L. G. Votta. “Parallel Changes in Large-scale Software Development: An Observational Case Study.” In: *ACM Trans. Softw. Eng. Methodol.* 10.3 (July 2001), pp. 308–337. ISSN: 1049-331X (cit. on pp. 34, 41).

[RH08] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering.” In: *Empirical Software Engineering* 14.2 (Dec. 2008), p. 131 (cit. on pp. 23–26, 29, 38, 39, 41, 57, 65, 67, 68, 72–74, 77, 79, 104).

[SBZ17] M. Shahin, M. A. Babar, L. Zhu. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices.” In: *IEEE Access* 5 (2017), pp. 3909–3943. ISSN: 2169-3536 (cit. on pp. 1, 5, 9, 10, 12–14, 16, 71).

[Sch01] U. Schöning. *Theoretische Informatik kurzgefasst. Spektrum*. 2001 (cit. on p. 85).

[SDG+16] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, M. Stumm. “Continuous Deployment at Facebook and OANDA.” In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. May 2016, pp. 21–30 (cit. on pp. 1, 5, 11, 15, 31).

[Shn10] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010 (cit. on pp. 96, 97).

[Sma11] J. F. Smart. *Jenkins: The Definitive Guide: Continuous Integration for the Masses*. " O’Reilly Media, Inc.", 2011 (cit. on p. 11).

[Spi05] D. Spinellis. “Version control systems.” In: *IEEE Software* 22.5 (Sept. 2005), pp. 108–109. ISSN: 0740-7459. DOI: 10.1109/MS.2005.140 (cit. on p. 13).

[Sta17a] Statista.com. *Facebooks Quarterly Global Revenue*. https://www.statista.com/statistics/277963/facebooks-quarterly-global-revenue-by-segment/. 2017. (Visited on 01/22/2018) (cit. on p. 7).

[Sta17b] Statista.com. *Number of monthly active Facebook users*. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. 2017. (Visited on 01/22/2018) (cit. on p. 7).

[Tra18] Travis. *Travis Licensing*. 2018. URL: https://travis-ci.com/plans (visited on 02/06/2018) (cit. on p. 14).

[URS+17] F. Ullah, A. J. Raft, M. Shahin, M. Zahedi, M. A. Babar. “Security Support in Continuous Deployment Pipeline.” In: *arXiv preprint arXiv:1703.04277* (2017) (cit. on pp. 12–14, 31, 69, 83, 90).

[Wol16] E. Wolff. *Continuous delivery: der pragmatische Einstieg*. dpunkt. verlag, 2016 (cit. on pp. 5, 9, 11, 14, 16, 18, 38).

[Woo04] A. Wool. “A quantitative study of firewall configuration errors.” In: *Computer* 37.6 (June 2004), pp. 62–67. ISSN: 0018-9162 (cit. on p. 1).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012 (cit. on pp. 73–76, 78, 79).

[Yin03] R. K. Yin. “Applications of case study research. Applied social research methods series.” In: *Thousand Oaks: Sage Publications. Yu, KH (2013). Institutionalization in the context of institutional pluralism: Politics as a generative process, Organization Studies* 34.1 (2003), pp. 105–131 (cit. on p. 72).

[YMZ+11] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, S. Pasupathy. “An empirical study on configuration errors in commercial and open source systems.” In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 159–172 (cit. on pp. 32, 37).

[You16] Youtube. https://www.youtube.com/watch?time_continue=39&v=Wh_1966vaIA. 2016. (Visited on 01/23/2018) (cit. on pp. 12, 15).

[ZBC16] L. Zhu, L. Bass, G. Champlin-Scharff. “Devops and its practices.” In: *IEEE Software* 33.3 (2016), pp. 32–34 (cit. on p. 5).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature