Adoption of DevOps Practices in the Finnish Software Industry: an Empirical Study

Paul Laihonen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 17.09.2018

Supervisor

Prof. Casper Lassenius

Advisor

M.Sc. Tuomas Keränen

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Paul Laihonen

Title Adoption of DevOps Practices in the Finnish Software Industry: an Empirical Study

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Casper Lassenius

Advisor M.Sc. Tuomas Keränen

Date 17.09.2018 **Number of pages** 64+1 **Language** English

Abstract
DevOps is a software development model with an objective to increase collaboration and information sharing between teams, produce fast feedback through frequent loops, and reduce error prone human interaction. This thesis researches the usage of these practices in the Finnish software industry by analysing a number of DevOps audits and by conducting post-mortem interviews for some of the audited organisations. As DevOps does not have an official definition, the DevOps maturity model was used as a baseline for the audits. Research methods included open and axial coding of the audits, and analysing the interviews. The aim was to understand how widely DevOps practices are being used and what are the most and least problematic areas.

Results show that a lot of generally accepted software development practices which are included in the DevOps methodology are being used in the majority of organisations. These practices include the use of agile practices and proper version control. The more DevOps specific practices are used less although organisations found them equally useful. For the majority of the researched organisations, DevOps is now a standpoint for all software development. A comparison between organisations focusing mainly on the software industry and organisations focusing mainly on other industries shows that there is a clear distinction of problematic areas. Software organisations have relatively more issues in the quality assurance category, whereas the other organisations' have relatively more problems on the environments and release category. The audited organisations were given a number of suggestions to improve their processes in the audits. The interviews showed that organisational priorities are more in the culture and management categories, whereas quality assurance and technologies and architecture were left on a smaller focus.

Keywords devops, software development, automation

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Paul Laihonen

Työn nimi DevOps praktiikoiden käyttö suomalaisessa ohjelmistoteollisuudessa

Koulutusohjelma Automaatio ja sähkötekniikka

Pääaine Säätötekniikka, robotiikka ja autonomiset järjestelmät **Pääaineen koodi** ELEC3025

Työn valvoja Prof. Casper Lassenius

Työn ohjaaja DI Tuomas Keränen

Päivämäärä 17.09.2018 **Sivumäärä** 64+1 **Kieli** Englanti

Tiivistelmä
DevOps on ohjelmistokehitysmalli, jonka tarkoituksena on lisätä tiimien välistä yhteistyötä ja tiedonjakoa, nopeuttaa palautteen saamista lyhyillä palautesilmukoilla, sekä vähentää ihmisen virhealtista vaikutusta. Tämä diplomityön tutki näiden käytäntöjen harjoittamista suomalaisissa it-organisaatioissa. Tutkimus tapahtui analysoimalla aiemmin tehtyjä DevOps auditointeja sekä haastattelemalla osaa auditoitujen organisaatioiden henkilökunnasta. DevOps maturiteetti mallia käytettiin käsitteen pohjana, koska sillä ei ole virallista määritelmää. Tutkimusmenetelmät sisälsivät aksiaalista sekä avointa koodausta ja haastattelujen tarkempaa analysointia. Tavoitteena oli saada ymmärrys DevOps-käytäntöjen levinneisyydestä Suomessa, sekä niiden ongelmallisimmista osa-alueista.
Tuloksista voi päätellä, että ohjelmistokehityksessä yleisesti hyväksytyt toimintatavat, jotka sisältyvät myös DevOps-käsitteen alle, ovat käytössä suurimmassa osassa organisaatioista. Nämä toimintatavat sisältävät esimerkiksi oikeaoppisen versionhallinnan sekä ketterien ohjelmistokehitysmenetelmien käytön. DevOpsille ominaisemmat menetelmät sen sijaan olivat selkeästi vähemmän käytettyjä, vaikkakin haastateltavat pitivät niitä yhtä hyödyllisinä. Suurin osa haastatelluista piti DevOpsia tämänhetkisen ohjelmistokehityksen lähtökohtana. Tutkimuksessa vertailtiin pääosin ohjelmistoteollisuudessa toimivien organisaatioiden ja pääosin ohjelmistoteollisuuden ulkopuolella toimivien organisaatioiden ongelmallisimpia alueita. Tulokset kertovat selkeitä ongelma-alueita kummallekin: ohjelmisto-organisaatioilla oli huomattavasti enemmän ongelmia laadunvarmistuksessa, kun taas muilla organisaatioilla oli enemmän ongelmia ohjelmistoympäristöissä ja ohjelmistojen julkaisussa. Auditoiduille organisaatioille annettiin vaihteleva määrä ehdotuksia auditointien yhteydessä. Haastattelujen perusteella organisaatioiden prioriteettina on ollut kulttuurilliset ja johdolliset kategoriat, kun taas laadunvarmistukseen ja teknologiaan sekä arkkitehtuuriin liittyvät suositukset jäivät pienemmälle huomiolle.

Avainsanat devops, ohjelmistokehitys, automaatio

Preface

I want to thank my supervisor Casper Lassenius and my advisor Tuomas Keränen for their support and feedback throughout this project. I also want to thank Eficode for making this thesis possible, and for providing all the required resources.

Espoo, 17.09.2018

Paul M. Laihonen

Contents

Abbreviations

CALMS	Cuture, automation, lean, measurement, sharing
CD	Continous delivery
CI	Continous integration
DevOps	Development and operations
IT	Information technology

1 Introduction

The production of software has long been done through practices concentrating solely on the process leaving people and culture on a smaller attention. Now, software projects are still divided to clear distinct parts where for example testing and Infosec activities happen only towards the end of the project. This creates long lead times, too many handoffs and problems are ultimately found very late in the process (Kim et al., 2016, pp. xxi–xxii). DevOps is a software development methodology initially generated to tackle these problems, by including all stakeholders in the process with efficient collaboration and information sharing, and reducing error prone human work by utilising automation as much as possible. The term DevOps is coined from two words: *development* and *operations*. It was initially generated to emphasise the need for bringing down the silos around developers (who create the software) and operations (who upkeep the software).

This thesis focuses on researching how well the practices known as DevOps are being used in the Finnish software industry. The research is done by analysing nineteen DevOps audits conducted by Eficode between 2016 and 2018. These audits were done by interviewing organisation employees and investigated how organisations produced software and what points were found beneficial and problematic from a defined DevOps point of view. In addition to the audit analysis, post-mortem interviews were conduced to nine out of nineteen audited organisations. The goal of investigating these audits and post-mortem interviews was to gain an understanding on how widely DevOps is being used, what practices are applied the most, and which of them where found the most and the least problematic. These goals are also represented as the Thesis research questions in Section 1.2.

Received results implied that DevOps has not reached organisational software development as a whole concept, but in pieces through its practices. The most common good and generally accepted practices in the software industry — such as proper version control and the use of agile software development frameworks — are used in most organisations, whereas more DevOps specific practices like in-depth cultural collaboration are yet to be implemented more properly.

1.1 Background and Motivation

The advances in technology and the adoption of agile practices since the early 2000's have gained DevOps more popularity for the past ten years. Before that, new software was a huge investment in any organisation. The development was slow and expensive, and developing and deployment time could be measured in years. Come to the first decade of the new millennia, time required to develop and release software had dropped from years to weeks or months, but time required to deploy software into production would require additional weeks or months and the probabilities for the outcomes to be catastrophic were very high. By 2010, with the introduction of DevOps, software development and deployment are done with fewer problems and more cheaply than ever. Currently new software lead times (which represents the time from creating an idea all the way to a working feature in production) are measured in weeks to production deployments can be done multiple times a day (Kim et al., 2016, pp. xxii–xxiii).

DevOps gaining popularity has resulted in the fact that more papers related to it are released each year in both scientific research as well as in the popular press (Erich et al., 2017). The topic has steadily gained popularity over the time among people, and an illustrative example of this can be seen from Figure 1, which shows the evolution of Google Trends[1] graph for the keyword "DevOps" since the first DevOpsDays in late 2009.

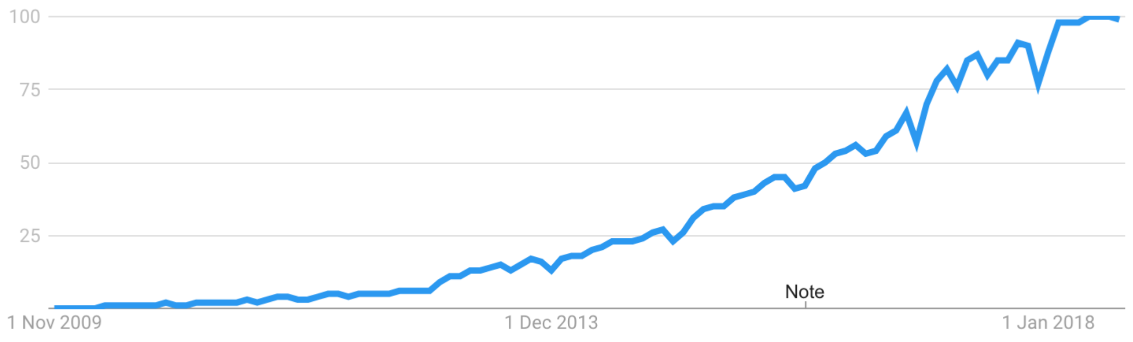

Figure 1: Google Trends graph for DevOps.

There has been little research related to the usage of DevOps in the software industry (Erich et al., 2017). Although the concept lacks a official definition, many software development practices have become established parts of DevOps. Researching how much and in which ways practices defined as DevOps are used, a more diverse understanding on the topic would be gained.

1.2 Research Questions

This thesis tries to unveil the researched topic through two research questions. These are as follows:

RQ1 How widely are DevOps practices used in the Finnish software industry?

RQ2 What are the most and the least problematic areas organisations have from a DevOps point of view?

1.3 Structure of the Thesis

This Thesis is structured so that the first chapter goes through the motivation and the general background to the research as well as states the research questions. The second chapter goes over previously done research and introduces the diverse concept of DevOps. Third chapter introduces the used research methods and subjects of which results are later presented in the fourth chapter. The fifth chapter discusses

[1]`https://trends.google.com/`

what these results mean. The sixth and final chapter presents overall conclusions to the research, talks about problems related to this Thesis and introduces topics for future research.

2 Literature Review

This chapter gives an overview to the existing research done on the topic discussed. DevOps is a fairly new concept and relatively little academic research has been done regarding the area. In addition to academic articles, also books, web articles, and blog entries have been used as source material. The credibility of non-academic sources has been maintained by mainly using sources that are commonly approved, referenced or written by credible authors.

The first part will go shortly over the history of DevOps: how and why the movement started. After that, the literature definition of DevOps describes the concept. It is split into two parts: the building blocks of DevOps and the core principles and practices of DevOps. The building blocks describe five components which build the very heart of the DevOps concept: even if the principles or technologies to implement it would change, these components would not. The next sections discusses the core principles and practices of DevOps and go over the current technological ways to implement DevOps. Emphasis should be put on the word *current* as these methods might evolve later with the forward moving technology. The final part of this chapter explains the current usage of DevOps worldwide.

2.1 History of DevOps

2.1.1 Origin of the Movement

Despite the current popularity of DevOps, the movement is only ten years old. One of the early influencers of DevOps — and later quoted as the godfather of the whole movement — was a Belgian consultant named Patrick Debois (Willis, 2012a). By 2008 he had worked in several IT-projects and had grown tired of the segregation of development and operations. Later that year he found like-minded people at the Agile conference in Toronto, Canada and ended up discussing the topic of agile infrastructure with a man called Andrew Schafer. This conversation was the first spark of DevOps, and lead later for example to forming the Agile System Administration group to Google. The initial popularity still remained rather modest. (Edwards, 2012)

Next year, in 2009, at the Velocity conference in San Jose, CA, John Allspaw and Paul Hammond gave their now-famous seminar presentation titled *10+ Deploys per Day: Dev and Ops Cooperation at Flickr*. The idea of the talk was operations working development-like. It really inspired Patrick and many others, and it quickly gained a steady group of followers (Edwards, 2012). Ultimately, it led to the first DevOpsDays which were organised by Patrick Debois in his hometown of Ghent, Belgium in 2009 (Kim et al., 2016). This is also when the term "DevOps" was initially coined.

After the 2009 DevOpsDays the movement started to gradually gather followers throughout the world and various DevOps-related events — such as the DevOps-Days of 2010 in Sydney, Australia and Mountain View, CA — started to emerge. Practitioners begun to find each other through different channels and for example the Twitter[2] hashtag `#DevOps` was a very important way to spread information in the early days. (Edwards, 2012)

2.1.2 Background of Practices

The DevOps practices originate from the best methods of physical manufacturing and leadership. According to Kim et al. (2016) this includes lessons learned from Lean, Theory of Constraints, and the Toyota Kata movement. The Lean movement and its relation to DevOps will be discussed later in Section 2.2.3. The Theory of Constraints was a core conflict in the 1980s manufacturing when the goal was to protect sales and reduce costs. Protecting sales meant a need for a bigger inventory and reducing costs the contrary: less tied up money with less inventory. This conflict was later solved with the lean principles, which patterns can be reflected in modern software development.

Some see DevOps also as a continuation to the Agile movement which started in 2001. Back then, software development was based upon a heavy set of values and principles such as the process of waterfall development. Kim et al. (2016) mention the main principles of the movement to include delivering working software frequently, working with small, self-motivated teams and working in a high-trust management model. Moreover, many of the movements in DevOps history emerged from the Agile community.

2.2 Building Blocks Behind DevOps

As opposed to for example the Agile Manifesto[3], there is no certified definition of what DevOps actually is, nor will there probably every be (Little, 2016). Erich et al. (2017) studied the common definition of DevOps and acknowledged that labels most commonly connected to it include a culture of collaboration, automation, measurement, sharing, services, quality assurance and governance. They state that none of these concepts are clearly presented in academic researches nor there are generally any metrics to measure their effectiveness. However, they mention that culture, automation, measurement, and sharing came up often in their research interviews as important parts of DevOps.

An early DevOps pioneer Willis (2012a) purposed already after the first US based DevOpsDays in 2010 that the main building blocks of DevOps would be these same four components. He writes that lean has been later added to the list by Jez Humble finalising the commonly used acronym of CALMS: culture, automation, lean, measurement and sharing, respectively. This section will shortly go through each of these components and discuss why they are an important part of DevOps.

[2]`https://twitter.com/`
[3]`http://agilemanifesto.org/`

2.2.1 Culture

Having the right culture is the cornerstone of DevOps and without it, all automation attempts will be futile (Willis, 2010). Many key aspects of DevOps include adopting a new process or a technical tool, but without the right culture DevOps can be considered merely another buzzword. One essential part of DevOps is changing the organisational culture towards increased collaboration and bringing down the silos around development and operations is a key move towards this (Wilsenach, 2015).

The friction between development and operations can be pinned to four cultural characteristics: open communication, incentive and responsibility alignment, respect, and trust (Walls, 2013). Open communication and more frequent collaboration is the biggest of these and the move from isolated teams towards a more open collaboration can be done by binding development and operations together. For example, locating teams closer to each other help as handovers and sign offs discourage from sharing responsibility and encourage a blame culture (Wilsenach, 2015). Humble and Molesky (2011) suggests enforcing more frequent collaborative encounters. An important step towards that is the involvement of operations personnel into the design and transition of the systems they operate. Their representatives should attend relevant retrospectives, planning meetings, and showcases of project teams to get involved and share their knowledge in the development process as early as possible. Furthermore, development personnel should equally attend meetings with operations and have the readiness to assist them in production related problems.

Humble (2012) says that the fundamental problem in the lack of shared responsibility is the bad behaviour that arises when people are abstracted away from the consequences of their actions. Silos between development and operations encourage this kind of behaviour and he proposes higher awareness and responsibility of the consequences. Sharing the operational responsibilities with development may help the developers to become more interested in optimising their product as well as motivate them in identifying ways to simplify development and maintenance (Wilsenach, 2015).

Willis (2012b) and Walls (2013) both underline the importance of mutual respect. It is considered as the base of any work culture enables a safe environment for any teamwork. Efficient collaboration with balanced risk management without the extra overhead can be resolved with more autonomous teams and applying changes inside teams without a convoluted decision making process requires trust (Wilsenach, 2015). This trust between management, teams and co-workers can be built on top of tools and automated processes such as version control and internal change approvals.

A "DevOps team" is often used as a collaborative solution in order to enable the right kind of culture (Erich et al., 2017). Humble (2012) and Wilsenach (2015) emphasise the irony in the creation of another functional silo to create better collaboration between functional silos, but Humble (2012) also notes that calling certain team a DevOps team is somewhat acceptable in the situation where operations and quality assurance personnel help developers understand their work, and vice versa.

2.2.2 Automation

Alongside the culture of collaboration, automation is a term most often used to describe and define DevOps (Erich et al., 2017). The use of automation is important especially in operational processes (Lwakatare et al., 2015), and fundamental in achieving low lead times and rapid feedback (Humble and Molesky, 2011). A set of different tools help stitching the automation together (Willis, 2010) and play an important part in DevOps and many of the practical steps linked to DevOps described later in Section 2.3 are based on automation.

In the centre of automation there is the deployment pipeline implemented by the development team. The development pipeline works as a single path to production for all changes made by the team — including changes made to code, infrastructure, database schemas, environments and so on. By using the development pipeline, each change is automatically validated with automated quality assurance, tagged with a version number, stored to version control and set ready for a possible release. In addition to these, the change will be ready for environment provisioning and configuration. Ultimately there would be no manual steps between a developer's commit to version control and the production system. (Humble and Molesky, 2011)

Automation is an important part of DevOps as it connects with almost every direction. Automating monotonous and repetitive tasks, such as manual testing or configuration, facilitates cultural collaboration and frees up people to focus on more valuable human activities reducing the change of human error in these areas (Wilsenach, 2015). In complex systems automating these tasks can also be time consuming and difficult to manage (Lwakatare et al., 2015). The following section discusses lean's importance to DevOps, and many of the principles mentioned also require automation for efficient use.

2.2.3 Lean

Although only a small number of organisations consider lean as part of DevOps (Erich et al., 2017), the same concepts that define lean can be applied to DevOps (Kim et al., 2016, p. 7). The lean way of thinking arose from practices used in manufacturing and the Toyota Production System used since the 1980s is often seen as the origin of these practices Humble et al. (2014). The Toyota Production system is a set of principles and practices which are considered the most widely known in any industry (Holroyd, 2014) initially used by the car manufacturer Toyota. Staats and Upton (2011) call the production system "the most important invention in operations since Henry Ford's Model T began rolling off the production line" and Holroyd (2014) cites to use its principles to explain DevOps to those unfamiliar with the concept.

Lean methodologies are designed to improve operational efficiency and reduce waste. Two main tenets among all others are the belief that (manufacturing) lead time is the best predictor of quality, customer satisfaction, and employee happiness, and that the most efficient way to reach short lead times is small batch sizes of work (Kim et al., 2016, p. 353). Lead time describes the period between when the request for a certain product or feature is made and when work is completed (Kim et al., 2016, p. 9). DevOps also aims to shorten the lead time as much as possible with small batch sizes, and automation. These create short feedback loops and ensure constant quality with automated testing and fast deployments to production through continuous delivery and modular architecture (Kim et al., 2016, pp. 10–11). Applying lean principles to management practices has a improving influence on IT performance, organisational performance, as well as lowering employee burnout levels (Forsgren, 2015).

Poppendieck and Poppendieck (2006) generated seven lean principles and thinking tools that help in translating each principle into software engineering practices. First of those principles is eliminating waste. In software engineering, waste can be considered anything that does not add value from a customer point-of-view. Shingo and Dillon (1989) initially introduced the concept of waste into lean manufacturing, and Poppendieck and Poppendieck (2006) transformed these wastes to fit software engineering. They include partially done work, features that the customer did not order, relearning, handoffs from one group of developers to another, breaking concentration with task switching, unnecessary delays, and software defects. The goal of DevOps is to systematically eliminate these wastes to achieve the goal of fast flow (Kim et al., 2016, p. 25).

The second principle is building quality in. This emphasises on the fact that the goal is not to put defects on the tracking board but to prevent defects in the first place. Quality assurance's main goal should be preventing bugs and not finding them. Creating quality code from the very beginning requires a disciplined organisation. The third and four principle — creating knowledge, and deferring commitment — define that decisions should be based on gained knowledge and made as late as possible. Architectural plans, for example, are often made before the actual implementation, but the knowledge whether the plan works in practice is not gained until the actual implementation. This means it would be unwise to lock the plans down in advance. In a changing market the decisions based on knowledge and facts create more predictable results than those based on forecasts (Poppendieck and Poppendieck, 2006). Knowledge is gained more often with early and frequent releases. They produce feedback which ultimately leads more often to success (MacCormack, 2001).

Frequent releases are also the key of the fifth principle: delivering fast. Fast feedback creates more frequent possibilities to improve products and processes. Continual movement also requires built in quality as well as capable and motivated people who know what they are doing without being constantly told so. The sixth principle is respecting people. As mentioned in the previous section, respect is an important part in building a working DevOps culture. Respect in lean means involving the front line people who actually do the work into decision making — and respecting the decisions they make. (Poppendieck and Poppendieck, 2006)

The last principle is to understand the whole. While experts often want to maximise their performance, it might not be good on a bigger scale. A lean organisation optimises the whole value stream and not only one separate parts of it (Poppendieck and Poppendieck, 2006). From the DevOps point of view this means higher collaboration between traditionally separated development and operations towards a common goal.

2.2.4 Measurement

Poppendieck and Poppendieck (2006) describe lead time as the most important lean measurement: how long it takes to go from a concept to cash? Measuring the value gives a good idea on how fast the organisation is able to deliver software, but in DevOps monitoring should go beyond that by monitoring as much as possible as often as possible (Willis, 2010).

Kim et al. (2016, pp. 209–214) splits measuring into five distinct levels: business, application, infrastructure, client software, and deployment pipeline. By monitoring and measuring all of these levels, a versatile view on the systems health is obtained. If problems occur, decisions can be based on facts instead of pointing a finger. This means, for example, that in order for developers to create more stable systems they need to provide a broad set of measurement data to see how releases affect the systems stability (Humble and Molesky, 2011). It is important to choose the measured key performance indicators with care, as people change their behaviour according to how they are measured (Humble and Molesky, 2011).

2.2.5 Sharing

Sharing extends the culture of collaboration discussed earlier and works as a loopback in the CALMS-model (Willis, 2010). In a working organisation, sharing contributes to several levels: Humble and Molesky (2011) mention an effective form of sharing for development and operations being the celebration of successful releases together. According to them, sharing can also mean spreading knowledge, development tools, and techniques. The more information is shared, the less separated teams become.

2.3 Core Principles and Practices of DevOps

Now that the base building blocks and ideologies of DevOps are introduced, it is easier to dive deeper into the practicalities. As DevOps is not merely a buzzword meant to be used by business and marketing to speed up sales, this section is going to introduce a more hands-on point of view: what are the actual steps and practices one has to implement when adopting DevOps.

The main approach used was initially introduced by Kim (2012) as "The Three Ways". He explains that all of the observed DevOps patterns can be derived from these three ways, represented in Figure 2. These ideas are more widely explained in the books *The Phoenix Project* (Kim et al., 2013) and *The DevOps Handbook* (Kim et al., 2016). The latter sums up all modern practices widely described as DevOps. The book is also going to be used as a base for most theories and practicalities discussed in this section.

As the name suggests, the Thee Ways describe three flows of which each enables an important stream of information from one group of stakeholders to another. The first flow runs from left-to-right — from development to operations to customer — and works as a core component in bringing the software product from merely an idea all the way to production and to the usage of end users. The First Way is generally described just as Flow.

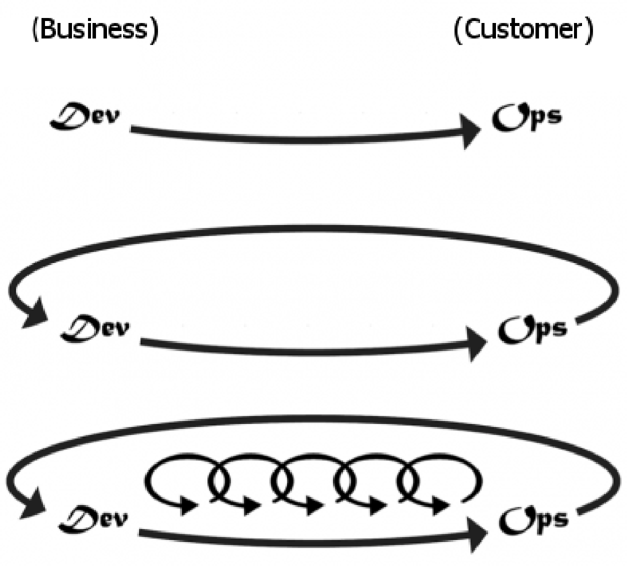

Figure 2: The Three Ways, as represented by Kim (2012).

The Second Way, Feedback, enables a fast and constant feedback flow from right to the left. This feedback can happen at any stage of the stream and it is meant to help in amplifying feedback for preventing problems from happening again, or enabling faster problem detection and recovery. This allows safer systems by creating quality at the source and a world where problems are solved before any catastrophic failure occurs.

The third principle leans on creating a dynamic culture of trust and constant learning. The Third Way, Continual Learning and Experimentation, approaches these ideas through a relentless urge for improvement and a scientific approach to experimentation and risk-taking. By taking the new knowledge into practice, feedback loops can be shortened and systems can be created even safer. Experimentation lets organisations learn faster and ultimately win in the marketplace.

The following subsections dive deeper into these principles and detail practical matters which should be considered alongside them. As explained later in Chapter 3, the usage of these principles and practices are used throughout this Thesis to describe the adoption level of DevOps practices.

2.3.1 Consistent Production-like Environments

In order to a create fast and reliable left-to-right flow, the environment behind developed software requires consistency. That means that all used environments need to be production-like at every stage of the stream. To achieve this kind of consistency, the environments need to be configured and built in an automatic manner. Ideally, developers could set up these environments independently whenever needed from scripts and configurations stored in version control. Enabling this automated, on-demand environment creation is one of the practical cornerstones towards an efficient DevOps culture. Processes are also easily repeatable when changes are not made directly to the system but through a configuration script (Morris, 2015).

When environments are configured automatically, developers will always run their code in a similarly configured environment, providing early and constant feedback on the quality of their code. In an ideal situation, any developer can set up his or her own servers without the need to get an approval to order them separately. This setup could be done for example with an electronic tool that builds a virtual environment replicating the production as much as possible. These servers would be set up almost immediately without the need to wait for hours or days for someone to handle the order. The servers could also be destroyed easily whenever a developer has no more need for them. This kind of self-service accelerates development process as waiting time is reduced when developers can more easily create and destroy environments on demand (Morris, 2015).

A development pipeline works best when software can be rebuilt and re-created anytime, on demand. Environments should be handled the same way: infrastructure should be easier to rebuild than to repair. Repeatable environment creation also enables easier horizontal scaling and disaster recovery. One of the famous quotes by former Microsoft distinguished engineer Bill Baker quipped that they used to treat servers like pets:

> *You name them and when they get sick, you nurse them back to health. [Now] servers are [treated] like cattle. You number them and when they get sick, you shoot them.*
>
> – Bill Baker (Kim et al., 2016, p. 118)

To ensure consistency between environments, all changes made to production should also be replicated in all development and test environments. Instead of manually logging to all servers and making changes one by one, the configurations have to be made in a way that ensures all settings consistent. This can be achieved by handling all infrastructure as code through configuration scripts and tools like Ansible[4] or Chef[5]. Another way of consistent environmental configurations is packing the whole environment in a container (e.g. Docker[6]) or a virtual image (e.g. Vagrant[7]) and dropping that artifact into a minimally configured environment that can run it (Kim et al., 2016, p. 114). Using automated configuration reduces the amount of environment-related faults and defects as the amount of error-prone manual work is minimised (Forsgren et al., 2017).

Version control has long been a mandatory practice in software development. A version control system enables collaboration between multiple developers and records changes made to files. However, customer value comes from both, software and the environment it is run in. Codifying the environment also enforces the use of version control and opens the common benefits of it, such as easy to see change history and

[4]`https://www.ansible.com/`
[5]`https://www.chef.io/chef/`
[6]`https://www.docker.com/`
[7]`https://www.vagrantup.com/`

centralised management. Usage of version control is one of the factors reflecting on how well the organisation is doing: Forsgren et al. (2014) state that the usage of it by operations is the highest predictor of both IT and organisational performance. Everyone along the stream (development, operations, quality assurance, infosec etc.) should utilise it to allow a repeatable and reliable reproduction of all components of working software. (Kim et al., 2016, pp. 115–118)

2.3.2 Automated Quality Assurance

Quality assurance was not described as a building block of DevOps in Section 2.2 nor has it been seen as one (Erich et al., 2017). Product quality does however have a huge impact on how often software can be deployed and the cost of each deployment. As DevOps aims to reduce the amount of human interaction with automation and continuous delivery. Out of all capabilities of continuous delivery, automated quality assurance is the single biggest enabler towards that goal (Forsgren et al., 2017). A common problem often confronted with manually executed testing is the uneven quality of carried out tests. As humans tend to get bored of repetitive and unilateral flows, manual tests are particularly prone to human errors. Also, the amount of tests run can be significantly higher when tests are automated (Rafi et al., 2012).

Early feedback loops on software quality are important. Preconfigured automated tests are easy to run repeatably and with smaller costs (Rafi et al., 2012). Finding defects after deployment can be often 100 times more expensive than finding and fixing them in the development phase (Boehm and Basili, 2005). The earlier a defect is found, the cheaper and faster it is to fix. An efficient way to assure constant quality is running a predefined test suite against the software automatically after every version control commit (Kim et al., 2016, p. 130).

Running the software against a full set of automated tests — which can be in some cases counted in thousands — can take hours. In general, automated tests fall into one of the three categories, from the fastest to the slowest: unit tests, acceptance tests, and integration tests. Feedback loops can be shortened by running the tests in an order which are in proportion to time spent executing them. (Kim et al., 2016, pp. 132–134)

Integrating non-functional requirements into the automated test suite is also important. Security tests need to be handled regularly to ensure lack of vulnerabilities and performance tests to make sure systems are able to handle required amount of stress. The fact that tests should be run in a production-like environment is especially important when running tests against non-functional requirements as they are more impacted by small configurational changes than running software. (Kim et al., 2016)

2.3.3 Continuous Integration and Delivery

Business' goal is more often than not to build quality into the product. This can be achieved in multiple ways, one of them being constant and continuous feedback. As previously mentioned, automated tests should be run in a production-like environment after every version control commit and work as a software quality gate: if the quality is feasible, it passes all required tests and moves forward automatically. This continuous process to handle a product automatically is called continuous delivery, first defined by Humble and Farley (2010). It ensures that all code checked in to version control is automatically tested and handled. When all developed code goes through the same procedure, one can be assured that the product is in a deployable and shippable state all the time, and delivering changes is painless (Morris, 2015). Continuous integration is a piece of continuous delivery and covers new software integration to existing software. Technically, this means triggering automated tests after version control commits to automatically validate new code. Gary Grover, the former director of HP's LaserJet firmware division, sums up the importance of automated tests in continuous integration:

> *Without automated testing, continuous integration is the fastest way to get a big pile of junk that never compiles or runs correctly.*
>
> – Gary Grover (Kim, 2014)

According to Humble and Molesky (2011) the automated deployment process has three important effects: results are automatically documented and errors can be fixed earlier, decision-makers can be notified automatically of incoming changes, and when automating all aspects of the pipeline changes to the environment can be locked down to make changes merely through a documented automated pipeline. Continuous delivery also makes the change management process more lightweight (Morris, 2015). Popular continuous integration tools, such as Jenkins[8] or Bamboo[9], can produce and store product build and configuration history. When done so, one can always tell which changes broke the build and what improvements should be made, which tests were run on which builds, which builds are deployed on each environment and so on (Kim et al., 2016, pp. 113–115). This kind of information is valuable for example when making improvements to the feedback cycle or general CI processes.

A general process for a development pipeline starts from the commit stage. That builds and packages the software, and runs automated tests as well as additional analysis, such as static code analysis and coding convention check. If successful, the acceptance stage is triggered, which deploys packaged code to a production-like testing environment running automated acceptance tests. Further on, the same packages are deployed to user acceptance testing or staging environments before accepted and deployed to production. One of the lean principles is to work in a single piece flow. DevOps equivalent to this is working with small batches. A continuous integration pipeline steers developers to commit often and small changes. The smaller the change the smaller is the risk to break the build. (Kim et al., 2016, pp. 113–115)

2.3.4 Suitable Architecture

To enable previously mentioned practices of automated testing, continuous integration, and continuous delivery the software architecture must be built to fit the

[8]`https://jenkins.io/`

[9]`https://atlassian.com/software/bamboo`

requirements. Software architecture can be roughly split into two archetypes: monolith and microservice (Kim et al., 2016, p. 182). A monolith architecture describes a system where everything is tightly coupled together. When large teams are working together with a monolith architecture, every change needs a lot of coordination (Balalaie et al., 2016). A microservice architecture means an architecture that contains many loosely coupled units with well-defined interfaces to communicate with other modules. In-between architectures naturally also exist.

Kim et al. (2016, p. 183) lists a number of pros and cons for monolith and microservice architectural types. Both of them are good for different types of projects. Monolith architectures are often the best choice for an organisation early in their development cycle due to the absence of excess complexity. Microservice architecture becomes a more suitable choice when the software grows bigger and more complex as it can then be split to more manageable pieces. When Amazon was going through their transformation from monolith to microservice architecture in 2002, they used Jeff Bezos' two-pizza team rule to limit team sizes: a development team should not be bigger than what two pizzas can feed (Choi, 2014).

The *2015 State of DevOps Report* (Forsgren, 2015) shows that high performing organisations have more likely adopted a loosely coupled microservice architecture, and that architecture is one of the top predictors of the development productivity and of how changes can be quickly and safely made. From a DevOps point of view, software with a monolith architecture works well and microservice-like in smaller sizes. The downsides are that it scales poorly, builds long and, when growing bigger, enforces a deployment strategy where the whole application either does or does not work. Running the test suite can also last long. Microservice architecture can make smaller software needlessly complex, but in a larger scale brings multiple benefits: simplicity through small units, independent scaling, performance and testing, as well as optimal tuning of performance (Kim et al., 2016, p. 183). A continuous integration pipeline works more efficiently when build and tested software modules are small. This way integrating changes to an existing system takes a shorter time and developers do not have to wait hours for tests to complete in order to see whether their code worked or not.

2.3.5 Monitoring for Constant Feedback

After safe and efficient development practices have been set, the next step is to ensure that the software runs steadily in production. That can be accomplished with monitoring. Monitoring means following the change of a certain variable. The more variables are followed, the more information the personnel running the production instances gain about their state. When something goes wrong, the monitored variables help in tracking down the problem and understanding how to prevent it from happening again.

A good practice is to gather as much information as possible. To do this, monitoring should be embedded as part of everyone's job, so all stakeholders in the value stream can find and fix issues. This way for example developers know when their code causes problems in the production environment as well as what the problems are so eventually the root cause can be found quickly. When the number of monitored variables grows, it is important for operations to keep in pace with them. A vast amount of constantly flowing information is easier to follow when it is categorised — for example according to the module they are produced by — and the monitoring logs are prioritised corresponding to their threat level — for example with labels such as *info*, *warning* and *error.* By doing this, the logs are given meaning and are easier to interpret whether or not a certain log entry should be addressed.

Turnbull (2014) presents a common monitoring architecture, developed and used by operations engineers at, for example, Google and Amazon (Kim et al., 2016). In the described architecture data collection happens at the business logic, the application layer and environments layer. In the collection layers, logs are gathered by applications either to application-specific files on each server, or preferably to a uniformly defined place. An event router is responsible for storing the data and for turning these thousands of separate log entries into statistical metrics. That enables the visualisation of logs these logs, alerts, anomaly detection and so forth.

The development pipeline should be monitored too. It produces a vast amount of important metrics which can be used to follow the state of product development. This includes variables such as a number of passed and failed builds, the duration builds take to create, test suite executions statistics and so on. The more information is gathered from the development pipeline the better general problems can be addressed and the better the pipeline itself can be optimised.

Monitoring is important and gives a technical advantage to business competitors. According to the *2015 State of DevOps Report* (Forsgren, 2015) high performing organisations could resolve production incidents up to 168 times faster than their opposition. The two most important practices to enable fast mean time to repair were listed as version control usage by operations and proactive monitoring of the production environment. In addition, the Microsoft Operations Framework study in 2001 showed that high service level organisations rebooted their servers twenty times less frequently than average. This was due to a disciplined approach to understand the root problem using monitoring statistics gathered from production environments (Behr et al., 2004).

In addition to technical metrics about the system health, statistical data following business metrics and the fulfilment of organisational goals should be also gathered. This can include the number of new users created, user session lengths, the usage rate of certain features and so on. When software changes are made frequently, it is important to keep up with the usefulness of these changes. Business decisions are easier to make when comprehensive statistics about whether a "great idea" actually creates value is available. All gathered data can be visualised through monitoring tools such as Grafana[10] or Graphite[11]. Visual graphs created from monitored data can help in creating an understanding about the system status with merely a glance. Letting teams access these graphs through self-service portals or physical information radiators can further promote their responsibility and desire to actively fix problems.

[10]`https://grafana.com/`
[11]`https://graphiteapp.org/`

2.3.6 Safe Change Management

System changes always incorporates risks. In software development change management can be done through two distinct bodies: internal, among technical stakeholders, or external, for example through change approval boards. One of the findings related to IT performance represented in the *2014 State of DevOps Report* (Forsgren et al., 2014) was that when external approval was required in order to deploy into production, IT performance decreased. On the contrary when approval was done internally, performance increased.

According to Kim et al. (2016) this is surprising, as in many organisations the change advisory boards serve an important role in coordinating the delivery process. They argue one possible reason for this being as the predicament of being in such a board reviewing complex change composed by thousands of lines of code changes with merely a hundred word description and without comprehensive background information. Staats and Upton (2011) give a similar claim towards internal approvals believing that *"people closest to a problem typically know the most about it"* so creating approval steps from people who are further located and further away from the work may reduce the likelihood of success.

The internal change approval process is often also referred to as peer review. That means that the change reviews are conducted by co-developers already familiar with the code so the technical team holds itself accountable for the quality of the code. Peer reviews can also be split into two categories: formal review and lightweight review (Huizinga and Kolawa, 2007). Formal reviews have been conducted for almost 40 years and include a heavy and rigid process which takes five times longer than lightweight reviews (Cohen, 2011). They are no longer the most efficient way to review code. Lightweight peer reviews can be done during or post development and can contain methods such as pair programming, "over-the-shoulder", email pass around, and tool-assisted review Kim et al. (2016, pp. 256–257).

In pair programming, two developers work on the code simultaneously, one writing the code and the other reviewing it. Pair programming can increase development time from 15% to 100% (Lui and Chan, 2006) but also decrease the amount of defects approximately by 15% (Cockburn and Williams, 2000). "Over-the-shoulder" is a method where a developer looks over the author's shoulder as the latter goes through the code, and in an email pass around the source code management system emails code to reviewers after code is committed. These two methods are described less effective to verify review results for example due to being too lightweight and simple and thus not through enough (Cohen et al., 2006). Tool-assisted reviews are conducted through specialised code review tools like Gerrit[12] or GitHub pull requests[13].

Bacchelli and Bird (2013) discovered that peer reviews come with multiple additional benefits to finding defects: they are a valuable way for knowledge transfer from developer to developer, and created increased team awareness as well as alternative solutions to problems.

[12]`https://www.gerritcodereview.com/`

[13]`https://help.github.com/articles/about-pull-requests/`

2.3.7 Continual Learning and Experimentation

When working with complex systems, no matter how safe the organisations software development practices and change management are, problems are inevitably to occur at some point. When they do, enabling a safe environment to process these failures is a key towards better practices and an opportunity for continual learning (Kim et al., 2016, pp. 271–273). Dekker (2016, p. 24) discusses the concept of *just culture*, a right balance in organisational culture between satisfying the demands for accountability and contribution to learning and improvement. He claims that eliminating the people who caused the problem — the bad apples, as he calls them — is an invalid solution as human error is not the cause of trouble, but merely a consequence of the design of the tools that are given to them. Because of this, instead of blaming and shaming the design itself should be improved, and the goal should always be to see problems as opportunities towards organisational learning.

One way to learn through failures is described by Allspaw (2012). He talks about a process of *blameless post-mortems* which are carried out soon after the incident has been resolved. In a situation where accountable quarters are punished for their failure they might be more interested in "covering their backs" than giving detailed information about the circumstances of the failure, or steps that lead to the failure. This creates lack of understanding how the accident occurred and guarantees that it will repeat. The blameless post-mortems include understanding what steps lead to the accident, how these steps could be prevented in the future, and what could have been done better. The best people to describe these errors are the ones that caused them, and when accidents are handled without the fear of blame or judgement, Allspaw (2012) found the accountable engineers to be much more enthusiastic in helping the organisation avoid the same error in the future. Ray Rapaport from Netflix agrees with the blameless methodology and gives an example through a situation they had:

> *In fact, it [an outage] was caused by an engineer who had taken down Netflix twice in the past eighteen months. But of course, this person we would never fire. In that same eighteen months, this engineer moved the state of our operations and automaton forwards not by miles but by light-years.*
>
> – Ray Rapaport (Kim et al., 2016, pp. 280–281).

2.4 Usage of DevOps practices

Little academic research exists about the usage of different DevOps practices. Erich et al. (2017) studied DevOps-related literature and did not find any research where an organisation would have implemented DevOps and provided qualitative data to observe its benefits. They also interviewed six organisations which had transferred to use DevOps. All of them found the transition smooth and the experiences positive. In the research, organisations wanted to achieve a variety of different goals with DevOps

The HELENA[14] project (**H**ybrid d**E**ve**L**opm**EN**t **A**pproaches) is an international study with a goal to "investigate the current state of practice in software and systems development". The project is still in progress during the writing of this thesis, but some initial results have been published. Current initial results show what software development frameworks and methods are the most common, for example, in Sweden, Europe and worldwide. According to Scott et al. (2017), 46% of the organisations worldwide use something they describe as DevOps either sometimes (23%), often (15%), or always (8%). In Sweden, the corresponding total is 44% of the organisations using DevOps either sometimes (16%), often (17%), or always (11%). Detailed information is presented in Figure 3.

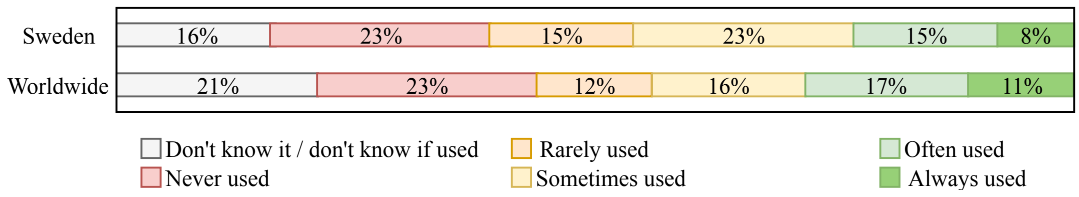

Figure 3: Usage of DevOps in Sweden and worldwide (Scott et al., 2017).

Scott et al. (2017) also list a number of development practices that could be regarded DevOps-related. The most popular from these both in Sweden and worldwide are coding standards, code review, automated unit testing, and continuous integration. A majority of organisations report using each of them at least sometimes. The least used practices include security testing and pair programming among others. A full list of these practices is shown in Figure 4.

Multiple large organisations, such as Flickr (Allspaw and Hammond, 2009), Netflix (Kim et al., 2016), Etsy (Rembetsy and McDonnell, 2012) and Amazon (Kim et al., 2016), have reported using DevOps with positive results. In addition to organisation specific stories, a number of more generic commercial reports are done regarding the practical usage and benefits of software development and DevOps. These include an annual *the State of DevOps Report* by Puppet and a quarterly *Currents* report by Digital Ocean (which does not solely concentrate on DevOps but software development in general).

According to Forsgren et al. (2017) high performing organisations capitalise on automation a lot more than other organisations, giving them technical advantage through its multiple technical benefits. As shown in Table 1, there are significant differences between high, medium, and low performers in terms of automated work. While Forsgren et al. (2017) claim that automation is a huge boon to organisations the high maintenance and implementation costs are seen as the highest reason to hinder automation implementations down (Kasurinen et al., 2010).

[14] `https://helenastudy.wordpress.com/`

Table 1: Percentage of manual work, by performance group. (Forsgren et al., 2017)

	Performance group		
	High	Medium	Low
Configuration management	28%	47%[a]	46%[a]
Testing	35%	51%[b]	49%[b]
Deployments	26%	47%	43%
Change approval process	48%	67%	59%

[a] [b] Not significantly different

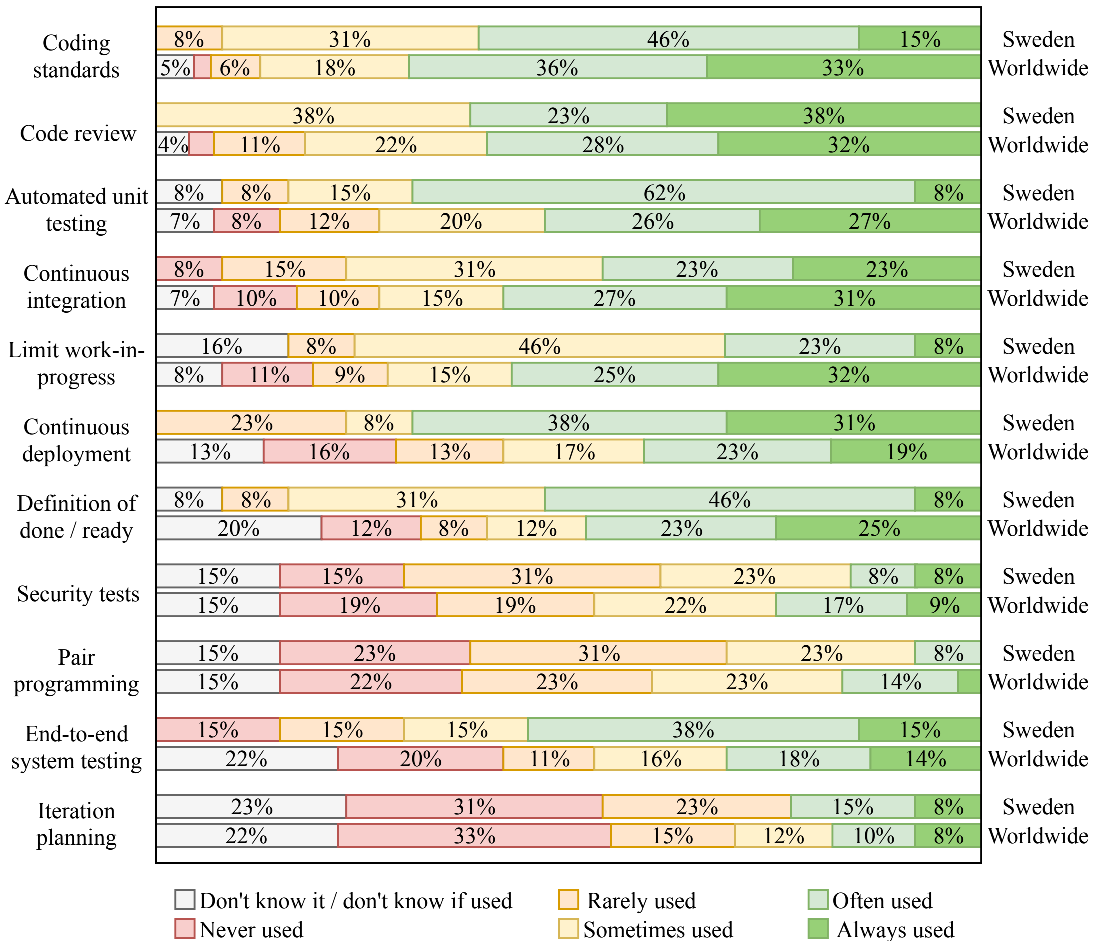

Figure 4: Usage of DevOps practices in Sweden and worldwide (Scott et al., 2017).

2.5 Summary

As a term, DevOps was first used in 2009 when Patrick Debois organised the first DevOpsDays in Belgium. Since then, the concept has spread worldwide and is now being used by the biggest software companies in the world. Figures 3 and 4 show that the practice has gained a steady foothold.

The basic goal of DevOps is simple: create the best software by the most motivated people as rapidly as possible while maintaining quality. The goals can be achieved through the core building blocks of DevOps. They are described as the CALMS components. These blocks enable the usage of a fast paced feedback loop through a working culture of sharing and trust. Automating as much from the development process as possible leaves the repetitive and error prone work to machines and people can concentrate on more sensible tasks. By combining them with built-in quality and constant measuring, it is possible to create a system that tells its operators about possible problems even before they occur.

Current technological practices to carry out efficient DevOps include giving automation a lot of responsibility. Automation can be used in every step of the way: from the automated provisioning of the development environments to quality assurance and deployments. When human interaction in these repetitive tasks is reduced, less errors occur and ultimately better quality software is created. To maintain quality in the parts where human interaction is still inevitable, simplicity is often the best choice. For example, reviewing changes through a rigid acceptance system has proven not be as efficient as colleagues reviewing each other's changes.

3 Research Methods

This chapter goes over the used research methods. The goal of this thesis was to gather and analyse enough information to understand how widely DevOps is used in the Finnish software industry, and what are the common problem areas. Two maturity models are used as a baseline for DevOps: the DevOps maturity model, and the automation maturity model. Both of these models were developed by Eficode. Eficode is a Finnish IT-company, originally founded in 2005, offering design and software services specialised in consulting, software development, and DevOps. At the time of writing this thesis Eficode has approximately 250 employees located at offices in Helsinki, Tampere, Gothenburg, Stockholm, and Copenhagen. Customers range from banking and healthcare industry all the way to small startups.

Eficode also carried out DevOps audits used as data in this thesis. The results of these audits were analysed by using axial coding and by comparing given results with previously mentioned maturity models. Also, possible patterns between different organisations were sought. In addition to the audit analysis, post-mortem interviews were done. The goal of these interviews was to further understand how beneficial the organisations saw these audits and how much they have assisted their software development processes.

The used DevOps and automation maturity models are opened up first in this chapter. After that, the used subjects are presented. Finally, used methods for data collection and analysis are explained in detail.

3.1 DevOps and Automation Maturity Models

In 2007 Eficode employees saw a need for a clear and visual representation of DevOps. This need came from both their own requirement to visualise the organisations core business somehow, but mainly because there was a clear business functional need for such. Organisation decision-makers often get the best understanding about something from clear and visualised data (Lurie and Mason, 2007) and thus came the idea of visualising the concept of DevOps. The initial visualisation is founded on the basis of DevOps practices: a CI/CD platform. On top of that are added business needs and arrows that help to understand how software development flow works. The result is represented in Figure 5.

First version of the DevOps maturity model was derived from previously mentioned platform representation in 2014. There was a business need for a clearer representation of DevOps itself with all its concepts, so they came up with the maturity model. The model is based on the ideas of long-time DevOps pioneers such as Patrick Debois and Gene Kim whose thoughts on the matter were later combined by Kim et al. (2016) in *The DevOps Handbook*. This chapter will refer to both the DevOps maturity model and the automation maturity model categorisation through *the Three Ways*, explained earlier in Section 2.3.

The first versions of the DevOps and the automation maturity models had five main categories: *organisation and culture*, *environments and release*, *builds and continuous integration*, *quality assurance*, and *visibility and reporting*. Approximately

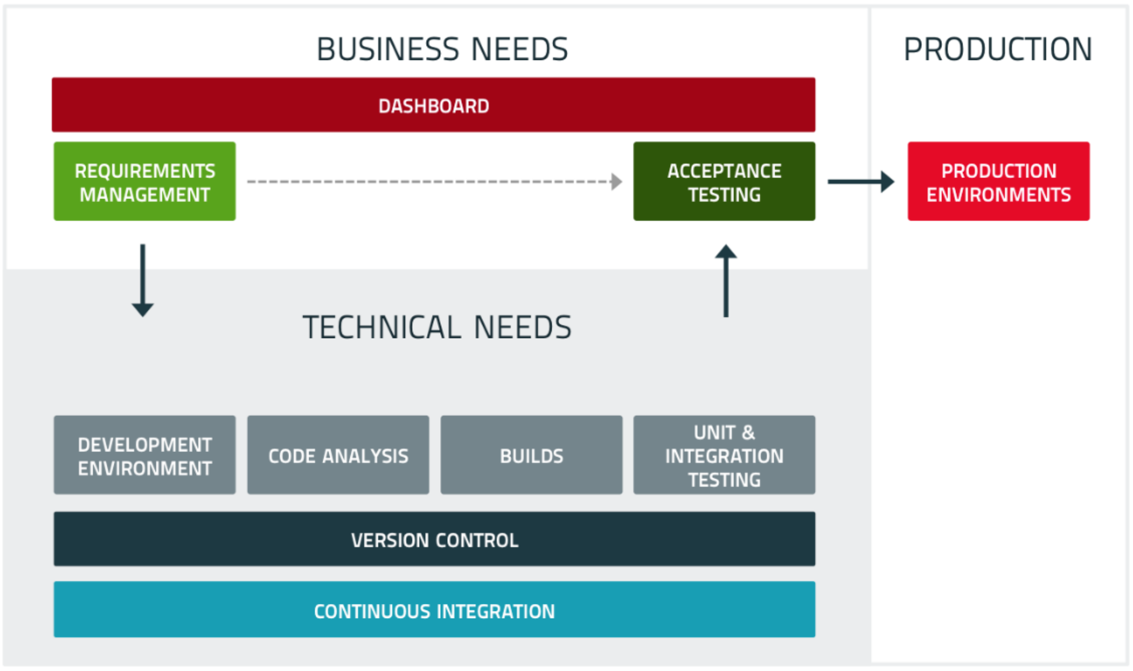

Figure 5: Visual representation of a DevOps platform.

two years later, *leadership* and *technologies and architecture* were added, so currently the model has seven categories. The DevOps maturity model is often presented as a maturity matrix so that each category has a ranking from one to four points. One point from a certain category means that it is on a poor level from a DevOps point of view whereas four points mean there is not much to improve. The total points, with certain weight given to each category, determines overall DevOps maturity.

Sections from 3.1.1 to 3.1.7 will go through the maturity model's categories. Technological categories (all excluding *leadership* and *organisation and culture*) also go through the automation maturity sections of that particular category and explain why each section is important.

3.1.1 Leadership

Good leadership is a two-way street. On one hand, a good leader can affect a team's ability to deliver code, architect good systems, and apply lean principles to how the team manages its work and develops products (Forsgren et al., 2017). On the other hand, a good leader does not necessarily know what the right decisions are if not provided with a sufficient amount of information.

When the systems of an organisation are comprehensively monitored and the gathered information is shared to all stakeholders, including leaders and other decision-makers, fact based business decisions can be made any time. Tactic knowledge about the current situation is a necessity in order for a good leader to act wisely. When system information is combined with business strategies and goals, decisions are made purely on verified knowledge.

3.1.2 Organisation and Culture

In order to create a working collaborative culture, possibly the whole organisation has to adapt. Continuous communication and cooperation between development, quality assurance, operations and other stakeholders is required to accelerate information sharing and knowledge sharing. Information sharing happens daily through real time messaging or daily meetings and knowledge sharing is ensured with up to date documentation. Section 2.2.1 covers why building a working DevOps culture is important, putting an emphasis on constant cross team collaboration and shared responsibility.

Organisational teams should work independently with clear responsibilities and focus areas. Development knows how to improve and actively works toward improving operational maintenance. Operations minimises configuration errors risen from manual adjustment to environments and gives a detailed information to development when their code not working properly. Between these, quality assurance personnel creates a versatile automation test suite to assure smooth transition from development to production.

3.1.3 Environments and Release

Environment configurations defined as code (also known as Infrastructure as Code), and fluent and frequent releases are not only important for software quality but also for the development flow. When releases are done automatically, personnel in developers and operations can concentrate on software development and system improvements without the constant worry of breaking the build or the environment with every minor change.

The category of *environments and release* concentrates on automating tasks that can often be seen as daily operations work as much as possible. The four automation maturity categories which reside under this are *access right management*, *server management*, *automatic service configuration*, and *development environments*.

Access Rights Management The goal is to centralise access management. Multiple tools that are handled each with separate user management can be frustrating for both for administrators and users: in the worst case scenario administrators need to keep multiple user bases up-to-date and users need to remember multiple usernames and passwords for different platforms. This can create security risks as people might choose simple passwords to help in remembering them or even write passwords down into a visible place.

Server Management Automated, on-demand environment creation ensures that developers run their code in a production-like environment, providing early and constant feedback on the quality of their code. In an ideal situation, any developer can set up his or her own servers without the need to get approval to order them separately. This setup could be done for example with an electronic tool that builds a virtual environment replicating the production as much as possible. These servers would be set up and destroyed immediately with short or nonexistent waiting times and without the need for someone to handle the order.

Automatic Service Configuration Handling server management as code and doing server configurations only through automated scripts ensures consistency and repeatability. These scripts can reduce the number of environment-related faults and defects as the amount of error-prone manual work is minimised. Using version control to store these configuration scripts, ensures that everyone knows where the environment setup can be found and that everyone has the latest version of this setup in use.

Development Environments With consistent development tools, producing documentation and giving support requires less resources. Every developer also knows how to help others setting up the development environment and newcomers are easy to integrate in.

3.1.4 Builds and Continuous Integration

Continuously integrating new changes into existing software produces rapid feedback loops. A CI pipeline works through automated triggers in the version control all the way to packaging the versioned software bundles into a package repository. Automation has a huge impact on accelerating software development (Ståhl and Bosch, 2013; Hilton et al., 2016, as an example). The automation maturity subcategories tackle key enablers of a working development pipeline.

Builds and Compilation Errors in repeated deployments and varying software quality can be reduced by wrapping the software up in an easily movable package. Packing the software enables repeatable installations of code and configurations into a specific environment. These packaging tools can be either environment specific (such as yum, npm, or OneGet) or framework-specific (WAR files on Java, gems on Ruby and so on). An alternative to putting the code into packages could be using deployable containers for example through Docker, Rkt or AMIs. This enables bundling all environment configurations into the same package and then dropping that package into the host machine. In that case the host machine often requires very little configuring itself. The CI should handle software testing and then build a deployable package. This package is stored into version control just as any other code. The package can be retrieved from the version control to the next environment during the CI process, and any time later when a specific version of the software is needed. Also the quality of these packages would be known all the time.

Deployment Deployment can be automated fully with continuous deployment. Deployment consistency is gained when they are done with predefined scripts the same way as for example servers are configured. Automated deployment can trigger after CI pipeline has successfully completed, or manually after the build and changes have been accepted by required quarter.

Package Management Applications are bundled into a deployable package. It is not often sensible to store these kind of packages into traditional version control. Instead, they should be stored with correct tags and version numbers in a sufficient package repository. Depending on the type of the package this repository system could be something like Artifactory[15] or Nexus[16]. When the relations are known, these systems can be easily queried to get information about requirement relations.

Levels of Continuous Integration Continuous integration can be implemented at multiple levels. Ideally, no human interactions are needed between a version control commit and the deployment readiness. Supportive documentation and notifications about the integration results are also provided automatically so relevant personnel can react immediately if so required.

3.1.5 Quality Assurance

The quality assurance category covers all aspects of quality assurance and endorses towards full automation in the area. Often the easiest and the most common parts to automate are static code analysis and unit testing. The prior can be done passively with tools such as SonarQube[17] and the latter is usually done by developers alongside coding. Other parts that require more effort could be automated acceptance testing, regression testing, security testing as well as performance testing.

Use Level of Test Automation Software test coverage measures the thoroughness of testing. The coverage percentage can be calculated in multiple ways, but generally 80–90% coverage is found appropriate. A perfect coverage of 100% might sound a target worth pursuing, but in real life targeting that would probably bring more harm than benefit as it is hard and often unprofitable to achieve. As stated earlier, automating the tests enables constant test quality and provides a tireless test executioner. Tests run automatically after every version control commit can provide fast and valuable feedback for the developer. For example, regression testing after every commit would be in many cases an unfeasible choice without test automation.

Code Quality Checking Good code quality comes from various different aspects. Static code analysis is one way to ensure quality, but it is not necessarily enough to ensure the absence of bugs. For example, pair programming and code review are proven to be very helpful in creating quality code. Ensuring standardised code quality helps in multiple ways so that the software is more maintainable (Baggen et al., 2012).

Automatic Quality Gates Quality is always defined within an organisation. Quality gates are a set of limits that need to be passed for the software to reach

[15]`https://jfrog.com/artifactory/`
[16]`https://www.sonatype.com/nexus-repository-sonatype`
[17]`https://www.sonarqube.org/`

a certain level of quality. This includes not only code but also functional quality. Predefined and automated quality gates are an easy way to define when the feature has reached desired level of quality. Automatic quality gates can also be reached through test driven development (Beck, 2003): this way the tests are written before coding anything and meant to work as a quality gate. Initially, the tests would fail. Later, when software implementations would be done, they would all pass and the software would have reached its desired quality.

Availability of Testing Tools All quality assurance can be very inefficient if a developer cannot run tests as close to a real life scenario as possible on their own development machine. Developers need to have access to all testing tools locally. Transferring code to external systems before crucial smoke tests can be run can be slower and more error prone than doing the same things on a local machine. External test dependencies should be mocked so that the tests can be run without limitations. Frequent local testing promotes higher product quality even further.

3.1.6 Visibility and Reporting

Moving towards a more transparent and visible way of reporting for example progress, errors and requirements makes it possible to find common pain points and get a big picture of the software quality. When general situation is known all the time, it is easier to base decisions on facts. Making improvements is also often more straightforward, as problematic areas are visible at any time. Automation maturity categories included under *visibility and reporting* are *transparency and information sharing*, *information sharing between development and production*, *automatic monitoring of the production environment*, and *version control*.

Transparency and Information Sharing Development and operations do not work well together if data is not shared in an efficient way. When information is shared constantly, all stakeholders can be sure that it is easily available and up to date. It is important to ensure that portfolio level information is available for management at all times.

Information Sharing Between Development and Production In the optimal situation developers and production have the possibility to follow problems in each area at any time. This can reduce common problems encountered on production level as developers are aware of what changes cause problems and how to prevent them in the future.

Automatic Monitoring of the Production Environment Real-time knowledge of production environment components and metrics, such as the amount of disk space being used and CPU load, is important to prevent unplanned malfunctions. This also enables automatic reacting to most situations and easier scalability.

Version Control Usage of source code management is one of the backbones of any software development process. As any other process, it should also have a documented structure and workflow which developers follow. Most version control systems, such as Git[18], have, among other features, a built-in tagging possibility which enable tracking bugs from production all the way back to version control.

3.1.7 Technologies and Architecture

Software development tools should always be chosen to fit the current situation, as well as future development. The tools should also be modern, well documented and well supported, so that developers can completely rely on them. Architectural decisions should also be thought from the point of view of the software, so that business targets can be reached efficiently. When working in an environment with many different tools, integration plays also an important part. For example, linking documentation and requirements as well as version control and CI-pipeline can make the environment easier to understand, easier to follow and more efficient.

Four automation maturity categories are *management of shared components*, *automatic monitoring of dependencies*, *software development tools*, and *technology and architecture*. Each of the categories emphasises on the importance of tool adequacy for each situation and on the possibility for integration between different tools.

Management of Shared Components Shared components, such as virtual images or APIs, are often used by a number of different stakeholders. Updated documentation and centralised storage location, where everyone in need of the components has access to, is of essence to ensure all personnel has the latest versions. Developing on old versions might cause problems when integrating work or even later in production. Possible notifications of updates are provided.

Automatic Monitoring of Dependencies Product components can have a vast amount of dependencies. These dependencies should be monitored automatically to ensure all of them are supported and up to date. Component dependencies are documented with a corresponding version and other information. This information is also available to other team members and other personnel working on the component.

Software Development Tools All shared tools are centralised to common environment. Teams can manage automated project creation inside the team. Tools enable different workflows for each team.

Technology and Architecture Product module technology should be either new or well supported and regularly updated, in order to receive new features and up to date security. The architecture of the system should be structured regarding to the

[18] `https://git-scm.com/`

requirements to best support product needs. If feasible, a microservice architecture is recommended.

3.2 Subjects

The audit data were collected by Eficode from nineteen Finnish-based organisations from various industries and sizes. The data collection was made between fall of 2016 and early spring of 2018 by both personal and group interviews and observing organisation working methods. The interviews were based on a set of pre-defined questions which tried to cover all aspects of the used DevOps maturity model. Research subjects are presented in Table 2. More detailed information about collecting data from each subject is represented in Section 3.3.

A large number of the subject organisations — a total of eight out of nineteen — were concentrated in the software industry, two of them in media, one in construction, two on heavy industry, two on telecommunications or software and one in each of electricity, medical, healthcare and pension. The organisational sizes varied from large (>250 employees) to medium (>50 and ≤250 employees) to small (≤50 employees). Total of six organisations were large, eight medium and five small. The number of interviewees was decided together with the organisation and the interviewer. The goal was to cover as good of a sample as possible.

3.3 Data Collection

This section discusses the different ways data used in this thesis was gathered. There were two different datasets: one gathered earlier through DevOps audits and post-mortem interviews done later based on these audits. The audits and audit reports were not made out by the author of this thesis, but instead by different consultants of Eficode between November 2016 and March 2018. The post-mortem interviews were carried out by the author during March and June 2018.

3.3.1 DevOps Audits

Subject data was collected by performing audits to selected organisations. They were done as interviews with a number of organisation employees or consultants that had been working within the organisations IT projects. Most of the time the interviews were personal but also a few group interviews were conducted. The number of interviewees varied between 4 and 20 person per project, with an average of 8.8 people and median of 8 people. The number of people interviewed in each organisation can be seen from Table 2. The interviewees were chosen, if possible, from different projects inside the organisation and with different backgrounds so the feedback would be as diverse as possible. In some audits, the audited organisation gave a list of interviewees that should be used. If the predefined list was not versatile enough, the auditor could have asked for a change of some of the interviewees. All interviews were held as a semi-structured interviews with an existing set of both

Table 2: Data subjects used in this thesis

				Maturity scoring	
Organisation	Industry	Organisation size	Number of interviewees	DevOps	Automation
MediaCom1	media	medium	5	14	17
MediaCom2	media	medium	6	56	46
ConstCom1	construction	large	6	50	48
HeavyCom1	heavy industry	medium	9	57	56
HeavyCom2	heavy industry	large	20	36	39
TeleCom1	tele	large	11	68	78
TeleCom2	software	large	15	49	46
SoftCom1	software	small	13	50	49
SoftCom2	software	medium	9	94	92
SoftCom3	software	medium	6	43	36
SoftCom4	software	small	5	43	56
SoftCom5	software	small	9	32	33
SoftCom6	software	medium	5	43	48
SoftCom7	software	medium	8	57	62
SoftCom8	software	small	4	50	47
EleCom1	electricity	small	5	82	79
MediCom1	medical	medium	14	71	74
HealthCom1	healthcare	large	10	48	44
PensiCom1	pension	large	8	50	47

open and closed base questions and areas to cover. The goal was to cover all aspects of the DevOps and automation maturity models with each interview.

These interviews were then transcribed, broken down, and the key points were written down either to an audit document or a set of representation slides.[19] These reports were constructed mostly in the same way: an introduction to the organisation and key findings, scoring in the DevOps maturity model and automation model, more detailed information about findings, a suggestion list based on earlier findings about how to proceed in the future to enable stronger DevOps practices and finally a more detailed description of the automation model scoring. In all reports, the key findings could be divided into three categories: the good and the bad things about the current situation, an ideal situation, and improvement suggestions how to move from the current situation to the ideal situation. The findings were grouped in an easily understandable and consistent way so that the problematic points were as clear as possible. This grouping was not necessarily the same in every audit but instead retold the situation with the organisation in hand.

As mentioned, each audit also included a list of improvement suggestions on how to move from the current situation to the ideal situation. This list had about thirty to fifty points. Each of the points had a description of proceeding with the issue and a reference on the amount of effort the implementation would require. These suggestions were also represented in a graphical roadmap that showed the timescale and order in which the suggested implementations should be done. The suggestions were divided into three or four implementation phases on the roadmap. The phases had approximately four to six months between them.

3.3.2 Post-mortem Interviews

A limited set of post-mortem interviews were conducted to selected audited organisations. The idea of these interviews was to gather information about the current situation of the organisations and to help reflect that on the earlier situation. Questions such as the usefulness of the audit, data gathering, the number of implemented suggestions, and rationality of the suggestions were asked. A full list of the post-mortem interview questions can be seen from Appendix A.

The interviews were conducted semi-structured, meaning the questions were planned in advance but the conversations could freely direct towards interesting aspects that came up during the interview. Semi-structured interviews are popular in case studies (Runeson et al., 2012). The questions were chosen to cover different points of view about the done audit as well as possible, and at the same time to get an idea about the implemented suggestions' situation.

Interviewees were chosen so that each of them would have had an extensive view about the DevOps situations development and could answer all of the questions as comprehensively as possible. All of the audited organisations were not chosen for the post-mortem interview either because the audit was conducted too recently of

[19]The audit document refers to a report written in a more paragraph-structured way and the report with the representation slides in a more bullet point-styled report.

because no decent interviewee was available from the organisation in question. Also, one of the excluded organisations audit reports did not include a suggestion roadmap.

The interview questions can be roughly split into three categories: questions about the audit process (questions 3–9), questions about the audit suggestions (questions 10–20), and general questions about the whole process and interview (questions 1, 2, 21 & 22). Two main goals were established. The first was to gain an understanding whether the audit process covered all aspects of the organisations software development practices with suitable comprehension. This goal helps in understanding the usefulness of the audits and verifies that the data gathering was sane and extensive. The second goal was to see which parts of the suggested DevOps process improvements had been implemented, when they had been implemented and why the organisation chose to implement these exact suggestions. Also, possible challenges in these implementations were asked.

Table 3: Summary of the interviews

Organisation	Interviewee title or role	Duration
SoftCom1	Development manager	00:48:16
ConstCom1	Product manager	00:43:15
PensiCom1	Chief Architect	00:56:09
SoftCom5	CTO (Chief Technology Officer)	01:13:17
SoftCom6	Product Line Manager	00:37:42
MediaCom2	Digital development manager	01:09:36
TeleCom1	Product owner	01:01:15
HeavyCom1	VPE (Vice President of Engineering)	00:35:29
SoftCom3	Product manager	00:41:15

Total of nine people were interviewed. Each of the interviewees came from a different audited organisation presented in Table 2. An overview on the people interviewed, their titles or roles in the organisation, and the length of the interviews are presented in Table 3. All interviews were performed between 25th May 2018 and 19th June 2018. Most of the interviews lasted between 35 minutes and an hour, and three of them (with an interviewee from organisations SoftCom5, MediaCom2, and TeleCom1) lasted a bit over an hour.

3.4 Data Analysis

3.4.1 DevOps Audits

The DevOps audit reports were analysed in a similar matter using the ATLAS.ti[20] qualitative data analysis and research software. All of these audits were done with a similar pattern and were based on the same version of the DevOps maturity model. The analysis methods are described later in this chapter.

[20]`https://atlasti.com/`

It has to be taken into account that the initial audit interviews were not conducted by the author of this thesis. This means that the reports conducted from these audits reflect the thoughts and points of view of the original writer and that some points can be seen subjective to him or her. The fact that the reports were premade, also means that the data was prepared and structured in a manner which the original writer found feasible. This includes deciding upon which high-level concepts – or categories, in this case – to write down and which concepts to ignore.

All analysis is based on the reports available and the content was not questioned. In a few cases, more detailed information was retrieved from saved recordings from the original interviews or the interviewers themselves.

Corbin and Strauss (2008, p. 198) mention the intersecting concepts of open coding and axial coding. Open coding means breaking data apart and finding recurring concepts. Axial coding, on the other hand, is the act of drawing relations between these concepts and categories, and binding them together. For this thesis, the directions for axial coding were drawn from the DevOps maturity model and its categories. This was done to help distinguish easily comparable points from each audit.

The analysis started by first scanning through all of the audit reports done by Eficode to check which of them fulfil the sufficient quality and content requirements. This dropped out some of the oldest audits done in 2015 and early 2016 and a total number of nineteen audits were accepted to the research. After this, the qualified audits were encoded with the open coding method. The coding was done so that one or more of the base codes of *current*, *on track*, *problem* and *suggestion* were used alongside each other code. The base code *current* means that the coded piece of data is a reflection of the organisations current situation. *On track* means that the data is a current situation and a beneficial, or a good thing from the DevOps point of view. *Problem* means the opposite of *on track*: it also reflects the current situation but the coded data is seen as a bad thing. Lastly, data encoded with a *suggestion* code is a suggestion done by the auditor on how to improve related practices. The codes *on track* and *problem* are in most cases used alongside *current* to underline whether the situation is seen as a good or a bad thing. These base codes are also closely related to the audit categorisation mentioned in Section 3.3.1: the current situation (*current*), the ideal situation (*on track* and *problem*), and how to move from the current to the ideal situation (*suggestion*). Example codes from a real case are represented in Figure 6.

The use of these base codes with other codes emphasises the context of the other codes. The amount of co-occurrences between base codes and the other codes can be later used to distinguish the most problematic points in the usage of DevOps practices. As an example, if the codes "lack of security tests" and "Problem" would be encoded together in the same data in total of fifteen out of nineteen audits it could be stated that security testing is an issue in many audited organisations. These co-occurrences were counted by filtering out recurrent pairs of code and base code from each audit (so that each pair is acknowledged a maximum of one time per audit) and calculating the total sum of these filtered recurrences in all audits. All unique codes were compared with the base codes on a co-occurrence table to see

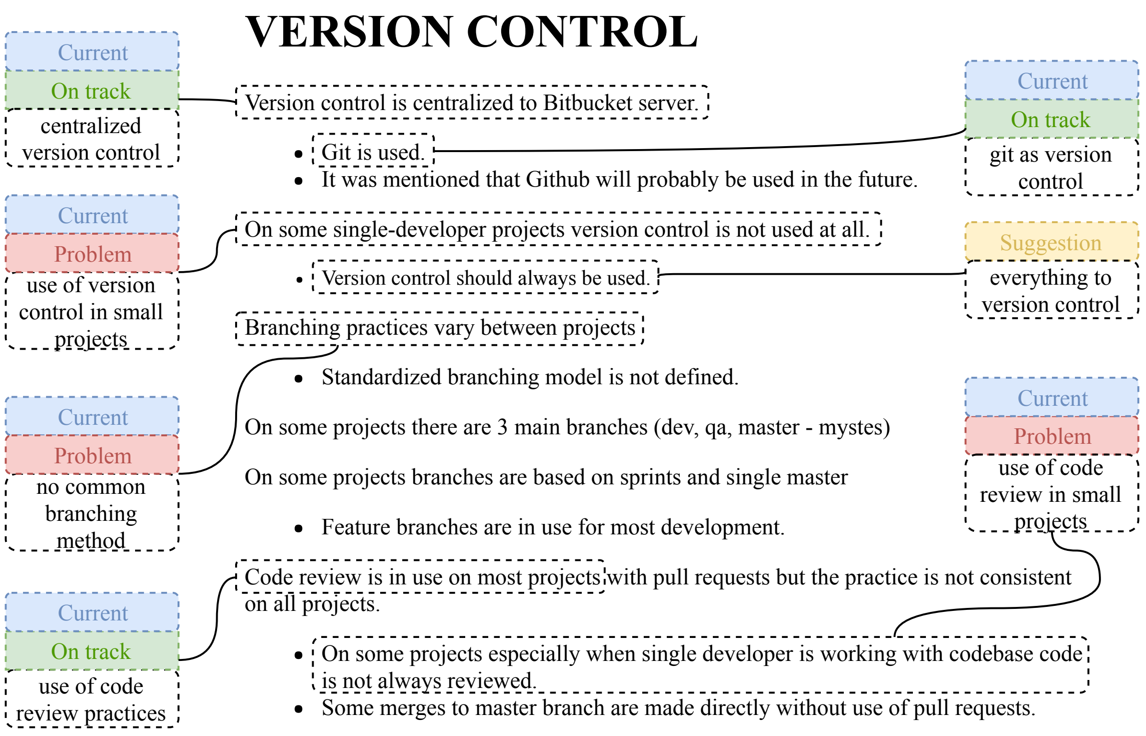
VERSION CONTROL
Current
On track
centralized version control
Version control is centralized to Bitbucket server.
Git is used.
It was mentioned that Github will probably be used in the future.
Current
On track
git as version control
Current
Problem
use of version control in small projects
On some single-developer projects version control is not used at all.
Version control should always be used.
Suggestion
everything to version control
Branching practices vary between projects
Standardized branching model is not defined.
On some projects there are 3 main branches (dev, qa, master - mystes)
On some projects branches are based on sprints and single master
Feature branches are in use for most development.
Current
Problem
no common branching method
Current
Problem
use of code review in small projects
Code review is in use on most projects with pull requests but the practice is not consistent on all projects.
On some projects especially when single developer is working with codebase code is not always reviewed.
Some merges to master branch are made directly without use of pull requests.
Current
On track
use of code review practices

Figure 6: Example coding done to the subject data.

which topics arise from each of them.

A total number of 846 codes were generated over 19 separate audit cases. All of the organisations had vast differences in most areas of their software development process, so 645 of the codes were used only once or twice. All codes were sorted into one of the maturity model categories described in Section 3.1. *Automation* was added as an extra category to summarise the codes related to automation covering partially also the automation maturity. The category was created by adding all codes which included the words "automat*", and "manual*" in it. The asterisk (*) symbolises the wildcard and works as any or no characters. *Automation* is the only category which includes codes that are also in other categories. Each of the codes was also grouped with one or more of *suggestion*, *current*, *problem*, or *on track* depending how the code is seen in the situation.

Many of the theoretical integrations (Corbin and Strauss, 2008, pp. 103–114) that will be discussed in Chapter 4 arise from the previously mentioned co-occurrences. These co-occurrences give a clear (but not bulletproof) and quantitative data about what issues are the most common, which suggestions were the most popular and so on. Especially when dividing the codes into categories according to the DevOps maturity model, a clear understanding about the usage of DevOps practices can be obtained.

3.4.2 Post-mortem Interviews

The post-mortem interviews were all recorded with a dedicated audio recorder. No notes were taken during the interview in order to achieve full concentration. After the interviews, notes were written down based on the recordings. Analysis was done by comparing each interviews notes to each other and drawing conclusions from that. All of the interviewees were fairly pleased with the results of the audits. As most of the audits were done over the course of just a few days with a tight interview schedule, they are not expected to dive into the organisations problems too deeply. Most of the issues raised by the interviewees would probably have been covered with a more in depth analysis, but generally it was not found feasible. The interview feedback implies that the audits were well done and can be taken into account as quality research material.

4 Results

This chapter covers results found through the coding and interview processes. The general findings are covered first, after which, the distinctions between the software industry and other industries are presented. Finally, comprehensiveness of the audits and given suggestions are opened through the interview results. All the findings are later discussed in Chapter 5.

4.1 General Findings

The combined totals of all *suggestion*, *current*, *problem*, and *on track* codes are visualised in Figure 7. Code groups in the graph are divided by maturity model categories. The graph takes into account all individual uses of each code so a specific code can be counted multiple times: for example in case it is used in multiple audits or in case it is grouped in more that one of *suggestion*, *current*, *problem*, or *on track*. This is why the total number of codes visualised here exceeds the number of unique codes, which is 846.

Figure 7 shows that *suggestion* related codes were the most used overall and either the most or the second most used codes in each category. All codes grouped as *problems* and *on tracks* were encoded together with in the *current* group so both formulate a smaller stake than it. Between *problem* and *on track*, *problem* is more common in every single category. This is expected considering the nature of the audits. The *quality assurance* category includes the most codes in every code category. *Environments and release*, and *visibility and reporting* categories include the least.

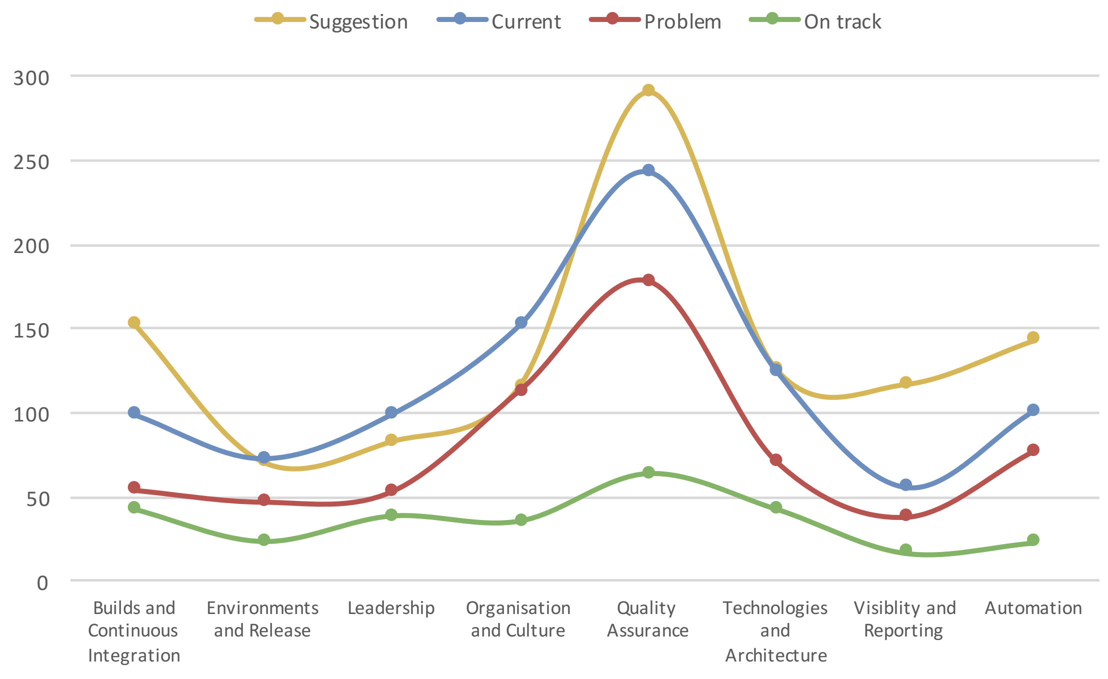

Figure 7: Total code group counts in each code maturity model category.

The overall total of all codes used in all audits, and the total number of unique codes used in all audits are visualised in Figure 8. In both cases, the totals are divided among eight maturity model categories. The overall total numbers are represented on the primary axis by columns and the unique numbers are represented on the secondary axis by lines. The axis, primary and secondary, are scaled so that the relation between these two graphs can be seen. Comparing overall totals to the number of unique codes gives an understanding which categories have a lot of recurring codes. For example, if the number of total codes used is significantly lower than the scaled unique codes line, it implicates that in that specific category unique codes are on average used less times than if the total codes column were higher. In this case, the *environments and release* category has 100 unique codes which are used 215 times in total and the *quality assurance* category has 212 unique codes used 776 times in total. This means that each code is used on average 2.2 times in the *environments and release* category and 3.6 in the *quality assurance* category. The higher average use of codes in a category can tell that the *problems*, *on tracks*, and *suggestions* are common between organisations.

Unique codes categorised as *quality assurance* cover the biggest single portion of all codes: a total number of 212 codes out of 846. The next biggest category is *organisation and culture* with 160 unique codes, following *technologies and architecture* (113), *environments and release* (100), *builds and continuous integration* (95), *leadership* (92), and *visibility and reporting* (74). Codes categorised as *automation* sum to 93 codes.

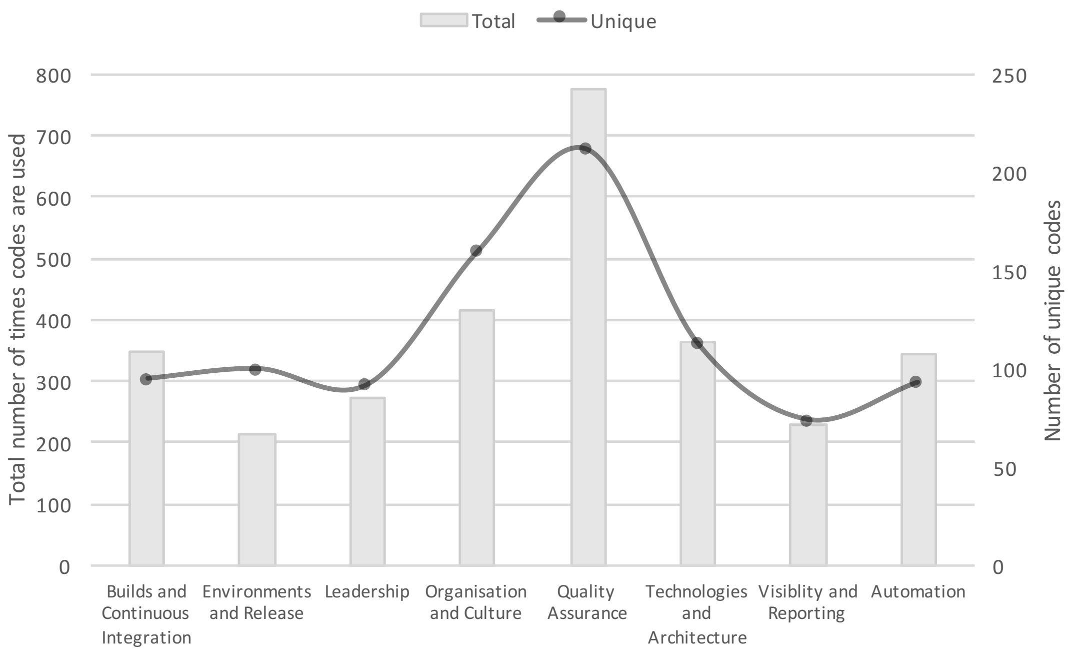

Figure 8: Total number of codes used (primary axis, left) and the number of unique codes (secondary axis, right) in each category.

4.2 Differences Between Industries

Nine out of nineteen of the audited organisations work on the software industry, so a division was made between the software industry and other industries. Figures 9, 10 and 11 represent the differences between these two in amounts of *suggestion*, *problem*, and *on track* codes.

As seen from Figure 9, the *suggestion* related codes follow each other relatively well and no huge observations are made. It can be said that on average the organisations working in the software industry got somewhat more suggestions on most categories. Figure 10 represents the amount *problem* related codes. Most categories are relatively equal. However, two categories stand out: *environments and release*, and *quality assurance*. In *environments and release*, organisations working in other than the software industry have on average over two times more problems than organisations working in the software industry. On the other hand, in *quality assurance* the software industry organisations have on average over 50% more problems. Finally, the *on track* related codes are represented in Figure 11. The average number of *on track* codes per audit is significantly lower than it was in *suggestion* and *problem* codes. Organisations working on other than the software industry stand out very clearly in most categories, whereas in the leftover three categories — *builds and continuous integration*, *environments and release*, and *automation* — the difference is minimal.

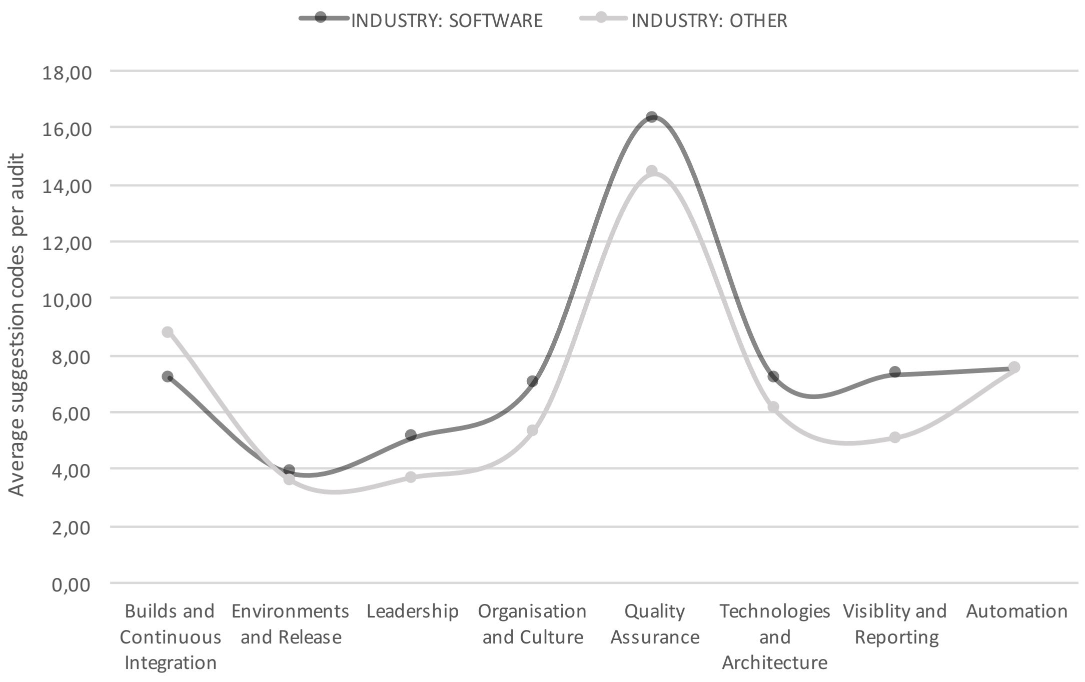

Figure 9: Average number of *suggestion* codes per industry.

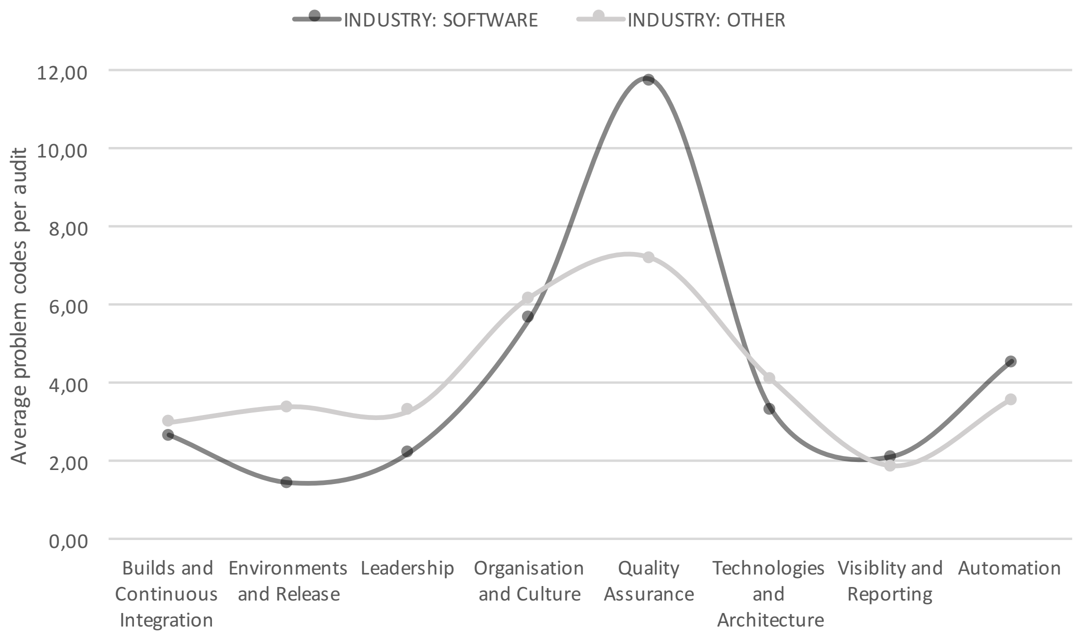

Figure 10: Average number of *problem* codes per industry.

4.3 Most Common Codes

To break down the code categories one level lower, Table 4 includes the most used *suggestion*, *problem*, and *on track* related codes. The top eight or nine codes were included in each code group. None of the codes were used in every single audit. This is because all of the audits were focused on different areas, depending on the situation of the organisation.

As in Figure 8, also the top codes are dominated by the *quality assurance* category whereas totally less coded categories are in minority in the table. The only code that appears on the table more than once is "use of quality gates". That is the most used code among both *suggestion* related codes as well as *problem* related codes. In the *on track* related codes "use of scrum practises" was the most used code, with almost two thirds of the organisations using scrum.

4.4 Comprehensiveness of the Audits

The concept of DevOps was varyingly familiar to the interviewees. Most had heard about the term and had a vague understanding about its content but a few came across it the first time because of the audit. The need for an audition of the organisations DevOps practices came from multiple directions. The product manager at SoftCom3 mentions that they had a known goal to automate their manual testing and eventually found a more versatile overview of their software development practices feasible. The CTO of SoftCom5 says that the need came from organisational growth as they wanted to make sure the software development practices would scale with the grow.

Table 4: Most used codes in divided by code groups. Usage combinations with *suggestion*, *current*, *problem*, and *on track* are shown.

	Code	S	C	P	O	Cat.
Suggestions	use of quality gates	14	8	7	1	(e)
	code analysis as part of workflow	12	0	0	0	(e)
	use of centralized task management	11	3	0	3	(f)
	automated tests part of CI	10	1	0	1	(a), (h)
	automated regression testing	9	1	0	1	(e), (h)
	automated acceptance testing	9	0	0	0	(e), (h)
	production system monitoring	9	6	3	3	(g)
	dashboards to support decision making	9	1	0	1	(c)
	quality gates as part of CI	9	0	0	0	(a)
Problems	use of quality gates	14	8	7	1	(e)
	use of coding conventions	2	8	7	1	(e)
	amount of tools used	0	7	7	0	(f)
	testing done mostly manually	0	7	7	0	(e), (h)
	deployments done manually	0	6	6	0	(a), (h)
	documentation spread into too many places	0	6	6	0	(d)
	lack of regular security tests	0	6	6	0	(e)
	acceptance testing done through intuitition	0	6	6	0	(e)
On tracks	use of scrum practices	3	12	0	12	(c)
	git as version control	7	10	0	10	(f)
	use of code review practices	5	9	3	6	(e)
	centralized documentation	8	6	0	6	(d)
	commits trigger CI	7	6	0	6	(a)
	experimeting with different working methods	0	6	0	6	(d)
	use of static code analysis	8	8	3	5	(e)
	use of real-time chat tools for team communication	4	6	1	5	(f)

S=Suggestion, C=Current, P=Problem, O=On track, Cat.=Categories.
Marked categories are: (a) Builds and Continuous Integration, (b) Environments and Release, (c) Leadership, (d) Organisation and Culture, (e) Quality Assurance, (f) Technologies and Architecture, (g) Visibility and Reporting, and (h) Automation.

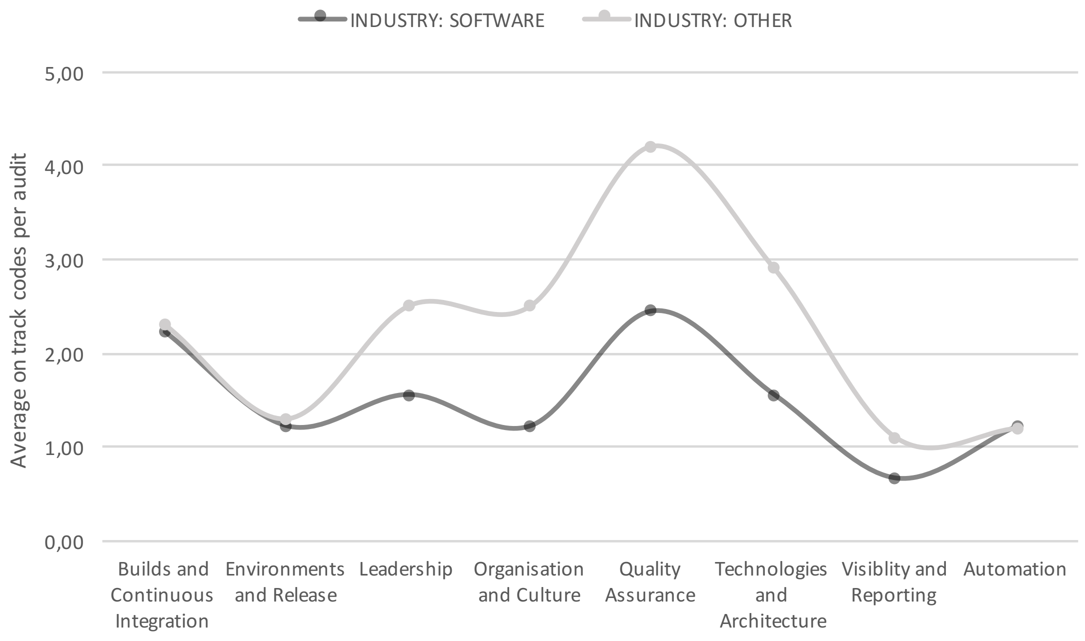

Figure 11: Average number of *on track* codes per industry.

> *The audit left a good aftertaste and gave a good momentum to the teams workflow. - - We gained what we were looking for, which was a viewpoint to our practices and cementing the faith that our way of doing was in the right direction.*
>
> – Product manager, ConstCom1

None of the interviewed organisations were displeased with the quality of the audit. The development manager at SoftCom1 did not come up with any ideas on how the audit would have been better as everything was treated as new and exciting at the time. There was a clear need for it in most organisations and as the chief of development in SoftCom1 mentions, maintaining competitiveness in modern markets requires modern software development practices. According to the product manager of SoftCom3, DevOps practices suggested in the audit gave a good perspective on how modern software development is done and what kind of tools are used. This was also mentioned as the best offering of the whole audit. The interviewees found that the audits covered the DevOps perspective of their software development practices very well. The current state review was realistic and clear pain points were pointed. Some interviewees already acknowledged the problematic areas and for them, the audit worked as confirmer. Some interviewees, however, found a few surprises from the big picture. The interviewee at SoftCom5 represents the audits' reflection to the real situation of the organisation well:

> *When the results came in and were reviewed, they really were on the "brutally honest" level. The results were not romanticised at all and*

eventually we really understood that the organisation is in early stages of good software development practices.

– CTO, SoftCom5

Only a few problematic points to each organisations current situation was presented. The product line manager at SoftCom6 says that a deep enough technical knowledge from their whole environment platform was not gained and the lack of that caused to miss a few important points, but they were still pleased on a general level. The missed points included open source software compliance (which was understood to be an important part of the organisations software development only later) and cybersecurity. The latter is taken into account in many audits, but somehow missed in SoftCom6's. The VPE of HeavyCom1 also said that the use of a few technologies was misrepresented in the report. This might have been caused for example by a misunderstanding during the interviews. The interviewee at SoftCom5 mentions that in some situations the current status could have been interpreted multiple ways and in those situations the interpretation was often done to the "worse" direction in order to motivate doing the enhancements.

The data gathering method as interviews was widely considered as the best possible way. A preliminary questionnaire in addition to the interviews was suggested by the development manager at MediaCom2 saying that it would have helped the interviewer to know better what kind of questions to ask the interviewee, and the interviewee to recall specific details of their development process and system before the actual interview. Some post-mortem interviewees argued the contrary: for example the product manager at SoftCom3 said that preliminary questions would have brought zero benefit.

4.5 Audit Suggestions

Given audit suggestions were taken very positively and almost all were considered important, at least from a certain perspective. In reality, when going through the suggestions together with the interviewees, it turned out some of the suggestions were considered very low priority or not fit to be implemented at all at the current time. For example, the development manager of SoftCom1 said that automated testing was tried, but it did not fit the current development process at all, continuing that automation caused too much maintenance and made the process too heavy to upkeep. Also, the product line manager of SoftCom6 said that they ignored containerisation suggestions for now, as the system architecture was not fit for that.

The suggestions were accompanied by a time schedule roadmap. The roadmap showed a suggestive order and a timeframe for the implementations. Most of the interviewees found the roadmap very indicative and said that although the idea was considered good, it had not been followed very slavishly. For example, the Chief Architect of PensiCom1 said that the schedule was too optimistic and did not fit the organisation at all due to the size of the suggestions and the time required to implement them in a big organisation like theirs. In most cases suggestions were cherry-picked from the audit and placed in the organisation's own process development roadmap. A few organisations however had used the given roadmap as is and started implementing suggestions in the recommended order.

The suggestions were gone through one by one with each interviewee. A general overview of suggestion implementation can be seen from Figure 12. The figure shows that effort towards quality assurance and automation related implementations was done relatively the least. Quality assurance and automation go hand in hand in the suggestions, as most testing-related suggestions are related to test automation. In both cases less than 50% of the suggestions had not even been planned at any level. Effort towards *organisation and culture* as well as *visibility and reporting* related suggestions was done the most, both about 80% of fully done, partially done, or planned.

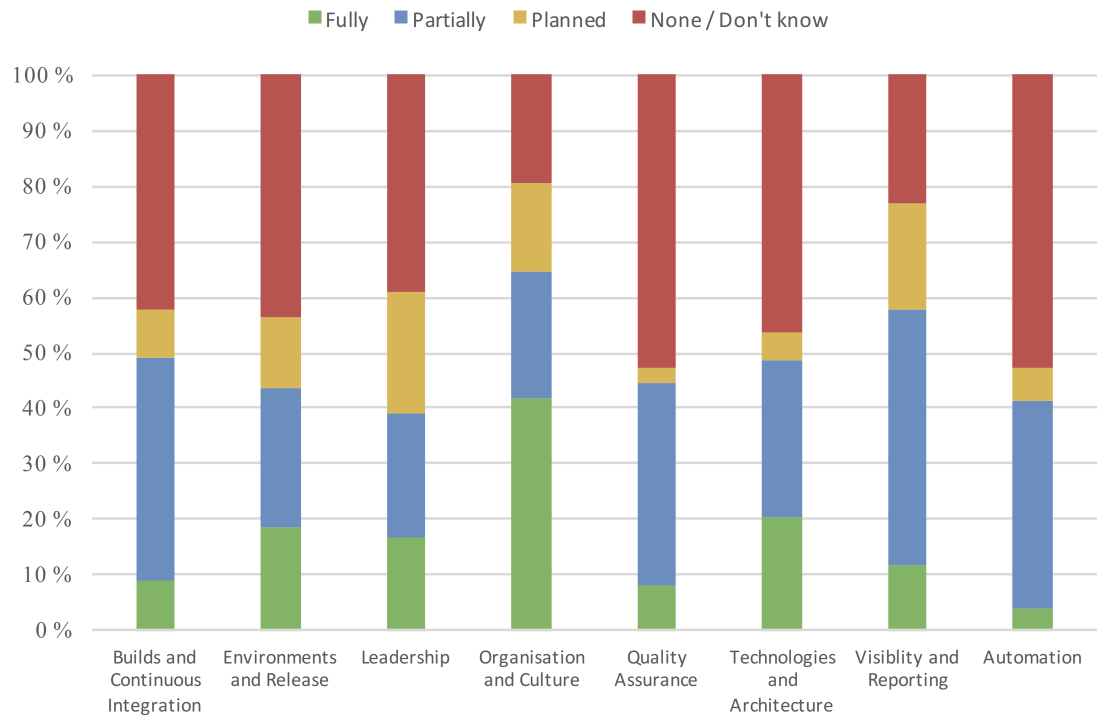

Figure 12: Amount of suggestions implemented in each maturity model category. *Fully* and *Partially* implemented suggestions are somewhat interchangeable as some interviewees argued that "nothing is ever fully ready", refusing to mark implementations as fully done.

Internal issues were by far the biggest blocks from implementing the suggestions. Selling the process improvement ideas to business, and management sides of the organisation in order to get resourcing were mentioned in most interviews as at least a minor roadblock towards a perfect DevOps-process. From the maturity model point of view, the most common mentioned challenges reflected somewhat Figure 12's guidelines. Quality assurance and automation related challenges or issues came up at least among SoftCom1, SoftCom5, MediaCom2, and TeleCom1. As mentioned earlier, SoftCom1 did not find test automation worth the added workload. Lack of pre-existing test automation infrastructure was also mentioned as an issue by the interviewees of MediaCom2 and TeleCom1. Building quality tests from zero, or automating a large number of manual tests brings challenges without the lack of existing knowledge on the issue. On the other hand, when a test automation infrastructure exists it is *"easy to automate even further"*, as quoted by the VPE of HeavyCom1.

Switching to a DevOps-process sometimes requires a change in the state of mind. The old ways of working might be familiar and feel safe, but are not necessarily the easiest or the most efficient way to do things. The attitude towards change was one of the contradictory issues that came up in the interviews. Although both developers and operations personnel are often eager to adapt new technologies and ways to doing work, new is not always felt as better:

> - - *personnel do not tend to adjust to the new culture just with a snap of fingers. Instead, a lot of supervising is required in the beginning to ensure new methods are followed. For example, say that new JIRA-tickets need to be written down in a certain way. In the beginning everything is good, but a few weeks later when in a rush the new ways might not be found so important and shortcuts are taken.*
>
> – CTO, SoftCom5

All interviewees agreed that moving towards a DevOps way of doing software has been beneficial although a little research had been done inside the organisations whether these benefits actually existed. Most of the interviewees mentioned that some process area had gained a boost. The product manager of SoftCom3 said for example that since the organisation changed its manual testing process they suggested way no bugs have been found in the production. The only organisation that could provide something tangible showed some very good results: ConstCom1 had done a lead time analysis in 2017 based on the audit-related changes they made. The analysis showed that their lead times had decreased up to 70%!

5 Discussion

This chapter discusses the research questions of this thesis based on the audit and interview analysis results represented in Chapter 4. Section 5.1 answers RQ1 and Section 5.2 answers RQ2.

5.1 Spread of DevOps

Although DevOps is considered a fairly new concept on the software development scale, the principles are not that far apart from the generally accepted way of developing software. As discussed earlier, the practices behind DevOps are based in manufacturing principles used for many decades. These practices have had the time to optimise themselves to the point where they are now. The optimisation has happened first by trial and error, and later through more researched methods, such as lean.

Some of the interviewees mentioned that they first came across the concept of DevOps with the introduction of the audits. Notwithstanding, a lot of software development practices perceived as the key elements of DevOps were already taken into active use and kept as self-evident truths. One reason for the best practices to spread could have something to do with the fact that software developers are very eager to constantly educate themselves. Multiple interviewees mentioned that personnel in their organisation, both from development and operations, constantly do active research on modern software development practices. This means that if they come across something that they think as beneficial to the current development process, at the very least it is mentioned in a periodical team meeting. DevOps has been one of those beneficial concepts and the organisations that had familiarised themselves with it, have picked the practices that they felt would suit their needs the most.

Table 4 shows most used codes in each code group. In all organisations, only "use of scrum practices" and "git as version control" are used grouped as *on track* in at least half of the cases. These two, and most of the other top *on track* codes, describe practices that were adopted already before DevOps became widely known. More DevOps specific concepts, such as continuous integration, automated quality assurance, and automation related codes, can be found from the *problem* and *suggestion* related top codes. As for example continuous integration and automated quality assurance are technically extremely important parts of DevOps, the observations implicate that organisations have not built their software development process on top of DevOps that much.

A few relations can be drawn from previously done research. One interesting connection between Table 4 and Figure 4 is that code review practices are very high in both rankings. Code review as well as static code analysis are both included the coding standards which is ranked the most popular out of all mentioned DevOps practices in Figure 4. Other similar observations also include security testing being a rather unfocused and continuous integration on the other hand being on the more used and good side of practices.

Generally, it can be said, based on the interviews, that the practices considered as DevOps are widespread across the Finnish software industry while DevOps as a whole concept is not as common as its pieces. Most common pieces used include agile using widely popular agile and version control practices, whereas the least common include automated testing and deployment as well as building quality into the product through quality gates and coding conventions. Nevertheless, the movement is definitely in DevOps' direction. Multiple interviewees mentioned that they value the ideas and practices which DevOps represents and are now looking at their processes more and more from its standpoint.

5.2 The Most and the Least Problematic Areas

5.2.1 Software Organisations Compared to Others

On the contrary what one would think, organisations focused mainly focusing on the software industry are not ahead of others in the quality of their software development process. Instead, when comparing pure software organisations with other organisations which have software development as a minor business, the scale tips a bit in the latter's favour. Figure 11 shows that organisations not focused on the software industry have gained more *on track* codes in five out of eight categories, whereas the residual three categories are even. Relatively, the biggest differences can be seen in the *organisation and culture* category, where other organisations have on average over 100% more *on track* codes per audit than software organisations. Other big difference categories are *leadership*, *quality assurance*, *technologies and architecture*, and *visibility and reporting* – in all which the difference between *on track* codes is from 60% up to 85% in benefit to other organisations.

Problem related codes are somewhat more balanced. As seen from Figure 10, only two categories pop out notably: *environments and release*, and *quality assurance*. In *environments and release*, other than software organisations have over 130% more *problem* related codes than their software focused competitors. In the *quality assurance* category software organisations have on average over 60% more *problem* related codes than organisations focused on other industries. Both of these differences can be explained. The movement towards microarchitecture and more frequent releasing have forced organisations that do their business out of software to adapt much more rapidly. For a software organisation to keep up with the market, it is crucial to release new updates and features often. For an organisation that focuses for example on heavy industry, it might be not and the architecture is kept the same for a longer time. This can create problems for example through legacy technologies, and knowledge which is often personified in these kind of situations. Frequent releases require a more modern DevOps serving architecture which also means that there are less problems from the audit point of view.

The difference in *quality assurance* problems can be concluded from the same cause. In other industries besides software, the need for constant change is much smaller. When the processes and software stay the same longer, tests and other quality assurance is easier to build in. Same tests can be utilised for a longer period of time and as less features are created, the old ones can be covered better. In a software organisation new features are created constantly. More often than not the amount of personnel responsible for writing tests is not scaled enough in relation to the amount of new software created, and quality assurance is left behind.

In spite of the big differences in *problem* and *on track* related codes, the amount of suggestions given to both software and other than software organisations is relatively equal — as shown in Figure 9 earlier. The number of *problem* related codes had big differences especially in the *environments and release*, and *quality assurance* categories but in the suggestions neither industry gets a peak in either category. This might be explained by assuming that when an organisation has a lot of problems in a certain area, the area is not built on a solid base. Then they are given fewer suggestions which require more effort to implement, but eventually build a solid foundation to that particular category. On the contrary when things are going better in some other category and the base is healthy, the organisation is given a large amount of smaller suggestions which are meant to help in taking the software development process even further.

5.2.2 Common Focus Areas

From a general point of view, Figure 7 shows that the biggest focus on the audits was in *quality assurance*. Every group has the most codes in that category. This might have to do with the auditors' preferences but can also be due to the fact that automated quality assurance is a very important part of DevOps. The mere total amount of codes in any category does not necessarily tell anything, except how much each auditor focused on each category. Any actual findings can be made by observing the relations between unique and total codes, and between code groups in each category.

One major relational difference between code groups is in the *organisation and culture* category. There the *suggestion* and *problem* related codes are basically equally popular. One reason for this might be in the schedule of the audit and the content of the category. As explained in Section 2.2.1, a DevOps culture is not built in a day or two. The same way its in depth problems are not fixed in an audit which was made over a short period of time. The main problems in the culture are acknowledged and written down to the audit, but as cultural issues often have a deeper cause than some technical problem, a good solution is hard to suggest. There are general cultural ways of doing things in a DevOps organisation — such as building feature oriented teams instead of task oriented teams — but those solutions may not resolve more unique issues that an organisation might have. Therefore, cultural problems can be identified in the audits just as any other problems, but the solutions (and suggestions) would require a more in depth analysis, and that goes out of the scope of these audits.

When observing the differences between the good (*on track*) and the bad things (*problem*) in Figure 7, two categories — *builds and continuous integration*, and *leadership* — contain the smallest relative differences. This implies that these areas are doing generally well from a DevOps perspective although improvements are suggested. The biggest differences in the good and the bad things are in *organisation and culture*, and *automation*. DevOps introduces a lot of cultural aspects which were not necessarily thought about before in a software organisation. This might explain the large difference in *on track* and *problem* related codes in the cultural category because organisations which have not adopted the DevOps culture are most likely doing things more traditional ways, and these traditional ways can be seen problematic from the audits perspective. Software automation is a fairly new concept, and automating as much as possible is not the goal of every software organisation. The audit perceives most of the manual work as a problem and this could be the reason for a high difference in *on track* and *problem* related codes in that category.

5.2.3 Frequency of Topics and Organisational Priorities

Figure 8 shows well in which categories the observations were more common and which categories they were more unique. The relation between total codes and unique codes represents how many times on average a certain code was used. In the *environments and release*, and *organisation and culture* categories the total column is lower than the unique line, which implies that the observations were more unique. This is expected especially in the cultural category as the audited organisations represent a wide scale of industries and sizes. Culture is greatly affected by these variables, so on average organisations have very unique bad and good things in it. As for *environments and release*, it is hard to name a single cause for the number of unique codes.

Builds and continuous integration, *quality assurance*, and *automation* are probably the most technology tool heavy categories out of all. Hence, it is natural that the same topics, such as utilising or not utilising a tool in a certain field, come up more often in them. *Automation* is also the reason each category is so tool dependant as automation by definition requires the use of some tool. In order to get efficient quality assurance or continuous integration, a lot of automation is required. These three categories also have more or less the same principles in any kind of software development, so it is only natural that same good and bad things come up more commonly in all organisations.

The post-mortem interviews revealed organisational focus areas in implementing given suggestions. Figure 12 shows in which categories the implementation of given suggestions were focused. Clearly, the most focused categories were *organisation and culture*, and *visibility and reporting*. This might be because of two reasons. First reason is that these categories were seen as the most problematic and critical. The second might relate to the fact that these categories are the easiest to grab as a leader. A decision-maker is directly responsible for the organisational culture and might see it as an easy place to make improvements. *Visibility and reporting* on the other hand includes subjects that enhance the flow of information from the software systems up to decision-makers and business. More information enables more knowledge or the systems and problems in other categories are ultimately easier to resolve. The *organisation and culture* category also had a big difference in good and bad grouped codes, so a need for improvement might have been seen. *Suggestions* categorised in the *quality assurance* and *automation* categories were implemented the least. These suggestions formed a large number of suggestions so the sheer number of these suggestions might have been a reason not to implement relatively so many.

5.3 Problems and Future Work

Probably the most problematic area of this Thesis was the usage of previously done audits. They were implemented to provide the best kind of benefit to the audited organisation and would perhaps have worked somewhat better to feed the needs of this academic research if done using it as an end-use target from the very beginning. Now the good things (presented as *on track* related codes) were left in a minority whereas the audits concentrated on problems found in the organisation and aspects that could fix these problems. On the other hand the total work related to all of the audits used would have concluded in such a workload that it could have exceeded the scope of a master's thesis. The DevOps model worked well and was very uniform throughout the audits and was not seen as a problem.

The audits and research done in the scope of this Thesis left it unclear, how much benefit adopting DevOps practices actually bring. The literature review presented some cases where individual practices showed to be beneficial. An interesting topic for future research would be the comparison how much and what kind of benefit an organisation using DevOps actually has comparing with an organisation not using DevOps. Another area for future research could include investigating the reasons why specific areas of the DevOps practices are clearly more ignored than others. This would help in further understanding whether the industry thinks mentioned practices are good in the first place or should some aspects be changed to one direction or another. As mentioned multiple times before, the software industry moves forward on a very rapid pace, and practices which were found beneficial ten years ago may not work now or in the next five to ten years.

5.4 Summary of the Results

It is hard to say when an organisation is using DevOps and when it is not. The audits and post-mortem interviews imply that a lot of practices defined as important parts of DevOps are used in many organisations, but not necessarily because of DevOps. The software development processes vary from organisation to organisation, but the most common DevOps-related methods seem to include good agile and version control practices. These two and most of the other common subjects were widely used already before DevOps became popular. While the concept of DevOps is becoming more and more common the more specific methods are also starting to gain popularity.

RQ2 does not have a clear answer. All of the categories have their own pros and cons. One of the areas where DevOps brings the biggest changes is culture. Cultural issues had the most bad things in relation to the good ones, and on the same time decision-makers were the most eager to fix problems risen in there. Technical tool heavy and automation dependant categories — *builds and continuous integration*, and *quality assurance* — had the most common problems between organisations. Good practices are known to spread, so these issues will probably decrease steadily over time in all places. Finally, when comparing organisations focused in the software industry to other organisations, clear distinctions can be seen. The software industry changes rapidly, which brings issues to those organisations especially in the *quality assurance* category, and the lack of change brings issues to other than software organisations in the *environments and release* category.

6 Conclusions

The DevOps movement saw its first light about a decade ago. During this time it has steadily gained more and more popularity among software related circles: first through the net of dedicated individuals and small conferences like the early DevOpsDays, and later attracting the interest of large multinational corporations, such as Amazon and Netflix. DevOps practices are ideologically based on the co-operating culture of trust and bring a number of benefits compared with the traditional ways of developing software. Its practices are based on the idea that coworkers whom trust each other share information more efficiently, and that automation is the key element in reducing problems as the amount of human interaction is reduced. Automation also enables faster feedback loops, which further produce more information and make more frequent product deployments possible. Decision-makers are able to base their decisions on facts when the large amount of information is split up into clear categories and presented constantly through efficient monitoring practices.

Nowadays the movement and DevOps related practices have also obtained a solid base in the Finnish software industry, and are being seen as beneficial and software quality enhancing. The evolution towards more efficient software development processes is moving constantly forward and as new software is developed now more than ever, good practices are adapted from the very beginning. Currently, software development processes and practices are often used without any connection to DevOps itself although they might be important parts of it. The more knowledge organisations gain of DevOps, the more they seem to realise how much benefits the practices bring as a whole.

References

Allspaw, J.
2012. Blameless postmortems and a just culture. `https://codeascraft.com/2012/05/22/blameless-postmortems/`, Accessed 10.06.2018.

Allspaw, J. and P. Hammond
2009. 10+ deploys per day: Dev and ops cooperation at flickr [video file]. Retrieved from `https://www.youtube.com/watch?v=LdOe18KhtT4`.

Bacchelli, A. and C. Bird
2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, Pp. 712–721. IEEE Press.

Baggen, R., J. P. Correia, K. Schill, and J. Visser
2012. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307.

Balalaie, A., A. Heydarnoori, and P. Jamshidi
2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

Beck, K.
2003. *Test-driven development: by example.* Addison-Wesley Professional.

Behr, K., G. Kim, and G. Spafford
2004. *The Visible Ops Handbook: Implementing ITIL in 4 Practical and Auditable Steps.* Information Technology Process Institute.

Boehm, B. and V. R. Basili
2005. Software defect reduction top 10 list. *Foundations of empirical software engineering: the legacy of Victor R. Basili*, 426(37):426–431.

Choi, J.
2014. The science behind why Jeff Bezos's two-pizza team rule works. `http://blog.idonethis.com/two-pizza-team/`, Accessed 04.06.2018.

Cockburn, A. and L. Williams
2000. The costs and benefits of pair programming. *Extreme programming examined*, 8:223–247.

Cohen, J.
2011. 11 proven practices for more effective, efficient peer code review. `https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/index.html`, Accessed 09.06.2018.

Cohen, J., E. Brown, B. DuRette, and S. Teleki
2006. *Best kept secrets of peer code review.* Smart Bear Somerville.

Corbin, J. and A. Strauss
2008. *Qualitative research.* Thousand Oaks, CA: Sage.

Dekker, S.
2016. *Just culture: Balancing safety and accountability.* CRC Press.

Edwards, D.
2012. The (short) history of DevOps [video file]. Retrieved from `https://www.youtube.com/watch?v=o7-IuYS0iSE`.

Erich, F., C. Amrit, and M. Daneva
2017. A qualitative study of devops usage in practice. *Journal of Software: Evolution and Process*, 29(6).

Forsgren, N.
2015. 2015 state of DevOps report. Technical report, Puppet Labs and IT Revolution.

Forsgren, N., A. Brown, J. Humble, N. Kersten, and G. Kim
2017. 2017 state of DevOps report. Technical report, Puppet and DORA.

Forsgren, N., G. Kim, N. Kerstnen, and J. Humble
2014. 2014 state of DevOps report. Technical report, Puppet Labs.

Hilton, M., T. Tunnell, K. Huang, D. Marinov, and D. Dig
2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Pp. 426–437. ACM.

Holroyd, W.
2014. Using the Toyota Production System to explain DevOps. `http://william.holroyd.name/2014/10/13/using-the-toyota-production-system-to-explain-devops/`, Accessed 06.07.2018.

Huizinga, D. and A. Kolawa
2007. *Automated defect prevention: best practices in software management.* John Wiley & Sons.

Humble, J.
2012. There's no such thing as a "devops team". `https://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team/`, Accessed 16.05.2018.

Humble, J. and D. Farley
2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader).* Pearson Education.

Humble, J. and J. Molesky
2011. Why enterprises must adopt DevOps to enable continuous delivery. *Cutter Business Technology Journal*, 24(8):6–12.

Humble, J., J. Molesky, and B. O'Reilly
2014. *Lean enterprise: How high performance organizations innovate at scale.* O'Reilly Media, Inc.

Kasurinen, J., O. Taipale, and K. Smolander
2010. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010.

Kim, G.
2012. The three ways: The principles underpinning devops. `https://itrevolution.com/the-three-ways-principles-underpinning-devops/`, Accessed 27.05.2018.

Kim, G.
2014. The amazing devops transformation of the hp laserjet firmware team. `https://itrevolution.com/the-amazing-devops-transformation-of-the-hp-laserjet-firmware-team-gary-gruver/`, Accessed 02.06.2018.

Kim, G., K. Behr, and G. Spafford
2013. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*, 1st edition. IT Revolution Press.

Kim, G., P. Debois, J. Willis, and J. Humble
2016. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.* IT Revolution.

Little, C.
2016. Why is there no devops manifesto? `https://devops.com/no-devops-manifesto/`, Accessed 10.06.2018.

Lui, K. M. and K. C. Chan
2006. Pair programming productivity: Novice–novice vs. expert–expert. *International Journal of Human-Computer Studies*, 64(9):915 – 925.

Lurie, N. H. and C. H. Mason
2007. Visual representation: Implications for decision making. *Journal of Marketing*, 71(1):160 – 177.

Lwakatare, L. E., P. Kuvaja, and M. Oivo
2015. Dimensions of DevOps. In *International conference on agile software development*, Pp. 212–217. Springer.

MacCormack, A.
2001. How internet companies build software. *MIT Sloan Management Review*, 42(2):75–84.

Morris, K.
2015. *Infrastructure as Code*. O'Reilly Media, Inc.

Poppendieck, M. and T. Poppendieck
2006. *Implementing Lean Software Development: From Concept to Cash*. Addison-Wesley Professional.

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

Rembetsy, M. and P. McDonnell
2012. Continuously deploying culture: Scaling culture at etsy. `https://www.slideshare.net/mcdonnps/continuously-deploying-culture-scaling-culture-at-etsy-14588485`, Accessed 23.07.2018.

Runeson, P., M. Höst, A. Rainer, and B. Regnell
2012. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons.

Scott, E., D. Pfahl, R. Hebig, R. Heldal, and E. Knauss
2017. Initial results of the helena survey conducted in estonia with comparison to results from sweden and worldwide. In *International Conference on Product-Focused Software Process Improvement*, Pp. 404–412. Springer.

Shingo, S. and A. P. Dillon
1989. *A study of the Toyota production system: From an Industrial Engineering Viewpoint*. CRC Press.

Staats, B. R. and D. M. Upton
2011. Lean knowledge work. *Harward business review*, 89(10):100–110.

Ståhl, D. and J. Bosch
2013. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th iasted international conference on software engineering,(innsbruck, austria, 2013)*, Pp. 736–743.

Turnbull, J.
2014. *The Art of Monitoring*. James Turnbull.

Walls, M.
2013. *Building a DevOps culture*. O'Reilly Media, Inc.

Willis, J.
2010. What devops means to me. https://blog.chef.io/2010/07/16/what-devops-means-to-me/, Accessed 16.05.2018.

Willis, J.
2012a. Devops culture (part 1). http://itrevolution.com/devops-culture-part-1/, Accessed 16.05.2018.

Willis, J.
2012b. Devops culture (part 2). http://itrevolution.com/devops-culture-part-2/, Accessed 05.07.2018.

Wilsenach, R.
2015. Devopsculture. https://martinfowler.com/bliki/DevOpsCulture.html, Accessed 16.05.2018.

A Post-mortem Interview Questions

1. General background in the organisation. (Who? What? How long? etc.)
2. Describe your relation to the auditing process.
3. Describe the quality of the audit.
4. Did the audit results reflect development organisation's real situation?
   4.1. If not, why not? Which aspects are incorrect?
5. Was data gathered in a good manner?
6. Was the result good?
7. Was there something important that the audit missed?
8. What could have been done better?
9. How is the development plan seen in general?
10. Were the suggestions good / relevant?
11. Has the suggestion roadmap been followed?
12. Was the suggestion roadmap reasonable?
13. Have the suggestions been taken into account in decision making?
14. What implementations are done based on the suggestions?
    14.1. Who helped with the implementations?
15. What have been the biggest challenges related to implementing the suggestions?
16. What have been the easiest suggestions to implement?
17. Have the implementations been useful and have they added any value?
18. Have any implementations been done outside of the suggestions?
19. Are any systematic ways used to track roadmap or implementation fulfilment?
20. How could the suggestions have been better?
21. General feedback about the audit.
22. Other things worth noticing about the audit.