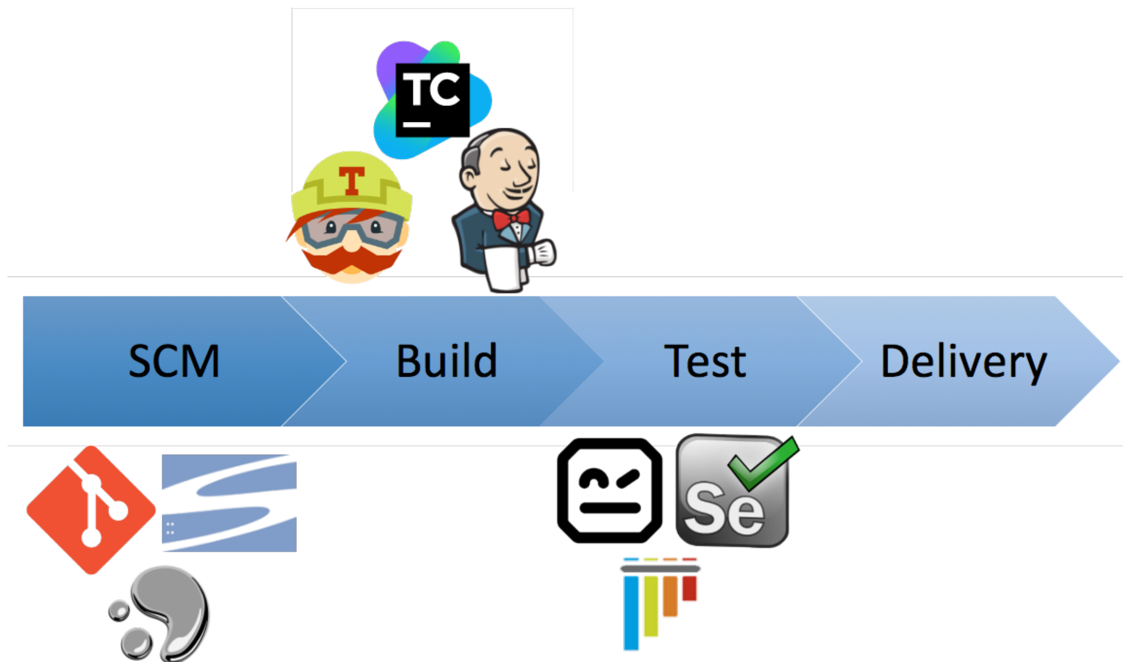
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Designing Continuous Toolchains

Using Proposed Guidelines and Tool Capabilities

Master's thesis in Software Engineering

ELSA MJÖLL BERGSTEINSDÓTTIR
HENRIK HELÉN EDHOLM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Designing Continuous Toolchains

Using Proposed Guidelines and Tool Capabilities

ELSA MJÖLL BERGSTEINSDÓTTIR
HENRIK HELÉN EDHOLM

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Designing Continuous Toolchains
Using Proposed Guidelines and Tool Capabilities
ELSA MJÖLL BERGSTEINSDÓTTIR
HENRIK HELÉN EDHOLM

© ELSA MJÖLL BERGSTEINSDÓTTIR, 2018.
© HENRIK HELÉN EDHOLM, 2018.

Supervisor: Eric Knauss, Computer Science and Engineering
Advisor: Rickard Nilsson, Cybercom Group AB
Examiner: Robert Feldt, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of a continuous toolchain with common tools for each component.

Typeset in LaTeX
Gothenburg, Sweden 2018

Designing Continuous Toolchains
Using Proposed Guidelines and Tool Capabilities
ELSA MJÖLL BERGSTEINSDÓTTIR
HENRIK HELÉN EDHOLM
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Continuous delivery and deployment are both relatively new software practices that can help to deliver software faster. A toolchain is typically constructed by integrating a set of tools and practices to implement these continuous activities. The aim of this study is to help software organizations design continuous toolchains. This is achieved by providing guidelines that can ease the toolchain design as well as by both shedding light on what tool capabilities are desired and which tools are currently available to best support these capabilities.

We conducted a case study where 17 industry professionals provided insights on what tools they use and how they design their continuous toolchains. A follow-up survey was then used to quantify the case study results. The survey gathered insights both on which tool capabilities are the most desired ones and moreover what current tools fulfill these capabilities. The synthesized results from both the case study and survey identified the toolchain structure, desired tool capabilities for each component of the toolchain and ranked them by importance. Recommendations for tools were given based on the tool capabilities importance. Guidelines for toolchain design were synthesized from the case study and provide general suggestions on how to maintain an efficient toolchain.

Designing a continuous toolchain is no simple task. By mapping the capabilities presented in this study with organizational needs software organizations can utilize our findings when selecting suitable tools for their toolchain. To further strengthen the design of their continuous toolchain and its maintenance, companies should incorporate the proposed guidelines into their workflow.

Keywords: Best practices, Common challenges, Continuous software engineering, Continuous toolchains, Guidelines, Software delivery, Tool capabilities, Tool selection.

Acknowledgements

First and foremost we want to thank Eric Knauss, our supervisor at Chalmers University of Technology, for always making time to provide us with valuable feedback and support.

We also want to thank Cybercom Group AB for the opportunity to conduct our thesis at their office and welcoming us with open arms. A special thank you to our industry supervisor Rickard Nilsson for providing us with continuous support and putting us in contact with software industry specialists.

Elsa Mjöll Bergsteinsdóttir & Henrik Helén Edholm, Gothenburg, May 2018

Contents

1 Introduction **1**

1.1 Purpose of the Study . . . 2
1.2 Research Questions . . . 2
1.3 Outline . . . 3

2 Background **5**

2.1 Terminology . . . 5
2.1.1 Continuous Integration . . . 5
2.1.2 Continuous Delivery . . . 6
2.1.3 Continuous Deployment . . . 6
2.1.4 Toolchain . . . 6
2.2 Continuous Toolchain Structure . . . 6
2.2.1 Source Code Management . . . 7
2.2.2 Build . . . 7
2.2.3 Tests . . . 8
2.2.4 Deploy . . . 8
2.3 Common Challenges . . . 9
2.4 Best Practices . . . 9
2.5 Tool Selection Process . . . 10

3 Methods **13**

3.1 Research Strategy . . . 13
3.2 Applied Methods . . . 14
3.2.1 Literature Review . . . 14
3.2.2 Case Study . . . 15
3.2.3 Survey . . . 16
3.3 Threats to Validity . . . 18
3.3.1 Construct Validity . . . 18
3.3.2 Internal Validity . . . 18
3.3.3 External Validity . . . 19
3.3.4 Reliability . . . 19

4 Results **21**

4.1 Demographics . . . 21
4.2 Continuous Toolchain Structure . . . 22
4.3 Common Challenges . . . 23
4.4 Best Practices . . . 25
4.5 Tools, Capabilities & Motivations . . . 26
4.6 Importance of Tool Capabilities . . . 30

4.7 Tool Selection 31

5 Discussion **35**
5.1 Designing a Toolchain (RQ1) 35
5.1.1 Technical Components of a Toolchain (RQ1.1) 35
5.1.2 Selecting the Right Tools (RQ1.2, RQ1.3) 36
5.2 Guidelines Supporting Toolchain Design (RQ2) 38
5.2.1 Comparison of Common Challenges 38
5.2.2 Comparison of Best Practices 39
5.2.3 General Guidelines 41
5.2.4 Source Code Management Guidelines 43
5.2.5 Continuous Integration Guidelines 43
5.2.6 Test Guidelines 44

6 Conclusion **45**

Bibliography **47**

A Abbreviations **I**

B Interview guide **III**

C Survey Questionnaire **VII**

D Xebialabs' Periodic Table of DevOps Tools **XI**

1

Introduction

Software delivery refers to the process of getting software from the developer to the customer or end-user, a process commonly known as *release cycle* [1]. In traditional software development it is not uncommon that release cycles last for years at a time where all developed functionality is delivered at once. These long development cycles introduce risks for the end product in an ever changing market, as it is nearly impossible to predict what products competitors will release in e.g. two years from now [2].

To get a hold of this issue, Agile software development emerged in the mid-1990's [3] and has since taken the world by storm [4]. The idea is to deliver working software on a regular basis with the help of frequent interactions and customer collaboration. An Agile release cycle commonly takes between one and three months [5]. During one release cycle new functionality is implemented and integrated into the end product which always should be fully functional. These agile release cycles can impose delays on features and bug fixes which are completed early in a cycle, unfinished features might however get delivered by the end of the cycle [6]. Organizations can overcome these obstacles by adopting continuous activities which will accelerate their time to market, improve the product quality and more [7]. According to Olsson and Bosch [8] this is a move that software intensive companies must make in order to stay competitive in today's market.

Continuous delivery (CDel) and continuous deployment (CDep) are two of the continuous activities [9] which have received a lot of attention in the past few years [6, 8, 10]. Their popularity comes from their numerous benefits, such as *accelerated time to market*, *improved productivity and efficiency* and *reliable releases* [7]. The two approaches stem from agile development, in particular the practice *continuous integration* (CI) which commonly constitutes of an *"automatically triggered process comprising inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding standard compliance and building deployment packages"* [9]. The main advantage of implementing CI as a practice is to detect problems in the code as early as possible and thereby reduce risks [11]. CI further allows developers to feel confident when pushing changes, knowing that a series of integration tests should detect any potential faults in their code.

CDel is a software development practice achieved when software artifacts can be released to production at any time. CDep differs from CDel by automating the actual deployment of the software to the production environment, thus taking the automation one step further. The challenges of adopting CDel and CDep have been widely reported [7, 8, 12, 13] and can be both of an organizational and technical nature. These challenges are often encountered when implementing a continuous toolchain, a term used in this study to refer to the set of

tools and practices that all connect to achieve full automation of the continuous activities, i.e. CI, CDel and CDep.

CDel and CDep are both relatively new practices with limited research attached to them [9, 14]. Current research primarily focuses on common challenges faced when adopting the practices but not much on what can be done to overcome these challenges [12]. Furthermore, guidelines for continuous toolchains and toolchain design, i.e. tool selection, has not been researched or published [7, 13, 14, 15]. If continuous toolchains are mentioned in literature their context and environment are usually missing [12], which greatly affects the ability to reuse the toolchain design. Hence there does not seem to be any standardized or organized way for how each software organization designs continuous toolchains. Thus, in this study guidelines and desired tool capabilities are provided. The guidelines aims at helping software organizations to design their continuous toolchains by pointing out important aspects which needs to be taken into consideration throughout the design process. The desired tool capabilities and their importance help software organization to select the right tools that fits the organizations use cases. Suitable tool capabilities complement the guidelines by encouraging their usage.

Even though benefits of adopting continuous practices are well known and have a large impact on organizations, little has been published on how to mitigate the challenges commonly faced when designing continuous toolchains. This study is of an exploratory and improving nature and provides guidelines on continuous toolchain design as well as key tool capabilities. The guidelines were derived through interviewing industry professionals to get insights into their thought process when selecting tools for a continuous toolchain and to get characteristics of desired capabilities in a tool. Additionally, a survey was conducted to validate the acquisition of the interviewees collective view and generalize the desired tool key capabilities from the case study.

1.1 Purpose of the Study

The purpose of this thesis is to provide guidelines for designing a continuous toolchain and identify desired capabilities for tools used in such a toolchain. There are a great deal of open source and commercial tools that can be used in a toolchain. As of now there does not seem to be any standardized or organized way for how organizations select tools for their toolchain. By studying literature and interviewing industry professionals this study aims to help software organizations design and select suitable tools in a more structured manner.

1.2 Research Questions

There are many aspects that need to be taken into consideration when designing continuous toolchains. Previous literature has studied the organizational and process related aspects [7, 13, 14], there are however less research that investigates the technical design decisions for the toolchain itself. This study will therefore investigate these decisions in order to provide both academics and practitioners with a more systematic tool selection approach.

RQ1 and its sub-questions aim to present practitioners current approach for designing continuous toolchains.

RQ1: How are continuous toolchains designed in the software industry?
- **RQ1.1:** What are the key technical components that construct a continuous toolchain in the software industry?
- **RQ1.2:** What capabilities guide practitioners when selecting tools for the technical components of a continuous toolchain?
- **RQ1.3:** Which tools best support each technical component of a continuous toolchain?

The main goal of *RQ2* is to provide practitioners and academics with technical guidelines which they should take into consideration when designing a continuous toolchain to ease the tool selection process.

RQ2: What guidelines can support continuous toolchain design?

By answering these questions we will provide insights concerning the thought process of toolchain design among practitioners and what tool capabilities the industry looks for when designing their toolchain. We will give suggestions on tools and key tool capabilities that we recommend when designing a toolchain. These suggestions will be presented in the form of guidelines which will help mitigate common challenges and the designing of a toolchains.

1.3 Outline

Chapter 2 primarily presents related work, but also defines key terminology used throughout this study. This chapter also covers the structure of a continuous toolchain along with a description for its respective technical components as defined by literature. The research methodologies used and the study's threats to validity can be found in Chapter 3. A representation of the gathered results are presented in Chapter 4. The results are then discussed in relation to the research questions in Chapter 5. Lastly, this study is concluded by a summary of the thesis work and suggested future work in Chapter 6.

2

Background

In a fast-changing world beating time-to-market can be essential for software organizations [8]. This requires them to work at a high pace to get their products out to their users as fast as possible. To cope with the speed of the market, organizations have adopted continuous activities in order to release software faster and more frequently [10]. By implementing CDel as a practice in the organization the software is always ready to be released into production and any build of that software that has passed all tests can be released. On the other hand, implementing CDep as a practice aims to deploy software to customers as soon as new code is developed. By releasing software often the risk of each release is reduced [16]. Faster feedback from users is one of the key benefits of adopting CDel and CDep as practices; shorter feedback loops give the ability to validate if the released software is indeed what the user wanted. The practices furthermore provide developers with quick and clear visibility of which software features that have been completed [10, 17]. In this chapter related studies and essential terminology will be introduced to establish a foundation for this study.

2.1 Terminology

In this section terminology will be established to emphasize our definitions of common terms used throughout this study. This is partly done due to the inconsistent definitions of *continuous delivery*, *continuous deployment* and *toolchain* found in literature [9, 11].

2.1.1 Continuous Integration

Continuous Integration (CI) is a practice that originated from Agile software development and Extreme Programming. The practice consists of developers integrating their code into a centralized repository where the integration is verified by an automated build of the software. CI is extensively used in the software industry to develop and release software more rapidly [18], as well as to reduce risks [19]. The aim of CI is to detect problems as early as possible so the software is always in a working state. Humble and Farley [10] describes CI as an enormous step forward in productivity and quality for most projects that adopt the CI practice.

2.1.2 Continuous Delivery

Continuous Delivery (CDel) is a software development practice where a toolchain is built in such a way that the software can be released to production at any time [10, 11]. According to Humble and Farley [10] CDel provides both a faster and safer delivery process by giving everyone involved an accessible and transparent process. The quick feedback provided by CDel is aimed at giving developers a confirmation that their code fulfills the end-users' needs [20]. Where CI integrates and build's the software, CDel deploys it to an environment where it can be released to production at any time.

2.1.3 Continuous Deployment

Continuous Deployment (CDep) is the practice of automatically deploying production ready code into the production environment [9, 10]. The deployment into production happens automatically once the software changes have passed a series of automated tests [21]. By applying CDep as a practice there will rarely be large builds which integrate a vast set of new functionality or changes to the existing functionality at once [22]. Instead there will be multiple updates deployed every day [11]. CDep automatically deploys the software to customers while CDel refers to the ability to deploy the software to an environment [9].

2.1.4 Toolchain

A *toolchain* is essentially a set of tools and practices which helps to achieve full automation of the continuous activities. The continuous toolchain has many names in literature (e.g. deployment pipeline [9, 23], delivery pipeline [14], build pipeline [20] and living build [10]), but no matter the name it is essentially an automated software delivery process [10].

2.2 Continuous Toolchain Structure

When it comes to the structure of continuous toolchains there is no one-size-fits-all solution to the complex problem of designing a toolchain [10]. The structure depends on the environment and the context which the toolchain is supposed to reside in, e.g. if the project is front-end facing or embedded. The front-end facing context refers to software that interacts with the user, e.g. websites and mobile applications, whereas the embedded context refers to software that is integrated into hardware of some kind, e.g. navigation systems and digital watches.

Based on previous literature [1, 7, 10, 12, 16, 18, 20, 24, 25] our synthesis concluded that the key technical components of a continuous toolchain are: *Source Code Management (SCM)*, *build*, *test* and *deploy*. Figure 2.1 presents a graphical representation of these technical components, their flows and how the different continuous practices relate to automation. Each component has a set of activities, practices and responsibilities which serves as a quality gate, i.e. the software cannot move to the next component if it did not fulfill the quality requirements of the previous component.

This section contains an introduction to the continuous delivery toolchain structure by introducing each component that make up a continuous toolchain. The purpose for each component is described along with the key activities which are carried out in each component.

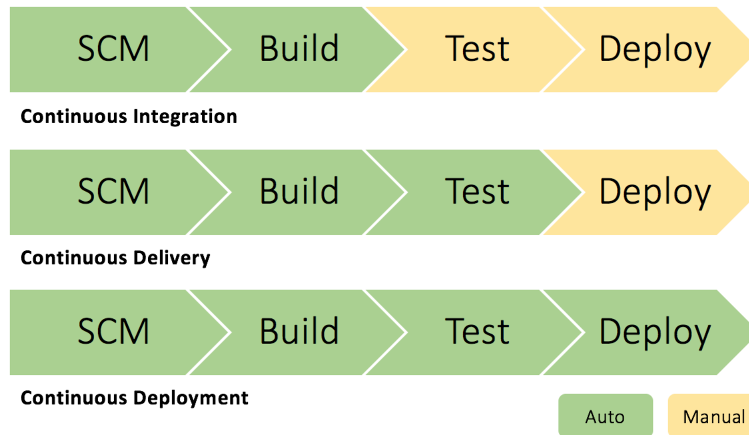


Figure 2.1: Continuous toolchain components as derived from literature

2.2.1 Source Code Management

Source Code Management (SCM), known in practice as version control, refers to a system which stores and manages all software artifacts that constitute a project [10]. These systems provide a project team with a platform where they can collaborate on the same source code and make incremental updates, i.e. commits, to the code base. Their strongest capability is to provide access to all versions of all files ever stored [26], moreover the version control system allows users to attach metadata to files [10]. It is common to run Static Code Analysis (SCA) to enforce coding standards at this step in the toolchain and fail any commits which do not adhere to predefined thresholds. Some useful SCA metrics mentioned by Humble and Farley [10] are test coverage, code style and cyclomatic complexity.

2.2.2 Build

The build component in the toolchain compiles the source code and provides the developers with information on whether the software works as intended through a series of automated tests, e.g. unit tests and integration tests. The build component is often called continuous integration server or *CI* server [10]. The main purpose of a CI server is to detect the most severe integration errors as quickly as possible and to always have the latest executable code available [19]. This is done by compiling the source code in an executable environment by running a build script whenever code has been added or changed in the repository. The CI server compiles and executes the source code and verifies if the code works or if the build failed. Using a CI server allows the team to see the build's status, which increases the project's visibility. The CI server can also run a series of tests focused on SCA, e.g. coding

standards, code duplication and cyclomatic complexity. One of the most important tasks of the CI server is to provide immediate feedback to developers concerning the integration build's results.

2.2.3 Tests

A collection of tests that have a specified set of behaviors is called a test suite. In this component of the toolchain there should be comprehensive and large test suites that cover all of the software requirements for the whole application. The software needs to be tested and built in a way where it can be automatically and safely deployed to production at any time. Rigorous test automation is a key success factor for CDel [27]. The testing component can be split into two general categories: acceptance tests and User Acceptance Tests (UATs).

Acceptance tests are used to verify that the software requirements have been met. Well written acceptance tests are crucial and determine the quality of the toolchain. These tests should cover the entire application. Automated acceptance tests can be functional or non-functional and should be tested by deploying the software to a production-like environment [1, 10].

The UATs validate and verify whether the software meets the end users needs. The purpose of the UATs is to detect faults in the software and to ensure that what the developers developed is what the end-user actually wanted. When UATs are automated the problem domain needs to be very narrow so it can be executed by a piece of software or a tool. The customer involvement is required only when constructing the automated test case, but not during execution [28]. Manual UATs are still very common in the industry and are sometimes necessary because it is difficult to automate the User Interface (UI) of the software [1].

The test suites in this component should cover all requirements of the software to help reduce the time between faults in the software and their detection, with the aim of eliminating root causes more effectively [13]. The automation of these test suites is seen as the way to achieve more frequent delivery of software [8].

2.2.4 Deploy

The purpose of the deploy step is to transfer software artifacts from an environment or a binary storage into the end-product. This transfer is traditionally done every one to six months [16], this irregularity increases risks related to the deployment and furthermore lengthens the feedback loop between customers and developers [16]. These risks can be mitigated by adopting the practice of CDep where deployments are done on a daily basis. In order to reach CDep this step needs to be automated, "*without any interruptions all the way from code to delivery*" [28].

2.3 Common Challenges

In a recent literature review by Laukkanen et al. [12] frequent challenges were found regarding testing, integration, system design and human activities of software development that emerge when CDel is adopted. The most reported challenge in the literature review was testing. Ambiguous test results, tests that randomly fail and time-consuming testing were frequently stated challenges. The testing challenges were caused by both system design challenges and other testing challenges. System design challenges reported were system modularization and unsuitable architecture, challenges caused by system design decisions. That suggests that system design is one of the root causes for CDel adoption challenges. Integration challenges reported were mostly issues that arise when the code is integrated into the code base, i.e. large commits, merge conflicts, broken build and work blockage.

If the above challenges are not taken into account while deriving guidelines it is likely that said guidelines will become impractical. The challenges mentioned are summarized in Table 2.1.

Common Challenges	**Description**
Human activities	Refers to the challenges that relate to human aspects in CDel and CDep, e.g. lack of discipline, experience and motivation.
Integration	Refers to challenges that arise when software changes are integrated to the repository mainline, e.g. broken build, large commits and merge conflicts.
Testing	Refers to the challenges of testing software. The most critical challenges are: time consuming tests, ambiguous test result, flaky tests and multiple platform testing.
System design	Refers to challenges that were caused by system design decisions, e.g. that software architecture limits CDel or that many dependencies between different parts of the software.

Table 2.1: Common challenges for continuous delivery [12]

2.4 Best Practices

Over the years researchers have synthesized best practices used by practitioners of continuous development [10, 21]. Some of the best practices are crucial for successful continuous development, whereas others are mere extensions adding more capabilities to the project at hand.

Rahman et al. [21] have identified 11 software practices used by practitioners of continuous deployment, many of which are enabled by adopting continuous activities. In order for the development to be *continuous* all the activities need to converge through automation.

Thus, the practice of *automated deployment* is a must for CDep whereas *automated testing* is a necessity for all continuous activities. It is the extent of the automated testing that makes the difference between CI and CDel. Another practice which are used by all case companies of Rahman et al. [21] is *repository use*, also known as source code management (see Section 2.2.1). The other eight practices do not make up the bare necessities for continuous development, but can make it easier to retain the practice (*intercommunication* and *shepherding changes*) and extend it with new capabilities (*staging*).

The practices promoted by Humble and Farley [10] have a more technical aspect and are largely based on the authors' own experiences of what works and not. They do however strongly recommend all adopters of continuous deployment to follow these practices "*in order to get the benefits of this approach*" [10]. These practices are: *only build your binaries once*, *deploy the same way to every environment*, *smoke-test your deployments*, *deploy into a copy of production*, *each change should propagate through the pipeline instantly* and *if any part of the pipeline fails, stop the line.*

Software processes in the context of CDel have been widely researched [6, 7, 8]. To implement CDel as a practice in a software organization there are key factors that need to be in place in the organization for continuous delivery to succeed. These factors have been identified by many practitioners, e.g. well established organizational culture, adopting agile principles, cross functional teams and customer involvement [7, 12, 16]. This study will not be exploring software processes further or the key factors that have to be in place for a organization to implement continuous activities. The scope of this study is about providing guidelines for designing a continuous toolchain with a dedicated focus on the technical aspects. Thus, software processes are out of scope for this study. The best practices are summarized in Table 2.2.

2.5 Tool Selection Process

Selecting a new tool for software development can have a critical impact on a project [29]. It is therefore important that this task is taken seriously and thought through, rather than done in an ad-hoc fashion. During the tool selection process it is necessary to take both the adopted software process and context into consideration [30].

When looking into previous research on the topic it becomes apparent that there are many ways for how to select the ideal tools. Pereira et al. [31] states that there are many studies which aim to facilitate tool selection, although they commonly only look at very specific tools or a smaller toolset. In their systematic literature review they instead look at a wider range of tools and compare their main characteristics, which they find through the tools' documentation. Another way to find tool characteristics is to use a theoretical framework to evaluate how well a tool fulfills a certain practice, such as refactoring [32]. Tool selection can also be done in a less formal manner by e.g. analyzing blogs and tutorials [33].

Once a new tool has been selected it is critical that organizational usability standards are developed and agreed upon, since they will mitigate implementation problems [30]. It is further suggested that the adoption of the new tool is done in a small pilot project to test how well the tool integrates with the prior standards. Such an approach makes it possible to get a greater perspective of the tool's impact without harming any critical projects.

Best Practices	**Description**
Automated deployment	Refers to the practice of making software available to end-users automatically.
Automated testing	Refers to the practice of automated techniques to perform various testing activities.
Deploy same way to every environment	Refers to the practice to use the same setup and settings when delivering software, irrespective of target environment.
Intercommunication	Refers to the practice of sharing all necessary development and delivery information among software team members.
Repository use	Refers to the practice to use a repository that uses branching strategies and can push code to the mainline of the repository continuously.
Staging	Refers to the practice of executing a specific set of techniques after software changes are written and before software changes are deployed to end-users, e.g. deliver software in environments that are replica of the production environment.
Stop the line	Refers to the practice that if a delivery to an environment fails the whole team owns that failure. They should stop and fix it before doing anything else; if any part of the toolchain fails, stop the line.
Shepherding changes	Refers to the practice of developers making software changes and being responsible for those software changes throughout the entire delivery process.
Smoke test deployments	Refers to the practice of running smoke tests when the application is deployed, which ensures that the application is up and running.

Table 2.2: Best practices for continuous delivery [10, 21]

3

Methods

This chapter provides a detailed description of the research methodologies used throughout this study. It furthermore describes the motivation behind the selected methods and how literature was found to lay a foundation for the study.

3.1 Research Strategy

In this study three research strategies were used: 1) literature review, 2) case study and 3) survey. Figure 3.1 depicts the flow of these strategies by presenting the relation between the process steps.

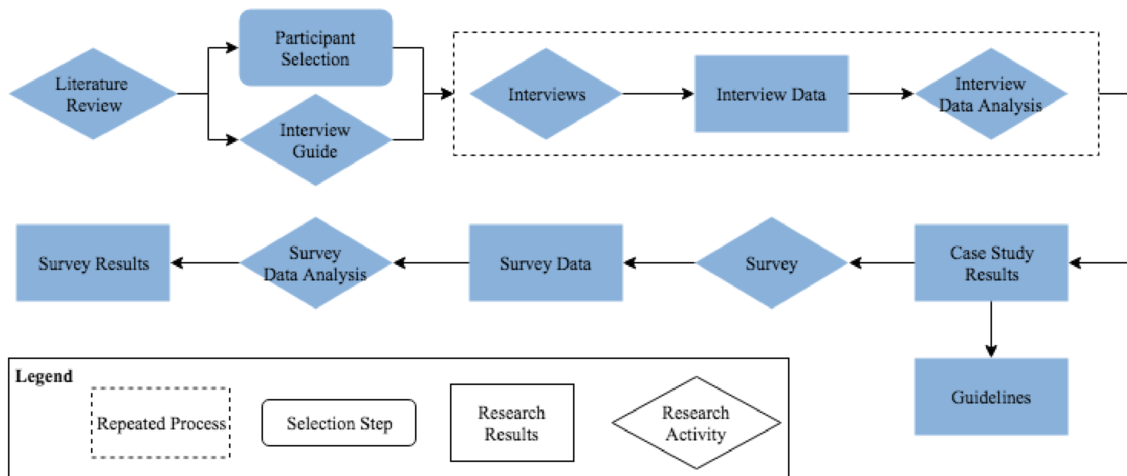


Figure 3.1: Overview of the study's research process

A literature review was conducted in order to establish background on previous literature that has been published on the topic of continuous software engineering.

The case study is of an exploratory and improving nature *"finding out what is happening, seeking new insights and generating ideas and hypotheses for new research"* and *"trying to improve a certain aspect of the studied phenomenon"* [34]. The case study approach was chosen to increase the generalizability of the results. An investigation of how industry professionals design their continuous toolchains in real life settings was performed and its outcome served as a basis for the proposed design guidelines.

A survey is a *"collection of standardized information from a specific population, or some sample from one, usually, but not necessarily by means of a questionnaire or interview"* [34]. In this study a survey was conducted in the form of a questionnaire and sent out to all case study participants in order to be able to validate the capabilities derived from the case study.

3.2 Applied Methods

This section describes this study's applied methods, i.e. *literature review*, *case study* and *survey*. The methods are described in detail in their respective sections. Each subsection starts with a description of the method, followed by an explanation of how it was used in this study.

3.2.1 Literature Review

Literature review is an approach used to analyze literature in a specific area and as a means of developing reasoning about the significance of intended research [35]. In order to establish a baseline for this study a literature review was performed. By reviewing literature about continuous activities an assessment of the current state of research on the topic was attained. This eased identification of the current research gap of formulating guidelines for continuous toolchains. The literature review was furthermore used to determine which methodology approach would be suitable for this study by identifying which strategies were used in other studies on this topic.

The literature review was conducted in a semi-structured manner using Google Scholar as the search engine. Both inclusion criteria and search queries were defined to find the most relevant papers. The inclusion criteria for this study were threefold:

1. Reports must be written in English.
2. Reports must be focused on the software domain.
3. Reports should not be older than 5 years, i.e. 2013.

For the studies related to continuous activities we had three primary search queries, all of which contained the word *software* to support inclusion criterion number two. Our search queries included keywords such as: *deliver**, *toolchain* and *pipeline*. The search queries were furthermore extended by containing synonyms to the used keywords. As an example, *deploy** was expanded with *release*.

The literature review was extended via snowballing of papers found through the search strategy. Snowballing was done by identifying research that had been referenced multiple times in the literature found through the search strategy. For papers found through snowballing, inclusion criterion number three was removed as it would enforce well established research in the field to be overlooked.

3.2.2 Case Study

A case study is a research methodology which provides a deeper understanding of contemporary phenomena by studying the phenomena in their natural context [36]. In this study we interviewed a set of industry professionals to get a good grasp of continuous practices, challenges and how tools are selected. The data collected through the interviews are qualitative and the resulting data set will work as a basis for the output of this method, i.e. guidelines for tool selection in continuous toolchains.

Data Collection

The procedures of the data collection for the case study are of high importance to ensure the study's relevance. When sampling interviewees for this study the aim was to keep maximum variation with regards to system context. Although there are multiple system contexts within the software industry, this study focused on the front-end facing and embedded contexts as these cover a majority of software projects and have a clear distinction. The distinction between front-end facing and embedded is essential since the two contexts face different challenges from one another. Regulations is a challenge that the embedded domain faces, because of heavy regulations it is problematic to deploy software automatically to hardware. A common challenge for front-end facing systems is that their UI's frequently change which makes it difficult to set up automated tests [12].

The initial participants were contacted through convenience, based on our industry supervisor's contacts. In addition all IT-related companies that were attending a local career fair were contacted. Participants were furthermore reached out to via Microsoft Teams and LinkedIn. In order to extend the sample each interview ended by asking the interviewee if they knew anyone who might be able to provide more information. There was a criterion in place to ensure that the interviewee's organization has continuous integration as an established practice. Since the study is targeted towards all continuous activities we also used criterion-e sampling [37] in order to retrieve cases exceeding CI.

All of the interviews were held in a semi-structured fashion with both closed and open-ended questions. This means that an interview guide (see Appendix B) was created and worked as a reference for the researchers, but the order of the questions was dictated by the conversation [36]. All interviews were audio recorded with permission from the participants. In each interview both researchers were present, one held the interview while the other took notes and chimed in wherever necessary to ensure that key aspects were not missed. After an interview had been conducted a transcription of the interview and verification was made by sending it back to the participant to enable correction of raw data as Runeson and Höst suggest [36].

Data Analysis

The analysis of data was done in parallel with the data collection. This is a common practice within qualitative research as it allows new insights to be taken into consideration when collecting upcoming data [36]. The interviews were all transcribed using intelligent verbatim transcription that excludes all fillers, repetitions, laughter and pauses through-

out the interview. To ensure accuracy of the transcripts one researcher worked as the transcriber and the other as the proof-reader for each interview. The proof-reader read through the transcript while listening to the audio recording and took notes of any inconsistencies. If inconsistencies were found in the transcript both researchers sat down together and went over that segment of the recording.

Once a recording was transcribed the researchers coded it in an emergent fashion by attaching memos to each transcript independently, before sitting together to synthesize their respective findings. During these synthesis sessions the two sets of codes (per interview) were merged, categorized and finally tabulated in a spreadsheet. The tabulation provided the researchers with a better overview of the data [36] and was organized as follows: the rows were divided into code categories where each row represent a code, each column represent an interviewee and if any memo was attached to a code it was added as a comment to the specific cell.

The data was then synthesized in relation to **RQ1** and its three related sub-questions. In all cases the synthesis was first conducted separately by the researchers before it was merged. The merging was done by first grouping related statements into categories and then group the categories that overlapped. Firstly, for **RQ1.1** the primary focus point was Question 2.2 in the interview guide (see Appendix B), when the codes were inadequate to provide a full picture of the toolchain the researchers re-read the transcripts to get sufficient detail. Secondly, in the case of **RQ1.2** codes which were categorized in *tool selection process*, *desired tool capabilities* and *view on alternate tools* were taken into consideration. These codes were either related to why a specific tool was selected, what capabilities the interviewees would want to see in a specific type of tool or how a tool relates to another tool. Thirdly, when it comes to **RQ1.3** codes related to all tools mentioned throughout the interviews were synthesized. The tool codes were split into sub-categories, e.g. *SCM*, *CI*, *tests* and *delivery* which have a direct one-to-one mapping with the toolchain components (see Section 4.2).

Lastly, data related to best practices, common challenges and tool selection were synthesized in relation to **RQ2**. The data was coded separately by the researchers into sub-categories equivalent to each toolchain component, in addition *general* and other categories were created and then merged. The guidelines were generated by both researchers from the synthesis depending on how many interviewees mentioned the code category.

3.2.3 Survey

Surveys are one of the most common methods for collecting data and is something that people tend to meet in their daily lives. At first glance it might look like an effortless task to create a survey, but there is more that needs to be done than just assembling the instrument [38].

The survey used in this study (see Appendix C) aims to generalize tool capabilities mentioned in the case study interviews (see Section 3.2.2). Since the interviews were semi-structured and the interview guide emerged throughout the case study, many insights gathered in the latter interviews were impossible to bring up with the interviewees who partook in the earlier interviews. Thus, in order to ensure the acquisition of the interviewees collective views it was necessary to incorporate a method to evaluate the findings.

Survey was the ideal method to use since the main objective is to generalize findings [39], without it recommendations regarding tool selection would be unfeasible.

Instrument Design

Various tool capabilities, desired or not, were mentioned throughout the case study. In several interviews tool capabilities were partly mentioned and in some cases not at all. The tool capabilities were furthermore never ranked against one another. Without such a ranking it is difficult to generate good recommendations on tool selection for practitioners if each toolchain component's key capabilities have not been generalized.

The selection of the sample population was driven by the research questions. The sample selected as survey respondents were the previous case study participants. All of the participants had prior experience with tools in the domain of continuous activities and had at least implemented or used CI as an established practice. They were therefore suited at defining what capabilities exist in the tools they have used and rank them.

The survey instrument was designed from scratch since it is based on the output of the case study. As the survey targeted prior study participants that work close to the topic at hand, limited effort was put into simplifying terms and questions related to the continuous activities. Focus was instead put into making the survey more valuable by clarifying the goals of the survey to the respondents in addition to formulating the response alternatives to enhance comprehensibility and thereby reduce misinterpretation by the respondents. The seven-scaled level of importance Likert scale by Vagias [40] was used to keep the response alternatives standardized and straightforward. By keeping the survey standardized, short and to the point it is less likely that the survey felt intimidating or unclear to the respondents. Anonymity of the survey aimed at making participants feel comfortable to avoid evaluation apprehension, because people are often not comfortable with being evaluated and that can influence the outcome of any study.

Kitchenham and Pfleeger's checklist [38] was used when specifying the information and instructions of the survey. To reduce researcher bias and question order effect the questions were automatically shuffled by the survey tool, i.e. Google Forms[1]. As proposed by Ghazi et al. [39], the survey instrument should be evaluated once designed. The evaluation was performed by expert reviews where academics with past experience in survey research provided feedback on the instrument. The feedback enabled improvements on both language and response options.

Data Collection and Analysis

The survey was sent out to all participants at noon, since respondents tend to answer emails right after their lunch [39]. The survey was open for receiving responses for a week and a reminder was sent out three days before the survey closed to mitigate the risk of a low response rate. The analysis of the survey data was based on Kitchenham and Pfleeger's process [38] and thus included data validation, partitioning the responses and coding of the data.

[1]https://www.google.com/forms/

The means of validating the data was done by reviewing the answers for consistency and completeness. The answered questions were inspected for characteristics of outliers to mitigate the risk of introducing systematic bias. Every question in the survey required an answer and the response rate was 83%, thus not needing to handle incomplete questions. Every question was furthermore evaluated for validity and consistency. Partitioning the responses into subgroups was not required since the survey did not ask any demographic questions because such information had already been obtained through the case study. The data was coded and ordinal scales were converted to its numerical equivalent to analyze the data as if it were simple numerical data. The data coding was done during the design of the instrument rather than the data analysis because the survey was split into clear categories, as suggested by Kitchenham and Pfleeger [38].

3.3 Threats to Validity

When conducting research there are internal and external factors which impact the work in such a way that the research's validity is questioned. Runeson and Höst [36], as well as Easterbrook et al. [41], divide these factors into four distinct categories, namely *construct validity*, *internal validity*, *external validity* and *reliability.*

3.3.1 Construct Validity

Construct validity is a threat if the research design is vague and therefore possible to both measure and interpret in multiple ways [41]. This can for example refer to ambiguous interview questions which lead to too diverse responses that hinder coherent synthesis.

By having continuous discussions concerning the methodology with both the academic and industrial supervisors we feel confident about where the research design landed. One of the biggest construct threats has been ending up with a weak interview guide. This was mitigated by 1) creating an outline, 2) discussing it with the respective supervisors, 3) re-iterating and making a mapping from research questions to interview questions, 4) removing ambiguity and ensuring that questions are formulated open- or closed-ended as intended, 5) pilot testing the interview guide and lastly 6) reordering the questions to increase the interviews flow.

Unsuitable survey design was a threat when the survey was constructed. Such a threat could lead to incorrect measurements and thereby false conclusions. This risk was mitigated by making the goals of the survey clear. The instrument was furthermore validated through expert reviews to raise the confidence and completeness of the survey.

3.3.2 Internal Validity

Internal validity relates to the design of the study with a main focus on whether the results really reflect the data [41]. A common mistake is to misuse statistical data and e.g. make causations out of correlations.

As mentioned in Section 3.2.2 the data analysis of the case study was done independently by the two researchers to minimize the risk of influencing each other's perceptions of the interview, thus strengthening the data set as all data have been analyzed at least twice. The internal validity is furthermore increased by assuring anonymity to the interviewees, as well as encouraging them to scrutinize the transcription for any factual errors, as it allows the interviewees to talk more freely. Lastly, we are fully aware that there are factors which have an impact on continuous toolchains which we intentionally overlook (e.g. software processes). By constantly keeping it in consideration, especially during data analysis, the risk of finding false positive causal relationships is severely reduced.

To mitigate internal validity threats for the survey method Kitchenham and Pfleeger's process [38] was followed during the data analysis. Incomplete questions did not need to be handled and there were no outliers, furthermore the respondents were assured of their anonymity. Since the survey sample were past case study participants with experience of practicing continuous activities all answers were deemed relevant and valid.

Overall the study results relating to tool recommendations might be conserving as a large focus is on what tools the participants already know, rather than what they might need. This could cause this study to only suggest tool capabilities and tools which the participants currently know about. A reduction of this risk was done by deriving generic tool capabilities desired by the interviewees which identifies the software organizations real needs relating to the toolchain, rather than what tool can solve this need.

3.3.3 External Validity

External validity correlates with the generalizability of the results [41]. A common way to increase the external validity is by increasing the sample size. By deepening the sample, i.e. looking at more similar cases, the relevance within the specific context increases. If the research instead broadens it will be generalizable to more contexts.

Since the study's data is based on interviewees working in northern Europe the synthesis of the results will give a good representation of local practitioners outlook on e.g. how to select tools for their toolchains. This does however raise a concern since practitioners in e.g. Silicon Valley, Hong Kong or Tel Aviv might have a completely different thought process when they select their tools.

Insufficient sample size is a validity threat for any survey, Kitchenham and Pfleeger describe that inadequate sample size negatively impacts a survey [38]. The sample population for the survey were the participants of the case study. This introduces a threat of how generalizable the findings would be for a larger population.

3.3.4 Reliability

When it comes to the study's reliability the concern is whether other researchers would end up with the same results if the study was replicated [41].

Through the transparency of the research methodology, founded in a systematic approach,

the study's reliability should be considered rather high. The primary concern is that the study is heavily rooted in west Sweden and might therefore differ from a replicated study in e.g. North America or Asia.

4 Results

In this chapter results gathered from both the conducted case study and the survey are presented. These results are of both qualitative and quantitative nature and are later used in Chapter 5 to answer the research questions. Section 4.1 introduces the study's participants and the characteristics of their case companies. The continuous toolchain structure derived from the case study is described in Section 4.2. Common challenges and best practices of continuous activities are detailed in Sections 4.3 and 4.4. Tools, their desired capabilities and motivations are presented in Section 4.5. In Section 4.6 desired tool capabilities are ranked by importance and Section 4.7 presents the results of tools supporting these desired capabilities.

4.1 Demographics

During the case study there have been 16 conducted interviews with a total of 17 interviewees. Based on the demographic questions asked during the case study interviews we can see that the interviewees were rather diverse in certain aspects, while similar in others. The four aspects we have payed the closest attention to are: *company size*, *context*, *experience* and *level of continuous*.

When it comes to context (see Section 2.2), 29% out of the 17 interviewees worked in the front-end facing context while 71% worked in the embedded context, as can be seen in Figure 4.1. The interviewees had a very diverse work life experience ranging from nine months to 38 years, as presented in Figure 4.2. A vast minority (12%) of the interviewees had a work experience of less than two years. 29% had between three and five years experience, whereas 24% of the interviewees had six to nine years of experience. The largest block (35%) were interviewees with more than ten years of experience. This means that the majority of the case study sample have vast experience of working with software.

EU's standardized subdivision of SMEs distinguishes between small (10-49), medium (50-249) and large companies [42]. Our interviewees cover all of these types with a majority (81%) working at large companies, as can be seen in Figure 4.3.

Level of continuous refers to the degree of which the case companies have implemented the continuous activities, i.e. CI, CDel and CDep. Out of the 16 case companies there was one which did not implement any of the continuous activities, six that implemented CI and nine which implemented CDel, as presented in Figure 4.4. There was however no case company that implemented CDep.

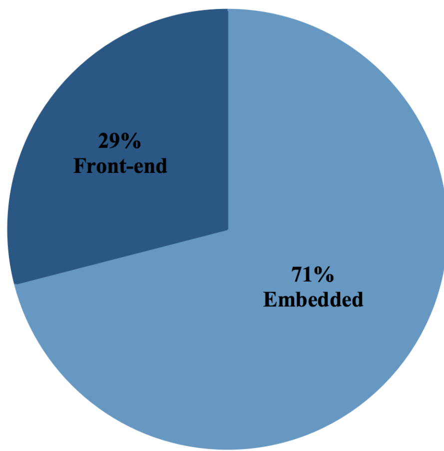


Figure 4.1: Context of participants case companies

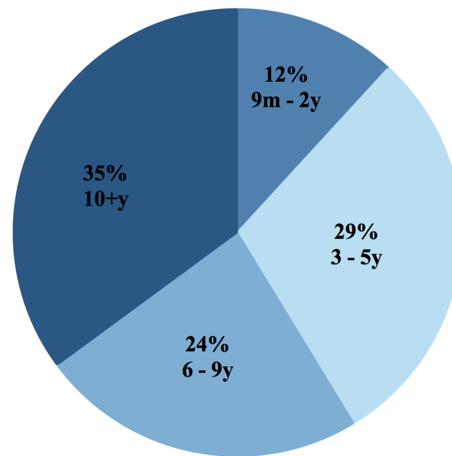


Figure 4.2: Participants work experience

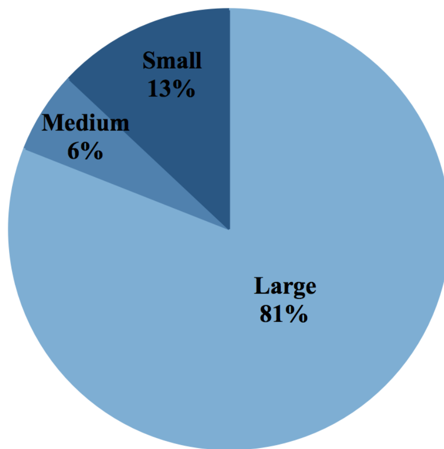


Figure 4.3: Participants case company size

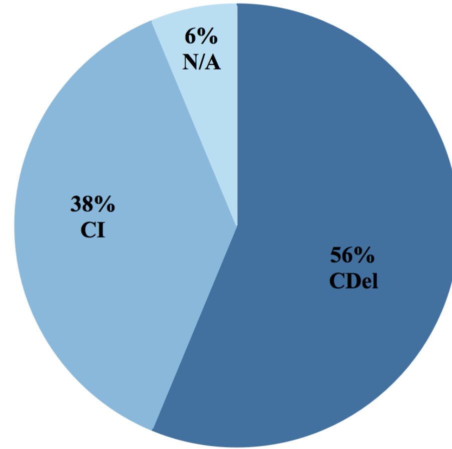


Figure 4.4: Case company implementation of continuous activities

4.2 Continuous Toolchain Structure

In the case study interviews, participants were asked: "*draw a picture while describing your current process from when you write code until it is released*" (see Question 2.2 in Appendix B). While drawing, the participants often mentioned tools that they use in each stage of their process and furthermore provided a thorough elaboration of their toolchain's structure.

The similarity between the different toolchains illustrated an evident pattern; 15 of the interviewees described an *SCM* component followed by a *build* component. The same 15 interviewees described a *test* component thereafter. 13 out of these 15 interviewees mentioned some sort of *delivery* component subsequent to the tests. Although there were some other, primarily test, components described by the interviewees there were not a majority which described them. Thus, the general outline of the toolchain is as depicted in Figure 4.5.

Figure 4.5: Continuous toolchain components as derived from case study interviews

4.3 Common Challenges

When conducting the interviews, participants frequently discussed challenges that they commonly face when implementing continuous activities and designing a toolchain. A challenge was considered relevant and common if it was mentioned by more than two interviewees, as this indicates a pattern [43].

The challenge of *excessive manual testing* related to the **test** component of the toolchain was frequently discussed. This was done in terms of both time consumption and difficulty when requirements are changing throughout development. One of the interviewees said that it is "*hard to have manual tests if everything changes all the time*". Furthermore, excessive manual UI testing does not provide constant steady feedback on quality and performance.

Integration challenges concerning both the **SCM** and the **Build** component of the toolchain were considered unanimously to be the most difficult hurdle to overcome in a continuous toolchain. Integration issues regarding open source software were also mentioned; "*Open source projects are lagging behind, so often we find defects and integration issues*".

A **general** challenge that was brought up multiple times concerned the *mindset* of the employees as well as the organization as a whole. One of the interviewees stated that: "*A big part of the organization is thinking Waterfall and puts fixed deadlines on the agile teams*". Another went so far as to say that: "*... the organization is not prepared to be a modern software organization*". Generally it is difficult to "*change the way of working for people who have grown accustomed*" to other ways. Often the interviewees felt that the organizations were not focusing enough on educating their employees and thereby not managing to establish a continuous mindset.

Another commonly faced challenge is *platform issues*. The interviewees mentioned that it is "*really frustrating sometimes when you move files from one operating system to another*". One of them stated that they commonly face "*license issues with Windows build environments*", while one said that "*if you are working on Windows you have big problems*".

Regulations refers to the challenge of having software fulfill regulations and standards before deployment of software is carried out. This challenge was frequently discussed by the interviewees, mostly in terms of safety critical software. As stated by an interviewee: "*A lot of regulations need to be fulfilled, so this is really another toolchain, just for the tests*". One interviewee explained that it was not plausible for them to have CDep because they were under regulations and would always need a manual step for sign-offs: "*we are*

under regulations and cannot get around some verification's and document sign-offs."

Overall *toolchain complexity* is a challenge that developers face when implementing continuous activities. Many sets of tools and practices need to be integrated, configured and maintained that all add to the toolchain complexity. As expressed by our interviewees: "*Quite difficult to figure out how to run CI, CDel things*" and "*Big problems are presented when you try to scale the toolchain*". There are however many other activities that can contribute to the toolchain complexity.

Visualization is a challenge for many organizations. It becomes really problematic to visualize the overall flow when there are a lot of pipelines flowing through the chain. If the developers cannot follow their changes they will have a tougher time seeing the whole picture, according to our interviewees. As explained by one of the interviewees: " *When changing code that is affecting a lot of things, if you can't visualize that it's a huge problem.*"

The challenges mentioned above are summarized in Table 4.1 and are primarily related to the toolchain and its components. These challenges are further discussed in Section 5.2.1 where a comparison to challenges found in the literature is made.

Common Challenges	**Description**
Excessive manual testing	Refers to the challenge to have too many manual tests in the software. With ever changing requirements manual testing does not provide consistent feedback on quality and performance of the software.
Integration	Refers to the challenge of continuously integrating software changes.
Mindset	Refers to the challenge to establish a continuous mindset for software organizations and employees as well as lack of experience practicing CDel and CDep.
Platform issues	Refers to the shortcomings of using certain platforms for continuous activities, but can also relate to e.g. licensing.
Regulations	Refers to the challenge when software needs to fulfill certain regulations.
Toolchain complexity	Refers to the challenge of overall toolchain complexity, commonly introduced when scaling the toolchain.
Visualization	Refers to the challenge to visualize the overall flow of the toolchain from when a developer commits until the code is deployed.

Table 4.1: Common challenges from interviews

4.4 Best Practices

From the case study interviews, interviewees frequently described best practices on efficient and effective ways to accomplish continuous activities. A best practice was considered relevant if it was expressed by two or more interviewees, as this indicates a pattern [43]. The tools that make up a toolchain need to follow these best practices to a degree and more importantly not hinder them. Best practices mentioned during the case study are summarized in Table 4.2 and further discussed in relation to best practices found in literature in Section 5.2.2.

Merge often and *mainline should always be in a working state* are two best practices that relate to the **SCM** component of the toolchain. The former refers to the practice of integrating software changes often, as stated by interviewee: "*You want to fail fast, merge often*". The latter refers to the practice that the mainline in an SCM system must be buildable and should furthermore always be in a working state.

The **test** component of the toolchain uses *multiple environments* as a practice that refers to testing the software in different environments that test several different capabilities of the software. Furthermore, the practice *manual testing is required for safety critical software* refers to always manually test safety critical software before deploying it to users. One interviewee stated that: "*The product is not finished before manual tests have it approved*".

Best practices that relate to the **general** aspect of the toolchain were also identified. The practice to *accept that some changes will fail* was brought up by several interviewees: "*One important thing is to let things fail. You need to be able to trust the toolchain*", "*Commits will often fail*" and "*You have to be okay with commits always failing*". This practice acknowledges that software changes that go through the toolchain will fail occasionally and that both developers as well as stakeholders need to accept this reality.

Automate as much as possible refers to the practice to automate as many components of the toolchain as possible. This practice was mentioned by several interviewees: "*Build for full automation*", "*Automate as much as possible*" and "*The overall chain needs to be automated*". Most interviewees recognized the need for toolchain automation to be able to implement continuous activities.

Rollback support refers to the practice of automatically reverting the toolchain to a previous working state if the toolchain fails. According to an interviewee: "*Every good software project should have the ability to revert the code*". By having the chain automatically reverted to a previously working state the downtime of the toolchain is reduced compared to having manual interventions when the toolchain fails.

Lastly *visualizing the flow of the toolchain* refers to the practice of being able to visually observe the flow of the toolchain, from when a developer commits until the code is deployed. This was mentioned frequently as a challenge (see Section 4.3), but it was also prescribed by several interviewees as a best practice. One interviewee described the practice as: "*Visualize the flow for each change so that you can follow in real-time what happens to your change and everybody else's*".

Best Practices	Description
Mainline should always be in a working state	Refers to the practice that the mainline of a repository should always be in a working state and must be buildable.
Merge often	Refers to the practice of integrating software changes frequently.
Test in multiple environments	Refers to the practice of testing software in different environments that test several different capabilities of the software.
Manual testing required for safety critical software	Refers to the practice to always manually test safety critical software before deploying it to users.
Accept that some changes will fail	Refers to the practice that developers and stakeholders need to acknowledge that software changes that go through the toolchain will fail occasionally and accept it.
Automate as much as possible	Refers to the practice to automate as many components of the toolchain as possible.
Rollback support	Refers to the practice of reverting the toolchain to a previous state in case of failure.
Visualizing the flow of the toolchain	Refers to the practice of being able to visually observe the flow of the toolchain, originating from when a developer commits until the code is deployed in the end-product.

Table 4.2: Best practices from interviews

4.5 Tools, Capabilities & Motivations

In this section tools and their desired capabilities are presented in Table 4.3. The desired capabilities are explained in Tables 4.4 to 4.7. This section furthermore mentions common motivations for selecting tools, provided by the case study participants. The tools mentioned in this section were selected based on commonality. A tool were deemed common if it was mentioned by more than 30% of the interviewees, irrespective of context.

Component	Tool Capabilities
SCM	*Actively maintained*, *Big user base*, *Easy to work with*, *Powerful branching strategies*, *Powerful integration with a CI*, *Powerful merging system*
CI	*Actively maintained*, *Different configuration options*, *Minimum setup*, *Modern UI*, *Plugin support*, *Scalable*, *Platform independent*, *Visualize the flow*
Test	*Keyword-driven*, *Low complexity*
Delivery	*Rollback support*, *Support customer deliveries*

Table 4.3: Capabilities mentioned in the case study

SCM Tool Capabilities	Description
Actively maintained	For an SCM tool to be actively maintained there needs to be regular compatibility and feature updates, bug fixes and patches being released.
Big user base	For an SCM tool to have a big user base there needs to be a great amount of users using the tool on a daily basis.
Ease of use	An SCM tool that is easy to work with in many different aspects subjectively or not, e.g. simple learning curve and easy configurations.
Powerful branching strategies	Branching strategies is a feature where two or more parallel versions of the same software is being developed at the same time on different branches. These separate versions have the same repository, but their development are kept separate until they are merged.
Powerful CI integration	A feature of an SCM tool is that it can integrate well with different CI systems.
Powerful merging system	Merging system is a system that is very powerful when integrating software changes.

Table 4.4: Description of SCM tool capabilities

CI Tool Capabilities	Description
Actively maintained	For a CI tool to be actively maintained there needs to be regular compatibility and feature updates, bug fixes and patches being released.
Different configuration options	A CI server should have various configuration options, e.g. managing the communication between integrated tools and configure jobs.
Minimum setup	A CI server has minimum setup if it is easy to get it up and running rapidly.
Modern UI	The user interface of a CI tool should have a modern design.
Platform independent	The CI server is platform independent if it works with different operating systems.
Plugin support	The CI server should support plugins and be extendable.
Scalable	A CI server is scalable if it can scale up and down in the amount of jobs it handles.
Visualize the flow	The CI server should visualize the flow (build jobs) of the delivery.

Table 4.5: Description of CI tool capabilities

When selecting **SCM** tools it is common to go for legacy tools already in use by the organization. The main motivation seems to be that it is hard to make the move to something modern and potentially better. When referring to Subversion one of the interviews said that: "*It's legacy and hard to get rid of*". The decision can sometimes be based on that

Test Tool Capabilities	**Description**
Keyword-driven	A testing tool is keyword-driven if a tool abstracts the way test cases are written and executed by using action word based testing.
Low complexity	A testing tool has low complexity if conducting activities that concerns both the setup and user interface are perceived as simple to carry out.

Table 4.6: Description of test tool capabilities

Deployment Tool Capabilities	**Description**
Rollback support	Rollback is an operation in a deployment tool that returns it to a previous working state if the current deployment fails.
Support customer deliveries	Deployment tool supports customer deliveries if it can deploy software to customers automatically.

Table 4.7: Description of deployment tool capabilities

the company already has invested resources in learning the tool and has thereby gained a lot of experience, which is another reason for selecting an SCM tool. Nonetheless, the most common reason for selecting an SCM tool is that it is superior to its competition. It is however important to recognize the aspects in which a tool is superior. Based on the interview data it is apparent that there are two SCM tools which stand out as the most common ones, i.e. Git and Subversion as can be seen in Figure 4.6. Git was mentioned in nine out of the twelve embedded interviews and all of the four front-end interviews, Subversion on the other hand was also mentioned in nine out of the twelve embedded interviews but only in one of the four front-end interviews. According to most interviewees, Git outperforms Subversion since it has a better *merging system* and a stable *user base growth*. As described by an interviewee: "*Git, many people know it. We used SVN for a long time but that is going down in popularity, because it wasn't as powerful*". Concerning the *ease of use* there is conflicting views where some people deem Subversion easier to use than Git and vice versa.

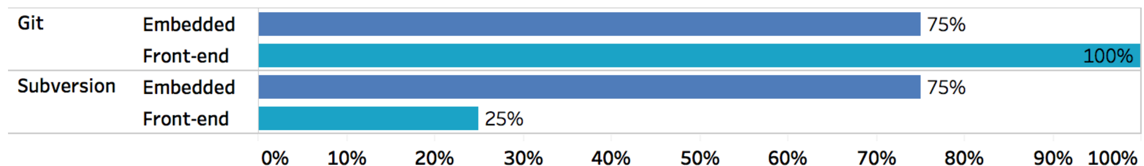


Figure 4.6: Mention frequency of SCM tools

Looking at **CI** tools there were two tools which clearly stood out from the rest in popularity, presented in Figure 4.7. The by far most mentioned was Jenkins, which was mentioned in all embedded interviews and in three out of the four front-end interviews. The interviewees see Jenkins as the current industry standard, in both contexts. The second most mentioned tool for CI was GoCD, mentioned in five embedded interviews and three front-

end facing. The interviewees seem to consider GoCD superior to Jenkins in regard to having better *visualization* capability. Regardless of context, experience seems to be a big motivational factor for choosing a CI tool, two interviewees stated, "*The decision was entirely based on the fact that I have a lot of experience with it*"and "*Tools that we are using currently depends on if we have used it before and have knowledge*". Another important motivational factor for selecting CI tool is that it needs to be open source.

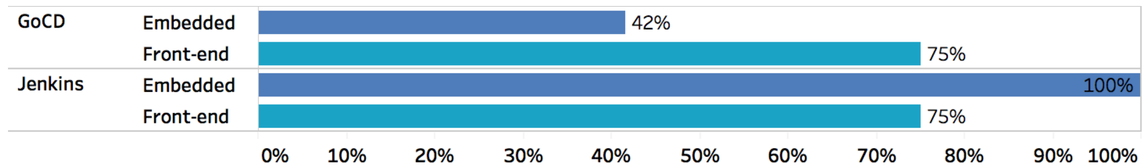


Figure 4.7: Mention frequency of CI tools

Test tools mentioned throughout the interviews have almost all been context specific, i.e. the tools have solely been mentioned by cases in either the embedded context or the front-end facing context. As presented in Figure 4.8, the two most common test tools in the embedded context are Robot Framework and Google Test, whereas Selenium is the most common tool in the front-end facing context. In an embedded context it seems common to select test tools based on the knowledge base, whereas front-end test tools are selected due to their general popularity. It is however not uncommon that certain test tools provide very context specific capabilities and is therefore the go-to option.

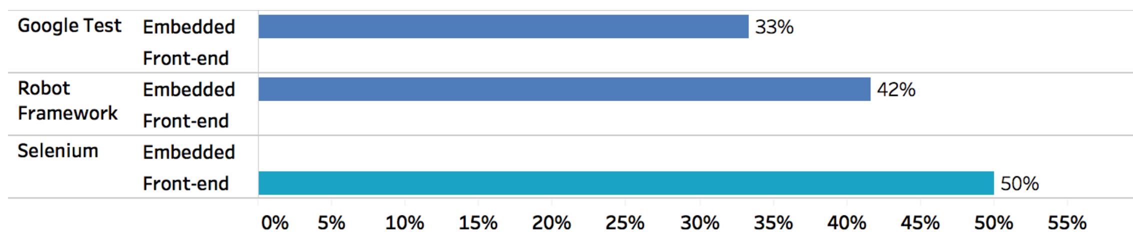


Figure 4.8: Mention frequency of test tools

Tools used during CDel are often intertwined with CI and deployment tools, their only purpose is not only to move software from one place to another. Tools mentioned in the interviews regarding delivery were tools used for all kinds of purposes, e.g. containerization, CI, storing binaries or deploying software. It is therefore impractical to generate dedicated tool capabilities for the **delivery** component.

When it comes to tools for **deploying** software there has been no tool which has emerged as the common ground. The interviewees have mentioned tools but none of them have been mentioned by more than one interviewee.

Within the front-end context the general viewpoint is partly that a tool should be popular and have an open source community surrounding it. For embedded projects the selection process seems to be more difficult and one of the first decisions is whether to use freeware or pay for the software. The embedded teams must furthermore create their own tools in order to attain desired capabilities, not because they want to but because there are no alternatives readily available.

When it comes to who decides on the tools to use there is a difference between the companies, irrespective of context. In six cases the organization or IT department decides on the tools, in some of these cases it is however possible for the teams to request what tools they would want to have. The decision for what tools to use is taken on a team level in four of the cases.

4.6 Importance of Tool Capabilities

This section presents the results from the survey that validates capabilities from the case study (see Section 4.5) that are desired for a tool in each component of the toolchain. Each capability was rated using a Likert scale of importance [40] mentioned in Section 3.2.3.

Important **SCM** capabilities according to the survey results can be seen in Figure 4.9. SCM capability that is of highest importance is *powerful merging system*, with 64% of respondents considering it to be extremely important. *Actively maintained* is deemed very important, whereas *powerful branching strategies* and *powerful CI integration* are more spread out. The last two capabilities *big user base* and *ease of use* are of moderate importance according to the respondents.

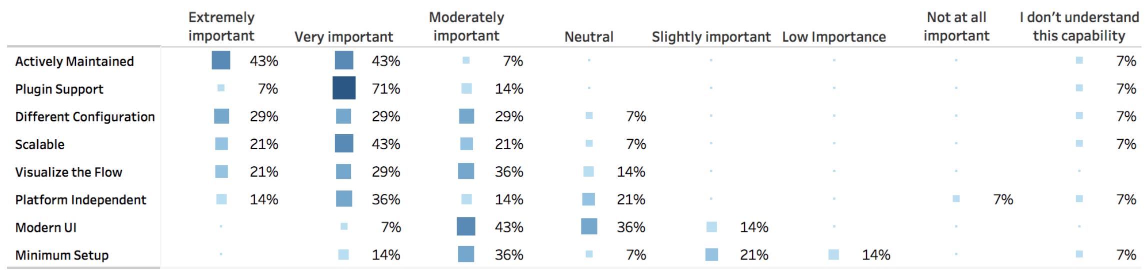
	Extremely important	Very important	Moderately important	Neutral	Slight Importance	Low Importance	Not at all important	I don't understand this capability
Powerful merging system	64%	21%	7%	7%				
Actively maintained	29%	43%	14%	14%				
Powerful CI Integration	29%	21%	36%	14%				
Powerful branching strategies	29%	43%	7%	14%		7%		
Ease of use	21%	36%	21%	21%				
Big user base	7%	36%	36%	21%				

Figure 4.9: Importance of SCM capabilities

CI tool capabilities importance according to the survey results can be seen in Figure 4.10. The majority of the respondents think *actively maintained* to be the most important capability of an CI tool. *Plugin support* was rated the second most with *scalable* and *different configuration options* was of equal importance thereafter. A capability that is of high importance is *visualize the flow*, ranging from extreme importance to moderate. The responses were more scattered for *platform independent* capability, ranging from neutral to extremely important. Most respondents considered *modern UI* and *minimum setup* to be of moderate to slight importance, these two capabilities were distributed mostly throughout the lower level of the importance scale.

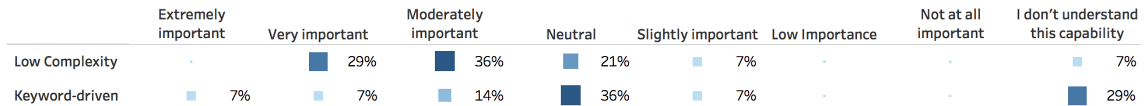
Figure 4.11 details the responses on **test** tools capability importance. Overall respondents selected *low complexity* to be of moderate importance and *keyword-driven* testing had respondents being mostly neutral and there were fair amount of respondents not understanding the meaning of this test tool capability.

For **deployment** tools, 43% of the respondents selected *rollback support* (see Figure 4.12) as extremely important capability in a deployment tool. That a tool *support customer*



	Extremely important	Very important	Moderately important	Neutral	Slightly important	Low Importance	Not at all important	I don't understand this capability
Actively Maintained	43%	43%	7%					7%
Plugin Support	7%	71%	14%					7%
Different Configuration	29%	29%	29%	7%				7%
Scalable	21%	43%	21%	7%				7%
Visualize the Flow	21%	29%	36%	14%				
Platform Independent	14%	36%	14%	21%			7%	7%
Modern UI		7%	43%	36%	14%			
Minimum Setup		14%	36%	7%	21%	14%		7%

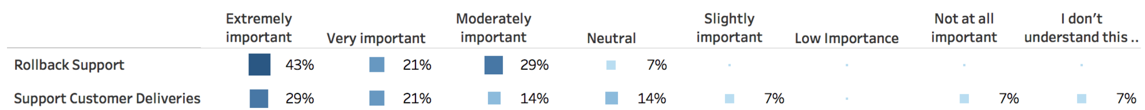
Figure 4.10: Importance of CI capabilities



	Extremely important	Very important	Moderately important	Neutral	Slightly important	Low Importance	Not at all important	I don't understand this capability
Low Complexity		29%	36%	21%	7%			7%
Keyword-driven	7%	7%	14%	36%	7%			29%

Figure 4.11: Importance of test tools capabilities

deliveries is of lesser importance, although half of the respondents consider it to be either extremely or very important.



	Extremely important	Very important	Moderately important	Neutral	Slightly important	Low Importance	Not at all important	I don't understand this ..
Rollback Support	43%	21%	29%	7%				
Support Customer Deliveries	29%	21%	14%	14%	7%		7%	7%

Figure 4.12: Importance of deployment tools capabilities

4.7 Tool Selection

This section presents tools that support the desired capabilities according to the survey results, these capabilities were ranked by importance (see Section 4.6). Tools that could be selected with supporting a capability in the survey were tools previously mentioned in the case study interviews as well as tools taken from XebiaLabs Periodic Table of DevOps Tools (see Appendix D). XebiaLabs published a table containing a list of tools that support continuous activities and more. The tools were chosen by individuals in the software industry as well as XebiaLabs employees and then ranked by popularity. By presenting tools from XebiaLabs and the case study interviews, the respondents would have an opportunity to prioritize and match desired capabilities with tools that they did not mention in the interviews but might have knowledge about. Thus, widening the tool selection that can be used when designing a toolchain.

For the **SCM** component of the toolchain respondents were almost unanimous that Git supported all the desired capabilities of an SCM tool as can be seen in Figure 4.13. The capabilities are ordered by importance, the most importance capability first and the least important one last. Subversion does not support a *powerful merging system*, the most important capability of an SCM tool, according to respondents. Subversion is believed to be *actively maintained* and half of respondents think that it supports *powerful CI integration*. *Powerful branching strategies* is not considered to be supported, however a *big user base* Subversion supports. Mercurial only seems to be *actively maintained* and

respondents do not agree that it supports strongly any other capabilities. ClearCase does not support the most important capabilities but it does support the two least important ones, i.e. *powerful branching strategies* and *big user base*, according to the survey results. Unfortunately, there was no data collected for the *ease of use* capability in relation to the SCM tools, due to a human error during the instrument design.

	Clear Case	Git	Mercurial	Subversion
Powerful merging system	25%	93%	50%	9%
Actively maintained	25%	100%	100%	82%
Powerful CI integration		100%	50%	55%
Powerful branching strategies	75%	93%	50%	18%
Ease of use	N/A	N/A	N/A	N/A
Big user base	75%	100%	25%	91%
This tool has none of the capabilities mentioned		7%		9%

Figure 4.13: SCM tools capability support (capabilities ordered by importance)

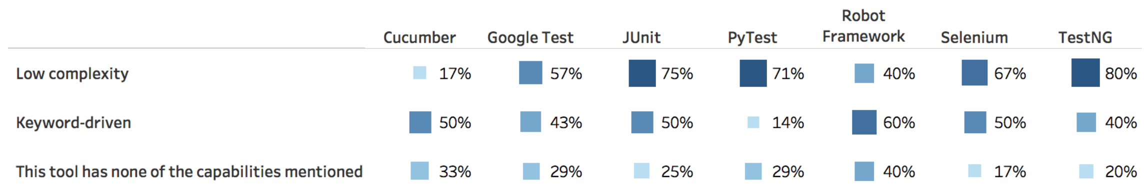
According to the respondents, Jenkins seems to support all desired capabilities of a **CI** tool, except for *modern UI* and *minimum setup*. Travis CI also supports nearly all of the desired capabilities, except for *plugin support* which was rated the second most important capability of a CI tool. In addition *platform independent* and *modern UI* respondents did not agree that Travis CI supported those capabilities. GoCD supported capabilities were rather distributed through the importance ranking, respondents agreed that GoCD is *actively maintained* and supports *visualize the flow*. It does however not support important capabilities such as *plugin support* and *scalable* according to the survey results. TeamCity was quite consistent with 67% of respondents believing it supports all the important capabilites, however nobody though TeamCity had a *modern UI*. Bamboo mostly supported capabilities that were ranked less important, rather than the other ones.

	Bamboo	Go CD	Jenkins	Team City	Travis CI
Actively maintained	67%	80%	92%	67%	100%
Plugin Support	33%	40%	92%	67%	50%
Have different configuration options	33%	60%	92%	67%	100%
Scalable	33%	20%	67%	67%	100%
Visualize the flow	67%	80%	83%	67%	75%
Platform independent	33%	40%	100%	67%	50%
Modern UI	67%	60%	33%		50%
Minimum setup	67%		42%	33%	75%
This tool has none of the capabilities mentioned	33%				

Figure 4.14: CI tools capability support (capabilities ordered by importance)

For **test** tools Selenium and JUnit support both capabilities, according to the survey results. Respondents believe Google Test, PyTest and TestNG to support *low complexity* as a capability, however they do not support *keyword-driven* testing. On the other hand Cucumber and Robot Framework support *keyword-driven* testing but do not have *low complexity* according to survey results. There are several respondents who that the tools mentioned above do not support any of these capabilities.

When it comes to tools that support **deployment**, respondents did not have knowledge about most of the tools listed in the survey. That result did not come as a surprise since no case company implemented CDep.



	Cucumber	Google Test	JUnit	PyTest	Robot Framework	Selenium	TestNG
Low complexity	17%	57%	75%	71%	40%	67%	80%
Keyword-driven	50%	43%	50%	14%	60%	50%	40%
This tool has none of the capabilities mentioned	33%	29%	25%	29%	40%	17%	20%

Figure 4.15: Test tools capability support (capabilities ordered by importance)

5

Discussion

This chapter discusses the results in relation to the research questions. Section 5.1 provides answers to **RQ1** and its sub-questions through a discussion on how to design continuous toolchains. Section 5.2 answers **RQ2** by presenting guidelines which aims at strengthening the overall toolchain structure.

5.1 Designing a Toolchain (RQ1)

The aim of **RQ1** and its sub-questions is to present the current approach for designing continuous toolchains in the software industry. Before designing a toolchain the structure needs to be decided, based on various aspects of the software project. Furthermore, there are capabilities and motivational factors that need to be decided on along with how important they are to the software project. Both the structure, capabilities and deciding factors will drive and help the design of the toolchain.

5.1.1 Technical Components of a Toolchain (RQ1.1)

According to industry professionals the key technical components that construct a continuous toolchain are *SCM, build, test and delivery*, as illustrated by Figure 4.5 in Section 4.2. It is however not uncommon to incorporate more components. These added components are primarily extra testing components, e.g. code reviews, SCA and pre-commit procedures. Although these components are necessary in software development to achieve high code quality, they were not evident enough during the case study to be considered as separate components in the continuous toolchain.

The assumption made in Section 3.2.2 that context would be the focal point to affect a project's level of continuous and that it is essential to make a distinction between the two contexts seems to be without merit. Based on the case study results it is unreasonable to draw such conclusions since the majority (71%) of our interviewees were within the embedded context (see Figure 4.1). The results do however indicate that the project type, i.e. if the project revolves around safety-critical software or not, directly impacts the level of continuous that a project can be. Such projects tend to operate under regulations that require manual testing, making full automation unattainable.

The toolchain structure derived from industry is very similar to the structure in literature,

illustrated by Figure 2.1 in Section 2.2, which indicates that both academia and industry have similar views for how continuous toolchains are structured. This result is expected, especially since the literature used to derive Figure 2.1 is both industry-oriented and relatively new. The major difference between the two component structures lies in the final component. The academic toolchain structure ends with deployment, whereas the industry ends with delivery. This outcome is primarily due to the fact that no case company implemented CDep as a practice. It is likely that the outcome would have been different if a larger subset of the case companies were in the front-end facing context, rather than the embedded context, since embedded software tends to be more challenging to automatically deploy [14].

5.1.2 Selecting the Right Tools (RQ1.2, RQ1.3)

Tool selection needs to be dependent on the tool capabilities that are important to maximize the efficiency of each toolchain component. If tools are not complementing important capabilities then each component is not being maximized in efficiency, leading to a reduced overall toolchain quality. To little surprise many of the case study participants mentioned that the price of tools is an important motivational factor. Some promoted paid enterprise tools since they commonly include maintenance which reduces some of the workload from the project. The desire for open source was however larger. The interviewees argued that by choosing an open source tool the developers would receive complete visibility of the tool's internals as well as becoming able to make modifications to the tool. This self maintenance would allow alterations of the tool to make it better suit their needs. The developers could furthermore resolve bugs themselves rather than being dependant on vendors.

As presented in Figure 4.9 the most desired tool capabilities for SCM tools is *powerful merging system* and the tool needs to be *actively maintained*. These results were somewhat expected since one of the prescribed best practices for SCM tools is to frequently integrate committed code with the existing code base. The two tools that best conforms with these capabilities are **Git** and **Mercurial**, presented in Figure 4.13. The fact that they both are open source makes them even more desirable. Git was most commonly mentioned in the interviews for both contexts (see Figure 4.6), but Mercurial was only mentioned once for both contexts. This urges developers to look into both Git and Mercurial when selecting SCM tools.

The most desired capabilities for build tools is that they are *actively maintained*, *support plugins*, *has different configuration options* and is *scalable*, as can be seen in Figure 4.10. By supporting plugins, especially while being open source, the tool can be customized by the team to fulfill their needs. This customization is further enhanced by tools which have various configuration options. Scalability issues are common for CI servers and it is therefore important to select a tool which is scalable in order to postpone potential break downs for as long as possible. As shown in Figure 4.14, the build tools which best support these capabilities are **Jenkins**, **Travis CI** and **TeamCity** where both Jenkins and Travis CI are open source tools while TeamCity has a freemium licensing scheme. Travis CI and TeamCity are both cloud based meanwhile Jenkins needs a dedicated server. It is interesting to notice that most common build tools mentioned by interviewees were GoCD and Jenkins (see Figure 4.7). TeamCity and Travis CI were only mentioned once but both

support build capabilities that are of more importance than GoCD.

For test tools the most desired capabilities and their importance are presented in Figure 4.11. *Low complexity* is a capability that is of high importance when selecting test tools, whereas *keyword-driven* testing was deemed rather neutral as a capability when it came to importance. Although *keyword-driven* testing is of lesser importance than having *low complexity* it could stem from that all of our interviewees have technical backgrounds. *Keyword-driven* testing overall could be of more importance to a person that does not have technical background because the test cases are abstracted by keywords. By having test cases abstracted with keywords it may become easier to show that the software fulfills the requirements because they are written in similar fashion. The tools that support these desired capabilities can be seen in Figure 4.15. These test tools do however all serve different purpose and are often language specific, which is likely the reason why there only were two desired test tool capabilities. This makes it futile to provide a 'one tool fits all' recommendations for test tools.

Based on our case study and survey results general test tool suggestions are not plausible but suggestions can be made for specific test purpose and context. For front-end facing projects our study results suggest **Selenium** which is a browser testing tool that has *low complexity* and **Cucumber** (acceptance testing tool) which has *keyword-driven* testing as a capability. **Robot Framework** can be used for acceptance testing in embedded software projects. It can test both Java and Python code and strongly supports *keyword-driven* testing, it furthermore integrates well with **Selenium**. **Google Test** was commonly mentioned in interviews (see Figure 4.8) as a unit testing tool for the C++ programming language but surprisingly it did not support either of the desired test capabilitis strongly.

For the deploy component, a desired capability was *rollback support* which was of great importance. This did not come as a surprise since it is prescribed as a best practice in both literature (Stop the Line) and in the case study (Rollback Support). *Support customer deliveries* was however not deemed to be of much importance (see Figure 4.12). These two capabilities for deploy tools were the only ones derived from the case study. There was no interviewee which had implemented CDep and that is likely the reason that only two capabilities were synthesized. Since the respondents were not familiar enough with deployment tools that supported these two capabilities, no recommendation is provided for this toolchain component. The tool suggestions above are based on the capabilities which relates to a tool, but does not necessarily take motivators into account. Throughout the case study *experience* was one of the most frequently mentioned motivational factors for tool selection. Having past experience of a tool tends to speed up the initial toolchain setup and can also provide the team with a comforting feeling, since they already know how to utilize the tool. It is furthermore common to select a new tool based on an existing brand deal, e.g. a company might select Bamboo as their CI server if they already use JIRA and BitBucket which are all part of the Atlassian suite.

When it comes to tools there is rarely a tool which is the best candidate for all use cases and recommendations should therefore always be taken with a grain of salt. Tools have the potential to make software development easier if it is both used right and fits the user's needs [28]. But to fit those needs the user must first invest time investigating what is most important for them and then select tools based on that result. Without such an investigation it is easy to fall for tools which are trending at the time, but might not support the needs of the organization.

The goal of the discussion above is to raise awareness of both commonly important capabilities and tools which can improve your future software deliveries. It should be taken into consideration that the tools recommended are based on the case study and are just a subset of tools available on today's market. These tools do however seem to be among the forefront of the market when it comes to continuous activities (see Appendix D).

5.2 Guidelines Supporting Toolchain Design (RQ2)

This section presents guidelines which are likely to help industry and academia to design their continuous toolchains. Not only are holistic guidelines provided but each toolchain component also has its own dedicated guidelines to further strengthen the component. The guidelines are partly based on common challenges and best practices found during both the conducted literature review and case study. A comparison between the findings of these two methodologies are presented in Sections 5.2.1 and 5.2.2. The guidelines are furthermore based on case study results relating to the tool selection process (see Section 4.7) and can be found in Sections 5.2.3 to 5.2.6.

5.2.1 Comparison of Common Challenges

This section presents a comparison between the common challenges found in both case study interviews and literature. An overview is visualized in Table 5.1, followed by a discussion of the presented challenges.

Common Challenges Comparison

Interviews	**Literature**
Platform Issues	
Regulations	
Visualization	
Mindset	Human activities
Integration	Integration
Excessive manual testing	Testing
Toolchain complexity	System design

Table 5.1: Comparing common challenges from interviews and literature [12]

Platform Issues. A challenge frequently mentioned by interviewees, but not in any literature, was the shortcomings of using certain platforms in the toolchain. We had not come across this issue in any literature during our literature review. It might stem from that CDel and CDep are relatively new software practices and continuous toolchain platforms have not been focused on as research topic.

Regulations. Only interviews mentioned that it is a challenge to implement a continuous toolchain while fulfilling all software regulations. It is interesting to note that literature has brought little to no attention to the challenge of constructing continuous toolchain

while complying with regulations. We think that it might be because literature is not focused towards embedded software in terms of continuous activities, where the challenge is more apparent than in front-end software.

Visualization. Only the interviewees mentioned that it is a challenge to visualize the flow of the toolchain and it becomes really problematic to visualize the overall flow when there are a lot of build jobs flowing through the chain simultaneously. The literature only prescribes developers to shepherd their software changes [21] but not to visualize it in any way. Several interviewees did refer to the resolution of this challenge as a best practice (see Section 5.2.2).

Mindset vs Human activities. In interviews mindset was frequently mentioned as a challenge, especially when developers still apply traditional software development practices instead of focusing on the continuous practices. Lack of experience practicing CDel and CDep was also described as a challenge in the interviews. Lack of motivation, discipline and experience were some of the most critical challenges mentioned in literature [12].

Integration. Mentioned in literature and in interviews that integration is a substantial challenge for all continuous activities [8, 9, 12]. In the interviews integration of software changes and integration between tools were discussed as great challenges. Literature however was more specific when it came to integration challenges, e.g. large commits, merge conflicts, broken build, long-running branches and broken development flow.

Excessive manual testing vs Testing. Both literature and interviews mentioned that testing is likely the most critical challenge for continuous software development. According to the interviews the main challenge is that there are too many manual tests in the software and it becomes even more problematic when there are dependencies on other parties, e.g. subcontractors. With ever changing requirements manual testing does not provide consistent feedback on quality and performance of the software. In literature testing is described as quite the challenge. The most critical testing problems described are: time consuming testing, tests that are not explicit pass or fail or that it is not clear what broke the build, tests that randomly fail as well as multiple platform testing [12].

Toolchain complexity vs System design. According to both interviewees and literature toolchain complexity is a challenge. In interviews many activities that contribute to the toolchain complexity were discussed. From integrating different tools together that make up the toolchain to scaling it were all thought to be a major contributors to the toolchain complexity. In literature system modularization, unsuitable architecture, scaling and dependencies all contribute to the toolchain complexity. The challenge is to make the correct system design decisions in order to keep the toolchain complexity low [12].

5.2.2 Comparison of Best Practices

This section compares best practices found in both case study interviews and literature. An overview of these practices is presented in Table 5.2, followed by a discussion.

Manual testing as a precondition for safety critical software. Only mentioned by interviewees is the practice to conduct manual tests before deploying safety-critical software. This most likely stems from the challenge of fulfilling software regulations.

Little to no research has mentioned the implications that regulations pose on both CDel and CDep and is therefore something that future research should investigate.

Best Practices Comparison

Interviews	**Literature**
Manual testing as a precondition for safety critical software	
Accept that some changes will fail	
Automate as much as possible	Automated deployment and testing
Mainline should always be in a working state & Merge often	Repository Use
Test in multiple environments	Staging
Rollback support	Stop the line
Visualizing the flow of the toolchain	Shepherding changes
	Deploy same way to every environment
	Intercommunication
	Smoke test deliveries

Table 5.2: Comparing best practices from interviews and literature [10, 21]

Accept that some changes will fail. The practice to recognize and accept that some software changes that go through the toolchain will fail was only mentioned by interviewees. The purpose of this practice is to motivate developers to use the toolchain by pushing software changes more frequently, if developers are using the toolchain conservatively its purpose becomes irrelevant.

Automate as much as possible vs Automated deployment & testing. Both literature and interviewees prescribed the practice to automate the toolchain [7, 10] and focused on rigorous test automation as a key success factor for continuous delivery [13, 27]. Interviewees also frequently stated that the toolchain should be automated as much as possible with emphasis on various test activities.

Mainline should always be in a working state & Merge often vs Repository use. Mentioned by both literature [10, 21] and in interviews, it is the practice of keeping the mainline in a working state by using a repository to merge software changes. This practice becomes even stronger if merging is done at regular intervals since the committed code then will deviate less from what is currently merged into the mainline. It came as no surprise that there were frequent discussions relating to this practice throughout the interviews, since repository use relates to the toolchain's SCM component.

Test in multiple environments vs Staging. Mentioned in both literature [21] and in interviews is the practice to test software in multiple environments. These environments have varying capabilities, so if there are multiple target environments all must be tested to ensure that the deployment to the real environment will work as intended.

Rollback support vs Stop the line. Both literature [10, 12] and interviews mentioned the practice to either stop the toolchain and fix the problem immediately or automatically

revert the delivery to a previous state if the toolchain fails. In every case this is a redundancy mechanism that should be integrated to minimize the probability of deploying bugs.

Visualizing the flow of the toolchain vs Shepherding changes. Mentioned in both literature [21] and interviews is the practice of being able to follow software changes throughout the toolchain. In interviews this practice is directed at the developers as a means to help them follow their own software changes visually, which is not mentioned in literature. By visualizing the changes, external stakeholders could see the project's status as well.

Deploy same way to every environment. Only literature mentioned the practice of using the same setup and settings when delivering software, irrespective of target environment. This should be done to ensure that the build and deployment process is tested effectively and is consistent [10]. Interestingly this was not brought up in the interviews. Several interviewees did however recognize the importance of utilizing containerization so software can be identically rebuilt after several years from now.

Intercommunication. A practice only mentioned in literature is intercommunication which refers to sharing all necessary development information among team members [21]. This practice was not brought up in the interviews, but could relate to the best practice of *visualizing the toolchains flow* to a certain extent since it allows the entire team to see the project's status.

Smoke test deliveries. Only mentioned in literature [10] is the practice to ensure that the delivery was successful and to investigate that is is working as it should. Smoke tests came up frequently in the interviews when discussing activities conducted in the test component, but never as a prescribed practice.

5.2.3 General Guidelines

Drive tool selection with design. Before selecting tools for the toolchain there needs to be a design composed which drives the tool selection. The design of the toolchain depends foremost on what capabilities and motivational factors are important to the software project. Desired capabilities for each toolchain component can be seen in Table 4.3. The motivational factors to keep in mind when selecting tools, in no particular order are: *experience*, *open source*, *budget*, *time* and how well does the tool *integrate* with other tools.

Automate as much as possible. Imperative to CDel and CDep is the automation of as many components and activities of the toolchain as possible, to save resources and reduce risk of faults in the software. Both prescribed as a best practices in literature and interviews, the toolchain should be automated as much as possible.

Use extendable tools. Tools that are integrated together in a toolchain should both be extendable and handle integration with each other very well. The open-closed principle describes that tools should be closed for modification but should be open for extension, meaning that a tool can allow its behavior to be extended without modifying its source code. This principle should be considered to apply to every tools behaviour, which enables

the users to customize it for their needs. Furthermore, the ability to extend tools takes future growth of the tool into consideration.

Educate and prepare for a continuous mindset. It is important to spend resources educating and preparing the employees on how to deliver quality software continuously. Employees often need to alter their mindsets from traditional software development to accommodate this new continuous mindset. If the organization is prepared before development starts the employees will have grown accustomed to the concept before software starts flowing through the toolchain. It however takes a great effort and mindset changes happen over a long period of time, but helps the organizational and team dynamics when delivering software [7, 12].

Make developers shepherd their changes. Each developer is responsible for their code changes throughout every component of the toolchain. The aim of shepherding code changes is to get the developer involved in all activities of the toolchain. Shepherding enhances the developers responsibility and ensures software quality in the end product without a dedicated quality assurance team [21]. Shepherding changes can be incorporated by visualizing the toolchain's flow, therefore choosing a tool that strongly supports visualization of changes can help shepherding changes.

Visualize the toolchain's flow. The main goal of visualizing the flow of the toolchain is to be able to show the status of every software change. Visualization of the overall flow, from when a developer commits until the code is deployed in the end-product. Visualizing the flow was a desired build tool capability and a common challenge in the software industry. Choosing a tool that strongly supports visualization of changes can help the developer to think about the end product and be aware and take responsibility of their own commit all the way through the chain.

Trust that the toolchain works. It is important that developers have confidence in the toolchain. They need to be able to trust that the series of tests will successfully catch faults and deliver working software that is reliable and robust. Otherwise the developers are likely to use it more conservatively which beats the purpose of having a toolchain.

Incorporate rollback support. To have an efficient toolchain, rollback support is required. It was a desired capability for deploy tools and was mentioned both in literature and interviews. This feature is of high importance, since the toolchain should always be up and running. Tools that support rollback should drive tool selection.

Keep the toolchain infrastructure stable. It is of great importance that the toolchain's infrastructure is stable, developers need to be able trust that the toolchain is up and running all the time. If developers need to fix their software changes on a regular basis because of toolchain instability, a lot of resources are going to be wasted. Try to keep the toolchain infrastructure stable at all times.

Setup the toolchain before starting development. A toolchain should be up and running with all the tools integrated before software development starts. Resources thereby can be spent on the actual development, instead of adding components and integrating tools to the chain in middle of development.

Limit usage of UI to configure tools. Using the UI to configure tools should be limited as much as possible. Ideally there should be no execution or configurations done through

a UI of a tool since they rely on consistency. The toolchain should not be dependant on a UI that could easily break the chain. Configuration of tools should be done through scripts that makes it more flexible, easier to maintain and migrate than using a tools UI.

Have specialists manage the toolchain. In large organizations it is recommended to use a dedicated team for toolchain management. This may however not be feasible in a smaller organization. In such cases there should always be at least one developer responsible for managing the toolchain. This task of maintaining the toolchain should remain top priority for the dedicated developer(s), even if the stakeholders might think differently.

Limit usage of Windows as a toolchain platform. Throughout our interviews it became apparent that Windows as a operating system is not ideal and is losing ground as a platform for continuous toolchains. Thread issues, Windows tasks and licensing issues were frequently mentioned as disadvantages of Windows. There is believed to be more growth and diversity in Linux environments which seems to be serving well as toolchain platforms. Limiting the usage of Windows as a toolchain platform should therefore be considered.

5.2.4 Source Code Management Guidelines

Keep mainline in a working state. In order to have a stable SCM component it is important to never let the mainline fail. It is therefore good to use branching strategies and ensure that the development branch builds before it merges with the mainline.

Merge changes often. By frequently merging and integrating software changes there is a decreased risk of ending up in what many interviewees refer to as “merge hell”, since the software will not have deviated too much from the mainline of the repository.

Test before you commit. The fastest way for a developer to get feedback on whether or not the newly developed code works as intended is by running tests locally before committing. This way the developer will reduce the potential of affecting the toolchain negatively while also avoiding to waste resources on committing flawed software.

Tool capabilities supporting these guidelines: *powerful merging system*, *powerful branching strategies* and *powerful CI integration.*
Tool recommendation: **Git**.

5.2.5 Continuous Integration Guidelines

Use a CI server with powerful SCM tool integration. Some of the key responsibilities of a CI server is to merge and build code received via the attached SCM tool. It is therefore of paramount importance that the integration between the SCM tool and the potential CI server is immaculate.

Use a CI server with powerful SCA tool integration. Although static code analysis (SCA) tools are an optional addition to a continuous toolchain, it is a good aid to maintain good coding practices and thereby reducing technical debt. Common features are checking

code style and cyclomatic complexity.

Aim for only one CI server. In an ideal situation there shall only be one CI server in the toolchain. When migrating from one server to another it is an unavoidable fact that two will be in use at the same time. Since e.g. Jenkins and GoCD have different resource-allocators it would be preferable to make a full migration to the new system.

Tool capabilities supporting these guidelines: *plugin support* and *have different configuration options*.
Tool recommendation: **Jenkins**.

5.2.6 Test Guidelines

Try to automate all tests. Tests are an integral part of any continuous toolchain and the level of automation is to a large extent what separates the continuous activities. If the amount of manual intervention is reduced there will be both shorter feedback cycles and faster software deliveries [16]. The reduction of manual intervention will furthermore minimize the risk of human errors.

Maintain a high test coverage. In order to allow full trust in the toolchain it is important that there is a high test coverage. This will not in itself ensure a good end-product as the test themselves may be poorly written, but it is a step in the right direction.

Test in different environments. Many software products are intended for more than one environment, it is therefore essential that there are a vast set of test environments in order to ensure that the software behaves according to specification. Furthermore, different test environments test different capabilities of the software.

Adopt trust levels. By having different test levels and thereby scaling the test coverage, increasing trust in the code at hand, resource waste will be limited. The idea is that the easier and less resource heavy tests are run at first so that highly problematic code will not lock test components which take longer to compute.

Enable testers to focus on new tests. Making testers continuously update old tests is a waste of valuable resources. To avoid this the project must define clear requirements so that a good regression test suite can be attained early on, thereby enabling the testers to focus on upcoming test cases.

Only test affected code. By only testing affected code, resources are saved at the same time as the new changes moved to the mainline are ensured to not having introduced any new faults into the end-product.

Manually test safety critical software. Safety critical software revolves around laws and regulations which often pose requirements that the acceptance testing should be done manually. It is therefore important that organizations working close to safety critical software takes this into consideration.

Tool capability supporting these guidelines: *low complexity*.
Since each test tool has such a different purpose, e.g. acceptance testing or UI testing, there is not any one tool that can be recommended over the others (see Section 5.1.2).

6
Conclusion

Continuous activities such as CDel and CDep are becoming more prevalent in software organizations that strive for faster software deliveries. When adopting these continuous activities into software projects, challenges are frequently encountered while constructing continuous toolchain [7]. The purpose of this study was to provide guidelines for software organizations on how to design continuous toolchains and identify important capabilities for tools used in such a toolchain. The motivation for the study was acquired when a research gap that guidelines for continuous toolchains and toolchain design, i.e. tool selection, had not been widely researched or published was identified in literature [7, 13, 14, 15]. If continuous toolchains were mentioned in literature their context and environment was usually missing [12], which greatly affects the ability to reuse the toolchain design. We wanted to contribute to closing this research gap by conducting a case study where industry professionals were interviewed to gain a good grasp of the perceived challenges and best practices as well as how tool selection is conducted in software organizations so we could generate design guidelines for toolchains. To generalize the the tool capabilities derived from the case study and rank their importance a survey was conducted. It was necessary to incorporate a survey to evaluate the findings in order to ensure the acquisition of the interviewees collective views.

This study contributes to both industry and academia by shedding light on the difference of how the two fields perceive continuous toolchains. It further present tool capabilities which the industry desires for the tools used in their toolchains as well as the actual tools which support these desired capabilities. Moreover it points out integral motivational factors for tool selection which should not be omitted. The key contribution is nevertheless a set of guidelines which gives suggestions that are necessary to keep in mind while designing continuous toolchains. Software organizations and researchers can thereby use this study as guidance for selecting tools for their continuous toolchains, saving resources by searching and investing in less efficient tools and furthermore advise software organizations on how to maintain an efficient continuous toolchain. The general guidelines can be incorporated into workflows and tool selection activities which are not related to the toolchain to a certain degree, as they are centered around improvements of software development.

In order to better understand the usefulness of the proposed guidelines (see Section 5.2) future research should first and foremost design and implement continuous toolchains while taking the guidelines into consideration. A synthesis of several cases of implementation would provide a good overview of which guidelines are the most useful in what contexts. This is a critical next step since the guidelines have not been thoroughly verified and furthermore the desired tool capabilities might be too conservative as they are only based on what the case study participants already know, rather than what they might need.

Previous literature have reported that projects face distinct challenges depending on context [14, 28] and this study therefore proposed that context is likely to affect the structure of a continuous toolchain. Based on the results of this study no such conclusion could be drawn. Nonetheless future research should not discourage taking context into consideration, especially not if investigating projects implementing the practice of CDep. It is likely that the distinction will become more apparent when more cases of CDep have been studied in both the front-end facing and the embedded context.

More research is required on tools for both CDel and CDep. Due to the absence of case study participants working with CDep this study lack concrete tool proposals for the toolchains final component. Such tools should however have *rollback support* and *support customer deliveries*, two important capabilities which can function as a starting point for future research on the topic.

A clear research gap noted throughout the course of this thesis is that little to no research have mentioned the implications that regulations pose on both CDel and CDep. A common challenge mentioned by several interviewees, especially for development of safety critical software. Future research should investigate to what extent these regulations affect delivery and deployment as well as how companies must adapt their delivery process to accommodate for this.

Bibliography

[1] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[2] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality?: an empirical case study of mozilla firefox," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pp. 179–188, IEEE Press, 2012.

[3] B. Katumba and E. Knauss, "Agile development in automotive software development: challenges and opportunities," in *International Conference on Product-Focused Software Process Improvement*, pp. 33–47, Springer, 2014.

[4] T. Dingsøyr and N. B. Moe, "Towards principles of large-scale agile development," in *International Conference on Agile Software Development*, pp. 1–8, Springer, 2014.

[5] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *VTT Publications 478*, 2002.

[6] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *Agile Conference (AGILE), 2013*, pp. 121–128, IEEE, 2013.

[7] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72–86, 2017.

[8] H. H. Olsson and J. Bosch, "Climbing the "stairway to heaven": evolving from agile development to continuous deployment of software," in *Continuous software engineering*, pp. 15–27, Springer, 2014.

[9] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

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

[11] M. Fowler, "Continuous delivery," May 2013. Available at `martinfowler.com/bliki/ContinuousDelivery.html`.

[12] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—a systematic literature review," *Information and Software Technology*, vol. 82, no. Supplement C, pp. 55 – 79, 2017.

[13] B. Fitzgerald and K.-J. Stol, "Continuous software engineering and beyond: Trends and challenges," in *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, (New York, NY, USA), pp. 1–9, ACM, 2014.

[14] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software*, vol. 123, pp. 263–291, 2017.

[15] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond continuous delivery: An empirical investigation of continuous deployment challenges," in *Empirical Software Engineering and Measurement (ESEM), 2017 ACM/IEEE International Symposium on*, pp. 111–120, IEEE, 2017.

[16] A. A. Gerry Gerard Claps, Richard Berntsson Svensson, "On the journey to continuous deployment: Technical and social challenges along the way," *Information and Software Technology*, vol. 57, pp. 21 – 31, 2015.

[17] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, "The highways and country roads to continuous deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[18] N. Rathod and A. Surve, "Test orchestration a framework for continuous integration and continuous deployment," in *Pervasive Computing (ICPC), 2015 International Conference on*, pp. 1–5, IEEE, 2015.

[19] M. Fowler, "Continuous integration," May 2006. Available at `martinfowler.com/articles/continuousIntegration.html`.

[20] M. Soni, "End to end automation on cloud with build pipeline: the case for devops in insurance industry, continuous integration, continuous testing, and continuous delivery," in *Cloud Computing in Emerging Markets (CCEM), 2015 IEEE International Conference on*, pp. 85–89, IEEE, 2015.

[21] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing continuous deployment practices used in software development," in *Agile Conference (AGILE), 2015*, pp. 1–10, IEEE, 2015.

[22] H. H. Olsson, J. Bosch, and H. Alahyari, "Towards r&d as innovation experiment systems: A framework for moving beyond agile software development," in *Proceedings of the IASTED*, pp. 798–805, 2013.

[23] N. Wilde, B. Eddy, K. Patel, N. Cooper, V. Gamboa, B. Mishra, and K. Shah, "Security for devops deployment processes: Defenses, risks, research directions," *International Journal of Software Engineering & Applications (IJSEA)*, vol. 7, no. 6, 2016.

[24] J. Sandobalin, E. Insfran, and S. Abrahao, "End-to-end automation in cloud infrastructure provisioning," in *Proceedings - 26th International Conference on Information Systems Development, ISD*, 2017.

[25] F. Oliveira, T. Eilam, P. Nagpurkar, C. Isci, M. Kalantar, W. Segmuller, and E. Snible, "Delivering software with agility and quality in a cloud environment," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 10–1, 2016.

[26] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 322–333, ACM, 2014.

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

[28] S. Mäkinen, M. Leppänen, T. Kilamo, A.-L. Mattila, E. Laukkanen, M. Pagels, and T. Männistö, "Improving the delivery cycle: A multiple-case study of the toolchains in finnish software intensive enterprises," *Information and Software Technology*, vol. 80, pp. 175–194, 2016.

[29] T. Bruckhaus, N. Madhavii, I. Janssen, and J. Henshaw, "The impact of tools on software productivity," *IEEE Software*, vol. 13, no. 5, pp. 29–38, 1996.

[30] D. B. Smith and P. W. Oman, "Software tools in context," *IEEE software*, vol. 7, no. 3, pp. 15–19, 1990.

[31] J. A. Pereira, K. Constantino, and E. Figueiredo, "A systematic literature review of software product line management tools," in *International Conference on Software Reuse*, pp. 73–89, Springer, 2015.

[32] E. Glynn and P. Strooper, "Evaluating software refactoring tool support," in *Software Engineering Conference, 2006. Australian*, pp. 10–pp, IEEE, 2006.

[33] M. Taheri and S. M. Sadjadi, "A feature-based tool-selection classification for agile software development.," in *SEKE*, pp. 700–704, 2015.

[34] C. Robson, "Real world research. 2nd," *Edition. Blackwell Publishing. Malden*, 2002.

[35] A. Bryman and E. Bell, *Business research methods.* Oxford University Press, USA, 2015.

[36] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, p. 131, Dec 2008.

[37] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, and K. Hoagwood, "Purposeful sampling for qualitative data collection and analysis in mixed method implementation research," *Administration and Policy in Mental Health and Mental Health Services Research*, vol. 42, no. 5, pp. 533–544, 2015.

[38] B. Kitchenham and S. L. Pfleeger, "Principles of survey research: parts 1-6," *ACM SIGSOFT Software Engineering Notes*, Nov 2001 to Mar 2003.

[39] A. N. Ghazi, K. Petersen, S. S. V. R. Reddy, and H. Nekkanti, "Survey research in software engineering: problems and strategies," *arXiv preprint arXiv:1704.01090*, 2017.

[40] W. M. Vagias, "Likert-type scale response anchors. clemson international institute for tourism," *Recreation and Tourism Management, Department of Parks, Clemson International Institute for Tourism & Research Development, Clemson University*, 2006.

[41] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering*, pp. 285–311, Springer, 2008.

[42] U. Loecher, "Small and medium-sized enterprises–delimitation and the european definition in the area of industrial business," *European Business Review*, vol. 12, no. 5, pp. 261–264, 2000.

[43] J. Saldaña, *The coding manual for qualitative researchers.* Sage, 2016.

A

Abbreviations

Abbreviations	Description
CDel	Continous Delivery
CDep	Continous Deployment
CI	Continous Integration
SCA	Static Code Analysis
SCM	Source Code Management
UAT	User Acceptance Test
UI	User Interface

B

Interview guide

0 Prior to interview start

0.1(What) We want to ask you some questions about your background and your experiences with continuous integration, delivery or deployment.

0.2(Purpose) The purpose of this interview is to get insights of how industry professionals select tools for their continuous toolchain and to get opinion on key tool capabilities.

0.3 Show the consent form. ***Stress*** that it's entirely optional to comment on the transcript.

0.4 Ask if they have any questions before we start

<table>
<tr><th>ID</th><th>Related RQ</th><th>Questions</th></tr>
<tr><td>1</td><td></td><td>Background</td></tr>
<tr><td>1.1</td><td>-</td><td>Could you shortly describe your role at your current company?</td></tr>
<tr><td>1.2</td><td>-</td><td>How large is the organization/company you work for?</td></tr>
<tr><td>1.3</td><td>-</td><td>How long have you worked at this company?</td></tr>
<tr><td>1.4</td><td>-</td><td>How long have you worked in the industry?</td></tr>
<tr><td>1.5</td><td>-</td><td>What is the domain of your current project?</td></tr>
<tr><td>2</td><td></td><td>Toolchain</td></tr>
<tr><td>2.1</td><td>-</td><td>Does your current project implement Continuous Deployment OR Continuous Delivery OR Continuous Integration?</td></tr>
<tr><td>2.2</td><td>RQ1.3</td><td>Can you draw a picture while describing your current process from when you write code until it is released?

If yes draw and go to question 3

If no:
Show the printed out picture
<table>
<tr><td>2.2.2</td><td>RQ 1.3</td><td>Can you describe if this picture is in any way similar to what you are using ?
<table>
<tr><td>2.2.2.1</td><td>RQ1.3</td><td>Which components from the picture are automated?</td></tr>
</table>
</td></tr>
</table>
If no: go to question 3</td></tr>
<tr><td>3</td><td></td><td>Tool selection</td></tr>
</table>

<table>
<tr><td>3.1</td><td></td><td>For each component X drawn by interviewee:</td></tr>
<tr><td></td><td></td><td>
<table>
<tr><td>3.1.1</td><td>RQ2.1</td><td>What tools are currently being used for this component X?</td></tr>
<tr><td>3.1.2</td><td>RQ2.2,
RQ 2.3</td><td>How was the decision made for selecting these tools?</td></tr>
<tr><td>3.1.3</td><td>RQ2.2,
RQ 2.3</td><td>What do you think are the pros and cons of these tools?

- Follow up question:
Is this the capabilities that you would look for in this tool: summarizing pros and cons</td></tr>
<tr><td>3.1.4</td><td>RQ2.2</td><td>Was any alternatives explored?
If yes:
<table><tr><td>3.1.3.1</td><td>RQ2.1</td><td>Which?</td></tr></table>
If no go to question 3.2</td></tr>
</table>
</td></tr>
<tr><td>3.2</td><td>RQ1.2</td><td>Which components from the picture are automated?</td></tr>
<tr><td>3.3</td><td>RQ2.2</td><td>What tool in the toolchain are you the most satisfied with?
<table><tr><td>3.2.1</td><td>RQ2.2</td><td>Why?</td></tr></table>
</td></tr>
<tr><td>3.4</td><td>RQ2.2</td><td>Would you want to change any of the tools in the toolchain?</td></tr>
<tr><td></td><td></td><td>If Yes:
<table>
<tr><td>3.3.1</td><td>RQ2.2</td><td>Why?</td></tr>
<tr><td>3.3.2</td><td>RQ2.2</td><td>How?</td></tr>
</table>
If no go to question 3.5</td></tr>
<tr><td>3.5</td><td>RQ 2.3</td><td>How would you decide on tools if you would extend your current toolchain to CDel/CDep?</td></tr>
<tr><td>3.6</td><td></td><td>What do you think are the main obstacles in moving from Continuous integration/CDel to CDel or CDep?</td></tr>
</table>

4		**Feedback**
4.1	-	Anything important that you want to add which we forgot to ask about?
4.2	-	We appreciate the time you took for this interview. Could you recommend anyone else that could provide us with more information?

C

Survey Questionnaire

Continuous Delivery Toolchains

This survey is part of a thesis which aims to provide guidelines for designing continuous delivery toolchains. There are a great deal of open source and commercial tools that can be used in a toolchain. As of now it does not seem to be any standardized or organized way for how each organization uses tools in continuous delivery.

The survey works as an instrument to validate key capabilities of tools used in a toolchain. You as a participant was selected for this survey based on previous activities in this thesis.

The survey is anonymous and is expected to take approximately 6 - 8 minutes.

Thanks in advance for your participation.

*Required

Source Code Management (SCM)

1. **Importance of SCM Capabilities** *
   How important do you think that the capabilities below are for a Source Code Management (SCM) tool?
   Mark only one oval per row.

	Not at all important	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important	I don't understand this capability
Actively maintained								
Powerful branching strategies								
Big user base								
Powerful merging system								
Powerful integration with a CI								
Easy to work with								

2. **Capabilities of SCM Tools** *
   For each SCM tool mark what capability you think each tool has. You can choose more than one capability for each tool.
   Tick all that apply.

	Actively maintained	Big user base	Powerful branching strategies	Powerful merging system	Powerful integration with a CI	This tool has none of the capabilities mentioned	I don't know this tool
Subversion							
Git							
ClearCase							
Mercurial							

Continuous Integration (CI)

3. **Importance of CI Capabilities** *
How important do you think that the capabilities below are for a Continuous Integration (CI) tool?
Mark only one oval per row.

	Not at all important	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important	I don't understand this capability
Platform independent								
Minimum setup								
Plugin Support								
Scaleable								
Visualize the flow								
Actively maintained								
Modern UI								
Have different configuration options								

4. **Capabilities of CI Tools** *
For each CI tool mark what capability you think each tool has. You can choose more than one capability for each tool.
Tick all that apply.

	Have different configuration options	Actively maintained	Modern UI	Minimum setup	Scaleable	Platform independent	Visualize the flow	Plugin Support	This tool has none of the capabilities mentioned	I don't know this tool
Circle CI										
Continuum										
CruiseControl										
GoCD										
Continua CI										
Shippable										
Travis CI										
Solano CI										
Bamboo										
Jenkins										
Codeship										
TeamCity										

Testing

5. **Importance of Testing Capabilities** *
How important do you think that the capabilities below are for a test tool?
Mark only one oval per row.

	Not at all important	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important	I don't understand this capability
Keyword-driven								
Low complexity								

6. **Capabilities of Test Tools** *

For each test tool mark what capability you think each tool has. You can choose more than one capability for each tool.

Tick all that apply.

	Low complexity	Keyword-driven	This tool has none of the capabilities mentioned	I don't know this tool
JMeter				
JUnit				
Karma				
Polyspace				
Google Test				
Cucumber				
Jasmine				
Robot Framework				
Gatling				
Selenium				
PyTest				
TestNG				
Mocha				
QUnit				
FitNesse				

Deployment

7. **Importance of Deployment Capabilities** *

How important do you think that the capabilities below are for a deployment tool?

Mark only one oval per row.

	Not at all important	Low importance	Slightly important	Neutral	Moderately important	Very important	Extremely important	I don't understand this capability
Support customer deliveries								
Rollback support								

8. **Capabilities of Deployment Tools** *

For each deployment tool mark what capability you think each tool has. You can choose more than one capability for each tool.

Tick all that apply.

	Support customer deliveries	Rollback support	This tool has none of the capabilities mentioned	I don't know this tool
ElasticBox				
XL Deploy				
Capistrano				
Otto				
CA Release Automation				
RapidDeploy				
Google Cloud Deployment Manager				
Urbancode Deploy				
CodeDeploy				
SmartFrog				
Deploybot				
OctopusDeploy				
JuJu				

Powered by
Google Forms

D

Xebialabs' Periodic Table of DevOps Tools

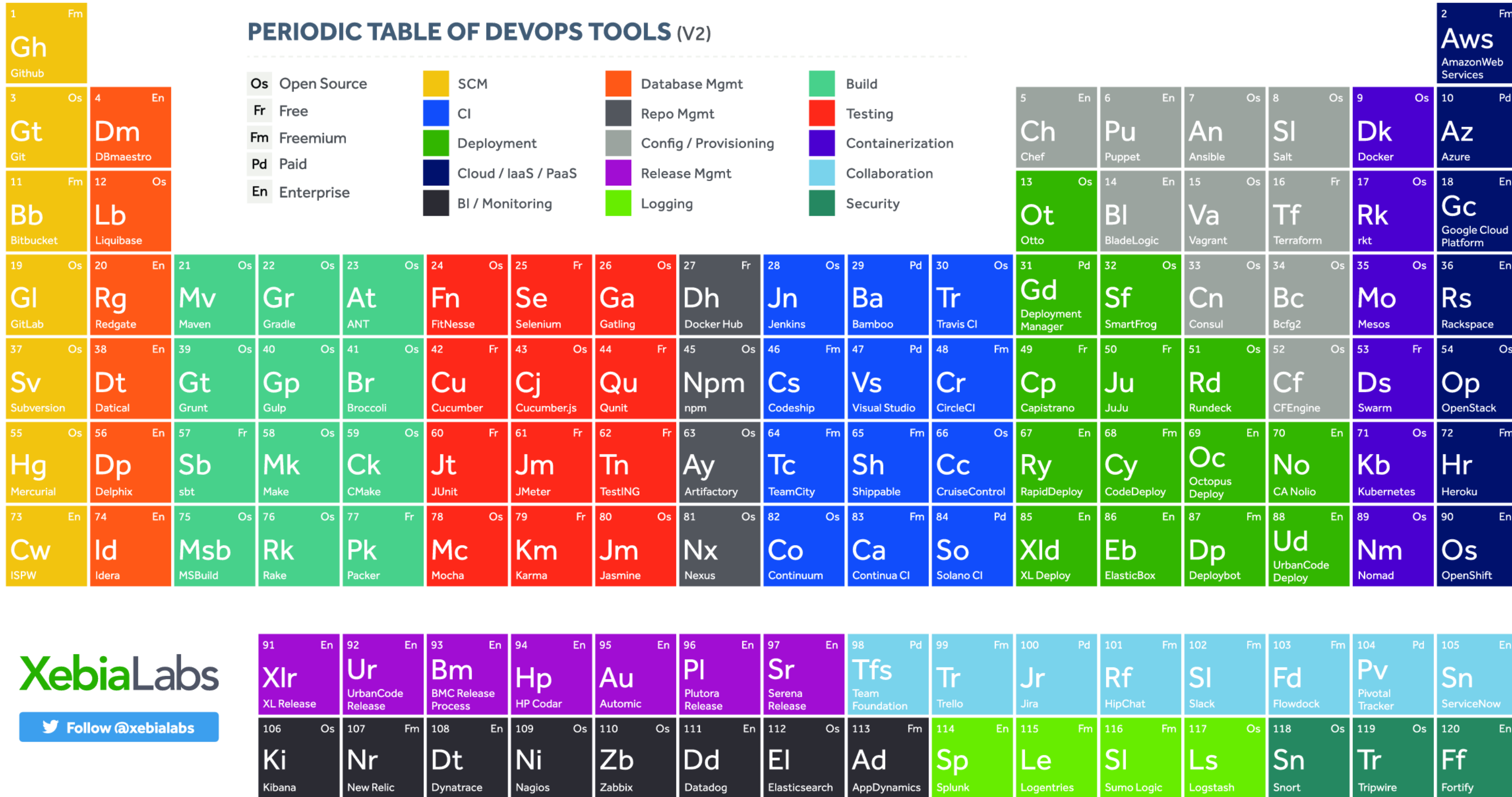


Figure D.1: Periodic Table of DevOps Tools (V2) by XebiaLabs[a]

[a] https://xebialabs.com/