Department of Computer Science
Series of Publications A
Report A-2023-4

Increasing Release Frequency by Accelerating Software Development Cycles in Software Engineering

Simo Mäkinen

Doctoral dissertation, to be presented for public examination with the permission of the Faculty of Science of the University of Helsinki in Auditorium CK112, Exactum building, on 28th April 2023 at 13 o'clock.

University of Helsinki
Finland

Supervisors
Tomi Männistö, University of Helsinki, Finland
Tommi Mikkonen, University of Jyväskylä, Finland
Antti-Pekka Tuovinen, University of Helsinki, Finland

Pre-examiners
Ville Leppänen, University of Turku, Finland
Ali Babar, University of Adelaide, Australia

Opponent
Eric Knauss, Chalmers University of Technology and University of Gothenburg, Sweden

Custos
Tomi Männistö, University of Helsinki, Finland

Contact information

Department of Computer Science
P.O. Box 68 (Pietari Kalmin katu 5)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911

Copyright © 2023 Simo Mäkinen
ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-9144-1 (paperback)
ISBN 978-951-51-9145-8 (PDF)
Helsinki 2023
Unigrafia

Increasing Release Frequency by Accelerating Software Development Cycles in Software Engineering

Simo Mäkinen

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
simo.makinen@cs.helsinki.fi

PhD Thesis, Series of Publications A, Report A-2023-4
Helsinki, April 2023, 130 + 66 pages
ISSN 1238-8645 (print)
ISSN 2814-4031 (online)
ISBN 978-951-51-9144-1 (paperback)
ISBN 978-951-51-9145-8 (PDF)

Abstract

In recent years, companies engaged in software development have taken into use practices that allow the companies to release software changes almost daily to their users. Previously, release frequency for software has been counted in months or even years so the leap to daily releases can be considered big. The underlying change to software development practices is equally large, spanning from individual development teams to organizations as a whole.

The phenomenon has been framed as continuous software engineering by the software engineering research community. Researchers are beginning to realize the impact of continuous software engineering to existing disciplines in the field. Continuous software engineering can be seen to touch almost every aspect of software development from the inception of an idea to its eventual manifestation as a release to the public. Release management or release engineering has become an art in itself that must be mastered in order to be effective in releasing changes rapidly. Empirical studies in the area should be helpful in further exploring the industry-driven phenomenon and understanding the effects of continuous software engineering better.

The purpose of this thesis is to provide insight into the habit of releasing software changes often that is promoted by continuous software engineering. There are three main themes in the thesis. A main theme in the thesis is seeking an answer to the rationale of frequent releases. The second theme focuses on charting the software processes and practices that need to be in place when releasing changes frequently. Organizational circumstances surrounding the adoption of frequent releases and related practices are highlighted in the third theme.

Methodologically, this thesis builds on a set of case studies. Focusing on software development practices of Finnish industrial companies, the thesis data has been collected from 33 different cases using a multiple-case design. Semi-structured interviews were used for data collection along with a single survey. Respondents for the interviews included developers, architects and other people involved in software development. Thematic analysis was the primary qualitative approach used to analyze the interview responses. Survey data from the single survey was analyzed with quantitative analysis.

Results of the thesis indicate that a higher release frequency makes sense in many cases but there are constraints in selected domains. Daily releases were reported to be rare in the case projects. In most cases, there was a significant difference between the capability to deploy changes and the actual release cycle. A strong positive correlation was found between delivery capability and a high degree of task automation. Respondents perceived that with frequent releases, users get changes faster, the rate of feedback cycles is increased, and product quality can improve.

Breaking down the software development process to four quadrants of requirements, development, testing, and operations and infrastructure, the results suggest continuity is required in all four to support frequent releases. In the case companies, the supporting development practices were usually in place but specific types of testing and the facilities for deploying the changes effortlessly were not. Realigning processes and practices accordingly needs strong organizational support. The responses imply that the organizational culture, division of labor, employee training, and customer relationships all need attention.

With the right processes and the right organizational framework, frequent releases are indeed possible in specific domains and environments. In the end, release practices need to be considered individually in each case by weighing the associated risks and benefits. At best, users get to enjoy enhancements quicker and to experience an increase in the perceived value of software sooner than would otherwise be possible.

Computing Reviews (2012) Categories and Subject Descriptors:

Software and its engineering → Software creation and management → Software development process management → Software development methods
Software and its engineering → Software creation and management → Software verification and validation
Software and its engineering → Software notations and tools → Software configuration management and version control systems
Software and its engineering → Software creation and management → Collaboration in software development → Programming teams
Social and professional topics → Professional topics → Management of computing and information systems → Project and people management
General and reference → Cross-computing tools and techniques → Empirical studies

General Terms:
software process improvement, continuous software engineering, release engineering, agile software development

Additional Key Words and Phrases:
continuous deployment, continuous delivery, continuous integration, DevOps, refactoring, maturity models, deployment pipeline, release models, lead time

Acknowledgements

Pursuing a doctoral degree is akin to a journey that is unique for everyone who decides to embark on the journey. For me, this thesis marks an end to a journey that began a decade or so ago. Speaking metaphorically, one of the publications in the thesis speaks of highways and country roads for achieving certain objectives. I am inclined to think the road taken for this thesis journey resembles a country road. A long windy country road at that. Luckily, I have not had to travel alone. Here, I would like to express gratitude to all the people I have had the good fortune of meeting along the road, starting from the very beginning.

Before I begin my story, I would like to offer my sincere thanks to the honored opponent Eric Knauss hailing from the Chalmers University of Technology and the University of Gothenburg, and the pre-examiners of the thesis. Pre-examiners Ville Leppänen from the University of Turku and Ali Babar from the University of Adelaide did the initial screening of the thesis, making good remarks in the process. Academia thrives from collaboration. Also, review and critical evaluation are at the core of science. Without international and national collaboration, assessing the merit and worth of doctoral theses would not be possible. I appreciate the comments of the reviewing parties, and the time and effort invested in the evaluation of this thesis.

The preparations for this academic voyage of mine were already made a few years before I actually started my doctoral studies. While I was still a graduate student in 2011, university lecturer Matti Luukkainen hinted that there was a teaching assistant position open at the University of Helsinki. A new professor had just started at the department and needed an assistant for his courses. Matti Luukkainen is an amazing lecturer who is genuinely interested in bringing in the best possible teaching methods to his students' benefit. Matti also supervised my master's thesis. Thanks Matti for pointing out the way.

The period before my doctoral studies during which I worked for professor Jürgen Münch was an important one. Research activities in the soft-

ware engineering research group oriented me towards research work. Seeing the role and importance of international conferences and journals was an important lesson for the coming doctoral studies. Writing an article on test-driven development together with Jürgen was particularly instructive for me. Vielen danke, Jürgen.

For the past several years, many of us have worked remotely due to the pandemic caused by the coronavirus. As a result, there have been fewer chances to meet people at offices in many trades and professions. Fortunately, the world was different in 2013. Tomi Männistö had then just started as a professor of software engineering at the University of Helsinki. His office was across the hall in the same corridor where most of our research group was situated at the time. One day I decided to knock on the door and greet the new professor.

From the very first visit, Tomi struck me as an easily approachable person who was willing to lend an ear to anyone walking in the room. Tomi's office was always open for those passing by. I was glad when Tomi accepted to supervise my doctoral thesis. I am grateful not only for Tomi's involvement in all of the thesis publications but also for other support and advice. Over the years, I have had the privilege of hearing about Tomi's travels around the world as well. I have truly enjoyed listening to tales from places like the south pole where penguin colonies inhabit the frozen land. While the trip with the thesis has mostly been an intellectual one, thank you Tomi for being with me on this journey.

At the same time, I would like to express my gratitude to my other two doctoral thesis supervisors Tommi Mikkonen from the University of Jyväskylä and Antti-Pekka Tuovinen from the University of Helsinki. With his wide experience, Tommi has given excellent feedback especially during the writing process of the thesis introduction. Tommi is always looking forward and thinking about the next step, which helped a lot when we were working on the maturity model topic. With Antti-Pekka, I worked closely on many fronts, including collaboration in all research aspects for the refactoring study. Antti-Pekka is well versed in software engineering. It was helpful for me to be able to throw out ideas on the table related to the thesis and discuss them with Antti-Pekka. Having not just one great supervisor but three was more than I could hope for. Thank you Tommi and Antti-Pekka for your support and wisdom as well.

This thesis would not have been possible without the Need for Speed research program (N4S) that launched in early 2014. Need for Speed brought many universities and industrial companies together in Finland. This collaboration and the outline of the research program gave a solid framework

on top of which to build a doctoral thesis. There were many workshops, meetings and gatherings that shaped the research initiatives for the program and for this thesis, too. My warm thanks go to all the people who were involved with the research program. The Finnish innovation agencies of Tekes and later DIMECC (Digital, Internet, Materials, and Engineering Co-Creation) provided the financial and organizational basis that made the program flourish for years.

Working together with other researchers and industrial organizations in Need for Speed proved to be fruitful from the outset. In one of the early gatherings in 2014, we met researchers from the Tampere University of Technology and Aalto University who were also interested in finding out the mysteries behind continuously providing users with new software versions. In particular, I found a shared interest with Marko Leppänen from the Tampere University of Technology.

Research activities with Marko and the others started quickly after our initial meeting. It did not take more than a week or two after which designs for the first study were drawn. The interviews followed shortly after. By late summer, we already had an article draft ready. The joint efforts continued in the same good spirit in other studies. I am grateful for having had the chance to explore the topic with Marko. I also appreciate the efforts of Marko's colleague Terhi Kilamo from Tampere who brought clarity and form to many of the studies.

On the same note, I would like to thank all the other researchers who coauthored the thesis publications and helped with the studies. Many brilliant minds from the University of Helsinki, Tampere University of Technology, Aalto University, and the University of Oulu joined forces for the studies. Diversity of opinion brings forth ideas, viewpoints, and validation of hypotheses in all research stages. This thesis would not have been the same without the contribution of everyone involved in the studies.

Of course, any empirical study is only as good as the collected data. Many companies in the Finnish software industry both inside and outside the Need for Speed research program opened their doors to us. Hearing development experiences directly from people involved in software development was enlightening. My heartfelt thanks go to all the wonderful people and interviewees who dedicated their time to support software engineering research.

By no small measure, the working environment and atmosphere for doing research is created by persons you interact with on a daily basis. Besides our research group at the University of Helsinki, I would like to thank the

good people who walked the halls of the university and brightened the day with friendly conversations from time to time.

Sharing the same office space or hall over the years with group members Patrik Johnson, Max Pagels, Fabian Fagerholm, Hanna Mäenpää, Sezin Yaman, Leah Riungu-Kalliosaari, Myriam Munezero, and Juha Tiihonen was a delight. For instance, Fabian's presence was warmly welcome not only because he was a trusted friend and colleague but also because he could clarify any aspect in software engineering when I was in doubt. I enjoyed the company of everyone else, too. Thanks to Hanna, I got an introduction to the research of open source ecosystems, which was quite interesting. Likewise, I am grateful to Petri Kettunen from our group for providing thought-provoking hints and leads that made me consider the topic area more broadly. I would also like to mention Arto Hellas, whose visits and good humor lifted the spirits on many an occasion. Thank you all.

Ultimately, we all draw strength from our immediate family and loved ones. I would like to thank my family for their continued support over the years. My father introduced me to computers at an early age when *hacking* computers mostly meant bashing the keyboard of a historic Telmac. When she was still with us, my mother encouraged and supported me in my lifelong journey of learning. It is to her memory that I also dedicate this thesis. Finally, I would like to thank my sister and her family for making me feel at home whenever I visited them.

Helsinki, April 2023
Simo Mäkinen

List of Original Publications

Publication I	M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M. V.Mäntylä, and T. Männistö. The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72, March 2015.
Publication II	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*, 80:175–194, 2016.
Publication III	S. Mäkinen, T. Lehtonen, T. Kilamo, M. Puonti, T. Mikkonen, and T. Männistö. Revisiting Continuous Deployment Maturity: A Two-Year Perspective. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1810–1817, 2019. Association for Computing Machinery.
Publication IV	M. Leppänen, S. Mäkinen, S. Lahtinen, O. Sievi-Korte, A. P. Tuovinen, and T. Männistö. Refactoring-A Shot in the Dark? *IEEE Software*, 32(6):62–70, November 2015.
Publication V	L. Riungu-Kalliosaari, S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö. DevOps Adoption Benefits and Challenges in Practice: A Case Study. In P. Abrahamsson, A. Jedlitschka, A. Duc, M. Felderer, S. Amasaki, and T. Mikkonen, editors, *Product-Focused Software Process Improvement*, PROFES 2016, pages 590—597, November 2016. Springer International Publishing.

Author Contributions

Overall, the doctoral candidate had a substantial role in all publications included in the thesis collection while focusing on the primary research objectives of finding out the rationale of frequent releases, charting the software development practices needed to release frequently, and investigating the organizational implications of frequent releases that are the core contributions of this thesis. All the original five publications have been prepared jointly with a number of universities and authors.

The roles and responsibilities of the doctoral candidate in preparing and conducting the studies are highlighted here. Author contributions for each publication are described using the contributor role taxonomy (CRediT) (Allen et al., 2014; Larivière et al., 2021). The current version of the taxonomy has 14 contributor roles ranging from conceptualization of the study to writing manuscripts, matching the nature of work important in different research phases (Larivière et al., 2021).

Publication I: The Highways and Country Roads to Continuous Deployment

Publication I is based on a set of qualitative interviews related to continuous deployment. The doctoral candidate had a role in all research phases. In the conceptualization phase, the doctoral candidate planned the research objectives with the other authors. Preparing and formulating the questions to be used for the semi-structured interview was a joint task in which the doctoral candidate participated by writing and revising interview questions and matching the questions with research objectives. For project administration purposes, the doctoral candidate was in contact with certain companies to acquire company cases for the study. The investigation phase consisted of collecting data with interviews. Acting as an interviewer, the doctoral candidate lead several interviews with help from Author 3. Researchers from each participating university collected data from their respective geographical regions. Author 1 was in charge of most of the interviews.

In the data curation and analysis phases, the interviews were partially transcribed and analyzed with thematic analysis. The doctoral candidate listened to interview records, read interview notes and partial transcripts in order to find common continuous deployment themes from the interviews. Based on the analyzed data and joint discussion, the authors were able to create models for release frequency metrics and release objectives. The doctoral candidate also took part in validating the findings of other researchers by comparing and cross-checking interview notes with interview recordings.

Table 1: Contributor roles per author for Publication I according to the contributor role taxonomy (Allen et al., 2014; Larivière et al., 2021). Legend: *DC=Doctoral Candidate.*

Contributor Role	**Author 1**	**Author 2 (DC)**	**Author 3**	**Author 4**	**Author 5**	**Author 6**	**Author 7**
Conceptualization	●	●	●	●	●	●	
Data Curation	●	●	●	●	●	●	
Formal Analysis	●	●	●	●	●	●	
Funding Acquisition							●
Investigation	●	●	●	●	●	●	
Methodology	●	●	●	●	●	●	
Project Administration	●	●					●
Resources							●
Software							
Supervision							●
Validation	●	●	●	●	●	●	
Visualization					●		
Writing – original draft	●	●	●	●	●	●	
Writing – review & editing	●	●	●	●	●	●	●

The results of the analysis were drawn in the writing phase that included a round of revisions. The doctoral candidate focused specifically on writing the challenges of adopting continuous deployment. All authors participated in the review and editing phase after receiving a favorable initial decision from the publisher. The contributor roles for Publication I are further summarized in Table 1.

Publication II: Improving the Delivery Cycle: A Multiple-case Study of the Toolchains in Finnish Software Intensive Enterprises

Publication II builds on the semi-structured interviews and the data collected for the study reported in Publication I. Thus, the contributor roles for the conceptualization and investigation phases are similar to the role attri-

Table 2: Contributor roles per author for Publication II according to the contributor role taxonomy (Allen et al., 2014; Larivière et al., 2021). Legend: *DC=Doctoral Candidate.*

Contributor Role	**Author 1 (DC)**	**Author 2**	**Author 3**	**Author 4**	**Author 5**	**Author 6**	**Author 7**
Conceptualization	●	●	●	●	●	●	
Data Curation	●	●	●	●	●	●	
Formal Analysis	●	●	●	●	●	●	
Funding Acquisition							●
Investigation	●	●	●		●	●	
Methodology	●	●	●	●	●	●	
Project Administration	●	●					●
Resources							●
Software							
Supervision							●
Validation	●	●	●	●	●	●	
Visualization	●	●	●	●	●	●	●
Writing – original draft	●	●	●	●	●		
Writing – review & editing	●	●	●		●		●

butions as stated for Publication I. In the conceptualization phase, together with the other authors, the doctoral candidate set the research objectives to drill down on the software delivery processes of the companies.

During the data curation and analysis phases, the authors returned to the interview data such as process descriptions and diagrams drawn in the original interview sessions. The doctoral candidate extracted software delivery practices from the data and annotated the practices and used technologies with higher-order themes to be used later in the analysis. To help visualize the software delivery process, the doctoral candidate created graphical illustrations for the manuscript. The authors cross-checked the findings and validated the results of the analysis from company respondents.

In the initial writing phase, the focus of the doctoral candidate was on depicting the software processes used by the companies and making observations on the state of the companies' deployment pipelines. After receiving the reviewer comments, the doctoral candidate edited the manuscript, trying to make sure that all reviewer comments were promptly addressed.

The doctoral candidate had a leading role in the project administration of Publication II. As a corresponding author, the doctoral candidate was the primary publisher contact. The responsibilities included submitting manuscripts, planning and organizing revision rounds, and drafting response letters to reviewers. Table 2 shows how all the roles were assigned in Publication II.

Publication III: Revisiting Continuous Deployment Maturity: A Two-year Perspective

Publication III reports of a longitudinal case study from the industry where a survey was used to collect data about the maturity of software processes in the case company projects. Upon receiving the company survey response data, the doctoral candidate performed data curation especially to the open feedback responses by normalizing terms to make responses comparable. The doctoral candidate also quantified the normalized terms to show the evolution of software development practices in the company. Author 2 and Author 4 had a key role in the quantitative, statistical, analysis of the survey responses. Besides quantitative methods, qualitative methods were used to analyze the data. With Author 3, the doctoral candidate did qualitative analysis on the survey feedback to determine the fit of the custom maturity model to projects in the company.

Results derived from both quantitative and qualitative analysis provided the basis for writing the manuscript. In the writing phase, the doctoral candidate focused on writing down the analysis for project lead times that imply how frequently specific projects are able to release new software versions. The doctoral candidate also authored the technological overview of the projects and the impact of technology on project maturity while editing the manuscript for submission with the other authors. On the side of visualization, the doctoral candidate crafted visual aids for the manuscript in the form of graphs and created visualizations summarizing the results for the conference presentation.

The doctoral candidate had the lead in project administration. In the role of the corresponding author, the doctoral candidate submitted the manuscript for publishing and managed the communication to the publisher. The doctoral candidate also presented the work at the conference where the manuscript was submitted to. Author roles for Publication III are listed in Table 3.

Table 3: Contributor roles per author for Publication III according to the contributor role taxonomy (Allen et al., 2014; Larivière et al., 2021). Legend: *DC=Doctoral Candidate.*

Contributor Role	**Author 1 (DC)**	**Author 2**	**Author 3**	**Author 4**	**Author 5**	**Author 6**
Conceptualization	●	●	●	●		
Data Curation	●	●	●	●		
Formal Analysis	●	●	●	●		
Funding Acquisition					●	●
Investigation	●	●	●	●		
Methodology	●	●	●	●		
Project Administration	●	●			●	●
Resources					●	●
Software		●				
Supervision					●	●
Validation	●	●	●	●		
Visualization	●	●		●		
Writing – original draft	●	●	●	●	●	
Writing – review & editing	●	●	●	●	●	●

Publication IV: Refactoring-A Shot in the Dark?

Publication IV is a report on a study of refactoring practices in the industry. The doctoral candidate had a substantial role in preparing and conceptualizing the study. To gain an understanding of which themes were considered important for refactoring and could thus be useful in the study, the doctoral candidate sought out existing studies on refactoring, turning themes of previous studies into actionable interview questions. Based on a previous study on continuous deployment, the doctoral candidate also made a suggestion of interview questions that could be used to elicit and characterize the general maturity of software development in company cases. All in all, the case interviews and the semi-structured interview protocol for the study were planned cooperatively by the authors.

Table 4: Contributor roles per author for Publication IV according to the contributor role taxonomy (Allen et al., 2014; Larivière et al., 2021). Legend: *DC=Doctoral Candidate.*

Contributor Role	**Author 1**	**Author 2 (DC)**	**Author 3**	**Author 4**	**Author 5**	**Author 6**
Conceptualization	●	●	●	●	●	
Data Curation	●	●	●	●	●	
Formal Analysis	●	●	●	●	●	
Funding Acquisition						●
Investigation	●	●	●	●	●	
Methodology	●	●	●	●	●	
Project Administration	●	●			●	●
Resources						●
Software						
Supervision					●	●
Validation	●	●	●	●	●	
Visualization	●	●	●	●	●	
Writing – original draft	●	●	●	●	●	●
Writing – review & editing	●	●	●	●	●	●

Investigation and data collection in the study was made using semi-structured company case interviews. The doctoral candidate took part in the data collection effort by doing interviews with Author 5 whereas Author 1 had a leading role in conducting the interviews. Analysis of the results was done using thematic analysis where the doctoral candidate read the interview transcripts and annotated text passages with higher-order themes. The higher-order themes turned into headings in the manuscript during the writing phase in which the emphasis of the doctoral candidate was on the risks and benefits of refactoring. Considering the effects of refactoring can be seen important to the overarching theme of the thesis as continuous forms of development need to constantly assess when structural code and other changes are needed. Writing the manuscript also involved a revision round. An overview of the role assignations for Publication IV are further illustrated in Table 4.

Table 5: Contributor roles per author for Publication V according to the contributor role taxonomy (Allen et al., 2014; Larivière et al., 2021). Legend: *DC=Doctoral Candidate.*

Contributor Role	**Author 1**	**Author 2 (DC)**	**Author 3**	**Author 4**	**Author 5**
Conceptualization	•	•	•		
Data Curation	•	•			
Formal Analysis	•	•			
Funding Acquisition					•
Investigation	•	•	•		
Methodology	•	•	•		
Project Administration	•	•	•		•
Resources					•
Software					
Supervision					•
Validation	•	•	•		
Visualization	•	•			
Writing – original draft	•	•	•		•
Writing – review & editing	•	•	•	•	•

Publication V: DevOps Adoption Benefits and Challenges in Practice: A Case Study

Publication V introduces a case study on DevOps based on a set of interviews. During the investigation phase consisting of interviews, the doctoral candidate collected data with Author 1. Author 3 had a key role in the conceptualization of the study and in the data collection. The doctoral candidate verified the analytic themes grouped by Author 1 from the interview data. A model of the perceived benefits and challenges of DevOps was created in the process, which was visualized by the doctoral candidate in the manuscript.

The doctoral candidate had a lead role in writing the original draft of the manuscript. While writing the manuscript, the doctoral candidate selected the most important observations flowing from the analytical DevOps themes annotated earlier. Connecting to the original theme of the doctoral thesis, the doctoral candidate wrote down analysis of which factors hinder the adoption of DevOps and therefore limit release frequency. In addition, the doctoral candidate described the associated gains that DevOps and consequently an increase in release frequency could produce according to the respondents.

Publication V is a conference article which was submitted to an international conference. In reference to the visualization phase of the study, the doctoral candidate was responsible for preparing the presentation and eventually presenting the work at the conference. Table 5 displays the relationships between contributor roles and authors in Publication V.

Previous Thesis

Publication I, Publication II and Publication IV have been previously introduced in a thesis from the Tampere University of Technology (Leppänen, 2017). Publication III and Publication V are not part of previous theses.

Leppänen's thesis (Leppänen, 2017) can be seen as companion to this thesis. The general idea of the interaction of processes, architecture, and infrastructure with delivery speed presented by Leppänen is extended and magnified in this thesis. The blueprint for software processes described here further breaks down continuous software engineering practices to each development stage.

Additionally, the unit of analysis in this thesis is shifted to the level of organizations by offering a longitudinal perspective to software process improvement and focusing more on organizational behavior. In comparison, viewpoints on DevOps and its impact on organizations reported in the thesis extend the original scope of continuous deployment.

Contents

Chapter 1

Introduction

Good help is hard to find, they say. Especially if you are not quite certain what you are looking for. Yet in recent years this has been the case in the software industry. A thorough analysis of job ads shows that companies, both large and small, are looking for release and build engineers who could help them automate software releases (Kerzazi and Adams, 2016). The position of a release engineer is rather new and the companies are uncertain what a release engineer actually needs to know and do.

Supposedly, these information technology handymen should know a thing or two about continuous integration of changes, be adept at managing infrastructure for software systems, have in-depth experience in releasing changes to the public, and have enough understanding to automate all the possible workflows in a software engineering process (Kerzazi and Adams, 2016). That is quite a lot to ask from any single individual. But why are companies looking for these professionals?

Companies are in the market for skilled professionals to handle releases because the expectations in the software industry have changed. Software development in large internet companies has shifted to a more continuous form of development with no clear end in sight as is the case at Facebook (Feitelson et al., 2013). Facebook leverages the full potential of the web platform by being able to deliver smaller fixes every day and major updates every week to its users. Owing to the high frequency of releases and dynamic configuration of features, Facebook can experiment on real users when introducing new functionality without disrupting the experience of all users. It has been argued that other companies want to follow the lead in an effort to build similar competences with the help of the release engineers the companies are so eagerly looking for (Kerzazi and Adams, 2016).

While the software industry has been at the forefront in terms of developing methods for more continuous forms of software development, software engineering research is slowly catching up. Major conferences have established tracks and workshops around the topic labeled continuous software engineering. The 2014 workshop of Rapid Continuous Software Engineering serves as an early example (Tichy et al., 2014). Program chairs of the workshop acknowledged that continuous software engineering pushes the boundaries of agile software development from development teams to the level of organizations as a whole. In their view, to be able to release software changes rapidly, development and testing functions must be well aligned with overall product and release management. To keep up with modern-day software development, the software engineering research field needs to consider the impact continuous software engineering has on software engineering disciplines. At least areas such as software development methodology, requirements engineering, software and system architecture design, software testing, and release engineering are seen to be affected.

More empirical studies are needed to study the phenomenon of continuous software engineering with scientific methods in real industry environments. Prior to 2011, empirical studies on the topic have been scarce (Rodríguez et al., 2017). The scientific field is not completely barren, though. A fair number of studies have been conducted on continuous integration (Karvonen et al., 2017). Continuous integration is a practice that stresses the importance of frequent integration of changes from developers along with automated processes for building and testing software versions (Fowler and Foemmel, 2000). Continuous software engineering practices build on continuous integration but continuous integration alone does not imply frequent release of changes to users (Fitzgerald and Stol, 2014).

Studies that *have* featured continuous software engineering practices and frequent release of changes in the past have been important but have not always been of the highest scientific quality (Rodríguez et al., 2017). Without proper description of the research environment and the data collection and analysis methods, it is more difficult to generalize the results of the studies. Many previous empirical studies are also various experience reports from the industry that rely on interviews. Respondents might speak of benefits of continuous software engineering practices but often there is not much hard evidence to back up the opinions or claims (Rodríguez et al., 2017). Findings based on purely qualitative studies can thus be considered somewhat subjective and the research field could benefit from more quantitative studies where possible (Karvonen et al., 2017). Still, studying software development in a realistic industrial setting and collecting the experiences of industry professionals is essential because laboratory conditions cannot compare to the complex relationships between parties found in the field (Rodríguez et al., 2017).

This thesis aims to seek understanding whether the software industry's quest for release engineers is warranted. More specifically, the focus of the thesis is on release frequency. The research questions of the thesis touch upon topics such as the rationale and mechanisms of releasing changes frequently. Industry experience reports speak of the perks of having a more direct channel to end users but critical scientific evidence on the benefits has been limited so far. Practitioners from the industry have described what actions are necessary to release software frequently but the relation of these actions to software engineering disciplines has not been fully explored. Since continuous software engineering is believed to have an impact on the whole organization (Tichy et al., 2014), the thesis also covers the organizational and working method aspects of releasing changes frequently. By examining the phenomena in real world software industry cases with empirical research methods, it is possible to make a contribution to the scientific body of knowledge that is relevant to the industry as well.

Including the initial introductory chapter, the thesis has been divided into six distinct chapters. Continuous software engineering as a phenomenon is described in greater detail in Chapter 2. The background theory in the second chapter explains the role of different developer, development and operations practices, such as continuous integration, continuous delivery, and continuous deployment that are essential when releasing software changes (Ståhl et al., 2017). In Chapter 3, the focus is on the research method of the thesis. The fundamental research questions driving the thesis are presented in the chapter along with the specifics of the research design. In addition to describing the research methodology applied in the thesis, the chapter shows how data for the thesis was collected and analyzed.

The last three chapters present the actual results of the thesis studies along with further discussion of the implications of the results. Intertwined with the layout of the research questions, Chapter 4 gathers the results of the empirical studies for each research question. The three main themes of the thesis and the implications of the work are further explored in the discussion of Chapter 5. As part of the discussion, the chapter also points out the limitations of the inferences made in the thesis studies. A review of related literature in the chapter positions the findings made in the thesis to other work in the field. Finally, Chapter 6 wraps up the thesis with conclusions.

Chapter 2

Continuous Software Engineering

Who knows, perhaps continuous software engineering started to take form with the advent of agile values and software development processes such as Extreme Programming (XP) (Beck, 2000). Practices in Extreme Programming include continuous integration, which essentially means checking in work frequently and getting early feedback about integration problems. In continuous integration, checking in work refers to integrating developed source code and other artifacts to a version control system (Beck, 2000). At the same time, developers receive early feedback about integration problems when automated tests are executed after the changes have been received. Continuous integration (Fowler and Foemmel, 2000) became regarded as a developer practice in its own right, outside of Extreme Programming. Clearly, continuous integration is an important step towards more continuous forms of software development that it might have catalyzed. Still, continuous integration is a step amongst many in what has recently become known as continuous software engineering (Bosch, 2014; Fitzgerald and Stol, 2014). But what is the continuity about in continuous software engineering and has software engineering been particularly discontinuous in the past?

Software engineering has many practices and stages or phases that are possible to carry out in a continuous manner. In fact, the possibilities are numerous enough across the software development areas of planning, development and operations that there is a whole family of continuous practices dubbed continuous * (star) (Fitzgerald and Stol, 2014, 2017). Continuous software engineering and continuous star are well able to capture the idea that the practices from different development areas are interconnected and cannot work in isolation. In the pre-agile software development era, the stage-gates between development areas have firmly stood on the ground with companies trying later to fit agile processes into the whole (Wallin et al., 2002). Here, discontinuity has meant the rigidity of handing off software from one process phase to another. From development to verification and validation, and from there to launch, the gates have been heavy to open and close. Similar gates still exist with the practices related to continuous software engineering but the emphasis is on the effort to make the gates swing easily.

Each continuous practice has its own position around or between the gates. Continuous practices that have recently been most prominent in research can essentially be broken down into developer, development, and operations practices (Ståhl et al., 2017). Continuous integration is a developer practice whereas continuous delivery (Humble and Farley, 2010), which aims for release readiness at all times, is a development practice. Continuous deployment (Fitz, 2009; Humble and Farley, 2010) is an operations practice that goes beyond release readiness and emphasizes a streamlined release process with frequent software deployment to production. In a sense, these continuous practices can be considered as stepping stones from one stone to another that ultimately lead to a more responsive way of developing software: mastery in continuous integration is required before heading further down the path towards continuous deployment (Olsson et al., 2012).

Individual continuous practices as such do not yet win the day, though. It is the interaction between the continuous practices that in the end creates a solid foundation for a release engineering process of the modern era (Adams and McIntosh, 2016). Release engineering processes consider how source code from the developer's workstation finds its way through the pipeline all the way to the production environment having been tested prior to deployment and final release. Similarly, DevOps focuses on knitting some, if not all, of the continuous practices together to form continuous loops from continuous integration to continuous deployment (Ståhl et al., 2017).

This chapter explains the key components of continuous software engineering. The first theme in Section 2.1 is continuous integration, which forms the backbone of continuous software engineering. Continuous delivery and continuous deployment, which rely on continuous integration, are covered next in Section 2.2. Finally, the integration of continuous practices is examined by taking a look at DevOps in Section 2.3 that closes the chapter.

2.1 Continuous Integration

As a developer practice, continuous integration and its components can be seen to constitute the underlying infrastructure necessary for frequently checking in source code from developers, building new software versions and

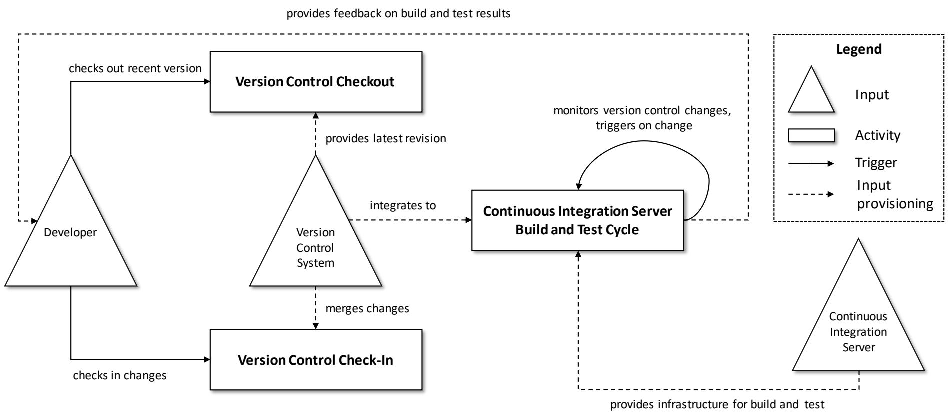

provides feedback on build and test results
checks out recent version
Version Control Checkout
provides latest revision
monitors version control changes, triggers on change
Developer
integrates to
Version Control System
Continuous Integration Server Build and Test Cycle
merges changes
Version Control Check-In
checks in changes
Continuous Integration Server
provides infrastructure for build and test
Legend
Input
Activity
Trigger
Input provisioning

Figure 2.1: Basic process of continuous integration (Beck, 2000; Fowler and Foemmel, 2000).

testing them accordingly (Fowler and Foemmel, 2000). Here, frequently means that integration and checking in code should be done as often as possible, at least once a day, as it has been argued that postponing integration of code changes will only make it more difficult in the end (Beck, 2000).

The centerpieces in continuous integration are the *developer* who is checking in and checking out the code, the *version control system* where the code is stored, and the *continuous integration server* or other build system that fetches newly checked in code from the version control and builds a new software version. This process is depicted in Figure 2.1 using the notation for modeling continuous integration flows (Ståhl and Bosch, 2014a). In the notation, triangles represent inputs, rectangles activities, circles and arrows triggers, and dashed lines input provisioning to other components. As shown in Figure 2.1, typically the developer does not interact directly with the continuous integration server. Instead, the developer works with the version control system that is actively monitored by the continuous integration server triggering the automated build and test when needed. Be it either success or failure, the flow halts in the end by informing the developer of the outcome of the build and test activities in some manner. It has been recognized that in reality, continuous integration practices and exact steps might vary considerably from case to case with certain cases being less continuous and involving activities not found in others (Ståhl and Bosch, 2014a,b). Regardless, the roles of core components and the basic principle behind continuous integration in the ideal model remain the same.

2.1.1 Version Control in Continuous Integration

A cycle of continuous integration starts when a developer has been working on code and feels that either the particular task has been completed or the code is otherwise ready for integration even if only partially complete (Fowler and Foemmel, 2000). The trigger for continuous integration is the developer who has the code changes ready on a workstation. On the receiving end, where the developer checks in code, is a version control system that has a repository for storing project files.

Version control systems have been around for a really long time (Rochkind, 1975; Tichy, 1985) and a good number of the operations are still similar today. Just as was the idea all those years ago (Tichy, 1985), a developer has a two-way connection to the version control system: files can be either *checked out* or *checked in*. When checking out, the latest versions of files in the version control system for a particular project are transferred or updated to the developer's local copy. Checking in is a reverse operation of checking out so that project source code or other files are sent to the version control repository. Besides storing the files, version control systems keep track of the version history of files that makes it possible to store the files efficiently by storing edit operations (deltas) rather than complete files, and to compare files, to merge different file versions, and to branch version graphs into multiple subtrees supporting parallel development of different versions (Tichy, 1985).

While the modern-day version control systems share some of the same characteristics and basic operations as the previous generation version control systems, there have been changes of late. Version control systems that are in use today can be divided intro centralized and distributed version control systems (CVCS and DVCS, respectively) (Muşlu et al., 2014). Previous version control systems were centralized, meaning that code and other files always had to be checked in to a central location, like a network server where the version control repository was hosted. Distributed version control systems, which have taken prominence lately (Brindescu et al., 2014; Muşlu et al., 2014), maintain a local repository that holds the complete history of all repository files instead of a single snapshot like the local repository copies in centralized version control systems (de Alwis and Sillito, 2009). Thus, operations like checking in, also known as committing, and checking out are directed initially at the local repository and not at the central repository. A central repository is in many cases used, nevertheless, as a place through which developers exchange fruit of their labor (Muşlu et al., 2014). Although a central repository has a role in the decentralized version control system paradigm, as opposed to the centralized version control system, in theory any repository could be selected to be a central repository since local repositories hold the full history of changes.

Distributed version control systems offer certain benefits. Developers seem to appreciate the fact that not all changes need to be checked in to the central repository; they can rather work directly with the local repositories allowing incremental workflows and the ability to work offline (Brindescu et al., 2014; Muşlu et al., 2014). Local repositories are particularly handy in open source projects where all project members creating patches might not have all the privileges to a centralized repository but they could still work on a local repository from a distributed version control system and then submit the patch (de Alwis and Sillito, 2009). Along with incremental modes of working, it also appears that the commits (i.e. check-ins) developers perform on repositories tend to be smaller and more frequent with distributed version control systems such as Git than with centralized version control systems such as Subversion (Brindescu et al., 2014). Even with centralized version control systems, daily integrations are certainly possible and many developers report to integrate as frequently as once a day with centralized version control systems (Brindescu et al., 2014). It looks like a fair share of developers are living up to the original ideal of frequent integrations (Fowler and Foemmel, 2000) in continuous integration. Beside these and other benefits, distributed version control systems have one distinct advantage that developers find appealing and that is the branches and branching in the version control tree that offer flexibility to software development (de Alwis and Sillito, 2009; Muşlu et al., 2014).

Branches in version control can be considered relevant for software development because branches affect how developers interact with version control repositories. There are many approaches to version control branch structures of which one alternative is to separate the current *release branch* frozen for specific releases, a *master branch* for on-going development of upcoming releases, and individual feature branches for isolating development of a feature that should be later merged to the master development branch (Adams and McIntosh, 2016). Specific branching models tailored for the decentralized Git version control system follow along these lines with slight alterations by having a *master branch* reflecting production, a *develop branch* for on-going development, *feature branches* for specific features, *release branches* for preparing for upcoming releases, and finally *hotfix branches* that are used to apply critical fixes to the production through the master branch (Driessen, 2010). Branches allow developers to work in isolation although merging branches can cause merge conflicts, which is why some might choose to avoid branches altogether (Adams and McIntosh, 2016). Because of local repositories, creating a branch is seen as a lightweight operation in decentralized version control systems, making it possible to switch between features and tasks with less effort and experiment with code more quickly (Muşlu et al., 2014). In centralized version control systems, branch creation is more expensive due to the non-private nature of branches that need to be synchronized with the central repository on each commit – whereas with decentralized version control systems branches can stay privately in the developer's local repository without the need to share branches with other developers (Muşlu et al., 2014).

Version control branches guide developer behavior that can affect how readily available source and other files are to the main development branch and ultimately the production environment as part of continuous integration. The frequency that developers check in their work to a branch might differ considerably from the frequency of integrating work to the main development branch. While checking in to the branch can be done daily, integrating the branch to the main line could take as long as three weeks or so (Mårtensson et al., 2017). This is reported to be partly due to the hardships and potential merge conflicts involved in merging branches with the main line. The good news is that at least decentralized version control systems make developers feel that they can somewhat increase the code velocity, that is, the speed at which changes propagate to the main branch (Muşlu et al., 2014). When thinking about continuous integration, it should be kept clearly in mind whether the focus is on actual code velocity, branch check-in frequency signifying daily developer activity, or both.

2.1.2 Automated Build and Testing

Once source code and other files required by a project have been checked in to version control, the next step is to build a package out of the files and subject the build to testing (Fowler and Foemmel, 2000). Building and testing are multi-step processes and it is up to the developer team to define which exact steps should be executed.

Building can involve such tasks as compiling source code files and packaging and assembling compiled code to compressed archives like the Jar archives in Java (Fowler and Foemmel, 2000). Of course, not all programming languages are comparable to Java and require compilation. With dynamic programing languages such as Ruby or Javascript, the emphasis of the build and test phase tends to be on testing (Meyer, 2014). Even without compilers spotting compile-time errors as part of the build (Fowler and Foemmel, 2000), continuous integration is still a recognized choice for many projects that have been made on top of dynamic programming languages. In fact, a fair share of the open source projects in Github working with dynamic programming languages are taking advantage of continuous integration: around two out of three projects using the dynamic programming languages Scala or Ruby have continuous integration taken into use as opposed to the compiled Java projects that have continuous integration in about one out of three projects (Hilton et al., 2016). Besides compiling, other build tasks may include preparing environments and databases for testing the build, and deploying the package to testing environments (Brooks, 2008). Regardless of the build steps taken, all necessary files like properties files or scripts required for a successful build should be checked in to version control (Fowler and Foemmel, 2000). Missing or invalid files in version control can cause build failures in the form of compilation errors, which are common reasons for failing builds along with other build configuration issues (Rausch et al., 2017).

For a developer, a build should be sufficiently straightforward to produce. The build steps should be automated so that the execution of a single command should result in a working build (Fowler and Foemmel, 2000). Such a requirement warrants the presence of a distinct build system to handle the build process although shell scripts that execute commands line by line can be a good start. Essentially, build systems and build tools can be categorized by the technology used to *low-level*, *abstraction-based*, *framework-driven* and *dependency management* build systems and tools (McIntosh et al., 2015).

Low-level tools such as make, Ant and Rake define specific targets that are mapped to a number of output files and the manner in which the output files should be created, and by which commands. Abstraction-based build tools can be used to specify and transform a generic higher-level build specification to a lower-level platform-specific build specification. As an example, CMake is an abstraction-based build tool allowing generation of files to several different platforms. For framework-driven build systems like Maven, it is characteristic to require less initial configuration in build specifications as the systems rely more on conventions, instead assuming where to find source code files and such without explicit configuration.

Yet another task for build systems is resolving external libraries for a project by reading required libraries and their respective library versions from build specification files, and downloading the libraries from external repositories as needed. Dependency management tools do just that. Maven – which also functions as a framework-driven build system – and Bundler are some of the known tools for dependency management. Regardless of the type, build systems tend to be rather programming language-specific. For instance, Maven and Ant are most often associated with projects favoring Java, Make and CMake with C and C++ whereas Rake and Bundler tend to be used with the Ruby programming language (McIntosh et al., 2015).

Besides building, automated testing is an important part of the continuous integration build and test cycle where running unit tests focus on verifying class-level functionality and acceptance tests focus on broader aspects of system-level behavior (Fowler and Foemmel, 2000). The degree of test automation should be relatively high when using continuous integration. Writing good automated tests can be tedious and not having enough automated tests in a project can sway developer teams to omit continuous integration altogether (Hilton et al., 2016). Manual or exploratory testing does complement automated testing in certain cases where continuous integration is applied by detecting problematic areas that automated tests miss (Mårtensson et al., 2017). Despite being helpful, excessive manual testing is seen as an impediment to continuous integration as manual tests make the integration cycles less straightforward with additional steps before code integration (Debbiche et al., 2014). Moreover, continuous integration boosts the usefulness of automated tests. Without an automated build and test system that executes tests, there is a danger that developers forgo running tests, resulting in failed builds when tests are eventually run (Hilton et al., 2017). With continuous integration, developers are more aware of the status of the test suite and are more inclined to write automated tests.

Having continuous integration in place does not guarantee that tests would not be broken or fail to pass, though. Test failures account for a fairly large portion of all build failures, at least in certain open-source projects (Rausch et al., 2017). Despite breaking the integration flow, failing tests are helpful in detecting faults and for giving developers timely feedback about potential problems in code and configuration before actual users experience any difficulties (Hilton et al., 2017). Feedback from automated tests might at times be less than immediate if there are many tests to be executed. Regression tests that can take several days to complete can hamper continuous integration cycles by prolonging the test feedback time to developers (Debbiche et al., 2014).

Developers would rather not wait too long for the whole continuous integration cycle with its build and test phases to complete. Ten minutes has traditionally been set as a threshold for build times considered acceptable (Beck, 2000). It appears that the idea of a ten minute build has been well internalized by developers since it is a commonly held belief that builds should not take more than ten minutes (Hilton et al., 2017). Fifteen minutes is already considered too long (Mårtensson et al., 2017). In reality, the average build times including testing might just be under 10 minutes for certain cases but build times around 25 minutes are not uncommon, either (Hilton et al., 2016). Build times can vary a lot from team to team in the same organization, too. A complete build with tests might pass in 5 minutes in one project and in another the build could take closer to 40 minutes (Brooks, 2008).

Continuous integration becomes less useful for developers the longer it takes to build (Hilton et al., 2017). As a side effect, long build times affect developer behavior. Code is no longer integrated frequently in small batches but the size of change sets grow and developers become more reluctant to change code with long build times (Brooks, 2008). For those who prefer working synchronously, a modest delay in integration can offer a chance to reflect on the recent changes – perhaps with a fellow developer – while waiting but when working asynchronously, task switching back to the original task in the case of integration failures can be difficult if there is a long delay between integration and the ensuing build feedback (Beck, 2000). Fortunately, measures can be taken to shorten the time it takes to build and test the most recent changes.

One way to speed up the continuous integration build and test cycle is to optimize potentially lengthy testing processes that are triggered by each checked in change. Known approaches for optimizing testing include the selection and priorization of test cases (Elbaum et al., 2014). If there is a large amount of test cases, it might not make sense to select and execute them all. Test cases or suites that have been known not to fail recently could be safely omitted as long as the frequently omitted tests get selected for execution every now and then (Elbaum et al., 2014). There are some dangers in aggressive test case selection and leaving out test cases as the ability to detect faults decreases and subsequently the chance of introducing undesirable regression increases. Besides merely looking at failed tests as such, test case selection can be based on the combination of analyzing source code changes and finding tests that fail alongside with source code changes (Knauss et al., 2015). In certain scenarios, associating historical test failure data with source code changes can more than halve the amount of tests selected and speed up test execution with a factor of three or so (Knauss et al., 2015).

The build and test cycle can usually be initiated both locally on the developer's workstation and on a remote server that has been tasked for the purpose (Fowler and Foemmel, 2000). Local builds prior to checking in code to version control can reduce the likelihood of introducing defects that could hamper the work of other developers. The remote server, dubbed the continuous integration server, is an essential component without which continuous integration would not be complete (Meyer, 2014).

The continuous integration server is at the core of continuous integration. Keeping watch in the background, the server build jobs are triggered soon after a developer checks in code as illustrated in Figure 2.1 (Beck, 2000; Fowler and Foemmel, 2000; Meyer, 2014). The version control system is continuously monitored for any sign of change at regular intervals like every few minutes. Whenever a change is detected, the continuous integration server checks out the most recent code version and starts executing all the build and test steps defined as part of the build (Fowler and Foemmel, 2000). Once the build and test run has been completed, the respective developers are notified of the results and whether all steps succeeded in some manner such as by e-mail or by other forms of communication (Beck, 2000; Fowler and Foemmel, 2000). Developers can then proceed to remedy broken builds, if needed.

Continuous integration servers are basically servers that have a specific continuous integration system installed. For instance, Jenkins is a continuous integration system that can be installed on-premise on proprietary hardware (Meyer, 2014). Not all continuous integration servers operate on-premises on dedicated hardware. Services that are hosted elsewhere by third parties and offered as a service, such as Travis, have become popular especially with certain open source projects. Out of a large number of open source projects in GitHub, over 40 percent used continuous integration and over 90 percent of these had chosen to use Travis as a continuous integration service solution (Hilton et al., 2016). Setting up continuous integration requires effort and configuring a continuous integration server or service is not as simple as it could be (Hilton et al., 2017, 2016). Having continuous integration in a project can well be worth all the effort, though. Not only do developers feel more confident with their code when it is double-checked by all the steps involved in continuous integration but frequent, small integrations make the whole product easier to push into the production environment (Hilton et al., 2017).

2.2 Continuous Delivery and Deployment

With the help of a number of practices, continuous integration can put developer teams into a working mode where they are in a better position to integrate code faster with shorter iterations and be provided with timely feedback on the quality of recently integrated changes (Hilton et al., 2017).

Out of the family of continuous practices, what more can continuous delivery and continuous deployment bring to the table?

In terms of software delivery, a key question is what happens to the whole set of recently packaged code and other bits of software after the initial rounds of continuous integration testing have successfully finished? With continuous integration, this is where the flow mostly stops. Continuous delivery (Humble and Farley, 2010) and continuous deployment (Fitz, 2009; Humble and Farley, 2010) take a step further by moving the focus to ensuring release readiness and actually shipping the goods over to production environments where the end users can enjoy the recent changes. Passing software bundles from one system to another and ultimately to production is no longer simply a matter of how the developers organize their daily work but a principle by which the organization or one of its projects live. To accommodate the increased interaction between different systems and the required extra effort to prepare software packages ready for release, continuous delivery is more generally a *development practice* instead of being merely a developer practice like continuous integration (Ståhl et al., 2017). When deliverables are frequently placed into production environments as part of continuous deployment, even more coordination and orchestration is required, making continuous deployment an *operations practice* (Ståhl et al., 2017). Continuous deployment induces a different, more experimental and agile, way of thinking about software releases and product planning with production environments facing end users who get to experience frequent software changes first hand (Olsson et al., 2012).

Continuous deployment puts the individual practices of continuous integration and continuous delivery together, and adds its own set of practices that are particularly geared towards shifting the deliverables from one system to another. Automated testing in its various forms is still an essential part of continuous deployment, even more so since there is a chance that any code or other recently constructed piece of software is finding its way to the production environment rather soon (Parnin et al., 2017). Once the preliminary tests pass, staging environments can then be used for further verifying release readiness. Testing in a staging environment is followed by sufficiently automated deployment to production environments, although production deployment does not necessarily mean that all features of the software update are *released* immediately to the users (Parnin et al., 2017). Features can be made invisible for the time being or released only to a small proportion of users to gain further understanding on how the users respond to the changes. Whether making a release available to a smaller sample of users or all of them, monitoring the production environment and the

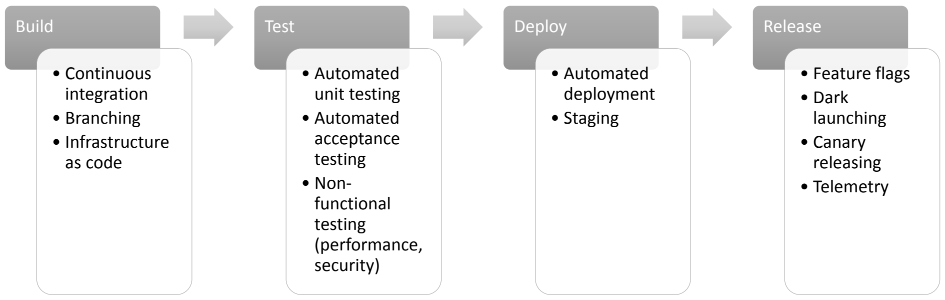


Figure 2.2: Practices of continuous delivery and continuous deployment in the stages of a deployment pipeline (Humble et al., 2006; Humble and Farley, 2010; Adams and McIntosh, 2016; Parnin et al., 2017).

running application for user behavior or signs of trouble are good habits to have when deploying often to production (Parnin et al., 2017; Adams and McIntosh, 2016).

Putting all, or most, of the practices of continuous integration, continuous delivery, and continuous deployment together form a pipeline from the hands of the developer triggering the software flow all the way to the production environment. A deployment pipeline (Humble and Farley, 2010) with all the parts in place have been called deployment production lines (Humble et al., 2006) or release engineering pipelines (Adams and McIntosh, 2016) although in practice a single project might not apply all of the principles (Humble et al., 2006). In this section, the deployment pipeline is deconstructed to smaller components to get an overview of the process, and a closer look is afforded to releases and releasing new software versions in production.

2.2.1 The Deployment Pipeline

Roughly speaking, a deployment pipeline is assembled from four larger parts or areas of practice: *building*, *testing*, *deploying* and *releasing* (Humble and Farley, 2010). Automated scripts for building and testing with proper configuration management along with automated deployment to various environments are the hallmarks of a functioning deployment pipeline. Figure 2.2 illustrates some of the practices and activities linked to the various stages of the deployment pipeline, commonly associated with continuous delivery and continuous deployment (Humble et al., 2006; Humble and Farley, 2010; Adams and McIntosh, 2016; Parnin et al., 2017).

Activities in the build stage of a deployment pipeline are well aligned with those found in continuous integration. Developers use build systems and check in code to version control systems, aware of the possibility to use version control branches for efficiently managing changes from multiple developers working on the same code (Adams and McIntosh, 2016). With the prospect of delivering the changes rapidly to further test, staging and production environments in mind, emphasis is put on efficient builds (Humble et al., 2006). The arrangement and building of software modules should be such that all necessary source code is compiled in a single run, preferably without many compiled intermediate binaries that other projects might rely on. Having fewer build and compilation stages of other projects triggered helps to minimize the time for the build stage of a single project. This mode of thinking that puts speed at the forefront is an overarching principle in the design and implementation of a deployment pipeline (Humble and Farley, 2010). Automated and manual processes that are part of the pipeline can be timed to understand which of the processes have the greatest effect on cycle time and thus have room for improvement.

A version control system is one of the primary junctions in a deployment pipeline also for the reason that a version control system can serve other purposes beyond storing software code and related artifacts. For ensuring proper configuration management, version control systems can be used to store infrastructure blueprints for servers and complete copies of virtualized servers in the form of virtual machine images (Humble and Farley, 2010). Stored infrastructure configuration written in a specific infrastructure programming language and virtual machines help to establish similar environments for development, testing and production, which facilitates the rapid flow of software changes from one environment to another (Humble and Farley, 2010; Adams and McIntosh, 2016).

Testing as part of the deployment pipeline consists of performing automated and manual tests split into test suites that test the validity of the built software version on different tiers with emphasis on automated tests (Humble et al., 2006). In the lower-level tiers, testing includes performing unit-level and other tests that focus on isolated code components and units. Unit tests are followed by higher-order functional and integration tests that can take advantage of production-like testing environments. Passing unit and integration tests indicate that the software version under test might very well be a potential release candidate and ready for the highest category of acceptance tests.

Deployment and the capability to deliver packaged software versions to other environments becomes important at this stage at the latest, as acceptance tests should at the very least be tested in an environment that resembles the production environment (Humble and Farley, 2010). Acceptance tests and the acceptance criteria for the tests can be either functional or non-functional. Functional acceptance tests, which can be automated, are driven by user stories to determine whether the implementation fulfills the needs set by the user regardless of the technicalities in the background. Working with the customer and users in short cycles, continuous deployment favors rapid validation of what is being implemented at any given time (Olsson et al., 2012). Acceptance tests as such have their place in ensuring that the perspective of the user is not forgotten. Acceptance test criteria can also be non-functional in the sense that properties such as performance and security can be tested in a production-like environment to get a sense of whether the system and the current release candidate respond according to the set performance thresholds without serious security vulnerabilities (Humble et al., 2006).

Preferably, acceptance testing should be done in an environment that resembles the production environment, which calls for proper configuration management and provisioning of test infrastructure. If the creation of testing environments for acceptance testing is not properly automated, developer teams might not get the immediate feedback necessary for continuous delivery and deployment. Setting up testing environments without automation takes considerable time not only from the developers but from other people as well and as a result acceptance testing tasks do not move in the development workflow towards deployment to production, piling up when testing resources eventually become available (Chen, 2017). Replicating production environments for testing is not an easy task when configuration between environments can still vary due to various reasons, making it difficult to ascertain the release readiness of the built version (Parnin et al., 2017). A high rate of incoming changes and inconsistency in environment configuration means testing is partly done in production, which is an undesired situation and increases the possibility of observed failures.

Fine-tuned deployment practices and automated deployment are prerequisites for efficiently testing a particular software version and ultimately shipping the tested version to a production environment with minimal overhead. Without proper deployment practices, the process of deploying a new version for testing in another environment can take days instead of the mere hour or so it takes with automated deployment (Humble et al., 2006). Continuous integration servers, such as Jenkins, enhanced with capabilities for workflow orchestration and deployment can provide assistance in automating the end of the deployment pipeline suitable for continuous delivery and continuous deployment (Armenise, 2015). Workflows in Jenkins can be composed of any number of arbitrary phases that build software packages, execute automated functional and non-functional tests in various environments, process the resources in any manner and deploy the packages to various environments. Phases that require human intervention can be configured to halt, waiting for additional permission to continue with the subsequent phases upon approval by *promoting* the build to the next phase. Workflow tools that are able to illustrate and visualize parts of the deployment pipeline (Armenise, 2015), help raise developer awareness of parts of the pipeline that still require attention in terms of continuous delivery and continuous deployment (Chen, 2017). Automating deployment gives developers the chance to deploy changes to production at will that can also strain and put pressure on developers as they are responsible for their own changes, needing to keep alert and monitor production systems for errors when the changed code is invoked for the first time by users (Parnin et al., 2017).

Deployment and specifically deployment to production environments does not necessarily mean that a new software version is immediately *released* to all users. Releasing software changes right away was incorporated in the original line of thought of continuous deployment (Fitz, 2009). Fewer changes at once was seen to make it easier to pinpoint and correct any possible failures in production, owing to the fact that there would be less ground to cover and where to look for failures. Chiefly, the idea still rings true and remains valid but in the continuous deployment parlance of late there has been a tendency to clearly distinguish between deployment and release. Deployment refers to the capability of moving necessary files related to a particular software version from one environment to another with appropriate environment configurations while releasing means making the version available to the end users (Adams and McIntosh, 2016). Continuous delivery and continuous deployment deal with both concepts – deployment and release – but the terms are not entirely interchangeable.

Continuous delivery can be seen as a precursor to continuous deployment. What separates the two is their relationship to deployment to production. There is a shared understanding that in continuous delivery, the deployment pipeline has been set up so that development teams have the capability to push and deploy recent changes to production but deployment and release have not been fully automated unlike in continuous deployment (Ståhl et al., 2017; Fitzgerald and Stol, 2017). Continuous deployment goes a step further in ensuring that changes made in development are effectively deployed to production and released to end users with a rapid schedule.

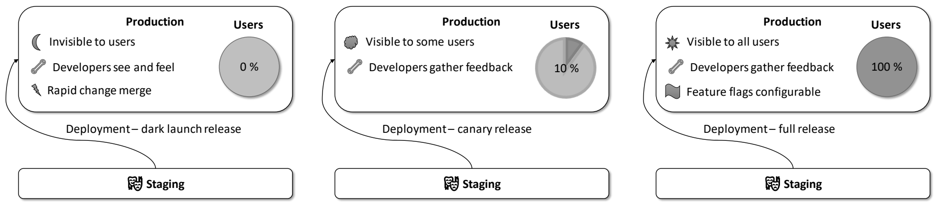
Conceptually, this difference in the operation of the last part of the deployment pipeline is the major difference between continuous delivery and continuous deployment although there has been some ambiguity in the use of the terms and their meaning has been mixed over the years (Fitzgerald and Stol, 2014, 2017).

An automated deployment pipeline can make the task of releasing new software versions to the end users easier, especially so if the pipeline extends all the way to the production environment as it should in continuous deployment. Releases and releasing changes, however, can take many forms. Strictly speaking, if a deployed software change or feature that is part of the release cannot be used by the end users or is not otherwise available, the change has not been released yet. Forms and modes of release exist where this is exactly the case as new code and other changes can be silently deployed to production but associated features remain disabled for end users with the help of feature flags or then the whole feature is built incrementally step-by-step with dark launches (Parnin et al., 2017). Other options include releasing changes only to some users but not to all of them with canary releasing (Humble and Farley, 2010; Adams and McIntosh, 2016; Parnin et al., 2017).

Beyond disabling individual features and releasing versions only to a small subset of users lies the most experimental release and deployment methodologies where end users are exposed to *different* versions based on a certain logic in dividing users into various groups. Experimental methods like A/B testing are part of this continuum in which implicit user feedback and telemetry from the production environment guides development efforts (Adams and McIntosh, 2016). In continuous software engineering, continuous innovation and continuous experimentation are linked to working methods that combine frequent releases of continuous deployment with user centered experimentation (Fitzgerald and Stol, 2017). Such methods form the basis of experiment systems that are at the top of the conceptual user centered agile software development staircase (Olsson et al., 2012) – methods that require the capability to deploy and release often to work in a sensible manner. The next section covers some of the release models that can be used in conjunction with continuous deployment.

2.2.2 Releases

A well-automated deployment pipeline makes software changes fluid, flowing from one end of the pipeline to the other with ease but a developer task related to a change cannot be truly defined as done if the change has not been released or made available to the users (Humble and Farley, 2010).



(a) Dark launching **(b)** Canary releasing **(c)** Full release

Figure 2.3: Software changes deployed to the production environment can be hidden from all or some users with dark launching and canary releasing models to prepare for a full release (Humble and Farley, 2010; Adams and McIntosh, 2016; Parnin et al., 2017).

Every change that is ready to be deployed or that is alternatively deployed directly to the production environment, however, does not form a meaningful feature that could readily be taken into use by the users (Humble and Farley, 2010). *Dark launching* and *feature toggles* support such partial deployment of changes and provide the minimal exposure or no exposure at all to users (Adams and McIntosh, 2016; Parnin et al., 2017). Gradual rollouts with *canary releasing* are useful to making the changes available to some, but not all, users to determine whether the changes work well enough and are received favorably by the users (Humble and Farley, 2010; Adams and McIntosh, 2016; Parnin et al., 2017). A change that is made available to all users is a release that has possibly gone through the various release phases but developer and release teams can also skip all of the intermediate phases by deploying the change to the production environment without restrictions on user visibility. Figure 2.3 illustrates the main features of the various release models starting from dark launching (Figure 2.3a) and canary releasing (Figure 2.3b) that are useful in preparing a full release (Figure 2.3c) later on.

Dark launching and feature flags defy logic at first glance. Why would anyone want to make a release that has no evident effect on how the system works from the users' perspective? One reason lies with the hardships involved in integrating work from version control system branches that have not been integrated to the main line of development in a while (Mårtensson et al., 2017). Developers practicing continuous integration saw benefits in integrating often to the main development branch with small batches of changes, and the same applies in continuous deployment with dark launching. A smaller set of changes is easier to merge and large architectural changes in software architecture less painful to piece together by deploying the changes often by the principle of dark launching (Parnin et al., 2017).

Likewise, feature toggles or feature flags allow the same flexibility as dark launching, making it possible for developers to try out and run code in the production environment by including the necessary software services in the set of changes deployed to production but hiding components from the users if certain Boolean expressions or other conditions in code evaluate to true or false (Adams and McIntosh, 2016; Parnin et al., 2017). Feature flags can be used either at compile time or at run time. Compile time flags that are disabled by setting the condition to false might not end up in the compiled code at all depending on the compiler. Run time feature flags are evaluated when the code is executed so the value for the feature flag – whether to show or hide a feature – can be read from a dynamic configuration source like a configuration server. What is helpful in development is also helpful when a released feature is not working as expected since the configuration source can be used to disable the feature without making a new release to fix the possibly faulty feature.

Release strategies where the changes are deployed and released to a share of the users while withholding the changes from others derive from the principle of canary releasing (Humble and Farley, 2010; Parnin et al., 2017). The decision which users get to interact with the released feature can be based e.g. on the geographical region where the user is from (Adams and McIntosh, 2016). Canary releasing functions much like a dynamically configured feature flag: if the result of the evaluation is true, the feature is shown to the user and if it is false, the feature is hidden. The configuration and evaluation of a release canary feature might just involve more logic and the value of the configuration is not globally the same to all of the users. Monitoring the production environment closely is a necessity with release canaries. Tracking feature usage through telemetry is needed to see how the users interact with the features and to see that the new features are not causing any system failures (Adams and McIntosh, 2016; Parnin et al., 2017). The general objective of canary releasing is to gather implicit or explicit feedback from the users (Humble and Farley, 2010).

Eventual full releases to all of the users follow the intermediate release phases if everything seems to be in working order and the quality metrics from the production environment look acceptable. Telemetry data is still useful when a release has been done to catch the occasional program exception or two and to generally get a feeling of how new features are being used after a release (Parnin et al., 2017).

Continuous deployment as a practice does not strictly dictate whether to silently push changes to production with dark launching, or to use release canaries to try out new features with a limited audience, or to deploy and release all changes without prior experimentation on users. The general expectation for continuous deployment is to have changes deployed and released quite often (Fitz, 2009). Since the release frequency is not locked as such, it has been suggested that the concept of *continuous release* should be added to the family of continuous software engineering practices to differentiate between deployment and actual release of changes (Ståhl et al., 2017). Continuous deployment still encourages developer teams to release more often, turning months between releases into weeks or even days (Chen, 2017).

2.3 DevOps

The primary concern of continuous deployment is getting changes quickly through the deployment pipeline and preparing a release to the users who can then enjoy the latest changes in the production environment. A developer who pushes changes to the deployment pipeline by writing the code and automated tests, and possibly deciding when to release a particular change, has a lot of responsibility. A developer has to be a master not only of development but of quality assurance and of operations as well. Development skills are evident in programming, quality assurance skills in testing and operations skills in orchestrating and managing network infrastructure and server environments before and after releases. As the name of the concept suggests, DevOps has its roots in this kind of thinking where the boundaries of particular roles in a development team have become blurred (Roche, 2013). No more strictly separated competences and people, throwing the ball over the wall to operations upon releases. Team aspects and bridging the separate worlds are important to DevOps, yet conceptually DevOps is a broader topic that covers the whole lifespan of a product (Roche, 2013).

DevOps has objectives that are similar to continuous deployment. The objective set for DevOps to reduce the time it takes for changes to propagate from development to production with sufficient quality ensured by automated tests (Bass, 2018) aligns well with the objective of continuous deployment. DevOps stresses the importance of automated build and continuous integration just as continuous deployment does (Ebert et al., 2016). The shared identity of DevOps and continuous deployment can perhaps be explained by the fact that continuous deployment is sometimes seen as a *part* of DevOps (Zhu et al., 2016; Ståhl et al., 2017; Bass, 2018) as illus-

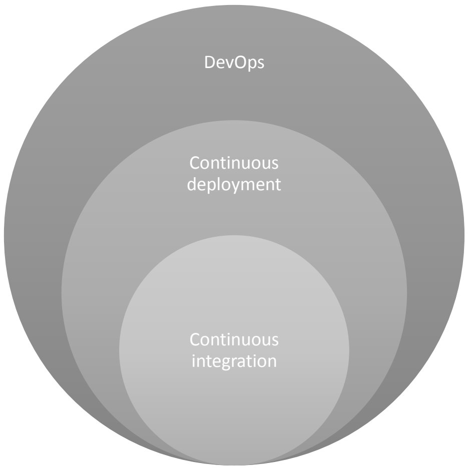


Figure 2.4: DevOps aims to unify development, testing and operations and its practices for reducing cycle time include continuous deployment and continuous integration among others (Zhu et al., 2016; Ståhl et al., 2017; Bass, 2018).

trated in Figure 2.4. Besides enclosing continuous deployment, DevOps has other notable characteristics that are related more to team structures, roles, and operations practices in general.

Bridging the chasm between developers and operations personnel with a certain regard to the whole lifespan of a product is driven by a different mindset to software development, development practices, and tools to support the way of working. According to the mindset, every person in the developer team is more or less responsible for the maintainability of a software product being developed and generally accountable for each change released to the users (Roche, 2013).

Team composition and structure of a DevOps team should reflect the added responsibility so that the team should have sufficient skills to deal with the operations context at its disposal. Instead of having horizontally aligned teams where each specialty forms their own department and team such as development, quality assurance and operations, there should be vertically aligned cross-functional teams that have the ability to develop, test, deploy, release, and monitor environments (Balalaie et al., 2016; Ebert et al., 2016; Dörnenburg, 2018). Such empowered teams serve the purpose of avoiding rigid handovers between teams and specialties where responsibility is shifted from team to team after finishing a particular phase in development; responsibility in DevOps teams is shared.

Preferably, DevOps teams should be small in size (Balalaie et al., 2016; Dörnenburg, 2018). A group of ten people or so well versed in development, quality assurance and operations can work more effectively on delivering individual software components or services. Thus, a single DevOps team might not actually work on the entire software product and its components but just a small part of it. These *microservices* are used to decompose software into independent services that communicate with each other through service interfaces that depend on service contracts (Balalaie et al., 2016).

Microservices and service-oriented architecture help teams to work with a degree of freedom on a service they are responsible for (Dörnenburg, 2018). A microservice can have its own deployment pipeline separated from other services and components that allows independent deployment and release (Balalaie et al., 2016). DevOps can be employed successfully without microservices since bridging development and operations is possible by other means but implementing microservices does require a streamlined deployment process such as advocated by DevOps (Ebert et al., 2016). In other words, DevOps does not need microservices but microservices can be helpful. Due to the independence of individual microservices, change sets can be small and focus on a single service, which can in turn help to attain the ideal of continuous deployment and DevOps (Bass, 2018).

Monitoring and telemetry were important in continuous deployment but DevOps puts even more emphasis on these areas, making sure that the operational environments are up and running properly. The infrastructure as code principle with separate version control repositories for server configurations and such is strongly encouraged in DevOps to recreate similar environments for all development phases (Ebert et al., 2016; Bass, 2018). Further assurance that all operations environments meet the same requirements and that the code has been deployed successfully can be made with specific compliance checking (Callanan and Spillane, 2016). Compliance checks look at the status of processes on the server and verify the existence of binaries and other libraries. Monitoring involves collecting and analyzing server and application health statistics with tools suited to the purpose (Ebert et al., 2016). Application logs on servers are an important source of information on application exceptions (Ebert et al., 2016; Balalaie et al., 2016). Logs can be centrally monitored in a DevOps fashion with good search indices in order to react quickly to any problem that might surface when deploying new versions or in the day-to-day operation of a software system.

DevOps is a broader concept than the other continuous software engineering practices described in this chapter. The true meaning of DevOps has been contested and it can be seen not as a single practice but a collection of different values, principles, methods, practices and tools to unify development, quality assurance and operations (Ståhl et al., 2017). DevOps is perhaps more about fostering a culture of collaboration than anything else (Ståhl et al., 2017; Fitzgerald and Stol, 2017). Nevertheless, the nature of this collaboration should bring cross-functional development teams closer and ultimately make it easier to ship releases faster while considering the whole lifespan of a software product or service.

Chapter 3

Research Method

Software engineering is a field that is open to many research methods such as case studies, surveys, controlled experiments, or simulations to name a few (Stol and Fitzgerald, 2018). Research questions drive the selection of a research method for studies as some types of questions are better addressed by specific methods (Yin, 2014; Wohlin and Aurum, 2015). The research setting, i.e. the environment in which research for the study is conducted, also has an impact on the applicability of research methods (Stol and Fitzgerald, 2018). *Natural settings* in the field are more realistic than *contrived settings* in the laboratory. Empirical observations can be made in the field and in the laboratory but in *non-empirical settings*, research is done at a purely theoretical level. In *neutral settings*, the researcher tries to minimize the effects of the environment because there is little control over the environment of the test subjects like is the case in questionnaires found in survey studies.

Research settings differ in terms of how *obtrusive* they are and how well the research findings can be *generalized* (Stol and Fitzgerald, 2018). Realistic settings in field studies are not very obtrusive but it is more difficult to generalize the findings compared to more obtrusive and controlled laboratory experiments in contrived settings. Control over the research setting comes at the price of realism.

This chapter describes the research setting and presents the research method for the series of field studies conducted for the thesis. A series of field studies is a good starting point to explore and understand in which light frequent software releases are seen in a realistic setting of companies engaged in software development. The important research questions that drive the selection of research methods and are at the core of the studies are covered in Section 3.1. A more detailed overview of the research design for each individual study in the thesis along with steps of case selection, data

Table 3.1: Research questions of the thesis.

Identifier	Research Question
RQ1	Why should software releases be frequent?
RQ2	How can a software engineering process be organized in order to release software frequently?
RQ3	What are the implications of frequent software releases to organizing work?

collection, and data analysis is covered in Section 3.2. An overview of the five publications in the thesis is provided in Section 3.3, where the research settings and outcomes are summarized study by study.

3.1 Research Questions

In the quest of examining the phenomenon of frequent software releases, the primary focus in this thesis is on three distinct themes divided into three research questions. The first theme and research question (RQ1) puts an emphasis on the rationale of frequent releases: why should releases be frequent in the first place and what is to be gained from releasing software frequently? Given that it is desirable to aim for frequent releases, what would it require from the whole development process and the development team in order to prepare releases with a rapid schedule? In the second theme (RQ2), the highlight is on development processes that enable frequent releases. Software development processes are not transformed on their own, however, but require the strength of willpower in those in charge and the collaboration of many people engaged in software development in different roles. The final research question (RQ3) seeks to explore what impact frequent releases carry in terms of organizing work and how people need to adapt to their new roles. These research questions are summarized in Table 3.1 and further elaborated in this section.

The research questions are explored in five distinct publications that each address one or more of the research questions. The rationale behind frequent releases for RQ1 is covered in the industry interview and surveys studies reported in Publication I, Publication II, Publication III, and Publication V. Software engineering process factors, which are at the core of RQ2, are present in all of the five publications. Organizational behavior, the changing role of development teams and their interaction with other key parties, and the organizational forces required to support the adop-

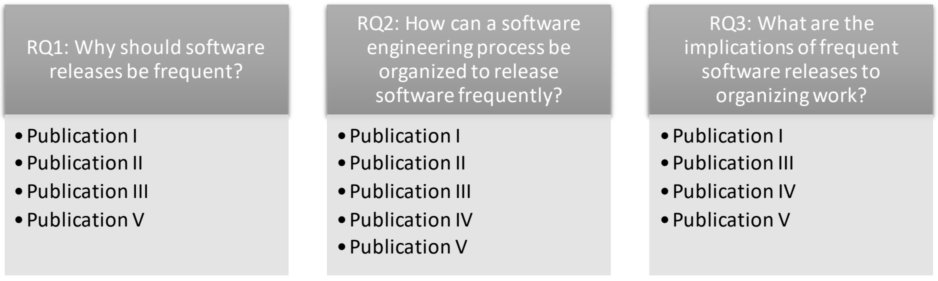

RQ1: Why should software releases be frequent?
• Publication I
• Publication II
• Publication III
• Publication V
RQ2: How can a software engineering process be organized to release software frequently?
• Publication I
• Publication II
• Publication III
• Publication IV
• Publication V
RQ3: What are the implications of frequent software releases to organizing work?
• Publication I
• Publication III
• Publication IV
• Publication V

Figure 3.1: Each publication in the thesis addresses one or more of the research questions.

tion of frequent releases are key elements of RQ3. Elements of RQ3 are especially considered in Publication I, Publication IV, Publication III and Publication V. Figure 3.1 further illustrates the relationships between research questions and publications in the thesis.

RQ1: Why should software releases be frequent?

Large companies such as Facebook have already introduced deployment processes in which developers can opt in to have their changes released during the same day and thousands of changes are pushed to the users each day (Savor et al., 2016). As speed is not a virtue by itself, it is only rational to ask, why should releases be frequent in the first place? What is the value that increasing the release frequency brings? Instead of having weekly releases, a software company could choose to release software biweekly, monthly or every six months or so, or whenever it suits them best. In some cases, companies have turned an infrequent release process where releases are six months or more apart into weekly and daily releases (Chen, 2017) but why weren't the lengthier processes suitable or good enough? It is worth also to inquire whether the domain for which the software is being developed makes a difference. These are some of the questions covered as part of the examination of RQ1.

RQ2: How can a software engineering process be organized to release software frequently?

Software development practices favoring frequent releases to production environments for all users to enjoy, such as continuous delivery (Humble et al., 2006; Humble and Farley, 2010) or continuous deployment (Fitz, 2009), have grown from the practical knowledge and ideas of software professionals in the field. The constituent parts of the deployment pipeline used in continuous delivery have been outlined earlier (Humble et al., 2006; Humble and Farley, 2010) but not all of the practical ideas have been written down as clear playbook guidelines for software engineering. Without a common understanding until recently, there has not always been agreement on what concepts like continuous delivery, continuous deployment, and DevOps mean in practice although they seem to share similar objectives (Ståhl et al., 2017). With this in mind, RQ2 seeks to find out which activities in a software engineering process help or hinder frequent releases. How should a modern deployment pipeline be formed so that design ideas flow freely from their inception through development, testing, and release to the end user? What is the difference to a traditional style of software development where the releases are more infrequent? Should the development team have a particular structure? Such concerns form the basis for RQ2.

RQ3: What are the implications of frequent software releases to organizing work?

A software process supported by the right technological choices can help people in a development team to work effectively in a uniform manner towards more frequent releases. Putting a practice into use, however, requires more than just processes and tools. Making a change to the current way of working requires changes in people, their attitudes, and the surrounding culture, which might all be much more difficult to influence than technology and tools (Fitzgerald and Stol, 2017). Development teams are not the only ones that need to adapt to new ways of working. The surrounding organization and its key partners such as the customers must redefine their relationships with each other if releases are to be more frequent. By studying the opinions, attitudes, hopes, and fears of the people who work in software engineering about increasing release frequency, it could be possible to gain some understanding in which way the organizational culture would need to be changed.

3.2 Research Design

Research questions are the driving force behind all research settings as the form of the questions can limit available research options (Yin, 2014; Wohlin and Aurum, 2015). Questions in the form of *how*, *why*, *what*, *where*, and *how many* have a relationship to research methods that are most suitable for answering questions in a specific form (Yin, 2014). Case studies, for instance, are especially suited for research questions in the form of *how* and *why* whereas surveys are useful when answers are sought for questions like *how many* or *how much*. Research questions and methods are also interlinked with research purposes. The purposes of research can for instance be to *explore* a particular phenomenon, which allows a wider range of research questions and methods to be used (Yin, 2014). *Descriptive* and *explanatory* research purpose types are types that go beyond exploring a phenomenon with the aim of better understanding why an event leads to an outcome in the case of explanatory research purposes (Runeson and Höst, 2009; Yin, 2014). The selection of a research method also rests heavily on the fact whether behavior of the subjects needs to be controlled like in controlled experiments (Yin, 2014). When studying past events or historical archives, directing test subjects is no longer an option so the study's chosen emphasis on contemporary or historical events has an impact on the research method, too.

The long path from research questions to research findings can be seen as a series of decisions that govern which type of study is being conducted, which research methods are applied, in which manner the data is collected and finally how the data is analyzed. For illustrating research designs, eight central decision points grouped into *strategy*, *tactical*, and *operational* phases can be used to depict the decisions made at each point and phase in research (Wohlin and Aurum, 2015).

The strategy phase consists of the decision points for research outcome (decision point 1), research logic (decision point 2), research purpose (decision point 3) and research approach (decision point 4). Research outcome defines whether the research is basic research or applied research seeking an answer to a specific problem. Research logic determines the logic behind reasoning for the study, be it either inductive reasoning from data or deductive reasoning for testing a theory. The decision point for research purpose relates to the exploratory, descriptive or explanatory nature of research and the overall research objective. Research approach deals with the ontological and epistemological foundations, the theory of knowledge, on which the study is based. Experimental studies tend to have a positivist approach to research while less objective approaches can for instance be interpretivist or even critical, rejecting the possibility to obtain truly objective knowledge.

In the tactical phase of research design, the decisions are about the research process (decision point 5) and the research methodology (decision point 6). The research process can be qualitative focusing on verbal or other accounts of study subjects, quantitative with a focus on numerical data, or a mixed process of the two. Research methodologies in the decision-making

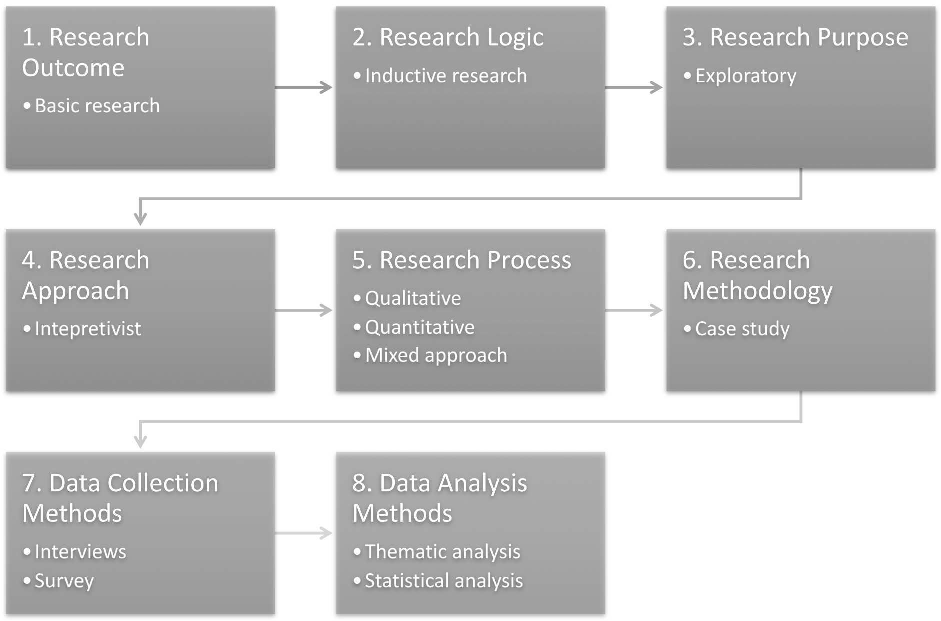


Figure 3.2: Research design for the thesis studies following the eight point research decision-making structure (Wohlin and Aurum, 2015).

structure are broader concepts than research methods but are identifiable by research method names such as case studies or action research.

Operational decision points in the last phase are concerned with data collection methods (decision point 7) and data analysis methods (decision point 8). Interviews are suitable for collecting qualitative data but the data collection methods are different for quantitative data, which is better collected with surveys and experiments. Data can be analyzed with the help of several distinct analysis methods that are coupled with the whole research process and data collection methods. Qualitative data from interviews can be analyzed, for instance, with thematic analysis or with grounded theory analysis methods whereas quantitative data can be subjected to statistical analysis. Ultimately, analysis of the data leads to the research findings, which is the final step of the operational phase in the research decision-making structure.

Putting all the decisions from the strategy, tactical and operational phases together, Figure 3.2 summarizes the decisions made for the research design of the studies in the thesis. These decisions are further elaborated and expanded in this section.

The first four strategic phase decision points (decision points 1–4) illustrated in Figure 3.2 sets the foundation for the the study design of the thesis studies. For the research outcome, the studies are best characterized as being basic research. Basic research is more fitting in this case since the focus is on understanding the circumstances when and how release frequency should be increased, not on solving an applied research problem as such. In the absence of a specific theory to be tested, the research logic is inductive with observations flowing from the data collected. Research questions in the form of *how* and *why* are usually associated with explanatory studies (Yin, 2014), but the primary research purpose is exploratory despite the explanatory elements. The purpose is to broaden the understanding of the phenomenon with exploratory research. Seeing the impacts of increasing release frequency and evaluating the whole software engineering process through the eyes of the study participants in their respective environments is important in gaining understanding, which makes the research approach interpretivist.

For the tactial phase (decision points 5–6), Figure 3.2 shows the research process and research methodology of the studies. By focusing on the subjective experiences of the study participants and their description of the circumstances in which they do software development, the research process is predominantly qualitative. Several studies in the thesis have, however, quantitative qualities and a mixed research process with both qualitative and quantitative components is used in several cases. A mixed process is used especially in the study reported in Publication II where qualitative interview responses for the reported release frequencies are quantified and treated as numerical data. The quantitative research process is most evident in a single survey study reported in Publication III that utilizes aggregation of standardized survey response data. Case studies are used to study phenomena in a real-world setting where the circumstances cannot be controlled in a similar manner as it is possible for controlled experiments (Runeson and Höst, 2009; Yin, 2014). Case studies are mostly suitable for exploratory studies (Runeson and Höst, 2009) and are suited to answering questions in the form of *how* and *why* (Yin, 2014). These factors make case study a logical choice for the encompassing research methodology of the thesis.

3.2.1 Case Selection

Selecting which cases should be part of a case study – the selection strategy – is an important step in planning and conducting case studies (Runeson and Höst, 2009). Cases should be selected in such manner that the research questions for the case study can be properly addressed (Yin, 2014). The real-world contexts for a software engineering study are companies and other organizations that are engaged in software development or other software engineering activities. To this end, the selected cases are from different domains of the Finnish software industry.

A total of 31 Finnish software companies were selected from industry domains ranging from web service development to companies involved in embedded software development and industry automation. The smallest selected companies had only 10 people or so working for them while the largest had thousands of workers. Chiefly, the companies that participated in the studies were active in a nationwide research program called Need for Speed (DIMECC, 2022). The program, which went on from 2014 to 2017 in Finland, had roughly 40 industry and academic partners. Letters of invitation from the academic collaborators were sent to the industry partners in the program, asking for their interest in participating in the studies. The companies had the liberty to accept or decline the invitation so the selection of the companies can be considered self-selection. To broaden the range of companies and to strengthen views for specific types of industries, more companies for the studies were enlisted outside of the research program. For the companies outside the research program, the selection was based on both the domain of the companies and convenience as the companies were known to the researchers. Overall, from the selected 31 companies there were 33 cases as there were multiple cases from several companies.

Defining and bounding a case to a specific unit of analysis is just as important as defining the research questions for a case study (Runeson and Höst, 2009; Yin, 2014; Wohlin and Aurum, 2015). The unit of analysis, i.e. the case, can be for instance individuals, small groups, organizations, processes or projects. Bounding of the case should also include the temporal dimension to determine at which period in time the real world phenomena should be studied (Yin, 2014). Multiple units of analysis are possible in case studies where the cases have many embedded units of analysis, focusing on more than one unit in the same study (Runeson and Höst, 2009; Yin, 2014). Besides having multiple units of analysis, case studies can also have multiple cases as opposed to having just a single case to study (Yin, 2014). Case studies with multiple cases are classified as multiple-case designs and those with just a single case as single-case designs. Single-case designs are good for longitudinal studies or in cases where the single case is critical enough to test a theory or unique in some other manner. Multiple-case designs are considered more robust and useful for replicating the studies in different contexts and environments, which in turn allows cross-case analysis (Yin, 2014). Multiple-case designs can contain embedded units of analysis just like single-case designs.

A multiple-case design with embedded units of analysis is used in a majority of the thesis studies. Software development companies in the studies form the real-world study context for the cases. One key unit of analysis is the *software development process* used to deliver and deploy software in each case. Many of the companies involved in the studies operate in a setting with many projects running at the same time. Software development teams and practices can change by project in such settings. Thus, the concrete case is a *software development project* that is developed by a *software development team*. Individual *team members* are also the embedded units of analysis since qualitative data is collected about their personal opinions and attitudes regarding software development and their understanding of the software process. The temporal bounding of the cases in the multiple-case design studies was the present moment, reflecting the state of the active projects and other matters when the data was collected. In the longitudinal single-case design case study reported in Publication III, the primary unit of analysis and case is the *organization* and the embedded units of analysis are the surveyed projects. In summary, organizations, software processes, projects, development teams and team members are all used as units of analysis in the studies.

3.2.2 Data Collection

Methods for data collection can be classified as direct or indirect methods (Runeson and Höst, 2009). Direct data collection methods are first degree methods where the researchers gather data by interacting with the study subjects such as by interviewing the subjects. Indirect data collection methods are second and third degree methods. In second degree methods, data is collected through some kind of automated mechanisms. The most indirect form of data collection methods are the third degree collection methods where the data already exists prior to its study and data is collected from previously stored documents or from other systems like organizational repositories. Figure 3.2 (decision point 7) shows the data collection methods used for the thesis studies with interviews being the main first degree collection method along with a single survey as a second degree method.

Interviews are used to elicit information and detailed views from study subjects with the help of a set of questions presented either locally or remotely (Wohlin and Aurum, 2015). Questions in interviews are generally *open questions* that allow more room for the answers or *closed questions* where the subjects are encouraged to answer questions in a particular form or structure, e.g. choosing an answer from a set of possible answers (Runeson and Höst, 2009). *Unstructured interviews* tend to follow the natural flow of discourse, having mostly open questions in the interview protocol (Runeson and Höst, 2009; Wohlin and Aurum, 2015). As opposed to unstructured interviews, *structured interviews* favor closed questions whereas *semi-structured interviews* try to strike a balance between open and closed questions (Runeson and Höst, 2009; Wohlin and Aurum, 2015).

An interview can be divided into phases and sections: an introductory phase traditionally begins the interview followed by background questions and the main questions for the interview (Runeson and Höst, 2009). Each question asked in an interview can also be classified according to its level of enquiry in a five-level classification (Yin, 2014). Level 1 questions are questions regarding the interviewee as an individual and the attitudes of the individual. Level 2 questions are questions about the case being studied and are thus the most common in case studies. Levels beyond Level 2 approach broader questions not only about the single case but more general questions about the cross-case findings, for instance.

Interviews for the 33 cases in the thesis were carried out during the years 2014 and 2015. Themes for the interviews varied by study so three distinct interview question sets were used for the interviews. The studies reported in Publication I and Publication II used the same interview protocol. In these two studies, case companies were approached with the idea that it would be helpful if the people who were interviewed were from software development teams and projects that were in a relatively advanced state. This way, the concrete cases could represent not only the typical case in a company but also reflect the best software development practices and processes available in the context of the company.

A single interview session was used in all of the 19 cases that were part of the first set of interviews. The people who were interviewed worked mostly as developers, architects or team leads in small software development teams developing a particular product. Around two hours were reserved for each interview. Usually, two researchers and one to two members of the software development team were present in the interview. The roles between researchers were generally divided so that one researcher asked the questions following the interview guideline while the other researcher took notes and asked clarifying questions when necessary. A digital voice recorder was used to record the interviews with the consent of the interviewees. The interviews followed a semi-structured pattern with both open and closed questions. Background questions about the company and the interviewee were asked in the first sections. Most questions in the interview were Level

2 questions about the concrete case, the project or product, with which the interviewees were working. These questions were aimed at gaining insight into the practical ways of software development, testing, and deployment of software releases in the setting of project or product developer teams. A small number of personal Level 1 questions about hypothetical situations and individual perceptions of software development were included in the interview protocol. For instance, the interviewees were asked what benefits or challenges they could see with a release model where releases and deployment to production environments were more frequent. Besides open and closed questions that could be answered verbally, the interview protocol included interactive parts. The interviewees were asked to depict their software development process from start to finish with a freeform process diagram on a whiteboard or on other similar surface. Notes taken in the interview about specifics of the development process were sent back to the interviewees for verification after the interview. In all but two cases the process descriptions were verified.

The second set of 10 case interviews reported in Publication IV had a similar design as the first interviews. With one or two researchers present, the semi-structured interviews on refactoring were recorded and transcribed. The people who were interviewed were senior software developers and architects from a wide range of Finnish companies engaged in software development. Background questions were used to contextualize the cases and characterize the individuals being interviewed. There were a number of questions that could be classified as Level 1 questions since the interest was also to gather information about how senior developers and architects understand and define refactoring. Although the primary unit of analysis and case was the project and the development process used in the project, individual development team members were the embedded units of analysis in the company's organizational context. Questions in the second interview set were mostly open questions but also several closed questions were asked in the course of the interview. The closed questions included, for instance, estimating the maturity of a project using a five step software development maturity model as reference (Olsson et al., 2012).

Interviews in the third interview set of the case study reported in Publication V touched on the topic of DevOps. Three cases from three companies were selected for the study. The data was collected in three separate interview sessions where questions regarding the development and testing practices of the companies were combined with reflective questions about the presumed benefits and challenges of DevOps. The interviews lasted around two hours and followed a semi-structured format. Two of the interviews were conducted on-site and one of the interviews was conducted remotely. All interviews were recorded and transcribed later.

Data collection for the longitudinal survey study reported in Publication III was done using an online questionnaire that can be classified as an indirect second degree method of collecting data. The survey measured project-level maturity of software development practices in areas such as test automation, quality, build and deployment, running and monitoring, and the lead time to release new versions over a two-year time period in a Finnish company. Selection of projects in the company was based on self-selection as project representatives were free to choose whether to answer the survey or not. Information about the survey was sent via e-mail to all active projects in the company. Initially, the survey had been conducted in 2015 and it was replicated in 2017 with minor changes to the questionnaire. There were responses from 43 projects in 2017, which is comparable to the 35 project responses in 2015. Since the company was involved in developing customer projects, the projects changed from time to time. Out of the 43 projects in 2017, 10 were the same as in 2015. Thus the bounding of the case is twofold. The context is the company itself with the organization being a unit of analysis as well as the individual projects that were active both in 2015 and in 2017.

3.2.3 Data Analysis

Research findings follow from the collection of data and its analysis. Qualitative data such as interview recordings and transcripts has different methods of analysis to that of quantitative data that can often be characterized in numerical terms. Figure 3.2 illustrates the two data analysis methods (decision point 8), *thematic analysis* and *statistical analysis*, used in the thesis studies. The primary source of data in the thesis studies were interviews to which the qualitative analysis method thematic analysis fits well. Statistical analysis was used in a single survey study (Publication III) to compare survey results between the two data points when the data was collected.

Qualitative data analysis relies on establishing a solid chain of evidence from events that have occurred or from subjective standpoints to the eventual conclusions (Runeson and Höst, 2009). The primary objective of interviews is to collect subjective data from people about their experiences and about their understanding of situations in their context. Traditional links in the chain of evidence are the sound recordings taken from the interviews that can be transcribed later on. Passages and quotes are then coded and grouped, forming a solid basis for the conclusions and completing the chain of evidence from the collected data.

Analysis of qualitative data begins with the coding phase (Runeson and Höst, 2009; Wohlin and Aurum, 2015). In the coding phase, researchers read through transcribed text and assign labels to noteworthy sections. Patterns or themes start to emerge when similar labels can be used to characterize sections multiple times within and across cases. Themes can be further organized to higher-level categories. Identifying codes and themes is the process of *open coding* and finding the relationships between themes and categories is *axial coding* (Wohlin and Aurum, 2015). Thematic analysis is an analysis method that can be used to structure analysis of qualitative data by further formalizing the analysis phases.

Thematic analysis wraps up general phases of qualitative data analysis to six different phases (Wohlin and Aurum, 2015). Getting to know the data is the first important phase. Researchers should immerse themselves in reading and understanding the collected data (Cruzes and Dybå, 2011). An initial set of codes can be defined in the second phase once there is enough understanding about the data. Coding in thematic analysis is typically open coding (Wohlin and Aurum, 2015) that requires multiple passes to get the codes right (Cruzes and Dybå, 2011). Coding approaches include inductive coding where codes are drawn purely from the data, deductive coding that favors predetermined codes to label data, and integrative coding that mixes the two approaches (Cruzes and Dybå, 2011). Phases three to five in thematic analysis are related to finding explicit semantic or latent, hidden, themes in the data and codes, and defining and reviewing the themes (Wohlin and Aurum, 2015). Codes can be plentiful so themes help to categorize the codes by narrowing down the amount of analytical units originating from the text (Cruzes and Dybå, 2011). Themes can also have relationships to higher-order themes if the themes relate to the same topic. A thematic map connecting themes to other themes and higher-order themes can be used to develop a taxonomy of sorts (Cruzes and Dybå, 2011). The final sixth step in thematic analysis is creating the report based on the analysis (Wohlin and Aurum, 2015).

Thematic analysis as an analysis method has been extended by the method of *thematic synthesis*. Thematic synthesis is similar to thematic analysis but the method emphasizes the importance of theme development in the analysis of data (Cruzes and Dybå, 2011). Thematic synthesis in software engineering has been noted to be of use with systematic literature reviews in classifying primary studies (Cruzes and Dybå, 2011) but the suggested guidelines for coding text and developing themes should be applicable for other qualitative data such as interviews as well.

All studies in this thesis took advantage of thematic analysis in various forms. Thematic analysis in studies reported in Publication I and Publication II was applied in a similar manner as the studies shared much of the same interview data. The interviews were recorded, partially transcribed on-site by researchers acting as scribes and later checked against recordings to keep the chain of evidence intact. Besides the partial transcripts, interview summaries made by the researchers were taken into account in the data analysis. An integrated approach to coding was used in these two studies. The higher-order themes to which the themes and codes were associated with originated from the layout of the interview questions e.g. for the rationale why projects should move to continuous deployment as seen by respondents. The process analysis reported in Publication II applied thematic analysis by coding process descriptions and technologies listed in the process diagrams drawn by the respondents. Software process phases were the higher-order themes used in coding to determine software development practices in each case and project.

Although the study reported in Publication III used quantitative data from a survey, the study also had a qualitative nature. The survey had an open feedback section used to gather feedback on the usage of the maturity model survey in the company. Open feedback from both the 2015 and 2017 survey was grouped and analyzed by thematic analysis. Since the format of the feedback was flexible, the data analysis was mostly inductive, drawing codes and themes from the data. Themes that appeared often in the text were highlighted in the report. Thematic analysis of the open feedback gave an idea of how the respondents perceived maturity models in the context of their company beyond the quantitative results.

As for the thematic analysis carried out in the study reported in Publication IV, the approach resembled that of the first two studies. Transcripts from interview recordings were coded using an integrated approach to determine how the respondents understood refactoring and how they saw refactoring taking place in their projects. The interview recordings for the study were mostly written out by professionals from third party transcription service providers, which added more detail to the transcripts compared to researcher notes. The themes of the questions in the semi-structured interview provided the framework for the thematic analysis but the coded responses gave form to the content of the analysis. Several researchers took part in the coding so the same transcript or parts of it could be coded by multiple researchers. Clarification requests were sent on occasion to the respondents if the original interview and the interview transcript failed to provide a sufficient answer to the overall questions posed.

Publication V provided results from a study that took advantage of thematic analysis. The pattern of analysis was similar to the other studies in that the recorded interviews were transcribed by professionals and later coded by the researchers. Coding was facilitated by a qualitative research analysis tool named Atlas.ti to which the detailed transcripts were loaded. Searching for codes and themes related to the adoption of DevOps was an inductive effort, relying on the interview data. Codes and themes were grouped to a thematic map that highlighted the main themes of the study. The discovered themes were finally used to organize the results in the study report,

Besides qualitative data analysis and thematic analysis, quantitative data analysis methods were used for the survey study results reported in Publication III. Statistical analysis of quantitative data can be used to compare between-sample means (Blaikie, 2011). The null hypothesis in the case is that there is no difference in the continuous deployment maturity of the case company projects as measured in 2015 and 2017. Because the survey utilized an ordinal scale in the survey questions where the respondents selected the most suitable maturity level for a process area, only non-parametric statistical tests are applicable. Parametric tests like the *t test* need the data to be normally distributed and thus parametric tests cannot be applied to non-metric data (Blaikie, 2011).

Instead, the non-parametric *Mann-Whitney U test* that fits ordinal data (Blaikie, 2011) was used to test for statistical significance from the survey responses. The statistical test was applied for the response means in the process areas of test automation, quality, build and deployment, and running and monitoring, taking into account the response means of both years 2015 and 2017, respectively. A metric for the *U value* is the result of Mann-Whitney U test that was in the end used to determine statistical significance based on lookup values of the metric.

3.3 Summary of Publications

All five publications in the thesis report of studies that have their respective research context and environment. This section introduces the context of each study and summarizes the key findings for the studies.

Publication I: The Highways and Country Roads to Continuous Deployment

The groundwork for the study of the continuous deployment phenomenon was laid by a series of semi-structured case study interviews in 15 Finnish software companies from various domains. Companies in the study included were involved in such fields as web software development, mobile applications, telecommunications and embedded systems in industry automation and medical systems. Themes of the semi-structured interview varied but for Publication I the main interests were the general state of the field in terms of continuous deployment with an emphasis on the perceived benefits and obstacles to continuous deployment in particular cases. The semi-structured interviews with open-ended and closed-ended interview questions were recorded on-site and analyzed using the qualitative, inductive, approach of thematic analysis and synthesis (Cruzes and Dybå, 2011).

Metrics yielded from the interviews included five distinct metrics that could be used to characterize the state of continuous deployment in a company case. These metrics showed how fast a company could release a small change if applicable in the domain, how fast it typically releases, how fast it prepares a release that could theoretically be deployed, and whether changes propagate automatically to the testing environment or the production environment, or both. In the best case, one of the companies could prepare and deploy a change within an hour if needed. None of the companies, however, had an automatic deployment pipeline all the way to the production environment. Some of the companies were releasing at regular intervals every few weeks but in domains with severe restrictions, the release frequency could be up to over a year. The industry domain made a large difference and the customers were not always prepared to accept releases frequently.

Coding and grouping the interview responses resulted in several theme groups for perceived advantages and obstacles for continuous deployments. A definite advantage according to the responses was that continuous deployment could lead to more frequent releases, which in turn would in all likelihood reduce the time to market and make the end users happier by creating value to them sooner than could be done with more infrequent releases. Automating repeated routine tasks might also save some effort in the future. With high-grade test automation and small change sets, the quality of the products could also be improved. Continuous deployment could also foster a culture of collaboration between developers and people involved in the operations line of work, those working with technological infrastructure such as servers.

Considering their current projects in the company and their attitudes towards continuous deployment, the respondents could also see several things that could be seen as obstacles to adopting continuous deployment. Cultures within the companies need to be receptive towards change in general to enable transforming the ways of working needed for continuous deployment. Not only the companies themselves might need the change, though. Customers were not always in the position to receive frequent releases although in some cases the companies had the capability to deploy and release at a faster pace. In some domains, the whole idea of continuous deployment is almost impossible. At the extreme, embedded medical systems rely on heavy standardization, which makes frequent releases difficult, and it is not easy to push changes to a factory plant running an automation system, either. Mobile applications suffer from the fragmentation of devices that according to the respondents cause the need to have manual testing and the marketplaces might also put extra constraints on delivery. Products that are viewed more like art such as games have a different relationship to releasing altogether with possibly long release cycles.

Testing has aspects that can require extra effort with continuous deployment, which was seen in the interview responses. Projects that have been developed without too many automated tests are at a disadvantage when moving to continuous deployment. Legacy systems are examples where the transition would be challenging. Manual or exploratory testing phases slow down the flow in the deployment pipeline. Performance testing and other non-functional testing are also the types of testing that can be hard to automate. At times there can be too *many* automated tests that take a long time to execute and could be an obstacle to continuous deployment. Multiple interdependencies with other projects or otherwise complex structures for building projects can be challenging just the same.

Continuous deployment requires confidence in the whole deployment pipeline and the test automation that is part of the pipeline. Companies are willing to invest in setting up a deployment pipeline to prepare releases at a quicker pace, recognizing at the same time that it might take a lot of effort to do so and might not be possible in all cases. Fear of breaking builds and shipping faulty releases to the end users is in the background. Trust is needed between developers and customers, and between developers and the deployment pipeline to overcome the anxiety related to continuous deployment. Many companies interviewed in the study have the potential to ship releases frequently but have opted not to do so. Taking the next step to continuous deployment could provide them with a shorter time to market eventually to the benefit of the customers and end users.

Publication II: Improving the Delivery Cycle: A Multiple-case Study of the Toolchains in Finnish Software Intensive Enterprises

Automation is one of the key aspects in building a functional deployment pipeline that can be used to enhance release readiness of software (Humble et al., 2006; Humble and Farley, 2010). Software changes and the release versions the changes amount to pass through phases in the deployment pipeline such as building, testing, and deployment, which all should preferably be automated. Automation implies the usage of automated tools that could perform these duties. In the study reported in Publication II, the focus is precisely on the processes and tools that companies use as part of their daily software development work. The study is an extension study of the study reported in Publication I. One company from the earlier study had to be left out because of insufficient data for assessing the level of automation in the company.

Understanding the degree of automation used in the case companies was an objective in the study as was finding out the reasons for not having tools in specific phases of the deployment pipeline. Another objective was to see if there would be any connection between the degree of automation and deployment capability. An important part of the interview protocol was to have the respondents draw their software development process and assign tools in use to the phases they illustrated in the process diagram. Deployment capability and release frequency metrics were extracted from the qualitative interview data and quantified. Analysis of data used both the qualitative coding of interview responses and the quantitative metrics in order to look for associations.

Six major areas or elements of software development were identified in the illustrations and process descriptions gathered from the respondents. A model of a modern software development was seen to consist of such elements as requirements, development, operations, testing, quality, and communication and feedback. Each element had several sub-elements like version control, build, continuous integration and artifact repository for the development parent element.

Analysis of the company cases revealed that certain elements and phases in the development process are better supported by technological tools than others. Requirements engineering has shifted towards agile software development and backlog management tools such as Jira. Version control systems like Git were found in all of the companies. Continuous integration was in many cases part of the basic inventory and remote continuous integration servers were triggering build phases from version control systems just as a working deployment pipeline should. Only a few companies took advantage of artifact repositories for storing compiled binaries. Virtualization and automated configuration management had found their way to about half of the companies, which allowed them to bring up environment-specific virtual machines with little effort and configure servers centrally.

Automated deployment was found to be rare but build promotion schemes were in place that allowed deployment to other environments after build approval. In the most advanced cases, deployment used new virtual machines for each release so that the update of the new virtual machines was followed by a switch from the old virtual machines that were discarded in the process, speeding up the release process.

Testing is an important part of the software development cycle, which was evident in the fact that most of the companies used automated unit tests in their work. There were a few areas, however, where automated testing was neglected, notably in acceptance testing and user interface testing. Exploratory and manual testing without tools were used often, and in certain fields predominantly so without any automated testing. Tools for performance, load, and availability testing were most prominent in the development of web applications. Code quality was sometimes kept in check with integrated code review tools allowing peer review of code, and static code analysis for detecting code anomalies.

Explicit and implicit feedback both within companies and with end users spurs development. Many companies had some sort of internal communication channel for discussing development issues such as a Skype or Slack channel. Although not that common, some of the communication tools were integrated with the development process in order to give developers feedback from continuous integration servers. Direct communication with the end users was rarely assisted by tools but implicit user data was at times collected to provide telemetry on user behavior.

Deployment capability and release frequency varied greatly between the cases, ranging from minutes to years. Based on the quantified metrics obtained from the interviews, release frequency is often considerably longer than the capability to deploy software. Grouping the cases by the degree of tool assistance in various software development phases, there was some evidence to support that those with a well-stocked deployment pipeline were in a better position to deploy rapidly with some exceptions to the rule.

Companies and developer teams employ tools in various areas of software development to help them in daily development activities. An automated deployment pipeline requires the reduction of manual phases but the manual work was still commonplace in the studied cases. Testing is a good example where companies still rely on manual or exploratory testing, and especially acceptance testing required human assessment instead of relying on automated acceptance tests. A tool for a specific activity might also not be used if the activity, like testing, is outsourced, or seen otherwise as irrelevant for the current product. Continuous integration and unit testing, for instance, were not seen as important in the mobile games sector as in other domains. Companies must constantly evaluate whether technology support in any development activity can aide them in developing their product further, in reducing effort, or perhaps in bringing releases more frequently to users.

Publication III: Revisiting Continuous Deployment Maturity: A Two-year Perspective

Maturity models such as CMMI (Capability Maturity Model Integration) can help organizations in process improvement by offering a set of generic guidelines to process areas and activities an organization might encounter in their line of work (CMMI Product Team, 2010). Due to their generic nature, models such as CMMI require interpretation when working with agile software development and thus might not be fully suited to evaluating software development practices like continuous deployment. Continuous delivery or deployment can be considered as a sign of agile software development but agility is difficult to define based on generic process adherence as suggested in CMMI (Fontana et al., 2014). Another option is to use more context specific maturity models for assessing continuous deployment maturity as presented in Publication III. The study reported in Publication III outlines the adoption of a company specific maturity model for capturing continuous deployment maturity. As a longitudinal study, the data was collected with an online questionnaire twice in the time span of two years. There were responses from 35 projects during the first year and from 43 projects two years later. All projects were from the same case company, a fairly large Finnish software development company.

The maturity model developed in the case company was a tiered maturity model including such process areas as test automation, quality, build and deployment, running and monitoring and typical lead time for deploying changes. Respondents self-assessed the state of their project with the help of instructions containing the development practices a project needed to have to reach a particular maturity level in a process area. The tiers ranged between tier 1 and tier 5, where tier 1 represented the least amount of maturity in a category and tier 5 represented the highest maturity level.

Survey responses from both years indicated minor changes in maturity levels of the company's projects. While not statistically significant, maturity in areas of build and deployment, and running and monitoring had improved a little. Individual projects that were still ongoing two years later were in some cases slightly better off but in other cases the perceived maturity had reduced. The typical lead time to deploy and release changes was around a month, which had not improved much. Continuous deployment during the same day was not a standard practice and project teams hardly even desired more frequent releases than every few weeks.

Self-developed maturity models and surveys can help companies to compare projects and understand which type of software development activities they should undertake to reach objectives such as those required by continuous deployment. As shown in Publication III, tiered maturity models have their challenges. Choosing between tiers is not easy nor is the maturity of a project always in the hands of the developer team. Active projects and those in the maintenance phase have different objectives. Customer preferences also take precedence over the development teams' considerations, affecting the maturity levels that can be attained.

Publication IV: Refactoring-A Shot in the Dark?

Refactoring is a development practice in which internal code structures are altered with the hope of improving software design without changing the expected behavior of software (Fowler, 1999). Continuous deployment encourages rapid code changes that incrementally build up the design and architecture of software. In a sense, architectural design is continuous as well with the idea of continuous architecture being that design decisions are postponed until the very last minute decisions are required (Erder and Pureur, 2016). Continuous architecture can benefit from continuous deployment by allowing the integration of the flux of structural changes. Refactoring is a necessary practice to allow the structural changes to happen when required.

In the study reported in Publication IV, to better understand how refactoring fits into the development cycles in the industry, semi-structured interviews were carried out in nine companies involved in software development. The companies were from different industry domains ranging from industrial automation to web development. Themes for the semi-structured interviews touched on views of refactoring, the manners in which companies integrate refactoring, and the benefits and risks related to refactoring. The interviews were recorded and transcribed, followed by a qualitative coding and analysis phase of the transcriptions.

Refactoring was seen to be an important part of software development. Developers spend roughly at least a fifth of their time refactoring code depending on the project. Refactoring is needed because the needs and understanding of required code structures evolve as development progresses. At the same time, there appears to be no good metrics for quantifying refactoring needs. Developers need to rely on intuition, which makes it harder to assure other parties that refactoring is needed. Large refactorings cannot be slipstreamed into normal development cycles as easily as smaller structural code changes and require more coordination.

Responses of the interview study reported in Publication IV indicate that by refactoring code, developers make an investment in the future. Without refactoring, development could later be more difficult or perhaps not possible at all. Through refactoring, code becomes easier to understand and reuse. In the background, there is always the risk that refactoring might not provide benefits at all, or that refactoring might break the code and cause failures in the field. Regardless of the industry domain, refactoring is an everyday development practice that involves weighing the future benefits and risks of refactoring code structures.

Publication V: DevOps Adoption Benefits and Challenges in Practice: A Case Study

Continuous deployment can be seen as a practice of DevOps, which is a broader concept including cultural aspects as well as aspects of collaboration between developers and operations personnel (Ståhl et al., 2017; Bass, 2018). Publication V presents an industry case study looking to gain insight into what companies see as the benefits in DevOps and the factors they see hindering the adoption of DevOps. The case companies were three software companies where semi-structured interviews were held to collect data for the study. Thematic analysis was used to code interview transcripts and to group themes of the benefits and challenges of adopting DevOps.

DevOps was seen as a pathway to more frequent releases due to the increased code integration velocity made possible by DevOps and ultimately to more features being implemented with low overhead release processes. Thanks to the test automation advocated by DevOps, the quality of releases could also go up. When collaboration between developers and operations people is increased, knowledge can be exchanged between people and their skills are put to better use. A higher release rate was also seen to lead to increased feedback from the customers or users, which allowed features to be more readily available and make an experimental approach to software development possible.

Several factors can slow down or make it difficult to adopt DevOps. While the idea of DevOps is to promote communication between specialties, bridging the communication gap may fail if not given enough attention and people are left to mind their own territories. Developers might care about getting releases out quickly while the operations people are more concerned about server uptime. People need to adapt to new roles.

A cultural shift is required, which according to the interviewees in the study, is more difficult to grasp than the technical challenges. DevOps might not suit all business domains where access to production environments is limited, or to environments in which replication of production environments for development is difficult. Conceptually, the term DevOps has also been overloaded to some extent in the past, leaving practitioners wondering what the true DevOps way to follow is. Results from the study reported in Publication V suggest that there are technical challenges to adopting DevOps but many of the perceived obstacles relate to the cultural context and the social behavior of groups and individuals in a work setting.

Chapter 4

Results

Reducing the period between releases and taking continuous software engineering practices into use that can accelerate development cycles requires understanding both about the technical and cultural conditions that can be specific to industrial domains. It is also worth considering what the justification is to increasing release frequency and does it ultimately pay off. Adopting the necessary changes to practices and processes may have implications to organizational behavior that need to be addressed, too. In this chapter, the focus is on the three research questions and main themes of the thesis, which are reviewed in turn by evaluating results of the original publications ranging from Publication I to Publication V.

Regarding the rationale of increasing release frequency, the results for RQ1 are covered in Section 4.1. As part of analyzing software process factors for RQ2, Section 4.2 casts an overview on the elements of the software engineering process that need to be in place when moving towards more frequent release cycles. Section 4.3 investigates organizational implications: the role of management and leadership in adaptive organizations considering continuous software engineering practices, employee attitudes, and changing customer relationships, which are all part of RQ3.

4.1 Principles for Managing Releases and Release Frequency

Releasing software is a tricky process that involves a fair number of steps starting from integrating the developer's changes and constructing a working version that can be subjected to a series of tests and ultimately deployed to a production environment where the changes are finally released to the end users (Adams and McIntosh, 2016). Being able to perform such a feat every week or even every day is no small task. It can take years to build the competence and the deployment pipeline required to do frequent releases every week (Savor et al., 2016; Chen, 2017). Yet some succeed. At Facebook, for instance, developers can decide themselves whether to release their changes in a weekly or a daily release (Savor et al., 2016). Is it worth the effort to convert a six-month release cycle (Chen, 2017) into a weekly or daily cycle or would six months suffice in many cases? This is one of the questions this section tries to answer, focusing on exploring RQ1.

4.1.1 Metrics for Characterizing Deployment and Release Cycles

A release cycle consists of all the activities that take place between two releases and release cycle time is essentially the period of time between the releases measured in minutes, hours, days, weeks, months or years. A release cycle time of a month means a new software version of a particular system is released every month. In reality, to better understand deployment and release capability, the release cycle needs to be broken into smaller pieces.

Excluding the actual development of a feature, deployment capability of the deployment pipeline can for instance be characterized by how much time is spent on individual parts of the deployment pipeline (Bass, 2016). *Deployment time* can thus be seen to consist of nine factors that each have an effect on how quickly a change can be released (Bass, 2016). These factors include the time it takes to build a version, run integration tests, deploy changes to a staging environment, run acceptance and other tests in staging environments, and the time it takes to reach a decision on whether the version in the staging environment is ready for release or not. When canary releasing is used, additional time is spent on deploying release canaries to selected users and figuring out how the canaries work out. The final factors are then deploying the version to the production environment, effectively releasing it to all users, and keeping an eye out for unexpected failures in the production system.

Time in the deployment pipeline is partly spent on activities preparing the release, which do not have a direct impact on the end users as they are internal to the company or organization. Only when a change is released, can users interact with the version with the changes. Internal deployment capability could in addition be measured by measuring the time it takes to have the next version ready at the staging environment, just waiting for the go-ahead from people responsible for the acceptance test phase. This leads to several metrics useful in characterizing deployment capability and release frequency: cycle time to *potentially* deployable software (de-

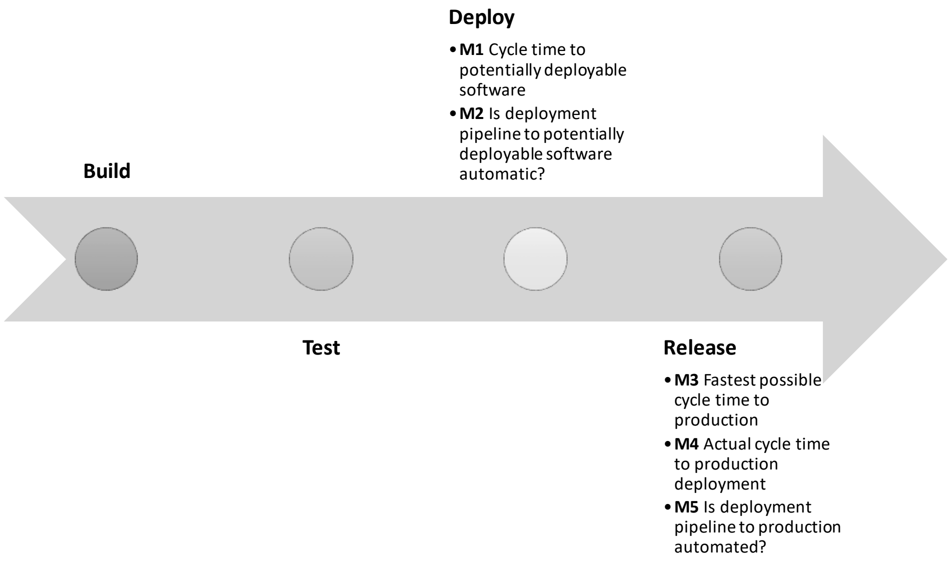

Deploy
• M1 Cycle time to potentially deployable software
• M2 Is deployment pipeline to potentially deployable software automatic?
Build
Test
Release
• M3 Fastest possible cycle time to production
• M4 Actual cycle time to production deployment
• M5 Is deployment pipeline to production automated?

Figure 4.1: Five metrics for characterizing deployment capability and release cycles in stages of the deployment pipeline (Publication I).

noted M1 in Figure 4.1) and *actual* cycle time to production deployment or release (M4) (Publication I; Publication II). Understanding whether the deployment pipeline is automated up to the staging environment (M2) or all the way up to the production environment gives additional yes or no metrics (M5), signifying the degree of automation (Publication I). Inertia of the deployment pipeline can be reflected on by thinking about the time it takes for a small change like a change to a single line of code to propagate through the deployment pipeline to the production environment (M3) (Publication I). The five metrics that can help to understand the internal capability for preparing releases and actual release practices are mapped to the respective deploy and release stages of the deployment pipeline and further illustrated in Figure 4.1.

In practice, there seems to be a great difference in the cycle time to potentially deployable software and the actual cycle time to production deployment (i.e. release cycle time). This means that in many cases the deployment capability or deployment readiness is much higher than the rate at which software is actually released. Company cases illustrated in Figure 4.2 show in months the relation of release cycle times depicted in gray to potentially deployable software cycle times depicted in black (Publication II). Release cycle times are often in the order of several months although the

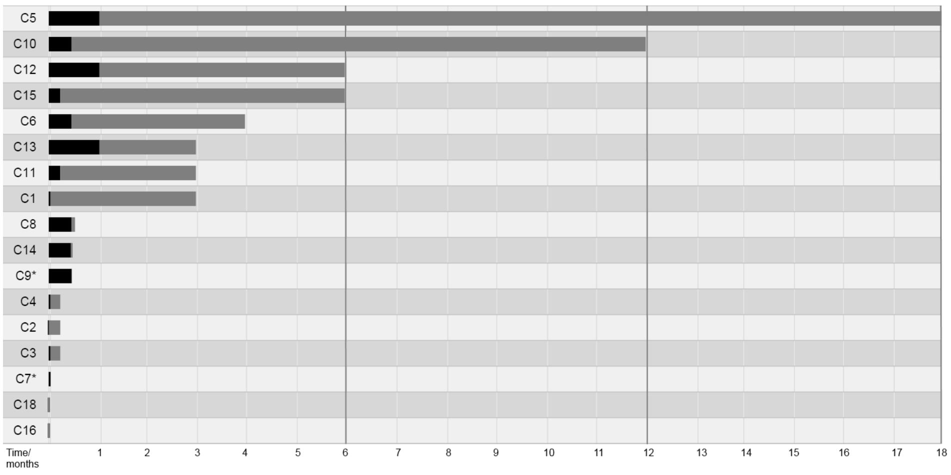


Figure 4.2: The cycle time to release (gray) is in many cases considerably longer than the cycle time to potentially deployable software (black) (Publication II).

cycle time to potentially deployable software suggest that there would be capability to release at least every month or every few weeks. In some cases, the difference is notable as for instance in cases C5 and C10 where the release cycle time is a year or over but the cycle time to potentially deployable sofware is a month or less. It appears that in practice continuous delivery is more likely or at least more common than continuous deployment to production.

There are a number of factors that can at least partly explain why there is such a large discrepancy between release cycle times and potentially deployable software cycle times. Factors such as the domain in which a company operates and the platform for which the software is built seem to make a difference. Companies that had the longest release cycles developed software for embedded systems, telecommunications devices, or mobile devices in the mobile games domain (Publication II). When access to the devices is constrained in some manner such as is the case in the telecommunications industry or in a factory housed with embedded systems, there is a lesser tendency towards continuous deployment (Publication I). Continuous deployment and frequent release cycles are also less likely in the embedded medical systems domain where devices are required to adhere to quality standards for ensuring the safety of devices and users. Longer release cycles in mobile games may be due to games being treated more as a form of art where testing is more subjective and not only focused on functional correctness but on the overall user experience (Publication II). Reasons for ramping up or scaling down release frequency are further explored in the following section.

4.1.2 Choosing the Right Release Frequency

At first, choosing to release changes as quickly as possible seems like an easy choice to make. Why would anyone deliberately choose to release software slower than they theoretically could deliver? A rapid rate in releasing does seem to bring certain benefits to the table but rapid releases might not suit all situations nor is releasing frequently always the objective. This section highlights the potential benefits and obstacles of increasing release frequency, and reviews objectives and motives of selecting release frequency for software versions.

An increase in the number of releases and frequent releases overall may benefit both the development side internally and the external end users. Adopting practices such as continuous deployment and DevOps that aim to improve release frequency are seen as pathways to improve the time to market and the speed at which end users get to enjoy new features (Publication I; Publication V). When versions of software do not have to wait on the shelf for months but can be released promptly, customers get serviced better with a possible perception of added value brought by the set of changes included in the release. Indeed, this added value provided in a release may be one of the key reasons why releases *should* be frequent. If a particular feature has been tested and is ready for release but not shipped until say two months later when there is a release date, there is value loss to those users who might find the feature useful. Whether there actually is loss of value for a particular user and how much likely depends on such matters as the frequency of the usage of the application and the feature in question, and the amount of value users see in the feature for themselves. With more users, the question of value may be more significant, too.

Short release cycles can also be helpful to developers. Feedback from the users is more immediate and seeing how users interact with features helps to develop the features further (Publication I; Publication V). Fast user feedback allows an experimental development approach where reactivity is higher and ideas can be tested more quickly with real users (Publication V). An indirect benefit of an increased release frequency to developers comes from the practices and tools used in the deployment pipeline.

A sufficiently automated deployment pipeline has continuous integration servers running test suites that can be used to send direct feedback to developers on test failures (Publication I). Automated tests ensure that fewer errors get through the deployment pipeline, which improves product quality or at least gives a sense of better quality to developers (Publication I; Publication V). Releases done more frequently reduce the size of each individual release so that releases are not overloaded with features, which may itself improve product quality by limiting the space where errors may occur. Automation also has the upside that repeated tasks requiring manual work in the release process can be automated, saving effort every time there is a release (Publication I). Releases or preparing the deployment pipeline still require some level of coordination between developers and operations people in charge of the technical infrastructure such as network servers. Coordination leads to collaboration between people and as a bonus brings people together, allowing them to combine their talents to solve the issues at hand (Publication I; Publication V).

Beneficial as increasing the release frequency may be with its direct and indirect benefits, continuous deployment or DevOps practices and short release cycles may not perfectly fit all situations. One of the most important factors to consider is the application domain as discussed in the previous section. Technological advances in the embedded systems domain in a connected world certainly provide new avenues for software updates but it is hard to see that embedded system chips in a factory automation system or medical devices for patients would in the near future be updated with the same degree of freedom as non-safety critical web services (Publication I). An overlapping concern with domain considerations is the customer's willingness and ability to accept frequent releases from the development organization (Publication I). When multiple parties are involved in delivering and releasing software, release practices must be compatible.

A host of other matters may make it more difficult to reach tiers of continuous deployment or DevOps where production deployments are done every day. Manual or exploratory testing in its many forms may well inhibit continuous deployment if testing takes a long time (Publication I). It is possible that testing is partly manual because phases in the deployment pipeline such as acceptance testing have not been automated (Publication II). Missing automated tests can slow down flow in the deployment pipeline but too many automated tests can do the same, taking hours or more to finish (Publication I). A project with a modular structure hanging on other projects might make builds for the project slow or more difficult to automate, too. Slow builds have room to improve, though, and measures can be taken to reduce build times from hours to minutes by optimizing the build process and installing new hardware (Publication II). When staging and production environments do not have similar configurations or vary significantly enough, developers might be more hesitant to try continuous deployment with frequent releases if the quality of releases cannot be sufficiently guaranteed (Publication I; Publication V). Encountering any of these other issues does not necessarily mean that releases could not still happen during the same day as the changes are done by the developers. Continuous deployment is still an option to consider and processes can be improved to streamline releases, which is less likely in the case of tighter domain constraints or regulations.

What is the right release frequency then, should all strive for continuous deployment and daily releases or even more often than that? It depends on the objectives one has, which are in part tied to the expectations and possibilities in a specific industry domain. Four distinct release frequency objective categories were identified when interview respondents from the industry were asked to state their current release practices and point in the direction where they wanted to be: *fully automated continuous deployment*, *continuous deployment capability*, *on demand deployment*, and *calendar-based deployment* (Publication I). Table 4.1 summarizes the categories explained further in the section.

Release frequency objective categories fully automated continuous deployment and continuous deployment capability correspond roughly to practices of continuous deployment and continuous delivery, respectively. Continuous deployment without too many manual phases in the process and proper deployment practices into production was an objective for some companies but not for too many. Continuous delivery or continuous deployment *capability* as it was named, emphasizes internal delivery capabilities of a company to have release-ready versions at hand frequently but not deployed to production. This mode of release frequency was just about as popular as the fully automated version of continuous deployment. Of these categories, fully automated continuous deployment has the potential to provide value fastest to the end users but the ability to quickly provide release-ready development versions for internal testing should not be overlooked, either.

On-demand deployment is the most sporadic of the release frequency objectives mentioned in the interviews. In this category, releases are infrequent or happen when a suitable release time is booked together with customers that all parties agree to. Because releases are infrequent, it seems likely that there is less motivation to automate the deployment pipeline. Less automation and more manual work have at least the potential to make releases more error-prone. On-demand deployment is still fit for industries where

Table 4.1: Categories for release frequency objectives. (Publication I)

Category	**Description**
Fully automated continuous deployment	Release changes frequently to production. With a fully automated deployment pipeline, changes propagate automatically through build, test and deploy stages to staging environments. Upon passing the tests, changes are continuously deployed to production, making daily releases possible. Corresponds to the standard notion of continuous deployment.
Continuous deployment capability	Deploy changes to internal testing and staging environments but hold release until current version accepted for release later. An internal version for testing is readily available. The practice of continuous delivery that aims for release readiness matches this category.
On-demand deployment	Release changes upon agreement with customers and other stakeholders. Sporadic releases are especially suitable for specific domains where release cycles are long for one reason or the other.
Calendar-based deployment	Release changes periodically to production using a fixed schedule. Maintain a fixed release period such as every week, every two weeks or every month. Introduces predictability to the release process.

releases are major milestones, amounting to years of craftmanship preparing products of artistic value such as found in games and entertainment.

A fixed schedule for releasing is certainly a good method for adding predictability to the development and release process. Calendar-based releases have fixed intervals at a rate specific to a project. A release can take place, for instance, every week or every two weeks. With calendar-based releases, developers know when the next release train is departing and several companies seem to appreciate the fixed schedule of calendar-based deployment (Publication I; Publication II). Interestingly enough, in about third of the cases, the companies had the internal capability to release changes in a week or two (Publication II). In another setting, where the lead time of several dozen consulting projects was surveyed, the median value of responses for lead time was two to four weeks (Publication III). It would not be an overstatement to say that a fair share of the companies involved in the studies either release or are prepared to release new features following the contours and spirit of weekly to monthly calendar-based deployment. Perhaps the resemblance is coincidental but could they all be following a similar software process framework?

Scrum is a software development process that advocates development sprints in which features are implemented and reviewed in one to four week cycles (Schwaber, 1997). If Scrum is indeed the prevalent software development process for many of the companies in their projects, Scrum could also drive the deployment capability and release frequency to match the sprint intervals. In order to increase release frequency beyond weekly releases, the underlying software development process and methodology might need to be changed as well, if Scrum is in itself somehow incompatible with daily releases. Many reasons could lead to weekly or monthly releases so the causality between Scrum and the observed deployment capability and release frequency is purely hypothetical but nevertheless still interesting to consider.

Release frequency determines how often new software versions are placed in a production environment where users can interact with new features or otherwise experience the updated version unless changes are mostly structural. The best release frequency is the frequency that suits all parties – developers, customers, users, and other stakeholders – in a given situation that might be unique to a project and the domain in question. Interest in a fully automated deployment pipeline with daily releases seems to be relatively low (Publication I; Publication III) although users might benefit from frequent releases. Being able to produce a development version quickly to testing and having the capability to deliver appears to have some importance even if releases are not that frequent (Publication I). Calendar-based releases with a fixed schedule are quite well balanced somewhere between sporadic on-demand releases and daily releases (Publication I). Attitudes towards daily releases can of course change over time but the predominant culture has to change, too. Software processes need to be solid enough to support the other changes required for frequent releases so the next section focuses on the processes in greater detail.

4.2 Meeting Process Demands of High Frequency Releases

Software engineering practices that emphasize making the fruit of labor of developers frequently available in the production environment such as continuous deployment requires mature software processes. Repeated tasks need to be automated as much as possible in order for continuous deployment to succeed. A blueprint for a deployment pipeline (Humble et al., 2006;

Humble and Farley, 2010) gives a fair idea of which parts are needed but assembling all the parts and welding them with software processes already in use is an effort in itself. Continuity is needed throughout the development process, yet in some parts the needs are more pressing than in others if continuous deployment is the objective. Improving on the aspects related to releasing software frequently builds upon the understanding what the necessary process elements are and the ability to evaluate process maturity in a given situation. In this section, the aim is to explore the second research question regarding the software process requirements of high frequency release by looking at the activities found in modern software development and the methods to evaluate the maturity of processes.

4.2.1 Activities in a Software Engineering Process

Software engineering is a field where multiple skilled professionals work together from the inception of a development idea to the release of a software product to end users. Every release is a milestone in developing software but in most cases maintaining software, monitoring software systems and gathering implicit and explicit feedback from people and systems calls for continuity in the development process beyond releases. The key to preparing releases frequently lies in automation and tools but making swift decisions in the development process demands flexibility from the whole process. Activities in such software process areas as *requirements*, *development*, *operations*, *quality*, *testing*, and *communications and feebdack*, summarized in Figure 4.3 (Publication II), characterize the manner in which software is developed in the modern era.

Requirements are the fuel that keeps the deployment pipeline flowing. Without requirements, there would be nothing to test, deploy or release. Requirements elicitation entails finding out from persons inside and outside of development organizations, from existing systems, or from other sources, what the software system in question would need to be like to serve its potential users well. Requirements or reported defects in modern software development turn readily into backlog management items that are requirements that are to be implemented in upcoming sprints. Backlog management items and sprints are part of terminology and practice used commonly in the Scrum software development process (Schwaber, 1997). A continuous flow of requirements is a vital part of a functioning software development process since without requirements and a list of backlog items, developers do not have much to work on (Publication II).

Implementing the required changes captured in backlog items, user stories, or other requirements records calls for development practices and activ-

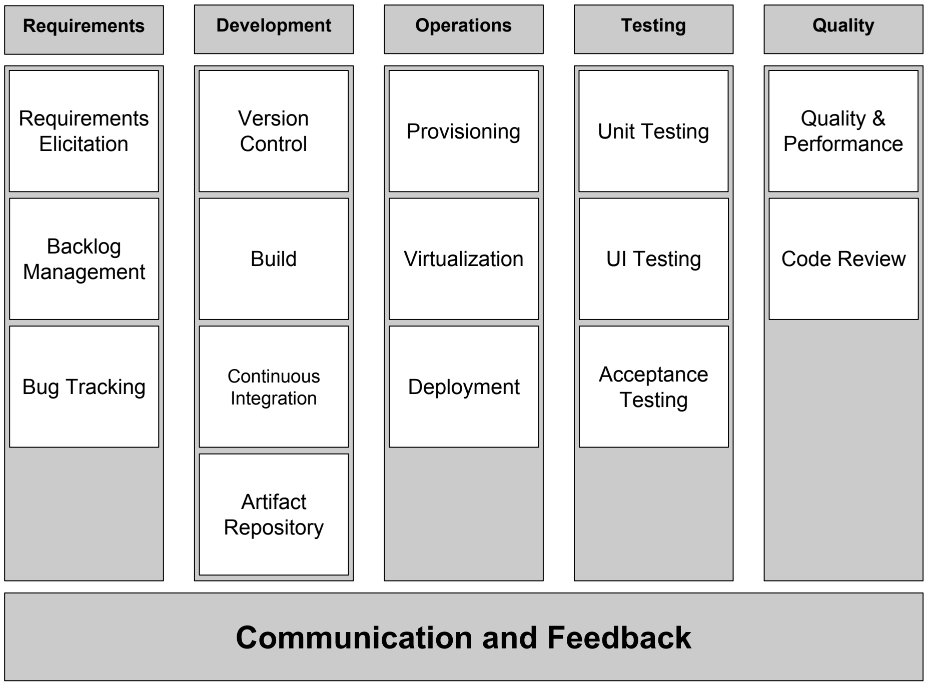


Figure 4.3: Software development in the modern era has a range of activities that support development, help setting up infrastructure, and verify and validate software products (Publication II).

ities. Developers store source code and other artifacts related to a change in version control systems. Distributed version control systems and different version control branch management strategies enable developers to integrate their work in shorter bursts at least locally, working in isolation when needed. Build systems help developers to assemble software packages by compiling source code, execute automated tests, deploy packages to development environments, and manage dependencies. Remote build systems operate on external servers where additional build steps can be executed. Continuous integration servers monitor changes from the version control system but double as remote build systems triggered by version control changes, combining multiple build steps and managing complex workflows such as orchestrating virtual machines for testing purposes (Publication II). Build systems can be further augmented by artifact repositories where compiled and tested binaries of in-house developed software can be stored to reduce build times. Both internal and external dependencies from various libraries or other programs are used in software development to compose software; artifact repositories especially help with internal dependencies.

Development and testing of software are usually done in separate environments, which are in turn separated from the production environment where the current software version is running. Setting up the environments and configuring the environments or network infrastructure is part of the operations work integral to the deployment pipeline. Provisioning and virtualization are used to build environments out of environment specifications on the fly. Virtual machines for network infrastructure such as servers have their blueprints that allow environments to be recreated and reused, time after time, for each build if needed. Configuring servers automatically with configuration scripts ensures that there are minimal differences between the environments. Replicating testing, staging, and production environments is important to avoid failures originating from characteristics of the environment itself (Publication II). Orchestrating the creation of virtual machines, environments and provisioning in general needs supporting technologies, which in some case may be provided by cloud service providers. Deployment is a necessary step in the deployment pipeline to transfer software packages between environments and deployment automation with scripts or dedicated tools help in the process.

Testing is a fundamental activity in a software engineering process to verify that the software is working as it should and that it generally meets the needs and expectations for which it has been made. Object-oriented programming guides programmers to write program code out of units such as classes that can have hierarchical relationships between them. Unit testing refers to testing the individual units, classes, in isolation to assess whether methods or functions within the smaller units work correctly. Unit testing is an activity that has proven to be a good candidate for test automation (Publication II). User interface testing focuses on testing whether users are able to interact properly with the displayed elements in the program and whether correct results are shown on the display. Visual elements, aesthetics and evaluating experience of users is by default more difficult to automate than unit testing. Acceptance testing is the step required in order to put a stamp marking acceptance or failure on the tested software version before continuing with deployment and release to production environments.

Software quality is not only evaluated by testing that individual features in the program work correctly. Aside from functional testing, non-functional properties such as performance, robustness of design or of code structures can be tested, too. Performance testing is an important aspect in the web domain where software systems may be subject to varying degrees of user load, making it useful to see how well a software system can operate under heavy load (Publication II). Load testing every now and then at given load thresholds sets a good baseline for web services but some forms of testing and quality assurance are more continuous in nature. Constant monitoring of software systems and servers, for instance, provides telemetry and health data that help to check the availability of services and catch failures early. Code designs and structures can be checked either with static code analysis tools or developer code reviews, which both serve as indicators of internal software quality (Publication II).

Communication and feedback processes within development teams and outside glue the other activities together. Developers send messages to each other and to third parties with instant messaging services that help development and assist in creating social identities for developer teams. Messaging services can even be connected with development processes so that continuous integration servers feed developers with real time information on the status of build and test jobs (Publication II). Not all messaging is instant like is the case with e-mail and meeting someone in person for a good old fashioned talk is still an effective way to communicate development needs.

Feedback from the users is essential when trying to build appealing software products that would meet the expectations of the users. In general terms, the initiative to provide feedback can come from the development organizations or from the users themselves. When developers ask users for feedback, it is a form of *pull* communication, and when users are the active party and supply the feedback to developers, it is referred to as *push* communication (Maalej et al., 2009). With either push or pull communication modes, feedback can be obtained explicitly by interacting with the users or implicitly by observing and analyzing user behavior. Support e-mails or contact forms on a website are classic examples for explicit push communication but modern push mechanisms include the possibility to use real-time interaction with users like chats that can be overwhelming for developers (Publication II). Gathering feedback from users with interviews and surveys are useful pull communication processes to elicit requirements. Web services, on the other hand, are especially suited for implicit pull mechanisms since user actions on web sites can be tracked and analyzed to increase understanding of user behavior (Publication II), directing the development effort. Rich internal communication and taking advantage of continuous user feedback in one form or another ensure that development stays on the right track.

4.2.2 Determining Maturity of Software Engineering Processes

Software engineering projects, or rather the people involved in the projects, apply different practices and activities that form the core of the software engineering process for a particular project. Even within a single software development organization, the process may very well vary from project to project. Projects are possibly in different phases in the software lifecycle, some old, some young, which lends to the varying characteristics of software processes. Beyond the projects, there is the organization itself that has capabilities embodied in its documented processes and imprinted in the minds of its personnel who follow a particular way of working. Capabilities can thus span many projects in an organization.

Software engineering process maturity in a given organizational context is defined by evaluating the capability and maturity the organization and its processes have compared to a theoretical baseline. Maturity models such as CMMI (CMMI Product Team, 2010) offer guidelines on evaluating capability and maturity in an organization but the guidelines may be too generic and rigid for agile software projects (Fontana et al., 2014). Applying both CMMI and agile methods like Scrum together brings more flexibility to planning and increases the predictability of releases at the same time (Sutherland et al., 2008). Even so, more precise methods to determine maturity in terms of continuous software engineering and its practices may be helpful for evaluating and improving processes and activities. One method is to take several representative projects in an organization and depict the whole development process from requirements to release by interviewing project personnel and collecting data corresponding to the various phases of the deployment pipeline (Publication II). A detailed breakdown of the development process yields plenty of information about the practices and activities in a project but getting the information involves a lot of work. In an environment where there is a plethora of projects and the objective is to get a general understanding of the maturity of the projects, a brief survey sent to key project members can suffice (Publication III).

Depicting the development process from different projects in a detailed manner helps to evaluate maturity and identify points of interest in the development process. By covering each activity that form the deployment pipeline, it is possible to see whether certain activities are properly automated and supported by a suitable technology stack for the purpose, and to compare project practices to other projects (Publication II). Detecting gaps in the deployment pipeline and in the level of automation shows where there is room for improvement. Figure 4.4 illustrates the degree of tech-

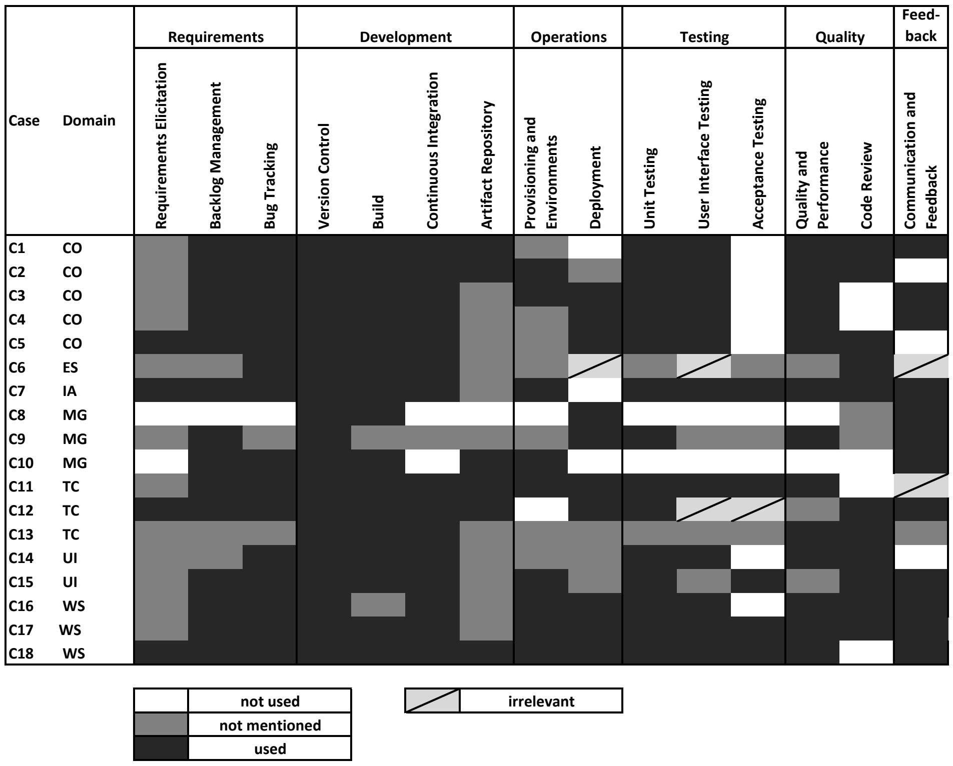


Figure 4.4: The usage of tools in case companies for each development phase categorized by domain suggest which phases involve manual work (Publication II).

Domains: **CO** = Consulting, **ES** = Embedded Systems, **IA** = Industrial Automation, **MG** = Mobile Games, **TC** = Telecom, **UI** = UI Framework, **WS** = Web Service

nology support, automation, and tooling across phases and activities in the deployment pipeline in several different companies and industry domains (Publication II). Evidently, there are areas in which the companies have laid a strong foundation and have mature, technology supported, processes in such areas as development consisting of version control, build, and continuous integration. Development activities for most companies require less attention than does, for instance, testing and acceptance testing that seem to require more manual work in these cases. Of course, the domain needs to be taken into account when considering the maturity of processes. Devices that do not have a user interface for the end users to interact with, as is the case in certain embedded systems, are difficult to subject to user interface testing and thus can be irrelevant in terms of project maturity. Maturity can also be hard to capture if the whole branch of industry seems to live by a different code where an activity appears to be somewhat neglected, mobile games and testing in general being the prime example. The appearance can be deceiving since companies can rely on outsourced testing or test their products manually with explorative testing techniques. Still, learning which activities have less technological support or are otherwise overlooked in a project or more widely in the organization can be an important driver in improving the processes needed to deploy and release versions frequently with good quality.

Surveying project practices with questionnaires aimed for project representatives is an efficient method for acquiring information about the maturity of projects in an organization (Publication III). In a setting where existing maturity models are not specific enough or are not applicable for some other reason, it is possible to construct and tailor a maturity model for specific needs. A theoretical maturity model can include a number of process areas for which there are defined capability levels based on the maturity of processes in the area, corresponding to the general conception of continuous representation in the CMMI model (CMMI Product Team, 2010). A maturity model tailored for measuring maturity in terms of continuous deployment capability, for instance, can include process areas from the deployment pipeline such as test automation, quality, build and deployment, running and monitoring, lead time, and security (Publication III). Each tier in a process area can include a set of conditions that have to be met in order to reach the level. Having a continuous integration server that polls the version control system for changes could for example be a requirement for reaching a particular level in the build and deployment process area. The higher the tier, the more advanced are the practices that need to be in place. Tiers can have a particular range that seems fit for the model like ranging from zero to five. Continuous delivery and deployment to production environment could be a practice on the highest tier of build and deployment. After constructing a maturity model, it can then be used as part of company-wide surveys to help understand the maturity of a company and the capabilities of its processes in projects.

Surveys that integrate maturity model self-assessment in project-level questionnaires can give an overview of the state of the company's processes and a sense of direction of software process improvement initiatives when replicated over a longer time period (Publication III). Performing maturity model surveys has a number of benefits but there are some drawbacks, too, and care must be taken when interpreting survey results. A positive signal is that just by participating in a survey, people in the project have to reflect on the practices used in a project. Comparing results and practices with other projects encourages healthy competition between development teams.

The downside is that different characteristics of projects make the practices used in projects difficult to compare (Publication III). Projects are different: the team structures, domain, customer involvement, lifecycle phase and project age can all vary. Small development teams working on maintenance tasks in a soon to be terminated project do not have the same incentives to keep improving aspects of the deployment pipeline as might reasonably sized development teams working on recently established and active projects. Customers who are directing the development effort as product owners in agile software development projects in collaboration with developers might also be less willing to invest in technical process improvements if the cost is having fewer features implemented. Companies can serve customers from different industry sectors and have a wide range of projects, each of which is tied to technological and other constraints inherent in the industry domains. Comparing maturity survey results between company projects in these circumstances can be difficult. A tiered maturity model with predefined requirements for capability levels in process areas does not fit all projects, either. Choosing the correct capability level is somewhat subjective as a project might not pass all the requirements to reach a particular level but still the project might apply development practices linked to higher tiers in the hierarchy. Despite the shortcomings, maturity model surveys provide a sense of the state of processes in a given environment, making it easier to place resources in software process improvement activities where needed and to raise the awareness of people about project practices that might need attention in order to excel.

4.2.3 A Process for Continuous Software Engineering

The deployment pipeline is long with its many stages of building, testing, deploying, and releasing (Humble et al., 2006; Humble and Farley, 2010; Adams and McIntosh, 2016) but there is more to continuous software engineering than just the deployment pipeline. Software development is part of the process but continuity is needed in product planning and maintenance of the product as well (Fitzgerald and Stol, 2017). The deployment pipeline needs a constant flow of requirements to keep going and effective communication to convey ideas or other feedback internally within development teams and externally with users and customers (Publication II).

Defining and working with requirements are fundamental activities in the continuous software engineering process. Keeping the release frequency high means there should be an equally high frequency of collecting requirements or at least ensuring that development teams have enough work to keep them busy during the short development iterations. Without tasks selected for development, developers have no choice but to find something else to do while waiting for the requirements to come in (Publication II). Interacting with end users and releasing new versions often creates opportunities to receive direct feedback, which development teams find beneficial when thinking about what to develop next (Publication I; Publication V). Engaging the end users and opening up communications channels for feedback provides good input for requirements but the added layer of interaction can wear down developers (Publication II). Implicit user feedback obtained by analyzing software usage data and other telemetry from environments in use may be less straining for developers. Developers should not be overwhelmed by development tasks so there should be a good balance between incoming requirements and tasks selected for development. Without having the right balance, developers might feel that they do not have enough time to develop high quality products, leading to less than perfect designs that need attention later (Publication IV).

Implementing a feature request or other requirement is part of the development cycle where solid development practices are needed to keep cycle times short. Version control is a central component that developers use to integrate their work frequently and it is hard to think about software development without version control (Publication II). Branching strategies for common distributed version control systems affect the check in behavior of developers. Integrating changes often to the main development branch from feature-specific version control branches is considered a good practice by developers and the practice also reduces the chance of merge conflicts when integrating changes (Muşlu et al., 2014). Automated builds are a fine companion to version control systems and continuous integration servers that are all needed to construct deployable software packages from changes checked in to version control. Build phases in the deployment pipeline should not merely be automated but be optimized for efficiency as well as fine tuned build processes can save hours in build times (Publication II).

A steady and continuous stream of requirements means that developers need to embrace change in development, too. Software structures, designs and the whole software architecture are subject to change when new ideas emerge. Ample time in development should be reserved for refactoring not only because developers are at the outset uncertain of how the system will evolve but because code and designs regularly need a certain amount of tidying up (Publication IV). Without keeping the code clean, it might be impossible to do future development on top of the old structures. Developers working with impractical structures are faced with hard decisions whether to try changing the existing code or scrap the code and start from scratch. Refactoring needs should be acknowledged by the whole development team and made visible to customers or other product owners as clearly as possible (Publication IV). Customers or product owners might not be able to grasp the evolution of software code and its structures as well as developers but refactoring work should not remain hidden, either. After all, refactoring might take considerable time and not always be successful so the decision to refactor is not for the developers to make alone. Any development team applying a continuous software engineering process with a high velocity of incoming feature requests and changes should consider refactoring and its place in the development process to keep the code clean, adjust software structures to current needs, and prepare for the future by making structures easy to change.

A high velocity of features and changes puts great demands on testing, too. Testing and automated testing in particular are central to the delivery and deployment pipeline (Humble et al., 2006; Humble and Farley, 2010), and for good reason. Without automated testing and tests that cover a fair portion of program code, developers find it difficult to implement continuous deployment (Publication I). For one thing, automated tests ensure that there are no unexpected failures after changes and that there is no regression, i.e. failures in components that have previously worked fine. Regression testing is also an important safeguard for refactoring as structural changes can potentially have undesirable effects that developers would like to avoid (Publication IV). Automated tests executed after each change reduce the likelihood of end users coming across features that do not work properly. A higher degree of test automation required by continuous software engineering practices can thus be seen as beneficial in increasing the perceived quality of software (Publication I; Publication V).

Automated tests in a functional deployment pipeline should include tests from different phases. Unit testing for individual code components is a good place to start (Publication II) but unit tests alone are not enough. User interface tests or browser-based end-to-end tests might be harder to implement, or at least they are not as common as unit tests (Publication II; Publication III). Where applicable, automated user interface tests can still save someone from having to do exploratory testing or to otherwise check the behavior of a system through the user interface. Manual testing, inspections and getting early feedback from selected end users or other test groups prior to a release might have their place in certain cases where user experience is critical (Publication II). Nevertheless, any testing activity that requires significant human intervention such as exploratory testing slows down the deployment pipeline (Publication I). Making the call whether a release or a particular feature is ready to be shipped and deployed to production is another example of work that is commonly left to people rather than to a set of automated acceptance tests (Publication II). Automated acceptance testing is an area that should not be overlooked when considering the deployment pipeline.

Non-functional testing of aspects like performance and security usually falls to the same category in which high levels of test automation are difficult to achieve (Publication I; Publication III). Aspects such as performance and security are cross-cutting concerns for the whole software system being tested and it is reasonable to test that expectations are met for non-functional quality. If testing of non-functional quality attributes is not possible for every incoming change, perhaps manual quality checks can be repeated every once in a while if there are significant changes to system or software architecture.

Regardless of the exact composition of the automated test suite, test execution should be fast enough to allow frequent test runs. Thousands of automated tests take a while to run, which reduces the likelihood of being able to release new versions very frequently (Publication I). Developers might have to wait hours for tests to finish (Publication II). Rapid feedback to developers from automated tests is useful but getting the feedback fast requires a well-adjusted deployment pipeline.

Using matching technological environments for development, testing, and production is beneficial for getting reliable test results. Depending on the domain, environments consist of such components as operating systems, installed software, databases, configuration, and physical or virtual hardware like memory or processors. Tests are less trustworthy in environments that are not alike so constructing testing environments with similar characteristics should be a priority (Publication I; Publication II). Even having dissimilar databases, database structures or insufficient test data is likely to make it harder to adopt development practices suitable for frequent releases (Publication V).

Orchestrating or setting up similar environments can be facilitated with virtualization services (Publication II). Network infrastructure blueprints that are stored in repositories can be reused, which reduces the chance of unexpected failures due to differences in environment configuration. Infrastructures provisioned from public cloud vendors provide an alternative to proprietary hardware (Publication II; Publication III). Together, virtualization and cloud infrastructures assist in rapidly releasing new versions as servers can be provisioned and updated at will at least in the web domain where such services are available (Publication II).

Getting feedback throughout the development process is critical. Information should flow effortlessly between people and all the activities in various stages of the deployment pipeline. Feedback from external sources, like the end users, can be used in developing features for a system (Publication I) but internal communication is just as important (Publication II). Practices like DevOps where the stages are integrated, are harder to implement when people do not engage and talk to each other (Publication V). Instant messaging services and chatting have the potential to bring people closer by offering a direct line of communication between team members (Publication II). Efficient communication inside development teams is complemented by keeping communications channels open from systems as well as people. Developers appreciate timely feedback from systems such as continuous integration servers to keep track of the state of the code and automated tests (Publication I).

Development teams can thus leverage feedback from within and without to increase their reactivity to changes in development, spotting errors early on in the deployment pipeline, and responding promptly to external feature requests. Higher reactivity with enhanced communication and feedback can be seen as an enabler for increasing release velocity, supporting and ultimately completing the process for continuous software engineering.

4.3 Organizing Work in Continuous Software Engineering

A software development organization employs software engineering experts who work with support functions in various kinds of branches, groups and teams that might have managerial boundaries between them. Software development organizations develop software either for their own use internally or for the use of other external customers. A blueprint of processes needing change for continuous software engineering gives an idea of what should be changed but above all changing processes means changing organizational behavior, behavior to which people have been used in the past. In fact, the technical challenges in adopting continuous software engineering practices like DevOps are thought to be easier to face than than the changes required for the organizational culture (Publication V).

People might have reservations about the new roles they might need to take. If the experts internally might have reservations related to implementing new processes, so might internal and external customers for whom the software is being developed. Management should also consider its own role in the change process.

This section covers the results for RQ3 that explores the implications of frequent software releases to organizing work. The first part looks at the managerial and leadership considerations that organizations may need to address before implementing continuous software engineering processes and increasing their release frequency. In the second part of the section, the attitudes and concerns of people and their need to adapt to new circumstances surrounding continuous software engineering are discussed further. The third part shows how the customer relationships might also need to change since customers have a definite say in which way software is developed and released to the end users.

4.3.1 Leadership and Showing the Way

An organization needs a supportive and encouraging atmosphere from all tiers of the organizations to carry out changes in its operating procedures like increasing the release frequency of software. Management needs to be behind improvement initiatives, lest development teams are left on their own and discouraged to substantially improve existing practices (Publication I). The organizations need to be receptive towards change (Publication V).

Besides being receptive to change, the organization and specifically its superiors should be active agents of change. Without stated objectives and a clear enough plan, development teams zero in on maintaining existing project practices and not necessarily on improving all aspects over time (Publication III). It is arguably more difficult to increase release frequency or halve lead times if there is not such an objective in a particular project. Management should give guidelines and a sense of direction to the development teams to increase the awareness of greater organizational ambitions.

A general understanding that project practices should be improved up to a certain level, can be helpful in pushing for an organization-wide change but improvements might not make sense for all projects. Projects that are barely on life support and are less actively developed are probably not first on the list to transition to rapid releases (Publication III). The same kind of feasibility evaluation is made when thinking of large structural and architectural changes to an existing project (Publication IV). Improving a software process or software architecture should reasonably benefit future development and deployment to be sensible. Insight is needed per project, gauging where the current practices are e.g. through maturity surveys (Publication III), and making it explicit what the desired level of maturity is for a project given its current and future standing in the organization's project portfolio.

Once there is a shared understanding in the organization about the desire to adopt continuous software engineering practices in projects and to perhaps increase the release frequency, the internal organizational culture needs to be shaped accordingly. A change involving multiple changes to the development process does not happen overnight. Development teams need time to learn the skills needed to change the software development process. It might be difficult to arrange sufficient time for process improvement in the hectic everyday working life of development teams (Publication I). Management should acknowledge the difficulties and allocate development teams enough time and resources that they need to transform and enhance old processes.

4.3.2 People, Attitudes and the Adaptive Organization

Time is not merely needed to learn new skills and improve processes, it is needed on a personal and interpersonal level to adapt to the possible change in roles with added responsibilities. When processes get more streamlined and development is closer to operations and infrastructure like in DevOps, people might need to share their previously exclusive territory, which can cause stress and concerns (Publication V). An increase in release frequency might mean sharing responsibility over the operations environment and redefining who is in charge of monitoring the environments for system health after releases.

Development teams also have reservations about the development style associated with frequent releases and the deployment pipeline with its automated building, testing, deploying and releasing of changes. Relying solely on automated tests makes teams uneasy and worried that the tests might miss faults in code and releases, sometimes rightfully so (Publication I). Replacing a trained eye with automated tests is not an easy task. Automated tests also play an important role in refactoring where automated tests verify that changed structures still work as intended after changes (Publication IV). The fear of breaking code without being able to see what is broken after refactoring is a similar sentiment to the fear of releasing broken builds to production environments in the first place.

Shipping broken builds and features to the users is obviously a genuine concern among development teams. Features and changes should undergo testing, automated or manual, so that teams could trust the releases to be free of defects. Building trust in the deployment pipeline is essential (Publication I). It follows from the idea of building trust and alleviating fear that development teams should strive for a decent code coverage with automated tests if possible.

Deploying changes to the production environment immediately or at frequent intervals can be risky. The risk that a failure is encountered after a release will in all likelihood be realized sooner or later. The risk factor increases but it might be worth to take the risk. When the releases are more frequent, change sets are smaller and it is easier to get direct or indirect feedback from the real users (Publication V). A reactive development style allows the operations environments to be monitored and errors can trigger alerts to the development teams who can work on the fixes. Besides, if the release frequency is high, deploying a quick fix to the production environment should not take long either. Companies and development teams can thus work with high frequency release by accepting an elevated level of risk in domains where failures are not life threatening or excessively costly (Publication I).

Despite having concerns, developers have also expressed a range of positive emotions when thinking of high frequency releases. Releasing new versions can be stressful but less so when development teams do not have to work with too many changes at a time. Smaller releases can be seen to lead to a better working atmosphere overall (Publication V). A positive working atmosphere can be a source of happiness to developers, which can also be seen in refactoring. Developers have intrinsic motivation factors and it appears that developers find it rewarding when they have a chance to refactor code structures to a better shape according to their development ideals (Publication IV). Shorter release cycles and the possibility to refactor can thus reduce stress and give a boost to spirits at work.

4.3.3 Reshaping Customer Relationships

A software development organization is well positioned to increase its frequency of software releases or take other continuous software engineering practices into use if the management is committed and the organization has the needed capabilities along with a motivated staff. There are multiple cases, however, where software is not being developed for an internal customer but for an external one. The relationship between the developers and the customers may vary.

Customers may not be willing, for instance, to accept frequent releases for some reasons even if the development organization has the capability to do so (Publication I). The situation is even more delicate if the development organization has been hired to work as part of the customer organization. Development teams or individual members who are working as consultants do not have the same access to operations environments and have limited possibilities to change processes in the customer organization (Publication III). A development organization that is in a consultancy relationship with a customer organization might have better readiness to release changes more frequently than the customer organization.

Customer organizations might not be willing to invest heavily in process improvement as such given the choice (Publication III). This is understandable since development tasks related with improving process aspects and the deployment pipeline do not progress the project in a similar manner as does typical feature development. If the consultancy relationship is one in which the customer simply oversees development and the customer organization does not have a development team of their own, gaining the approval for process improvement from the customer could be harder still.

The challenge in developer-customer relationships does not only concern process improvement but also structural and architectural changes that do not have directly visible results to the customer like in refactoring code. Without good metrics, it is not easy to show to the customer that refactoring is really needed and justified by hard data (Publication IV). Whether putting effort into refactoring or improving software processes with a higher release frequency in mind, development teams and organizations should be transparent enough so that customers understand how project resources are being used.

Negotiating with the customers and convincing them of the potential benefits of adopting continuous software engineering practices should probably be done at the outset before starting development of a project. Only then can the customer get to know the working habits of development teams and gradually gain understanding about high frequency release models in domains where they are suitable. Both the benefits and drawbacks should be discussed so that the customer can decide which release model to choose for a project. Consultants integrated into in-house development teams should similarly bring their expertise to customer projects. Prior to starting development, consultancies and customers should agree whether customer processes need attention in terms of fine tuning the deployment pipeline and taking other continuous software engineering practices into use. Changing attitudes to be more favorable to continuous software engineering practices and redefining organization relationships might take time but the change should start from somewhere, little by little.

Chapter 5

Discussion

Results of the empirical studies covered in this thesis have provided enough information for formulating answers to the original research questions of the thesis. In this chapter, it is time to revisit the research questions and summarize the key findings for each question. As with all research, the findings and the inferences made from the results of the studies are not without limitations. The limitations are discussed more broadly in the validity threats section. Following the section on validity threats, the findings of the thesis are compared to previous work done in the field. Given the boundaries set by the validity threats, the findings can be seen to have a certain degree of merit as noted in the final closing section of the chapter concerning the theoretical and practical implications of this thesis.

5.1 An Overview of the Research Questions

The three research questions posed in this thesis focus on different aspects of increasing release frequency and its implications in software development organizations. To recap, RQ1 ponders the rationale of doing frequent software releases. In RQ2, the setting of the question is the landscape of software processes and deployment pipelines needed to provide software changes frequently to end users. The final research question RQ3 has its focus on the organization of work and managing change towards more frequent releases. This section summarizes the discussion and findings for all three research questions in turn.

RQ1: Why should software releases be frequent?

An increase in release frequency can be a boon to software development, helping developers and users in a number of ways. When the period between releases shortens, software features become available to users quicker and the time to market for features improves. Developers benefit from shorter release cycles as well. Shorter release cycles allow for a more experimental style of development, where feedback of implemented features flows back to developers, given that feedback channels exist and production environments are monitored sufficiently. Increases in release frequency require a certain degree of test automation that can have positive effects on product quality. Preparing releases calls for the interplay of plenty of people with different skills in the development team and beyond. Working together can thus bring people together and improve collaboration of people, at least before most stages are automated.

Theoretically, the idea of added value to the user can be considered one of the driving factors behind increasing release frequency. If users are able to enjoy features almost as soon as feature development is completed, there could be a perceivable value gain for the users over more infrequent releases.

A high rate of releases might not suit all domains, though. There are circumstances in which it is difficult to think about software changes streaming into systems and devices with minimal preparation and less than comprehensive testing. Obviously, medical devices have high standards of safety that can require long certification procedures that add up to the length of the release cycle. Embedded domains in which circuitry controls whole factories and their production lines are not much easier to hook up with immediately propagated software changes given the possible consequences of updates gone awry. Again, some domains are less compatible with piecemeal releases because the software changes made over time constitute a single, coherent, piece of work. Such works of artistic quality can be found, for instance, in the entertainment sector and mobile games.

In domains where a high frequency of releases are possible, other factors can slow down the rate of adoption. The deployment pipeline has many parts and any part lacking automation makes it a little bit more difficult to deploy changes directly to production environments. Testing phases may be manual or otherwise exploratory, or require physical devices to verify correct software behavior. Especially acceptance testing is an activity usually without automation. Environments used for development and testing might also vary too much from the production environment to make development teams uncomfortable with fully automated deployments. Even with adequate automatic testing, rapid releases might be difficult due to the lengthy test runs. Improvements in the testing processes and hardware can still bring the testing time down considerably.

Having the most streamlined testing and deployment process in place does not guarantee that development teams could push their changes to production at will. Customer readiness to receive frequent releases limits the options development teams have in providing new releases to end users. The willingness of customers is one matter to deal with but at times the development teams and organizations are themselves hesitant to take high frequency releases into use in their projects.

There are many release and deployment models to choose from. Fully automated pipelines have the capability to push changes from developers directly into production environments when selected tests pass, corresponding to continuous deployment. Keeping staging environments up-to-date and putting the release on hold until a decision has been made leans towards continuous delivery. In continuous delivery, release readiness is high with a good capability for releases without actually pushing the changes to production automatically. A fixed release schedule is a predictable way of releasing after a fixed period like every few weeks or every month, for instance. Releases performed more or less on-demand when the situation calls for it are more sporadic in nature.

Rather few development teams or organizations desire very frequent, same day, deployments and releases to end users. Even having continuous deployment capability as understood by continuous delivery is not a common objective for software development teams or organizations. Projects in the industry seem to have release cycles on the order of months or at least several weeks. Underlying software development processes and development sprints in processes such as Scrum could hypothetically impact release cycles in projects. Longer release cycles and on-demand releases are reasonable for software in domains where it is harder to split features into smaller releases. An interesting notion is that in many cases development teams working with projects would be capable of releasing their changes much faster to users than they do. Release readiness is thus at a higher level than the release frequency.

For a good number of reasons, releases ought to be frequent to provide users with software that matches their need, gradually evolves over time, and provides added value in a timely manner. Then again, there is a fair share of reasons and circumstances where releases ought not to be frequent. Releasing changes from developers immediately to the production environment during the same day is certainly not the only choice but a choice among others as far as release cycles go. The perfect release cycle for any project, frequent or not, is the one that makes all parties such as developers, customers and end users equally happy and satisfied with the result.

RQ2: How can a software engineering process be organized to release software frequently?

Releasing software frequently is no small task. Between gathering requirements for a software system and releasing new software versions to users lie many phases and stages. Processes and activities in stages such as requirements management, development, testing and quality assurance, and operations and infrastructure management, must be well integrated with each other without too much delay when moving through stages. The processes and activities roughly constitute the deployment pipeline that generally needs to have a high degree of automation for frequent releases to be possible.

The first facet in the continuous software engineering process is the continuous management of requirements. Feedback should be collected implicitly and explicitly from the end users through monitoring systems, asking for feedback, and providing direct feedback channels to users. Feedback helps to develop future versions and keeps the development teams occupied with enough features on their backlog. If direct feedback channels to users are provided, care must be taken that development teams are not overwhelmed by requests. The rate of incoming feature and change requests either from internal or external sources must match the team's development capability.

Development is the second facet in the continuous software engineering process, encompassing various activities developers are involved in every day. Developers should have access to a distributed version control system to which they can frequently integrate changes, first locally and then to a shared repository. A smart branching strategy guarantees that version control branches used for feature development are short-lived enough not to cause merge conflicts. Build systems used as part of the development workflows have the responsibility of compiling code and assembling packages for subsequent development stages. Continuous integration servers have a role in monitoring selected version control branches and triggering further build and testing jobs in the deployment pipeline when changes are available. Build processes should not only be automated but optimized to keep build and testing times sufficiently short. Optimizing build processes may require hardware changes in the network infrastructure such as servers. Proper optimizations in build and testing processes with the right hardware can potentially save hours in automated build and testing times.

Development should be interlaced with regular, almost daily, episodes of refactoring code and other artifacts. A high rate of code changes and making architectural decisions on the go call for refactoring. Developing future versions and preparing new releases might not be possible if development teams are not allowed enough time to improve internal code structures to better fit the evolving environment. As much as a fifth of development time should be reserved for refactoring, time spent not only on developing new features but improving the structures. Refactoring needs to be part of the daily development routines to keep code in check. For better transparency of the development process, development routines and refactoring should also be appropriately explained to customers involved in development.

Refactoring requires automated test suites to verify that structural changes do not have negative effects. Automated tests are no less important in the continuous software engineering process with frequent releases in mind. Testing in its many forms is the third facet in the process. Lower-level code constructs such as classes and methods should be verified by respective automated unit tests on every turn of build and test cycles. Where possible, more comprehensive user interface tests can complement unit tests by providing the possibility to verify behavior the user of the system would experience. The user's perspective is also central to forms of acceptance testing that is aimed at resolving whether a particular feature or a set of features is ready to be shipped to end users. Acceptance testing requires thorough consideration of the sensibility of changes made and thus is one of the more difficult types of testing to automate. Non-functional properties like performance and security are not easy to test automatically either but assuring a smooth and safe user experience can be considered valuable. Since it is difficult to automate, perhaps testing of some non-functional properties could be tied to larger architectural changes that are more likely to affect such properties.

For many types of testing, emphasis should be given to test automation. Automating tests is an avenue to better perceived product quality, at least if judged by developers. When automated, tests should complete reasonably quickly to make sure that the deployment pipeline maintains a steady speed and activities for subsequent phases can proceed. Developers are not too keen to wait for hours for tests to complete. Tests that are not automated have to be skipped or tested by manual labour. Manual and exploratory testing have their places, especially when matters of opinion are involved in the aspects being tested. Nevertheless, the distance to a fully automated pipeline capable of deploying changes to users in a heartbeat grows with each manual step and stage required.

The fourth facet in the process, operations and infrastructure management, is important not only to testing but also to deploying and releasing new versions in various environments. Testing environments should be as similar as possible to production environments for testing to be effective.

Testing in environments that differ in critical supporting software such as database systems can give a false sense of security. Virtualization and orchestrating virtual environments provide a convenient means for replicating environments in development and in testing, reducing insecurity related to environments. Deployment as the act of shipping software packages across environments and making new versions available to users can also benefit from virtualization. In compatible domains, cloud infrastructure together with virtualization can make the transition from one version to the other easier by helping to ramp up a completely new virtual server infrastructure for new version on the fly. Regardless of the domain, the last mile to production environments and to the end users should be considered carefully with appropriate deployment and release strategies.

Perceiving software engineering processes from the different angles afforded by the four facets of the continuous software engineering process can help to realize which activities might need improvement. The sense of how activities should be organized is of great importance but more information about the state of practice in any given company or organization is needed to truly kickstart the change process. Companies should look towards reflective interviews of internal project personnel or broad company-wide maturity surveys to gauge the current situation in a company.

Detailed interviews help to illustrate which parts of the process are currently handled best and where more work would be needed to improve practices. The degree of manual work versus automated processes is also a good measure to have when conducting interviews. The downside is the amount of work involved in interviews.

Surveys that take advantage of company-specific maturity models offer another avenue for gaining an understanding of the maturity and capability of the organization and its projects. Maturity models can for instance include elements from the four facets ranging from development and quality aspects to infrastructure management as desired. Tiered maturity models can have a set of requirements for each tier for determining maturity. There lies a challenge in dividing maturity models to clearly progressive tiers because practices used in real projects vary. Surveys are tailored to reach a broad audience inside any organization and are helpful as such in spite of being less detailed than interviews.

Both interviews and surveys are helpful in collecting information about development practices and comparing projects inside companies. Because of the different nature of projects, it might be difficult to compare project practices with one another. The project domain, phase of development lifecycle, and available technologies can all be different and limit options for taking certain development practices into use. Taking the limiting factors into account, companies can take advantage of the collected data and direct resources to improving development processes in projects that matter the most.

RQ3: What are the implications of frequent software releases to organizing work?

Adopting continuous software engineering practices that enable frequent releases to end users can be a daunting task for any organization. Working habits must change with support from the whole organization and its customers. A cultural revolution of sorts is needed and it has implications on all levels of the organization.

As with any organizational level change, management must show leadership in carrying out the necessary changes. Management should seek to understand what the current status of its development practices is and set clear objectives for improvement where necessary. Not every project in the organization needs similar attention as projects are in different phases of development. Some projects are in active development and some are merely being maintained for the time being. Employees should be encouraged to actively reform processes, practices and deployment pipeline stages that in the end help to release software versions more frequently. A learning organization gives support and enough resources to its employees. Developers and other members of the development team might need to acquire new skills to improve existing practices sufficiently.

Reorganizing work to accommodate a more responsive development style suited to frequent releases may require shifting responsibilities. Development teams must work more closely with other functions to bring up and maintain infrastructure like servers as needed. It will take time before everyone is used to the arrangements and the situation can be stressing for employees. Members of the development team and others may also have reservations about the effectiveness of the deployment pipeline in catching defects before they turn into failures visible to users. Automated tests help in building trust in the deployment pipeline.

Comprehensive test suites also help with other important development activities like refactoring. Even so, there are certain risks involved in frequent releases and deploying directly into production environments. Depending on the domain, the risks may well be acceptable but erring on the side of caution in domains such as health and industry automation may be the safer choice.

On the positive side, making smaller releases more frequently may reduce tensions caused by infrequent and unavoidably larger releases. Developer spirits may well be lifted if the working atmosphere is positive and accepting towards practices such as refactoring where developers have a chance to perfect their code, embracing the change that comes with continuous software engineering and frequent releases.

Readjusting work and realigning attitudes internally in development organizations are critical factors when considering more frequent releases. Internal matters are important but so are the external relations a development organization has. Software is in many cases developed in partnerships with customers, perhaps even on customer premises. In such joint endeavours, developers might find it difficult to bring their expertise on frequent releases and continuous software engineering practices to the table. Customer organizations are not necessarily interested in investing heavily in process improvement. Feature development can have a higher priority for the customer, which is understandable. The problem is not limited to integrated development teams. Customer representatives in development projects might not consider process improvement tasks to be as valuable to them as feature development tasks.

Customers should be made aware of the possibilities of frequent releases and continuous software engineering practices even before starting a project. Equipped with enough knowledge, they can decide whether to pursue more frequent release models and improve software processes either for themselves or for the development organization. At times, the development organization's capability to deliver and deploy new releases far exceeds the customers' capability to handle them. Customers should be able to opt out of frequent releases if the idea does not suit their current way of working.

Reassuring the customer that frequent releases might be beneficial for them might be difficult, though. Akin to refactoring, good metrics showing indisputable benefits of more responsive development and frequent releases are hard to find. No matter the choice, the development process should be transparent enough so that the customer has a clear understanding of how development resources are being used. If in the end the decision is made to aim for frequent releases, at least the end users might have the chance to enjoy evolving applications that are rapidly molded according to their wishes.

5.2 Threats to Validity

The validity of studies is judged by the strength and correctness of *inferences* made in a study, meaning validity is not a property of the studies or research methods per se (Shadish et al., 2002). As the very concept of truth is contested, human judgment remains the primary method for assessing correctness of inferences in a study. Inferences might be undermined and threatened by overlooked or otherwise unknown factors that could also explain the findings (Shadish et al., 2002). These factors are the threats to validity of inferences made in a study.

Validity and thus threats to validity can be divided into several different classes. A common categorization of threats to validity include *internal validity*, *construct validity*, *external validity* and *reliability* (Runeson and Höst, 2009; Yin, 2014). Previous validity topologies have considered *statistical conclusion validity* as a class of its own although inferences based on analysis of statistical data are also part of internal validity (Shadish et al., 2002). Broadly speaking, *internal validity* concerns matters of causal relations inferences about whether a treatment leads to a specific outcome (Shadish et al., 2002; Runeson and Höst, 2009; Yin, 2014). *Construct validity* refers to the inferences made in turning the real-world phenomenon being studied into tangible operational measures (Runeson and Höst, 2009; Yin, 2014) or to higher-order constructs in other words (Shadish et al., 2002). *External validity* deals with the generalization of findings and how well the inferences of the study apply in other domains (Runeson and Höst, 2009; Yin, 2014). For external validity, it is reasonable to question whether the inferences would hold if a different group of people were observed in a different setting, or if the treatment or measurement variables would be different (Shadish et al., 2002). Inferences should also be repeatable across researchers, which is a concern specific to *reliability* (Runeson and Höst, 2009; Yin, 2014). Given the same data and research protocol, other researchers should be able to replicate the study and reach the same conclusions independently of the original researchers.

Validity concerns vary according to the type of study and the used research methodologies. When there is a high degree of control over the study conditions, for instance, the setting might be less realistic. Such conditions make threats to internal validity less plausible as the causal relationships can be more readily observed but external validity might be at risk due to the artificial conditions as found in laboratory experiments (Shadish et al., 2002; Stol and Fitzgerald, 2018). In contrast to laboratory experiments, in case studies threats to internal validity are less prominent since claims about causal relationships are generally not made in descriptive or exploratory studies (Yin, 2014). Since case study is the primary research methodology used in the thesis studies, the validity threats that apply to inferences made in case studies in particular are the most relevant.

Threats to validity can be mitigated using a set of countermeasures (Runeson and Höst, 2009) or tactics that are specific to different phases in research (Yin, 2014). If at all possible, the design of studies should be robust enough with design controls in place that reduce the threats to validity (Shadish et al., 2002). For example, the strength of inferences related to construct validity in case studies increase when multiple sources of evidence are used and there is a clear chain from data to findings that can be followed (Yin, 2014).

All classes of validity have their distinct threats that should be duly addressed. The plausibility of each threat depends on the conducted study. This section elaborates the threats to internal validity, construct validity, external validity, and reliability of the inferences made in the thesis studies.

5.2.1 Internal Validity

Causal inferences are claims that, based on observations, there is a reason to believe that it was in fact specific events that lead to the outcome. Such causal relationships are put under scrutiny when internal validity of inferences are considered (Shadish et al., 2002; Runeson and Höst, 2009; Yin, 2014). Besides the stated causal relationship, other unknown factors could be at play that might as well explain the findings (Yin, 2014). In fact, examination of causal relations begins by ruling out other possible causes until other possibilities are exhausted and only the most plausible one remains (Shadish et al., 2002).

There are numerous threats to internal validity that might have undesired effects on the study results and the inferences drawn from the results (Shadish et al., 2002). At times, the sequence of causes and effects can be fuzzy and it is unknown whether the effects occurred before or after administering a particular treatment, owing to the internal validity threat *ambiguous temporal precedence*. Another threat to internal validity is *selection*, which signifies the fact that any observed effect could in fact be due to the inherent attributes of the selected subjects and not the treatment in question. Even if the selected subjects were on level ground before starting a study, the situation can evolve during the study, leading to validity threats of *history* or *maturity*. Internal validity threats that are more plausible in experiments include *regression*, *attrition*, *testing* and *instrumentation*. The validity threats specific to experiments are less of a concern for inferences based on case study interviews. Respondents in interviews do not take or retake series of tests and the instrument of the study is mostly the person asking the questions in the interview.

Inferences may be subject not just to a single but many internal validity threats at the same time. The effects of threats can accumulate, weakening the inferences further. This threat of interaction is acknowledged as a threat of its own as *additive and interactive effects* (Shadish et al., 2002). Poor selection of test subjects, perhaps without randomization, may be enhanced by special local history events when the study is being conducted. The dual internal validity threats of selection and history would be in effect at the same time, causing uncertainty to inferences about the real effect of the applied treatment.

Threats to internal validity have to be considered whenever causal inferences are made in a study. Causal inferences do not appear in all types of studies, though. Case studies, for instance, can be exploratory or descriptive in nature and thus lack strong causal inferences (Yin, 2014). Data collection in case studies can consist of interviews with only limited direct observations where inferences are drawn from interview data (Yin, 2014).

As the primary research methodology for the thesis studies is case study and the research purpose is exploratory, there are not many causal relationships or inferences to examine. Still, several inferences made in the thesis studies need to be considered in respect to internal validity threats. According to one of the inferences, continuous deployment and DevOps can be seen to lead to a number of benefits. Another inference states that in software industry, the degree of tooling is positively correlated with the capability to deploy. For these inferences, only a few of the internal validity threats can be considered plausible.

To some degree, the plausible threats to internal validity in the thesis studies include *ambiguous temporal precedence* and *selection* as summarized in Table 5.1. The temporal ambiguity aspect relates to the case study interviews where the respondents were asked to name the benefits they saw associated with continuous deployment (Publication I) and DevOps (Publication V) practices. Because continuous deployment practices and automated deployment to production in particular were quite rarely used in the case study companies, the mentioned benefits are in most cases benefits the respondents thought the practices *could* have. In a sense, the application of continuous deployment and DevOps practices did not precede the moment when the interviews were conducted. If the usage of the development practices would have been more widespread, the implications might have been assessed in a different light by the respondents.

Table 5.1: Plausible threats to internal validity in the thesis studies.

Threat Category	Validity Threat
Ambiguous temporal precedence	Inferences for benefits of continuous deployment and DevOps are weakened by low adoption rates of said practices in the selected industry cases. Respondents might have had limited experience about the benefits prior to answering what they thought of as the benefits of the development practices.
Selection	The selected cases from the industry were from various domains but their background might have had an impact on the findings. Other rival factors based on the domain and specifics of the companies cannot be completely ruled out when examining the causal relationship between deployment capability and the degree of tooling.

Selection of cases in a case study is not quite the same as for controlled experiments as the selected cases are not intended to be statistically representative (Runeson and Höst, 2009). Selection bias is more likely in experiments where randomization of subjects to treatment groups has not been perfect (Shadish et al., 2002). Nevertheless, selection of cases has an impact on the findings of the thesis studies and on a number of drawn inferences. All in all, the selected 33 cases for the thesis studies came from 31 different companies in Finland. Many companies were partners in a research program and were from a wide range of industry domains. Effort was made to select at least several cases from similar domains to get an understanding of the operating principles in a specific domain.

The internal validity threat of selection applies in particular to the inference that concludes that there are signs of correlation between the degree of tooling and deployment capability in companies (Publication II). One strategy to tackle internal validity threats in case studies is to address rival explanations (Yin, 2014). The question is, does the level of tooling significantly affect the capability to deliver and deploy software releases, or could there be other factors that might have an effect? Other factors like the industry domain, company infrastructure, employee work experience, customers or many other factors could explain the deployment capability although the degree of tooling might reflect these factors.

5.2.2 Construct Validity

Concepts in general can have many meanings and it may be difficult to capture the true essence of a concept being studied. For research purposes, concepts or constructs should be specified in a manner that makes it possible to clearly distinguish and measure the construct by identifying valid operational measures (Yin, 2014). Constructs that should be well defined are not limited to outcome measures, i.e., how to measure a particular construct. Besides outcome measures, it is equally important to define the underlying construct and phenomenon that is being studied (Shadish et al., 2002; Yin, 2014). Likewise, labels can be attached to groups of people who are selected for the study and to the settings that offer the circumstances for a study (Shadish et al., 2002). Understanding the clarity of the definition of such categories is just as important as for outcome measures.

Construct validity threats include a host of issues that relate not only to the specificity of constructs but also to the operational measures and methods used in measurement, and to the various motivational factors that can affect both test subjects and experimenters (Shadish et al., 2002). The threat *inadequate explication of constructs* applies whenever the construct cannot be accurately described either because there is insufficient information to do so or the description is too specific. The application of invalid constructs altogether and mistakenly merging features of multiple constructs to a single construct fall into the same threat category. No matter how well the construct is defined, the characterization of other constructs may overlap leading to threats such as *construct confounding* or *confounding constructs with levels of constructs* used with ambiguous constructs.

The operational measures for a construct may sometimes offer too narrow a view to support the inferences (Shadish et al., 2002). Using only a single measure for a construct exposes inferences to *mono-operation bias.* Multiple measures for a construct strengthen the inferences as there are more sources of data to infer from, giving more confidence to the findings when results from the multiple outcome measures point the same way. The bias applies also to person constructs if only a single type of person is used. Another bias is the *monomethod bias* that threatens construct validity when data for outcome measures is collected using a single method. The data collection method may have its limitations that are reflected in the findings and the real effect may remain unobserved because of the method itself. A remedy to fight off such construct validity threats is to use many sources of evidence (Runeson and Höst, 2009; Yin, 2014), effectively increasing the certainty that a valid set of measures is being used and that the data collection methods are appropriate for the measures. Making use of multiple data sources and multiple methods is at times referred to as triangulation (Runeson and Höst, 2009).

Interaction between experimenters and test subjects in the experimental situation has potential to affect the responses, too (Shadish et al., 2002). Reactions from the experimenters in the experimental situation are not irrelevant to the outcome, either. Experimenters can project their own thoughts and emotions about the possible outcomes of a treatment to the test subjects. As a result of the *experimenter expectancies* threat, the subjects might modify their behavior according to the expectations, possibly altering the effect of the treatment. Even when interactions between the experimenters and the participants are neutral, the behavior of the participants might change due to the experimental situation. Merely being part of an experiment might induce *novelty and disruption effects* that are independent of the treatment. Introducing change might be seen as an exciting factor in itself. The introduced change and being involved in an experiment could also impede the work of the participants in some way, disrupting their routine processes. Applying a treatment to a group of people can lead to other construct validity threats that stem from the sensitivity of test subjects to treatment conditions. Sensitivity to group membership can be a factor in some cases but is of less significance in case study interviews where the respondents represent the case in question.

Notable construct validity threats in the thesis studies are related to the clarity of the concepts used in the interviews and surveys, the measures used in gauging the constructs, and the methods used in data collection. The three recognized threats to construct validity are summarized in Table 5.2 and elaborated further in the next passage. One of the challenges is that concepts used in the thesis studies such as *continuous delivery* and *continuous deployment* (Publication I; Publication II), *refactoring* (Publication IV) and *DevOps* (Publication V) are concepts that are hard to exhaustively define. Over the years, even experts have modified their views on the exact meaning of continuous delivery and continuous deployment (Fitzgerald and Stol, 2014, 2017).

When industry experts were asked in the thesis study interviews what they thought refactoring was, they generally understood refactoring as structural changes without behavioral changes but their views were broader than the original meaning of refactoring (Publication IV). Similarly, one of the findings from the DevOps study was that DevOps is not too clear as a

Table 5.2: Plausible threats to construct validity in the thesis studies.

Threat Category	**Validity Threat**
Inadequate explication of constructs	Constructs like continuous delivery and deployment, refactoring, and DevOps have no distinct boundaries. Inferences about their advantages and disadvantages may have been blurred by the lack of a clear definition. Release frequency and particularly the construct lead time without qualifiers are subject to multiple interpretations.
Mono-operation bias	Only a single measure was utilized for lead time in the survey study. Without multiple measures, the inferences for lead time are not as strong as could be.
Monomethod bias	Interviews were the primary means of data collection. Inferences for release frequency and refactoring measures derived from the interviews could have benefited from the usage of other supportive sources like version control systems. Similar triangulation might have been beneficial for the survey study and its lead time measure.

concept to industry experts, either (Publication V). While the variety of views was expected and acknowledged as part of the case study interview protocol, a threat of *inadequate explication of constructs* is a plausible construct validity threat to inferences made about the benefits or challenges of continuous delivery and deployment, refactoring, and DevOps, and operationalized measures such as refactoring frequency. When the respondents were asked to consider these aspects, their understanding of the constructs may have influenced the responses. To take an example, responders thought that DevOps leads to improved quality assurance that can be seen as an inference for a specific study in the thesis (Publication V). Since the meaning of DevOps is contested, the inference becomes weaker as it is not known *what* actually leads to improved quality assurance.

Release frequency is another tricky construct because software versions can be deployed to various staging environments before the release is actually made. There may also be shortcuts to releasing new versions if the situation calls for it, as is the case with urgent fixes. The same concerns apply for the similar construct of *lead time*. Akin to release frequency, lead time can be understood to characterize the time it typically takes for a change or a set of changes to be introduced into a production environment. It is less clear to define when time starts running for the lead time measure and when it stops. If the clock for lead time is started at the instant when a feature request is received and written down somewhere, the measure summarizes the reactivity of the whole development flow from the inception of the idea to release of the new version. Other development phases such as starting implementation of a feature or perhaps even finishing testing could be equally justifiable starting points for lead time if the objective is to judge the capability to develop and release changes.

Capturing all the possible release scenarios was a challenge in the semi-structured interviews (Publication I; Publication II). When asked how frequent the releases in a project are, respondents might in fact mean releases to one of the testing or staging environments, or answer about the release frequency to production. To mitigate the threat against inadequate explication of constructs and to *mono-operation bias*, multiple release frequency measures were used in the interviews (Publication I; Publication II). Respondents were asked to recall the whole release process in the case study project and to illustrate the flow with drawings of their own. They were asked about the typical speed of delivery to production environments and also the minimal time to release a small change. An in-depth elaboration was possible in the interview studies but less so in the survey study (Publication III).

The survey study that had a self-assessed maturity level for lead time (Publication III) and the inferences based on the quantitative results on the average lead time might have suffered from the threat of inadequate explication of constructs. Survey participants were given a survey guide to help them fill the survey. In a survey based on an on-line questionnaire there is no dialogue or similar interaction with the researchers so it is more difficult to address the threat. The mono-operation bias applies to the survey study since only one measure was used for lead time if not taking the target lead time the project was aiming for into account. Deployment to production is mentioned in one of the process area maturity level measures for build and deployment but the highest maturity level has other requirements as well besides deployment directly to production.

Another plausible construct validity threat that applies not only to the lead time and release frequency measures but also to other constructs in the thesis studies is the *monomethod bias*. For the survey study (Publication III), the lead time measure is based solely on self-assessment and there are no other sources to back up the respondents' answer to the question. The multiple release frequency measures used in the semi-structured interviews (Publication I; Publication II) and the fact that the responses about the development process were sent back to the respondents for verification mitigates the monomethod bias to a certain extent.

Nevertheless, the inferences would have been stronger if there had been proper triangulation with hard data from version control systems or other information systems where release tags and dates could have provided a secondary source of evidence. Analysis of version control system history might have also offered an additional data collection source when assessing the refactoring frequency of projects (Publication IV). Semi-structured interviews are essential in collecting data for case studies but getting data from other sources helps to confirm the findings and to build a stronger case.

5.2.3 External Validity

Research outcomes are generally more helpful if the findings are applicable to a wider population than just the sample population used in the study. External validity focuses specifically on this matter of generalizing findings and assessing the strength of causal inferences in cases beyond the ones in the study (Shadish et al., 2002; Runeson and Höst, 2009). In the general case, there could be greater variance in people and the environments, or the outcome measures and treatments could differ, too (Shadish et al., 2002). The generalization can go both ways from the narrow sample of cases to the broad population and from a broad population to a narrow, more specific case or setting (Shadish et al., 2002). Generalizing to other cases with similar characteristics is another form of generalization. Whether the findings are relevant for cases that have slightly different characteristics is of interest as well.

For instance, when investigating software engineering phenomena and cases in the Finnish software industry as in the thesis studies, the generalization could theoretically take many forms. From a sample of software industry companies, the generalization could be to the wider population of software development companies in Finland or perhaps even to software companies around the globe. When considering generalization from the general population to the narrower case, the findings could be from a general population such as software companies in Finland. The interest would then be whether the generic population inferences would hold in a specific domain and case, say for a publisher producing electronic educational material for public schools in Finland.

The generalization to a similar case seems most straightforward. A software company involved in providing non-critical software services through web interfaces with the software running on their own servers or in the cloud would match many of the cases in the thesis studies. Any inferences in the studies should have the best fit to such cases. But even cases that appear to be similar have differences. Perhaps the people working in the company have specific backgrounds, education or experience that makes it harder to draw inferences from companies in the study. Requirements for the industry domain could well differ, making the setting dissimilar.

As a rule, external validity for the inferences in a study is stronger the more variety is included in the persons, settings, and outcome measures but there are challenges to this strategy (Shadish et al., 2002). The downside with increased variety is that the arrangements for the study become harder to handle and the size of individual groups grow smaller, making it unfeasible to have too much variety in the studies.

For case studies, the rules for generalization are not quite the same as for studies that have statistically representative samples and where data is analyzed with statistical methods (Runeson and Höst, 2009; Yin, 2014). Case studies are focused on individual cases that do not represent samples from a broader population as such. *Statistical generalization* is for generalizing from samples to broader populations but without representative samples case studies rely on *analytical generalization*. In analytical generalization, the findings are generalized to a theory, not to a population.

External validity and generalizability can be threatened if some conditions in the study *interact* with causal inferences (Shadish et al., 2002). Inferences may be weakened because they no longer hold in a different setting, or when different units or outcome variables are used. If units of only one kind are used in the study like only males and no females as subjects, the inferences could be coupled with the unit and the threat *interaction of causal relationships with units* might apply.

Subtle or substantial differences in the setting may mean that the results are not transferable to other settings if certain features of the setting interact with the findings (Shadish et al., 2002). Such a threat of *interaction of causal relationship with settings* is present when the original setting, say a large city, has different features and possibilities that could not be found in a countryside town. Similarly, treatment effects can be tied to specific conditions that appear as treatment variations but which ultimately impact the inferences of a study. Thus, an inference might not hold elsewhere if there is *interaction of causal relationships over treatment variations*, which also applies in cases where combinations of treatments interact in a way that

Table 5.3: Plausible threats to external validity in the thesis studies.

Threat Category	**Validity Threat**
Interaction of causal relationships with settings	The operating domain of a company may significantly impact the inferences for the positive correlation between automated deployment pipelines and deployment capability in companies.
Interaction of causal relationship with outcomes	Given the deployment capability outcome measure used in the inference between automated deployment pipelines and deployment capability, the positive correlation might not be observed if another outcome measure such as actual release frequency is used. Many other company-specific factors can affect the actual release cycles and hide the potential to release more frequently.

changes the outcome. An outcome measure may be constructed in multiple ways so the selection of one can interact with causal inferences, too. The threat of *interaction of causal relationship with outcomes* is possible when inferences are good for a particular outcome measure but fail to hold for a related outcome measure.

Since external validity threats are concerned mostly with causal inferences, which are rare in the thesis case studies, only a limited number of external validity threats apply and are plausible. As was discussed in the internal validity section, causal inferences are found mainly in Publication II where the correlation between technological maturity and continuous deployment capability is explored. There were signs of a positive correlation indicating that companies with more tools and an automated pipeline had better deployment capability than companies with fewer tools and automated stages in their pipeline. Regarding the inference, it is possible that settings or outcomes might have interacted with the causal relationship as presented in Table 5.3.

After reviewing results from companies, operating in a range of domains as reported in Publication II, the results already hinted that the domain was a significant factor in determining deployment capability. Thus the threat of *interaction of causal relationships with settings* is a plausible external validity threat in the thesis studies. An industrial factory setting differs from a software service setting running on the web. The implication of the threat is that increasing the degree of automation and tooling might not help with deployment capability, if the operating principles of the domain are not compatible with the idea of pushing changes frequently to staging and production environments. The results are not generalizable to all domains.

Considering the chosen outcome measures and the deployment capability construct used in the inference, the external validity threat *interaction of the causal relationship with outcomes* is plausible. Release frequency can be measured by many different outcome measures. The measure for deployment capability was chosen because actual release frequency cycles in the company cases were fairly long and it seemed that many companies had the potential to release new software versions more frequently than they actually did. There is a strong possibility that the degree of automation and tooling in the deployment pipeline would not show such a positive correlation if another release frequency outcome measure like the actual release frequency would be used. Plenty of good reasons prevent companies from releasing their products to customers and end users as discussed in Chapter 4. Having an automated deployment pipeline helps companies in preparing the releases but the technological maturity does not directly lead to more frequent release cycles.

To mitigate external validity threats and enhance theory building from cases, case studies should apply replication logic when a multiple-case design is used (Yin, 2014). Except for the survey study, all the thesis case studies have a multiple-case design as explained in Chapter 3. For instance, the studies reported in Publication I and Publication II use literal replication inside particular domains to explore similar cases. At the same time, theoretical replication is used to explore cases in different domains, selecting cases in the web development and embedded system domains, for instance. Having a sound replication logic strengthens the external validity of inferences in the thesis studies although theory building is somewhat weaker in the included case studies that have fewer cases.

5.2.4 Reliability

Any finding in a case study should be based on a solid chain of evidence originating from the collected data and its analysis (Runeson and Höst, 2009). Reliability of a study's inferences means that other researchers should be able to follow the trail of evidence and come to the same conclusions given the same data (Runeson and Höst, 2009; Yin, 2014).

Table 5.4: Plausible threats to reliability in the thesis studies.

Threat Category	Validity Threat
Lacking documentation	A case study protocol was followed in all of the studies. Data was collected using semi-structured interviews in most cases. While the steps of the research protocol were documented, only themes of the interview sections were mentioned in the majority of reports. Specific interview questions were not made available, save selected questions used to characterize release frequency.

Reliability is threatened if others are unable to follow the trail of thought. A clear threat to reliability is lacking or missing documentation about the study protocol (Yin, 2014). Shortage of documented steps gives an impression that data collection and analysis have not been systematic or thorough. Theoretically, it would be difficult for other researchers to reach the same conclusions if they were to repeat the same study. The obvious solution is to document steps in the study well enough so that the study protocol could be followed (Yin, 2014).

There are some plausible threats to reliability for the inferences made in the thesis studies. For the majority of the studies, there was a case study protocol that was the same for all the cases in a study. Semi-structured interviews were used for data collection in the interviews and more or less the same set of questions were asked of all the respondents. There was slight variation to the protocol and to the order of questions if the respondent had already covered the topic in earlier responses. When possible, the interviews were recorded.

For the studies reported in Publication I and Publication II, there were close to 50 questions. Themes and a general outline of the questions are reported in Publication II but the complete set of interview questions has never been released. Clearly, lacking documentation of the interview questions hinders reliability although the analysis is based on the data collected using a solid interview protocol as summarized in Table 5.4. A case study protocol was also followed for the other case study interviews reported in Publication IV and Publication V but the complete semi-structured interview question sets were omitted from the reports, showing only the interview themes.

Analysis in the interview studies relies on thematic analysis that is done by coding transcribed passages with labels. Multiple researchers were involved in reviewing the identified themes in all of the cases, which supports the reliability aspect and strengthens the inferences made about the important themes. Especially with the cases found in Publication II, findings derived from interview data were sent back to the respondents for confirmation, further reducing the chance of errors made in the analysis.

As for the questionnaire used in the continuous deployment maturity survey study (Publication III), the questions and themes are described on a general level. The reliability for the inferences in the survey study is improved by the supplementary survey guideline describing some of the conditions the respondents weighed when choosing the correct maturity level for their project.

5.3 Related Work

Continuous software engineering and its related software development practices have been studied previously in different settings. While there are empirical studies from the industry concerning such topics as continuous delivery, continuous deployment, and DevOps, a number of more theoretical studies also exist.

In this section, some of the relevant earlier work is presented and compared to the results derived from the thesis studies. The section has been divided into two parts so that the first part focuses on giving an overview of the existing literature on continuous software engineering topics. The second part of the section digs deeper into the results of individual studies by weighing the results against the outcomes of the thesis studies.

5.3.1 An Overview of Literature

Research of continuous software engineering practices and phenomena such as continuous deployment has been carried out roughly around a decade or so with a clear increase in research output since 2011 (Rodríguez et al., 2017). Over the years, there have been empirical studies that have reported of case studies, surveys, panel discussions, and workshops, not forgetting the more theoretical work. Even if somewhat informal, various published experience reports from the industry also add up to the body of knowledge.

The existing literature consists of both primary studies and secondary studies. Primary studies report of direct observations or experiences, depending on the type of the study. Secondary studies survey the literature landscape for a given topic by analyzing the results of primary studies.

Literature surveys or reviews that gather the results of multiple studies provide a good starting point for a general overview of existing literature on continuous software engineering.

Systematic literature reviews and systematic mapping studies are examples of secondary studies that harvest existing primary studies to answer research questions (Kitchenham and Charters, 2007). What makes them systematic is the use of well documented search strategies and explicit criteria according to which primary studies are included or excluded from the study. Secondary studies following the systematic review protocol are more rigorous since the search strings and sources are known. The difference between systematic literature reviews and systematic mapping studies is that mapping studies are used to cluster information with a broader, less focused, approach.

To give an overview of existing literature, this section provides a limited tertiary review of systematic literature reviews and systematic mapping studies that have been conducted on continuous software engineering topics. These secondary studies have focused on topics such as continuous deployment, continuous delivery, and agile release engineering. Table 5.5 summarizes the benefits and challenges of continuous software engineering practices drawn from the three secondary studies selected for the tertiary review (Rodríguez et al., 2017; Karvonen et al., 2017; Laukkanen et al., 2017). The table also illustrates how many primary studies were included in each secondary study, ranging from 30 to 71 primary studies. While the themes and the emphasis of the studies are somewhat different, all studies use similar search strings for searching primary studies. The summary of the benefits show that there is some consensus to benefits such as increased responsiveness of development in terms of shorter lead time and improved feedback. For the challenges, testing seems to be a major concern mentioned in all of the secondary studies. The findings of each secondary study are further described in this section.

Rodríguez et al. (2017) conducted a systematic mapping study on continuous deployment. Using a wide array of search terms equivalent to continuous deployment, they selected 50 primary studies for closer analysis. Many of the studies reported of empirical experiences from the industry. The scientific quality of the primary studies is apparently slightly compromised as quite a few studies did not disclose too much information about the research context or describe the validity threats appropriately.

Rodríguez et al. (2017) were able to identify 10 recurring themes from the primary studies. Most studies mentioned the acceleration of release cycles, which can be seen as one of the key objectives of continuous deploy-

Table 5.5: Benefits and challenges of continuous software engineering practices reported in secondary studies.

Publication	Theme	Primary Studies	Benefits	Challenges
Rodríguez et al. 2017	continuous deployment	50	shorter time-to-market, improved release reliability, increased customer satisfaction, improved developer productivity, smaller testing scope, continuous feedback, rapid innovation	realigning processes and people, increased QA effort, embedded domain constraints, customer willingness
Karvonen et al. 2017	agile release engineering	71	frequent feedback, improved communication, customer satisfaction, improved lead time, improved developer productivity	domain constraints, insufficient testing resources, size and complexity, changing mindset and culture
Laukkanen et al. 2017	continuous delivery adoption	30		40 problems identified in 7 themes, system design and testing most critical with incompatible architecture and time-consuming and unreliable testing

ment. The role of automation in the development process was considered central in the studies partly because manually performed activities slow down the release flow. Quality assurance and testing in general is one of the major themes of continuous deployment. If deployment is to be continuous, so must testing be. There is a slight risk that by focusing on testing of individual changes, sight of the big picture is lost. At the same time, continuous testing is seen to foster a culture of quality. The change in the nature of testing also shows in that the amount of defects can increase in some cases and a great number of releases are used to fix defects introduced by rapid releases. Version control systems need to be able to keep up to speed of changes so configuration management with various branching strategies has been one of the themes in the studies.

The aggregated themes also reflect the reactive nature of development and possibilities of continuous deployment. Design and development of features can be made in smaller batches with feedback gathered from real users guiding planning and implementation of features and fixes. Customers and end users can be more heavily involved in the development process. Monitoring system health after releases is yet another possibility for gathering implicit user data for upcoming tasks. If something goes awry, individual features can be more easily rolled back thanks to the smaller size of releases. Flexibility in design and development adds up to potentially more experimental workflows as highlighted in the themes. Architecture of the system must bend together with the rapid changes, too. Constant architectural change may lead to incurring technical debt that needs to be paid later. The position of continuous deployment in relation to agile software development is brought up in the primary studies. Continuous deployment is thought to be compatible with agile and lean software development but the aspect of continuity makes it different.

Finally, organizational matters arise as an important theme from the primary studies according to Rodríguez et al. (2017). Changing the development process to support continuous deployment requires effort from all stakeholders across the organization. Previous team divisions according to function might not work because planning of features is continuous and features might be rolled out to customers before marketing, for instance, has had time to react. Teams need to be cross-functional with expertise from multiple fields made available in the same team. Development team members might face added responsibilities but all development activity should be kept transparent so everyone is on the same page about what has changed and when, even if the system breaks due to a bad release.

Aside from the major themes, based on the analysis of the primary studies Rodríguez et al. (2017) were able to identify a number of benefits and challenges of continuous deployment practices that are listed in Table 5.5. An often mentioned benefit in the primary studies is the ability to shorten the release cycle from several months to perhaps several weeks that shortens the time-to-market as well. Despite some of the worries related to quality of releases, automated tests give a sense of reliability to releases. Customers are also said to be happier with shorter release cycles since they do not have to wait so long for requested fixes and features. Because individual features can be shipped and released independently, the overall productivity of developers is reported to increase when continuous deployment practices are in place. Tests can be focused on the individual features being deployed and thanks to the smaller testing scope, it is easier to find defects. Feedback that flows continuously between developers and customers binds the parties stronger together, improving relationships and leading to the additional benefit of rapid innovation together with customers.

The challenges listed by Rodríguez et al. (2017) deal with concerns about changing the process to suit continuous deployment practices. Realigning processes and people takes great effort within the organization, posing a challenge. Another challenge is that compared to previous software development processes where releases were done less often, more resources might need to be allocated to testing. Several primary studies highlight the fact that a continuous process might not fit the embedded domain due to architectural incompatibilities. In some cases, the challenge is that customers may have doubts about the quality of rapid releases and are not willing to accept more frequent releases. Importantly, the listed benefits and challenges are noted mostly to be the *perceptions* of professionals and as such are not necessarily backed up by empirical evidence from the field.

In another systematic literature review, Karvonen et al. (2017) studied agile release engineering practices by analyzing 71 primary studies. Besides continuous deployment, continuous delivery and rapid releases, the search terms also included continuous integration. The primary focus of the review was to understand the various impacts agile release engineering has on software development and its outputs. Clustering of the primary studies resulted in several clusters for impact featuring the advantages and challenges of agile release engineering, methods to mitigate adoption challenges, improving and refitting of development processes, prevalence of the practices in the industry, the impact of the practice on various success factors, and a larger cluster of lessons learned.

Owing to the broader theme, the benefits and challenges summarized in Table 5.5 for the systematic literature review of Karvonen et al. (2017) contain elements associated with continuous integration as well. Frequent feedback is once more mentioned to be one of the benefits of agile release engineering. It is not only feedback from customers but also feedback resulting from the development process to developers and other parties that is to be considered important. Integrating changes quickly works well only if feedback from testing to developers, for instance, is prompt. Quick feedback in the development flow is seen to improve communication. While customer feedback is deemed important to software development, the actual methods for eliciting feedback are not covered well in the analyzed primary studies. Customers for whom software is being developed, are more satisfied with iterative development that is characteristic of agile software development in general.

When moving from long release intervals to shorter ones, the lead time of changes improves. Introducing changes or fixes to products has been found to take less time if release cycles are measured in several weeks, not in several months. Version control systems and continuous integration systems can handle workflows where developers work on multiple changes at the same time, which is seen to lead to the benefit of increasing productivity of developers.

The challenges of agile release engineering that appear in the primary studies analyzed by Karvonen et al. (2017) are related to many aspects of software development. Concerning the environment for development, constraints that are specific to domains such as telecommunications and mobile make it more difficult to release changes frequently. Even if the environment is suitable, establishing a proper development process for handling frequent changes can be difficult. Testing is recognized as an activity that takes the most effort when performing releases. Adopting agile release engineering practices is hard if there are not enough resources to allocate to testing. A development organization may not have the necessary technological capabilities with appropriate testing tools in place, either. Another challenge is that the system and its architecture may be too big or complex, making integration of changes harder to handle.

Karvonen et al. (2017) take note of several organizational challenges when people have to adapt to processes required by agile release engineering. The changes to previous ways of working can be disruptive and people can experience great strain because of the skill demands of the new process. Developers need to have the right mindset when working with a continuous stream of changes.

In addition to the benefits and challenges, Karvonen et al. (2017) searched for evidence of how commonly agile release engineering practices are used in practice. Interestingly enough, it seems that industry organizations rarely practice continuous deployment. Only a few known cases of continuous deployment are reported in the primary studies. In those cases where continuous deployment is in use, the platform is predominantly web. The share of desktop software that is updated continuously is reported to be small. Looking at the statistics for mobile application stores, only percentiles receive updates every week and for the rest the release cycle is more infrequent. Open source repositories allow a different viewpoint to assessing prevalence since the code and configuration of projects is freely available. The usage of continuous integration has not exactly skyrocketed in open source projects lately and the rate of adoption has remained roughly the same as before.

Based on the findings of the primary studies, Karvonen et al. (2017) have drafted a list of recommendations that organizations should adhere to if continuous deployment is the objective. According to the recommendations, agile software development should be the primary mode of working in the organization to start with and everyone should be comfortable with working agilely. Continuous integration should be an everyday practice in the organization. Whatever tests the continuous integration server executes, the tests should have a good coverage of code to promote trust in the tested versions. Acceptance testing should not be forgotten when testing the changes. In order for version control to work properly, any and all changes should be pushed to version control branches that only have a short lifespan. The architecture of the products and systems should support the frequent integration of changes. Deployed versions and new releases should not cause any adverse effects. The service for end users should remain uninterrupted by changes and information about any bigger change in software behavior should be made available to end users. Dependencies must be taken care of, too. Other third party applications that take advantage of the services offered should work equally well after the changes. The transition to continuous deployment itself should be made with full transparency and with consent of all involved parties. This list of recommendations serves as a good reminder what to take into account when considering the transition to continuous deployment.

Instead of focusing on potential gains of continuous software engineering practices, the systematic literature review by Laukkanen et al. (2017) drills down into specific adoption problems of continuous delivery. The detailed review analyzes 30 empirical primary studies that have reported on adoption problems related to continuous integration, continuous delivery and continuous deployment. Due to the low number of studies published on continuous deployment experiences, the main focus is on continuous integration and continuous delivery. Through analysis and synthesis, Laukkanen et al. (2017) also extract causes and propose solutions for the identified problems.

Laukkanen et al. (2017) were able to identify a total of 40 distinct problems that were categorized into 7 themes. The themes were build design, system design, integration, testing, release, human and organizational, and resource. Problems from the themes integration and testing appeared most often in the primary studies. In many of the analyzed cases, there was either an explicit or an implicit theme that could be considered critical. As illustrated in Table 5.5, the two critical themes with problems that raised the most concern were system design and testing. From the design perspective, the overall system and software architecture can be incompatible with the concept of continuous delivery. Software may be structured in such a way that the dependencies are hard to handle when making changes; the components may be too intertwined. Architecture that is too modular has a similar effect. Modular architecture has its benefits but making a build can be more complex. Critical testing problems revolve around the feedback tests have to offer. Testing might take such a long time that the waiting for the results kills the pace of development. Tests that do not consistently give the same result is yet another matter to consider as a critical testing problem. Problems from both of these critical themes lead to all kinds of other problems.

Besides the themes that were considered most critical in many cases, the analysis of Laukkanen et al. (2017) covers a host of other significant adoption problems in other themes. Build design problems are related to system design problems since complex designs may lead to complex builds and build scripts that take a lot of effort to maintain. Together with testing, integration is a major problem theme. Version control systems act as a central hub for integrating changes and any problems in integration has a direct impact on the work of developers. The probability of time-consuming merge conflicts increases if the integration frequency is low and the size of commits grow due to poor version control branch practices. Merge conflicts may then lead to broken builds, making it impossible for other developers to continue their work and stopping the development flow of developers.

Merge conflicts and broken builds may actually originate from problems in testing, as noted by Laukkanen et al. (2017). When testing takes ages to complete, developers are discouraged from integrating their work frequently. As a result, code commits grow in size, which can cause problems in integration. There is also a greater chance for broken builds if the tests are not reliable. In the end, the chain reaction is complete when work cannot be integrated frequently, causing merge conflicts and broken builds, which makes it harder still to integrate work. Testing that takes much time is classified as a critical problem for good reason. Testing needs to support the integration flows in order for continuous delivery practices to be possible. In environments where testing is more difficult due to hardware requirements or due to the necessity to test on multiple devices, there tends to be more adoption problems.

After testing, the logical next stage in the development process is the deployment and release of changes. Laukkanen et al. (2017) mention there is little empirical evidence of release problems found in the primary studies. In some cases, deploying changes has been reported to be difficult if the system is broken into too many small pieces. The concern for the end user or customer experience is generally reflected in the listed release problems. Frequent releases might mean frequent user exposure to failures caused by bad releases. Even if the release is good, deploying a new version may cause downtime observable by the user. Making a release fully backwards compatible with previous versions is a problem of its own. When third parties depend on the software functionalities, extra caution in releases is warranted. Due to the caused disruptions, certain users might not be favorable to frequent releases, either. When deployed, users might not be able to take advantage of new or changed features unless properly advertised. Documenting and marketing a versionless product is hard as well.

According to Laukkanen et al. (2017), human and organizational adoption problems are more general problems that are not particular to any specific phase in development. Organizational structures have an impact on the way resources are allocated and managed. Without the right resources, teams may lack the skills they would need to prepare changes frequently. Multi-disciplinary teams may be the answer but the accelerated pace of development also calls for increased communication and collaboration between teams, which may be lacking in problematic scenarios.

In addition to organizational structures and management, personal traits of team members influence the outcomes of a development process as suggested by the analysis of Laukkanen et al. (2017). Developers may find it hard to stick to the agreed development protocol and integrate their work frequently if the process is not fluent enough in testing, for instance. Perhaps the resolve of developers is not always strong enough to follow the process to the letter and the problem is the lack of discipline. Integrating changes may be less frequent and tests may not be written as often as they should be. The motivation of team members may waver especially when setting up the deployment pipelines at the outset if people are not allocated enough time to do the changes. Motivational issues are important but at the same time the effort required in the transformation process is a resource problem. In order to reach the maturity required by continuous delivery, team members need to master many skills, which can cause a great deal of stress especially to the less experienced team members who have much to learn.

Although there are numerous problems associated with continuous delivery adoption, the bright side is that Laukkanen et al. (2017) were able to identify solutions to most of the problems from the primary studies. Only some of the build design problems remained without apparent solutions. Problems in the system design and integration themes prevent changes from being integrated frequently. To keep the integration frequency of changes high, features that are not ready can be integrated but disabled with feature toggles until development is complete. Using only a single version control branch for all changes forces developers to integrate changes with care, which can solve integration problems. At the same time, it is essential to protect the version control system from breaking changes using appropriate review protocols since work is halted with broken builds. Making sure that developers can execute the build steps reasonably fast is a good antidote against infrequent integration of changes, too. The same principle should be applied to testing.

Running automated tests can take a long time and the delayed response and feedback was identified as a problem. Several solutions mentioned in the analysis of Laukkanen et al. (2017) can help to save time from testing. Changing the execution order of tests can reveal problems from critical tests earlier than without reordering tests. Testing can even be dynamic and adaptive so that tests that have failed most often in the past are automatically put to the top of the test queue. Executing tests in parallel can speed up the response time for tests if the existing testing infrastructure cannot cope with the amount of tests. In domains where software is developed for specific hardware, simulators can replace some of the need to test on actual devices. If defects slip through to production environments in spite of all the testing, it is good to have redundant services and the ability to roll back changes in the case of failure.

The suggested solutions for release problems include useful hints for release management. Advertising new features in blogs could help users in discovering features. Practical advice is offered for customers who dislike receiving frequent updates. In suitable situations, the discomfort of the customer can be reduced by letting the customer decide on their own whether they want to receive frequent updates or not.

Finally, problems related to the human and organization theme require broad solutions that address cross-cutting concerns in organization as described by Laukkanen et al. (2017). Management must be alert when introducing continuous delivery practices in an organization by devising a strategy and a plan that the organization can follow. Because of the higher skill requirements, employees should be offered ample opportunities for learning. It is all the better if at least some complexity can be hidden behind abstractions and tools of the trade to flatten the learning curve. Bringing about an open working atmosphere where people are allowed to freely share their concerns is one way to reduce anxieties and stress that operating in a different fashion can induce. In general, work can be directed so that developer teams can more easily address problems in development. Due to the disruptive nature of truly blocking problems such as broken builds or other hard problems, shifting the attention of the team momentarily on the problem can provide a quick solution. Collaboration between team members should be encouraged. In the case of motivational problems, being open about the value and expected benefits of continuous delivery practices can sway opinions of employees in doubt and give a sense of purpose to the improvement activities.

Summary of Literature Review Findings

The findings from the literature reviewed in the three secondary studies (Rodríguez et al., 2017; Karvonen et al., 2017; Laukkanen et al., 2017) share common ground, having similar inferences drawn from the primary studies. These findings are also well aligned with the findings made in the thesis studies as will be summarized in this section.

Looking at the identified benefits and challenges of continuous software engineering described in Table 5.5, common themes emerge from the summary. Adopting continuous software engineering practices can be seen to lead to a shorter time to market or to an improved lead time (Rodríguez et al., 2017; Karvonen et al., 2017). Both of the mentioned perks of continuous software engineering mean that it is possible to increase the release frequency; a fact that is also highlighted in the thesis studies.

The importance of feedback within and outside the development organizations is acknowledged in the literature reviews (Rodríguez et al., 2017; Karvonen et al., 2017). Receiving prompt feedback from customers to feed the development process is considered beneficial as is the rapid feedback developers can get internally when developing a particular feature. A tighter relationship with the customers and end users was deemed important and seen to improve feedback cycles in the thesis studies, too. Reacting swiftly to feedback and developing corresponding software functionality desired by customers and end users is seen as a pathway to improved customer satisfaction (Rodríguez et al., 2017; Karvonen et al., 2017).

Maintaining a high quality of releases together with an increased release frequency was seen as a concern in all of the three literature reviews (Rodríguez et al., 2017; Karvonen et al., 2017; Laukkanen et al., 2017). Quality assurance is a major theme in continuous software engineering because testing can take a lot of time and there might not be enough resources to carry out extensive testing when releases are made more often. Similar adoption challenges such as the required manual effort in testing, the difficulty of performing acceptance testing automatically, and testing taking too long a time were brought up in the thesis study interviews. The fear of introducing breaking changes when releasing often came up in the thesis studies and the fact was mirrored in the review studies (Laukkanen et al., 2017).

Findings from the literature reviews are in agreement with the thesis findings that impediments to adopting continuous software engineering practices have been recognized in specific domains. Software development in embedded and mobile domains makes for an environment in which it is more difficult to deploy changes quickly (Rodríguez et al., 2017; Karvonen et al., 2017). In particular, there seem to be more adoption problems in hardware intensive domains when testing is more difficult (Laukkanen et al., 2017). The architecture of software systems in the embedded domain might constrain software development and rapid deployment of changes (Rodríguez et al., 2017). Adoption challenges related to architecture are known to be issues in other domains as well (Laukkanen et al., 2017). Similar architectural issues such as complexity of the build process and modularization were also pointed out by the respondents in the thesis study interviews.

Adoption challenges relating to organizational aspects have not been forgotten in the literature reviews, either. The effects of adopting new practices can be disruptive to the current working habits and have an effect on people (Karvonen et al., 2017). Personal motivation is another human factor in adopting continuous software engineering and can be a challenge (Laukkanen et al., 2017). Successful adoption of continuous software engineering practices calls for an organization-wide plan (Laukkanen et al., 2017). Managerial responsibilities and human factors were emphasized in the thesis findings, too.

The guidelines for adopting continuous deployment offered in the literature reviews (Karvonen et al., 2017) is a close match to the continuous software engineering process blueprint described further in the thesis. Having an agile software development process in place with continuous integration servers running tests that have a good coverage is a fine recommendation for continuous deployment. The insights in the literature reviews regarding the guidelines are backed up by the analysis of the software development processes carried out in the thesis studies. Adopting continuous deployment requires many practices in place and it is not completely suited for all domains or environments where there are complex architectural dependencies. Perhaps the adoption challenges can partly explain why continuous deployment is not a reality yet in many domains (Karvonen et al., 2017). The findings from the thesis studies do corroborate the low adoption rates of continuous deployment identified in the reviews.

5.3.2 Reflections on Research Questions

While an overview based on literature surveys synthesizes results from existing primary studies well, there is room for a closer inspection of the results from the primary studies in order to compare the results to those obtained in the thesis studies. Following the structure of the three research questions of the thesis, this section reflects on primary study results individually for each research question.

Reflections on RQ1: Why should software releases be frequent?

Regarding the rationale of high frequency releases as discussed as part of the examination of RQ1, other studies have found benefits and challenges associated with continuous software engineering practices. Akin to the findings of the thesis studies, improved time to market of developed features is generally seen as the benefit of continuous deployment (Chen, 2017; Parnin et al., 2017). Increasing release frequency obviously decreases the time to market of particular features, as shown by Facebook where daily deployments are possible (Savor et al., 2016). Because continuous delivery, continuous deployment and DevOps require a certain degree of automation and fluency in the development process, it can also be more efficient to implement individual features (Itkonen et al., 2016; Chen, 2017; Snyder and Curtis, 2018). Although productivity is notoriously difficult to measure, increases in productivity have been observed when moving to shorter release cycles even if the deployments are not done every day (Snyder and Curtis, 2018). A similar notion of improved productivity was also brought up in the thesis interview responses.

For product quality, the experiences are mixed. Improving test automation was seen to lead to better product quality in the thesis interview responses. The reduction of observed defects and general improvements in quality have been noted as benefits also in other studies (Itkonen et al., 2016; Chen, 2017; Parnin et al., 2017). In contrast, the *reduction of product quality* has sometimes been identified as a challenge of continuous deployment due to the chance of frequently introducing unnoticed defects (Claps et al., 2015). The fear of shipping broken code resonates well with the thesis interview responses, too. Automated tests build confidence in releases but it is not always enough to prevent all failures from happening. Although the fear of deploying broken code may be warranted, even daily deployments do not appear to degrade software quality significantly more than is expected (Savor et al., 2016).

The role and importance of feedback from users is recognized in the thesis interview responses as one of the additional benefits. Other experiences and opinions from the industry seem to match the idea that with more frequent releases there is the possibility to collect user feedback more frequently and thus build better, more meaningful, products overall (Chen, 2017; Parnin et al., 2017). While gathering data from users is a definite advantage, monitoring and analyzing feedback requires skill and can be a challenge in itself (Claps et al., 2015).

The proposition of providing user value faster through rapid releases gains some theoretical support from previous work (Dingsøyr and Lassenius, 2016). Continuous deployment and high frequency releases can be seen to be on the same continuum with continuous value delivery. Ranging from business value to relative worth, value has many meanings. It may be difficult to measure value to the user before a feature is actually implemented. Continuous deployment may help in this so that it provides a means to quickly provide various features for user testing. Together with other continuous software engineering practices and monitoring production environments for user behavior, continuous deployment strengthens the ability to supply features that matter the most.

The suitability of continuously deploying changes to production systems running in specific safety critical domains was questioned in the thesis interview responses as was the difficulty of deploying changes in the mobile domain. There is agreement that the web domain is considered the least challenging of domains when striving for continuous deployment (Adams et al., 2015). A continuous stream of changes is considered more difficult in the mobile domain where there is no ownership of the whole ecosystem and harder still in safety critical domains (Adams et al., 2015).

The picture might not be so bleak for safety-critical domains, though. Steps have already been taken in the automotive industry to integrate continuous deployment activities in the complex development process (Pelliccione et al., 2017). Safety is a real concern and there are many stakeholders involved but with cars having open communication channels to infrastructure services, deploying changes on the road is not far from reality.

In addition, there are signs from the highly regulated healthcare domain that at least continuous delivery and more flexible development procedures are being considered if not completely adopted (Giorgi and Paulisch, 2019). Risks in the healthcare domain software need to be well controlled and evaluated throughout the product life cycle but mechanisms have been developed to cope with the regulations. Automated workflows that integrate directly with version control systems can assist in detecting changes in source code that require a safety standard compliance review (Stirbu and Mikkonen, 2020). When safety is so built into the daily development workflow, it is possible to guide the creation of the necessary safety-related documentation and architecture diagrams, even when the integration frequency is high.

It is not only the automotive industry or software development in the medical sector where advances in change and release management are observed. With the help of DevOps, release cycles seem to be shortening in the financial sector, too (Snyder and Curtis, 2018).

Reflections on RQ2: How can a software engineering process be organized in order to release software frequently?

Already the early illustrations of a pipeline suitable for continuously deploying changes to production systems (Humble et al., 2006) described the interplay between development stages and output artifacts such as versioned code and other binaries needed in the software development process. The discussion of RQ2 echoed the necessity that to be truly continuous, continuous software engineering needs a process of its own. Existing studies have also highlighted key features that a continuous software engineering process should have. As one of the arguments goes, current process metamodels are not even adequate at describing the parallel workflows required to handle individual incoming change requests as they come (Krusche and Bruegge, 2017). Apparently, an event-based process model where events trigger workflows is more suitable for continuous software engineering (Krusche and Bruegge, 2017). Such an approach is noted to be more appropriate for handling feedback loops and introducing continuous changes as part of the software development.

In practice, it has been mentioned that applications and the surrounding infrastructure should have certain architectural qualities that make it easier to deploy changes more frequently (Bellomo et al., 2014). For instance, any steps that can be taken to improve testability help in easing automated testing and thus doing deployments at a more rapid pace. Dividing a bigger component into smaller components can in a similar way reduce building, testing, and deployment efforts because there is less code to work with and fewer tests to execute on every test run. In this context, it was also noticed that deployment was in some of the cases made easier by harmonizing environments with virtualization techniques and by applying the blue-green release switching technique to deployments. These notions are quite similar to the findings of the thesis for the continuous software engineering process blueprint discussed as part of RQ2.

The journey from fixed releases to more frequent releases can be long. An example from the industry shows the steps a company had to take when moving to shorter release cycles (Neely and Stolt, 2013). The company wanted to cut down the release cycles from eight weeks so a release could be made without a fixed schedule. Old development processes no longer worked as they used to have. Without fixed releases, there was no need for regular planning meetings since work items were updated on the go. A cornerstone in the transformation was the mantra to automate the development process as much as possible. Especially automating and optimizing testing was considered critical because the duration of testing was seen to impact the minimum continuous delivery time. Both of these observations match the ones made in the thesis well. A keen idea in the company was to improve the processes gradually so the release frequency could improve slowly from eight weeks to two weeks and onward from there. In the end, the company had a continuous delivery process in place with which they could release new versions when they wanted. There was a learning curve to get all parts in place and people had to adapt to new working conditions but they succeeded in the end.

Reflections on RQ3: What are the implications of frequent software releases to organizing work?

Setting up a deployment pipeline and introducing new processes and roles for software development requires changes across the whole organization. Such profound changes cannot be done without the help of management who should show leadership accordingly as described as part of the thesis results ensuing after the discussion of RQ3. The essential nature of management has been noted in a number of other studies as well. If management is not committed to software process improvement, there may simply not be enough driving force to implement a demanding practice like continuous deployment throughout the organization (Claps et al., 2015). Management holds the key to resources so they may need persuasion (Chen, 2017). Managers who are not too keen to improve processes may hold back the true potential of development teams (Savor et al., 2016).

Development teams might have a hard time adjusting to the new roles required by continuous deployment and DevOps as pointed out in the thesis. Responsibilities are shared and work is distributed in different configurations across developers and people who were previously in charge of infrastructure. There is evidence from other studies that there are hardships involved in taking on duties that members of the development team are possibly unfamiliar with. Adapting to new roles has been found to be one of the challenges related to implementing continuous deployment (Claps et al., 2015). Different team configurations are possible in situations where development teams are striving for continuous delivery and continuous deployment. Either developers take more responsibilities from the operations personnel or collaboration between the two realms is substantially enhanced to allow for more flexible releases (Shahin et al., 2017). Teams can also be collocated or merged so that a single cross-functional team has all the capabilities it needs for developing and deploying releases (Shahin et al., 2017; Chen, 2017).

Joining development and operations functions, and working according to continuous software engineering practices is not an easy task. Individual team members may need a broader skill set than before, which naturally leads to the need to learn, as pointed out in the thesis. Other studies reinforce the idea that the experience of the team is requisite for the successful implementation of continuous deployment (Claps et al., 2015). Unfortunately, practitioners recognize that the curricula in universities and other schools is not completely up to date regarding essential skills for release engineering and deployment (Adams et al., 2015; Parnin et al., 2017). Those working in the industry have expressed that they feel like they have to learn all these valuable skills while working (Shahin et al., 2017). One proposed solution to limit the burden on individual teams is to form specific teams that are charged with building a deployment pipeline that can be used across the whole organization (Shahin et al., 2017; Chen, 2017). Such an arrangement is argued to free up resources especially in large organizations since project teams can call in help when needed and actual project team members do not have to invest so much time in building the deployment pipeline.

Be it either the end user who is using the software or the customer for whom the software is made, it is important to nurture the relationship between different parties. There is some evidence that by moving to more continuous forms of development, communication between developers and customers is accelerated and thus collaboration improves (Itkonen et al., 2016). Relationships have been seen to be less tense after the adoption of continuous deployment (Chen, 2017). In general, customers are thought to be more satisfied if continuous deployment practices can be applied (Parnin et al., 2017). Like the thesis results show, there are more sides to customer relationships and not all customers are ready for more continuous releases. The same phenomenon has been witnessed in other studies. Customers may not be willing to accept the continuous deployment model with rolling updates for their system due to preference or insecurities related to successful deployment of releases (Claps et al., 2015; Parnin et al., 2017).

5.4 Implications to Theory and Practice

The work published as part of this thesis can be seen both as a contribution to the scientific body of knowledge in software engineering and as a set of blueprints and guidelines for the industry. Being the primary research methodology for the studies in the thesis, a case study has the advantage of being rooted in practical reality while making it possible to devise broader theories about the phenomena in the field. This section shows a number of implications the presented work could have to theory and practice.

Theoretical implications can be seen to be elements in the research findings that bring up novel ideas or theories, or increase understanding and so advances science. The case studies from the software industry presented in the thesis contribute to the current understanding of how software is being developed and delivered to end users in different industry domains. An analysis of the release practices in the industry shows that release frequency has many aspects. The results from the case studies help build a theory for the rationale of frequent releases, that is, why frequent releases make sense in certain domains and less in others.

Release frequency can be characterized by many metrics. The actual period between releases is a useful metric but the additional release frequency and deployment capability metrics presented in the thesis show that a single metric is not sufficient for understanding the current state and potential of a software engineering organization and its projects. A theory devised from the case studies in the thesis proposes that deployment capability in particular is furthered by the availability of certain tools in proper stages of the deployment pipeline. By the devised theory, deployment capability is also strongly conditioned by the operational domain for which the software is being developed.

An analysis of the software development processes in the presented industrial cases helps to build a clearer picture of development stages required to prepare, develop and release new software versions to users. Derived from the analysis of the processes, the development blueprints show what should be taken into account in each stage in order to prepare the development and deployment pipeline to better support increasing release frequency. Further analysis of the development stages brings clarity to the essential development processes and practices by grouping development aspects into specific facets.

The empirical studies carried out in the industry also point out shifts in the development philosophy that high development velocity and increasing release frequency might induce. Supporting practices such as refactoring have a bigger role in the development flow since structural and architectural changes are more continuous. As shown in the studies, the shift in development philosophy involves changes not only in the way development work is organized internally in an organization but also in the manner how releases and development is managed together with customers.

Practitioners from the industry can leverage the research findings to further improvement of development practices in their own environment. Considerations of the rationale for releasing new software versions frequently can help to identify the release frequency that feels right for a specific product in a specific domain. If the informed choice is to move towards more frequent releases, understanding which metrics are relevant for characterizing release frequency assists in assessing the current situation and setting objectives for the future.

Any organization wanting to move forward should get a grip on its current state of affairs. Companies working in the software industry can use similar data collection methods as introduced in the thesis such as interviews of project personnel or company-wide maturity surveys tailored for the company. After having gathered enough information about the current situation, companies can compare their development activities, practices and processes against the continuous software engineering process blueprint presented in the thesis. Making the comparison can be helpful in forming ideas about areas requiring the most attention given the constrains in the domain and the individual objectives of the company.

Moving forward implies change to current ways of working. As the findings of the thesis studies show, there are many factors that need to be considered when managing change towards more frequent releases and improved development processes. Software development organizations can take note and reflect on the various organizational aspects of change detailed in the thesis. Leadership in management and supporting change in an organization are needed and employee attitudes should not be overlooked either. Every software project or product is different and can have internal or external customers who might have a wide variety of opinions about high frequency releases. The notes made in the thesis should help those in the industry in choosing the right release strategy and tactic in each case. A good relationship with the customer and transparent communication regarding the development process and release objectives pave the way for successful development right from the start.

Chapter 6

Conclusions

Software releases are milestones in software development where the fruits of labor are made public and available to its users. Every software release involves a great deal of work from the inception of ideas to development, testing, deploying and releasing the new version. The frequency of releases determines how often new versions are made available. Long intervals between releases mean that the users have to wait longer for fixes or improvements to current behavior. In the industry, the releases can be months or years apart. Reducing the time between releases and increasing release frequency seems like a sensible objective from this perspective.

Deriving from the experiences of software professionals in the industry, the empirical studies in this thesis have explored the idea of increasing release frequency and the continuous software engineering practices that are closely associated with increasing release frequency. The findings of the thesis increase the understanding of the phenomenon and give answers to essential questions such as what the rationale for frequent releases is and what the blueprint is for a continuous software engineering process that best supports frequent releases. The results of the thesis also offer views to the way work should be organized when dealing with the demands of high frequency releases and how the change process should be properly lead within an organization.

According to the findings of the thesis, the rationale behind increasing the release frequency in software development projects is the hope that software features and other revisions become available to end users at a quicker pace than would be otherwise possible. Theoretically, the quicker the pace of delivery and deployment of new software versions, the quicker can value be provided to users. Developers seem to appreciate the more experimental and accelerated mode of development allowed by frequent releases where feedback rapidly circles back to developers from users and monitored environments alike. An increase in release velocity drives the overall improvement of development practices and the deployment pipeline used to deliver new versions to end users. Advances in areas such as automated testing can have positive effects on product quality.

The results indicate that frequent releases in the order of days or a few weeks do not make sense in all circumstances. Environments and domains such as embedded systems, industrial automation and medical devices are notable exceptions where it is reasonable to advance with caution when software systems are updated. Software with artistic qualities like mobile games in the entertainment sector also have constraints in moving to frequent smaller releases. Regardless of the domain, making the transition to frequent releases may be difficult if the existing software development process relies heavily on manual testing or other labor that cannot be easily automated.

As shown by the studies in the thesis, there are many kinds of software release models being used in the industry. At the time when the studies were conducted, only in a few cases were the project development teams capable of rapid releases and even fewer chose to do so in practice. Releases might follow a more fixed schedule release train model or then be more sporadic with on-demand releases. What is important to note is that most observed cases had substantially higher capability to release software frequently than their actual release cycle was in reality. This could mean that in theory software development teams would be able to ship faster and are just perhaps holding on to previous well known release practices instead of adopting new ones. Of course, it is not only the development teams that need to adapt as the customers and other involved parties need to realign their ideas to releasing software rapidly. In the end, the release model that makes all parties satisfied is the correct one to choose.

Reaching a high frequency in releases requires fluidity in certain aspects of the software engineering process. Empirical observations of industry experiences described in the thesis help to create a blueprint for a continuous software engineering process, covering the whole deployment pipeline from managing requirements to deploying applications to end users. The main principle is to ensure the continuous nature of all activities in the process, be it either the continuous supply of requirements to development, continuous testing of changes or continuous and seamless deployment of releases to various environments.

Continuity in the development process is best realized when there are as few temporal gaps between finishing one activity and starting another down the line as possible. The duration of activities is also of interest since activities taking longer can delay later activities. Evidence from the thesis suggests that companies that have invested in technology and tools in various activities to potentially reduce manual labor are in a better position to deploy changes faster. More automation and tools seem to lead to the possibility of increasing release frequency. The effect is somewhat conditioned by the application domain, though. Optimization to the process can be either large or small as shown in the thesis cases. Merely optimizing the build process and switching hardware can save hours each time the code is compiled and tested after changes.

It appears that certain activities are more difficult to automate than others, even if technological maturity in a given organization is at a good level. Acceptance testing, which is meant to validate implemented functionality, is one such activity. Testing of non-functional properties such as security falls into the same category. More often than not, these activities include a human component required to either perform the activity or subjectively assess outputs in order to determine the release readiness of a set of changes.

In terms of increasing release frequency and achieving a state of continuous deployment, the last mile from staged testing environments to production environments is critical. Cloud technologies with flexible infrastructure and virtualization can enable a steady supply of changes to different environments. According to the industry experiences reported in the thesis, it is rare for companies to have projects with an automated deployment pipeline that could in one go directly provide end users with new experiences as changes in software emerge.

The essence of continuous software engineering processes is how to deal with changes. Understanding signals from end users by collecting feedback leads to perceived need to change. Any activity in the process should be aimed to keep the cycle time as quick as possible. Development teams should be dexterous enough to handle the changes rapidly and also refactor code structures when needed. The real challenge is to tailor the process so that the changes find their way to production environments and end users with sufficient quality and speed. The results from the cases show that such a process is not easy to implement to say the least, but it is possible.

In reality, making the best use of software development processes geared towards high frequency releases needs strong organizational support and a different attitude to developing and releasing software altogether. Companies need to have an understanding about which projects in their portfolio have the best fit for frequent releases. As the thesis results from maturity surveys highlight, projects in different life cycle phases have different objectives for process improvement. There is little sense in trying to transform processes in projects that are barely being maintained.

Understanding the current status of projects and setting clear objectives is part of the responsibility of management when thinking of reforming development practices across the organization. But there are other organizational matters that need to be taken into account, especially if the objective is to do frequent, almost daily, releases. Shifting to frequent releases might mean that an internal cultural revolution is needed with changed working habits and reshaped customer relationships.

Management should show leadership and take an active stance to support the change process. Employees should be given enough resources and the chance to learn new skills where needed to retrofit automated deployment pipelines to existing software development systems. A more responsive development style in which production environments are continuously monitored and development teams have a greater responsibility for infrastructure and releases requires the cooperation of many people. In the interview responses, this cultural shift was considered to be one of the issues people need to overcome. At the same time, shorter release intervals leading to smaller releases can be seen to benefit working conditions by reducing some of the anxiety commonly attributed to large releases. Developer spirits also seem to be boosted when they have enough time for refactoring and making the code a little bit better for the purpose.

By no means is choosing the release frequency and improving the software process simply an internal organization matter. Software is often done with and for customers who should have a say about which release model is used and how project resources are used. Naturally, customers might prefer feature development over software process improvement. For making informed decisions, customers should right away be given enough information about the good and bad sides of frequent software releases. Unfortunately, there are not too many distinct measures that can objectively show the benefits of increasing release frequency. There is always the risk that releasing a new version might be unsuccessful and end users are exposed to failures. That risk needs to be weighed in every case but with a functional automated deployment pipeline and a high rate of releases the fix ought to be just around the corner, too.

References

Bram Adams and Shane McIntosh. Modern Release Engineering in a Nutshell – Why Researchers Should Care. In *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 5, pages 78–90, March 2016. doi: 10.1109/SANER.2016.108.

Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh, and Kim Moir. The Practice and Future of Release Engineering: A Roundtable with Three Release Engineers. *IEEE Software*, 32(2):42–49, 2015. doi: 10.1109/MS.2015.52.

Liz Allen, Jo Scott, Amy Brand, Marjorie Hlava, and Micah Altman. Publishing: Credit Where Credit is Due. *Nature*, 508(7496):312–313, 2014.

Valentina Armenise. Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. In *Proceedings of the Third International Workshop on Release Engineering*, RELENG '15, pages 24–27, 2015. doi: 10.1109/RELENG.2015.19.

Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52, May 2016. doi: 10.1109/MS.2016.64.

Len Bass. Deployability. In Ivan Mistrik, Richard Soley, Nour Ali, John Grundy, and Bedir Tekinerdogan, editors, *Software Quality Assurance*, pages xxiii–xxvii. Morgan Kaufmann, Boston, 2016. doi: 10.1016/B978-0-12-802301-3.00019-3.

Len Bass. The Software Architect and DevOps. *IEEE Software*, 35(1):8–10, January 2018. doi: 10.1109/MS.2017.4541051.

Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

Stephany Bellomo, Neil Ernst, Robert Nord, and Rick Kazman. Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN 2014, pages 702–707, 2014. doi: 10.1109/DSN.2014.104.

Norman Blaikie. *Analyzing Quantitative Data.* SAGE Publications Ltd, 2011. doi: 10.4135/9781849208604.

Jan Bosch. *Continuous Software Engineering: An Introduction*, pages 3–13. Springer International Publishing, 2014. doi: 10.1007/978-3-319-11283-1_1.

Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How Do Centralized and Distributed Version Control Systems Impact Software Changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333. Association for Computing Machinery, 2014. doi: 10.1145/2568225.2568322.

Graham Brooks. Team Pace - Keeping Build Times Down. In *Proceedings of the Agile 2008 Conference*, pages 294–297, 2008. doi: 10.1109/Agile.2008.41.

Matt Callanan and Alexandra Spillane. DevOps: Making It Easy to Do the Right Thing. *IEEE Software*, 33(3):53–59, May 2016. doi: 10.1109/MS.2016.66.

Lianping Chen. Continuous Delivery: Overcoming Adoption Challenges. *Journal of Systems and Software*, 128:72–86, 2017. doi: 10.1016/j.jss.2017.02.013.

Gerry Gerard Claps, Richard Berntsson Svensson, and Aybüke Aurum. On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way. *Information and Software Technology*, 57(0):21 – 31, 2015. doi: 10.1016/j.infsof.2014.07.009.

CMMI Product Team. CMMI for Development, Version 1.3. Technical Report CMU/SEI-2010-TR-033, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2010. URL `http://resources.sei.cmu.edu/library/asset-view.cfm?=9661`.

Daniela S. Cruzes and Tore Dybå. Recommended Steps for Thematic Synthesis in Software Engineering. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, ESEM 2011, pages 275–284, September 2011. doi: 10.1109/ESEM.2011.36.

Brian de Alwis and Jonathan Sillito. Why Are Software Projects Moving from Centralized to Decentralized Version Control Systems? In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, CHASE 2009, pages 36–39, May 2009. doi: 10.1109/CHASE.2009.5071408.

Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. Challenges When Adopting Continuous Integration: A Case Study. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, pages 17–32. Springer International Publishing, 2014.

DIMECC. DIMECC N4S-Program: Finnish Software Companies Speeding Digital Economy, 2022. URL `https://n4s.dimecc.com/en/`. Retrieved: January 2022.

Torgeir Dingsøyr and Casper Lassenius. Emerging Themes in Agile Software Development: Introduction to the Special Section on Continuous Value Delivery. *Information and Software Technology*, 77:56–60, 2016. doi: 10.1016/j.infsof.2016.04.018.

Vincent Driessen. A Successful Git Branching Model, 2010. URL `http://nvie.com/posts/a-successful-git-branching-model/`. Retrieved: January 2022.

Erik Dörnenburg. The Path to DevOps. *IEEE Software*, 35(5):71–75, September 2018. doi: 10.1109/MS.2018.290110337.

Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, May 2016. doi: 10.1109/MS.2016.68.

Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245. Association for Computing Machinery, 2014. doi: 10.1145/2635868.2635910.

Murat Erder and Pierre Pureur. Chapter 2 - Principles of Continuous Architecture. In Murat Erder and Pierre Pureur, editors, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*, pages 21–37. Morgan Kaufmann, Boston, 2016. doi: 10.1016/B978-0-12-803284-8.00002-6.

Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013. doi: 10.1109/MIC.2013.25.

Timothy Fitz. Continuous Deployment, February 2009. URL `http://timothyfitz.com/2009/02/08/continuous-deployment/`. Retrieved: January 2022.

Brian Fitzgerald and Klaas-Jan Stol. Continuous Software Engineering and Beyond: Trends and Challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 1–9. Association for Computing Machinery, 2014. doi: 10.1145/2593812.2593813.

Brian Fitzgerald and Klaas-Jan Stol. Continuous Software Engineering: A Roadmap and Agenda. *Journal of Systems and Software*, 123:176–189, 2017. doi: https://doi.org/10.1016/j.jss.2015.06.063.

Rafaela Mantovani Fontana, Isabela Mantovani Fontana, Paula Andrea da Rosa Garbuio, Sheila Reinehr, and Andreia Malucelli. Processes Versus People: How Should Agile Software Development Maturity Be Defined? *Journal of Systems and Software*, 97:140–155, 2014. doi: 10.1016/j.jss.2014.07.030.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Martin Fowler and Matthew Foemmel. Continuous Integration (Original Version), 2000. URL `https://www.martinfowler.com/articles/originalContinuousIntegration.html`. Retrieved: January 2022.

Fabio Giorgi and Frances Paulisch. Transition Towards Continuous Delivery in the Healthcare Domain. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, ICSE-SEIP 2019, pages 253–254, 2019. doi: 10.1109/ICSE-SEIP.2019.00035.

Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 426–437. Association for Computing Machinery, 2016. doi: 10.1145/2970276.2970358.

Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 197–207. Association for Computing Machinery, 2017. doi: 10.1145/3106237.3106270.

Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010.

Jez Humble, Chris Read, and Dan North. The Deployment Production Line. In *Proceedings of the AGILE 2006 Conference*, AGILE'06, pages 118–124, July 2006. doi: 10.1109/AGILE.2006.53.

Juha Itkonen, Raoul Udd, Casper Lassenius, and Timo Lehtonen. Perceived Benefits of Adopting Continuous Delivery Practices. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16. Association for Computing Machinery, 2016. doi: 10.1145/2961111.2962627.

Teemu Karvonen, Woubshet Behutiye, Markku Oivo, and Pasi Kuvaja. Systematic Literature Review on the Impacts of Agile Release Engineering Practices. *Information and Software Technology*, 86:87–100, 2017. doi: 10.1016/j.infsof.2017.01.009.

Noureddine Kerzazi and Bram Adams. Who Needs Release and DevOps Engineers, and Why? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, CSED '16, pages 77—83. Association for Computing Machinery, 2016. doi: 10.1145/2896941.2896957.

Barbara Kitchenham and Stuart Charters. Guidelines for Performing Systematic Literature Reviews in Software Engineering, Version 2.3. Technical Report EBSE-2007-01, Keele University and University of Durham, 2007.

Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. Supporting Continuous Integration by Code-Churn Based Test Selection. In *Proceedings of the 2nd International Workshop* *on Rapid Continuous Software Engineering*, RCoSE 2015, pages 19–25, 2015. doi: 10.1109/RCoSE.2015.11.

Stephan Krusche and Bernd Bruegge. CSEPM - A Continuous Software Engineering Process Metamodel. In *Proceedings of the 2017 IEEE/ACM 3rd International Workshop on Rapid Continuous Software Engineering*, RCoSE 2017, pages 2–8, 2017. doi: 10.1109/RCoSE.2017.6.

Vincent Larivière, David Pontille, and Cassidy R. Sugimoto. Investigating the Division of Scientific Labor Using the Contributor Roles Taxonomy (CRediT). *Quantitative Science Studies*, 2:111–128, 4 2021. ISSN 26413337. doi: 10.1162/qss_a_00097.

Eero Laukkanen, Juha Itkonen, and Casper Lassenius. Problems, Causes and Solutions When Adopting Continuous Delivery—A Systematic Literature Review. *Information and Software Technology*, 82:55–79, 2017. doi: 10.1016/j.infsof.2016.10.001.

Marko Leppänen. *Vanishing Point: Where Infrastructures, Architectures, and Processes of Software Engineering Meet.* PhD thesis, Tampere University of Technology, 1 2017.

Marko Leppänen, Simo Mäkinen, Samuel Lahtinen, Outi Sievi-Korte, Antti-Pekka Tuovinen, and Tomi Männistö. Refactoring-A Shot in the Dark? *IEEE Software*, 32(6):62–70, November 2015a. ISSN 0740-7459. doi: 10.1109/MS.2015.132.

Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V. Mäntylä, and Tomi Männistö. The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72, March 2015b. doi: 10.1109/MS.2015.50.

Walid Maalej, Hans-Jörg Happel, and Asarnusch Rashid. When Users Become Collaborators: Towards Continuous and Context-Aware User Input. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 981–990. Association for Computing Machinery, 2009. doi: 10.1145/1639950.1640068.

Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Exploratory Testing of Large-Scale Systems – Testing in the Continuous Integration and Delivery Pipeline. In Michael Felderer, Daniel Méndez Fernández, Burak Turhan, Marcos Kalinowski, Federica Sarro, and Dietmar Winkler, editors, *Product-Focused Software Process Improvement*, pages 368–384. Springer International Publishing, 2017.

Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering*, 20(6):1587–1633, December 2015. doi: 10.1007/s10664-014-9324-x.

Mathias Meyer. Continuous Integration and Its Tools. *IEEE Software*, 31 (3):14–16, 2014. ISSN 07407459. doi: 10.1109/MS.2014.58.

Kıvanç Muşlu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 334–344. Association for Computing Machinery, 2014. doi: 10.1145/2568225.2568284.

Simo Mäkinen, Marko Leppänen, Terhi Kilamo, Anna-Liisa Mattila, Eero Laukkanen, Max Pagels, and Tomi Männistö. Improving the Delivery Cycle: A Multiple-Case Study of the Toolchains in Finnish Software Intensive Enterprises. *Information and Software Technology*, 80:175–194, 2016. doi: 10.1016/j.infsof.2016.09.001.

Simo Mäkinen, Timo Lehtonen, Terhi Kilamo, Mikko Puonti, Tommi Mikkonen, and Tomi Männistö. Revisiting Continuous Deployment Maturity: A Two-Year Perspective. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1810–1817. Association for Computing Machinery, 2019. doi: 10.1145/3297280.3297458.

Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Continuous Integration Impediments in Large-Scale Industry Projects. In *Proceedings of the 2017 IEEE International Conference on Software Architecture*, ICSA 2017, pages 169–178, April 2017. doi: 10.1109/ICSA.2017.11.

Steve Neely and Steve Stolt. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In *Proceedings of the 2013 Agile Conference*, pages 121–128, 2013. doi: 10.1109/AGILE.2013.17.

Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the "Stairway to Heaven" - A Mulitiple-Case study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of

Software. In Vittorio Cortellessa, Henry Muccini, and Onur Demirörs, editors, *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA 2012, pages 392–399. IEEE Computer Society, 2012. doi: 10.1109/SEAA.2012.54.

Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, Michael Stumm, Shari Whitaker, and Laurie Williams. The Top 10 Adages in Continuous Deployment. *IEEE Software*, 34(3):86–95, May 2017. doi: 10.1109/MS.2017.86.

Patrizio Pelliccione, Eric Knauss, Rogardt Heldal, S. Magnus Ågren, Piergiuseppe Mallozzi, Anders Alminger, and Daniel Borgentun. Automotive Architecture Framework: The Experience of Volvo Cars. *Journal of Systems Architecture*, 77:83–100, 2017. ISSN 1383-7621. doi: 10.1016/j.sysarc.2017.02.005.

Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *Proceedings of the 2017 IEEE/ACM 14th International Working Conference on Mining Software Repositories*, MSR 2017, pages 345–355, 2017. doi: 10.1109/MSR.2017.54.

Leah Riungu-Kalliosaari, Simo Mäkinen, Lucy Ellen Lwakatare, Juha Tiihonen, and Tomi Männistö. DevOps Adoption Benefits and Challenges in Practice: A Case Study. In Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, and Tommi Mikkonen, editors, *Product-Focused Software Process Improvement*, PROFES 2016, pages 590–597. Springer International Publishing, November 2016.

Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975. doi: 10.1109/TSE.1975.6312866.

James Roche. Adopting DevOps Practices in Quality Assurance. *Communications of the ACM*, 56(11):38–43, November 2013. doi: 10.1145/2524713.2524721.

Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study. *Journal of Systems and Software*, 123:263–291, 2017. doi: 10.1016/j.jss.2015.12.015.

Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, 14(2):131–164, 2009.

Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 21–30. Association for Computing Machinery, 2016. doi: 10.1145/2889160.2889223.

Ken Schwaber. SCRUM Development Process. In Jeff Sutherland, Cory Casanave, Joaquin Miller, Philip Patel, and Glenn Hollowell, editors, *Business Object Design and Implementation*, pages 117–134. Springer London, 1997.

William R. Shadish, Thomas D. Cook, and Donald T. Cmapbell. *Experimental and Quasi-experimental Designs for Generalized Causal Inference.* Houghton Mifflin, 2002.

Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar, and Liming Zhu. Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, EASE'17, pages 384—-393. Association for Computing Machinery, 2017. doi: 10.1145/3084226.3084263.

Barry Snyder and Bill Curtis. Using Analytics to Guide Improvement During an Agile–DevOps Transformation. *IEEE Software*, 35(1):78–83, 2018. doi: 10.1109/MS.2017.4541032.

Vlad Stirbu and Tommi Mikkonen. CompliancePal: A Tool for Supporting Practical Agile and Regulatory-Compliant Development of Medical Software. In *Proceedings of the 2020 IEEE International Conference on Software Architecture Companion*, ICSA-C 2020, pages 151–158, 2020. doi: 10.1109/ICSA-C50368.2020.00035.

Klaas-Jan Stol and Brian Fitzgerald. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology*, 27(3):1–51, September 2018. doi: 10.1145/3241743. Article 11.

Daniel Ståhl and Jan Bosch. Continuous Integration Flows. In Jan Bosch, editor, *Continuous Software Engineering*, pages 107–115. Springer International Publishing, 2014a. doi: 10.1007/978-3-319-11283-1_9.

Daniel Ståhl and Jan Bosch. Automated Software Integration Flows in Industry: A Multiple-Case Study. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 54–63. Association for Computing Machinery, 2014b. doi: 10.1145/2591062.2591186.

Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. Continuous Practices and DevOps: Beyond the Buzz, What Does it All Mean? In *Proceedings of the 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications*, SEAA 2017, pages 440–448, Aug 2017. doi: 10.1109/SEAA.2017.8114695.

Jeff Sutherland, Carsten Ruseng Jakobsen, and Kent Johnson. Scrum and CMMI Level 5: The Magic Potion for Code Warriors. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, HICSS 2008, pages 466–466, Jan 2008. doi: 10.1109/HICSS.2008.384.

Matthias Tichy, Jan Bosch, Michael Goedicke, and Magnus Larsson. Message from the Chairs. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages iii–v. Association for Computing Machinery, 2014. doi: 10.1145/2593812.

Walter F. Tichy. RCS - A System for Version Control. *Software: Practice and Experience*, 15(7):637–654, 1985. doi: 10.1002/spe.4380150703.

Christina Wallin, Fredrik Ekdahl, and Stig Larsson. Integrating Business and Software Development Models. *IEEE Software*, 19(6):28–33, 2002. doi: 10.1109/MS.2002.1049384.

Claes Wohlin and Aybüke Aurum. Towards a Decision-Making Structure for Selecting a Research Design in Empirical Software Engineering. *Empirical Software Engineering*, 20(6):1427–1455, December 2015. doi: 10.1007/s10664-014-9319-7.

Robert K Yin. *Case Study Research: Design and Methods.* Sage publications, fifth edition, 2014.

Liming Zhu, Len Bass, and George Champlin-Scharff. DevOps and Its Practices. *IEEE Software*, 33(3):32–34, May 2016. doi: 10.1109/MS.2016.81.