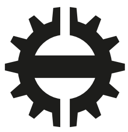


TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Marko Leppänen

Vanishing Point: Where Infrastructures, Architectures, and Processes of Software Engineering Meet

Julkaisu 1452 • Publication 1452

Tampere 2017

Tampereen teknillinen yliopisto. Julkaisu 1452
Tampere University of Technology. Publication 1452

Marko Leppänen

Vanishing Point: Where Infrastructures, Architectures, and Processes of Software Engineering Meet

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 27th of January 2017, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2017

ISBN 978-952-15-3898-8 (printed)
ISBN 978-952-15-3900-8 (PDF)
ISSN 1459-2045

Abstract

Leppänen, Marko

Vanishing Point: Where Infrastructures, Architectures, and Processes of Software Engineering Meet

Keywords: software engineering, agile software development, continuous deployment, continuous delivery, software architecture

In software project management, there exists a triangle-like relation connecting the required time to deliver a certain scope of software features with a certain cost. If one of these three is affected, others must compensate for this. For example, having a faster delivery means either a costlier product or less features, or both. Long delivery times are usually unacceptable in any case as the business environment is changing fast.

To deal with this, contemporary software is mostly produced with Agile methods, which emphasise developing small increments to deliver constant stream of value to the customer. A small piece of software is easier to produce and test. Also, rapid feedback can be gained with tight co-operation with the customer. As the increments have become almost infinitesimally small, the working software can be constantly improved as the changes can be delivered to the end user almost instantly. However, this is only possible when the increments are reliably tested and the delivery itself is rapid. Thus, automation in these crucial parts is a must. Furthermore, the customer is not able to comment on every change in person, so the collection of the feedback must be automated. Furthermore, the software product itself has to support the continuous delivery. There exists a certain relation between these aspects–namely the tool infrastructure, processes and the architecture—reminiscent of the project triangle of time, cost, and scope.

In this thesis, we examine the crucial properties these aspects in the context of increasing the speed of delivery–up to continuous delivery and deployment combined with the idea of continuous feedback. Also, the ramifications of rapid software delivery are studied. The research is carried out as interviews and related methods, such as surveys, to gain data from the companies involved in software development. Also, some quantitative analysis is used to back up the findings. As a result, a model is introduced based on the research. It can be used to explore the aspects and their interrelationships. We present a set of key enablers of increasing the delivery speed and present a set of side-effects that have to be considered. These can be used as a guideline in the companies which are striving to hasten their delivery pace. Additionally, a comparison of various companies based on their delivery speed is presented.

Preface

The title of the thesis refers to the point in a picture where parallel lines in space meet. It is a metaphor for the trichotomy between software architecture, process and infrastructure, which I hope to fade out a little bit in this thesis. However, it also carries a threefold meaning. As a researcher, I have travelled a long way to this thesis and the focus has been missing at times. Here I wish to thank all the people who have put the show back on the road. The title also reflects a bit of the road movie aesthetics I feel being a suitable tribute to the Need4Speed project which was the incubator of the thesis.

First I wish to thank my supervisor, Tommi Mikkonen, for not only commenting and supervising this work, but giving me the opportunity to join forces with such a great research team in Need4Speed project. I also wish to thank my earlier supervisors, Kari Systä, Kai Koskimies and the late Ilkka Haikala. A huge thank you to Kai for exposing me to the patterns and the team, which I wish express my gratitude next; the co-authors of the patterns book, Veli-Pekka Eloranta, Johannes Koskinen and Ville Reijonen. The discussions with you guys have been invaluable source of insight and this second book has been much easier effort after cutting my teeth with the patterns book.

Additionally, I would like to thank co-authors of my publications so far, especially Simo Mäkinen from Helsinki University. Without the collaboration and support from all of you, this thesis would not have been completed in such a short time. I am deeply thankful of your insights. Also, I would like to thank all my colleagues at the Tampere University of Technology for their support and comments during the research. The grants from Nokia Foundation have helped me financially through the writing of the thesis. In addition, my pre-examiners professor Jürgen Münch and Dr. Apostolos Ampatzoglou deserve much credit for improving the last iterations.

I also would like to thank my friends for encouraging me during the months it took to write the thesis. Your help has been indispensable. I would like to give a special thanks to Teemu Sundell for his comments and reviews of this work. Finally, I express my gratitude to my parents for their life-long support of my studies.

Nokia 19.12. 2016

Marko Leppänen

Contents

Abstract

Preface

Contents

Terms and Definitions

List of figures

List of tables

List of publications

1 INTRODUCTION 10

1.1 Motivation 12

1.2 Approach and Research Questions 13

1.3 Research Overview 14

1.4 Research Method 15

1.5 Thesis Contributions 19

1.6 Structure of This Thesis 20

2 BACKGROUND 21

2.1 Software Development Processes 22

2.2 Software Development Tools and Infrastructure 25

2.3 Software Architecture 26

2.4 Continuous * 28

3 COMPONENTS OF CONTEMPORARY SOFTWARE DEVELOPMENT 33

3.1 Architecture-Process relation 35

3.2 Process-Infrastructure relation 39
3.3 Infrastructure-Architecture relation 42
3.4 Impact of speed 45
4 CLOSURE 46
4.1 Research questions revisited 46
4.2 Related research 53
4.2.1 DevOps 53
4.2.2 SEMAT 54
4.2.3 Post-Agility 54
4.2.4 ChatOps 55
4.2.5 Continuous Deployment and Architecting 55
4.3 Limitations of the research 56
4.4 Future work 57
4.5 Introduction to included publications 58
4.6 Concluding remarks 60
REFERENCES 62

List of Figures

Figure 1 Continuous Delivery process as a pipeline. The developer does three check ins to the version control, gets the feedback from each of them, and the final one gets promoted to a release. Adapted from [4]. 11

Figure 2 Left side: The waterfall model, adapted from Royce [15]. Right side: the Scrum process, simplified from Scrum Inc. [16]. 13

Figure 3 Research timeline. The diamonds represent the research activities, while the boxes represent the publications based on them. 15

Figure 4 The PAI model—the components of software development. Infrastructure, architecture and process interact with each other related to the speed of development. 22

Figure 5 Elements of modern software development, adapted from Publication VI. 23

Figure 6 An example of a software process with elements of Continuous delivery and rapid feedback. The tools for each phase are also shown (Publication II). 31

Figure 7 An example of a Continuous Delivery maturity model, redrawn from Rehn et al. [11]. The model consists of five maturity levels on five different aspects of the software development, forming a matrix of 25 different possible combinations of maturity. 33

Figure 8 The publications included in this thesis and their rough relation to the elements of software development.. 35

Figure 9 An example from Publication V showing how refactoring is carried out in a software development process.. 38

Figure 10 The process presented in Publication II. The company uses proof-of-concept to measure end users if an idea has any business value. 40

Figure 11 A “social developer” requires a mature set of tools to get feedback and information from all relevant stakeholders and to monitor the development pipeline. .. 41

Figure 12 The overview of the tools that were found to be used in Publication VI. The development infrastructure consists of a selection of these tools, forming an architecture of its own. 44

Figure 13 A Kanban board and radiators as means of visual control. 50

List of Publications

This thesis consists of a summary part and the following original publications:

I. Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T. 2015. The Highways and Country Roads to Continuous Deployment, In: Software, IEEE, vol. 32, no. 2, pp. 64-72, Mar.-Apr. 2015, doi:10.1109/MS.2015.50

II. Leppänen, M., Kilamo, T., & Mikkonen, T. 2015. Towards Post-Agile Development Practices through Productized Development Infrastructure. In: Rapid Continuous Software Engineering (RCoSE), 2015 IEEE/ACM 2nd International Workshop on. pp. 34-40. IEEE. Florence, Italy. 23-23 May 2015. doi:10.1109/RCoSE.2015.14

III. Kilamo, T., Leppänen, M., & Mikkonen, T. 2015. The Social Developer: Now, Then, and Tomorrow. In: SSE 2015 Proceedings of the 7th International Workshop on Social Software Engineering, 2015. pp. 41-48. ACM, New York, NY, USA. ISBN: 978-1-4503-3818-9 doi:10.1145/2804381.2804388

IV. Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte, O. Tuovinen, A.-P., & Männistö, T. 2015. Refactoring–a Shot in the Dark? In: Software, IEEE, vol. 32, no. 6, pp. 62-70, Nov.-Dec. 2015 doi:10.1109/MS.2015.132

V. Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., & Lehtonen, T. 2015. Decision-Making Framework for Refactoring. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), October 2, 2015, Bremen, Germany. IEEE. pp. 61-68. doi:10.1109/MTD.2015.7332627

VI. Mäkinen, S., Leppänen, M., Kilamo, T., Mattila, A.-L., Laukkanen, E., Pagels, M., & Männistö, T. 2016 Improving the Delivery Cycle: A Multiple-Case Study of the Toolchains in Finnish Software Intensive Enterprises. In: Information and Software Technology, Vol. 80, 01.12.2016, pp. 1339-1351. doi:10.1016/j.infsof.2016.09.001

The publications are reproduced in the thesis in accordance of the publication permission schemes of the publishers. The main contributions of the included publications follow later in section 4.5. Detailed description of the role of the candidate per publication is discussed in the following:

In **Publication I**, the candidate planned, conducted and analysed the interviews material together with second, third, fourth, fifth and sixth authors. The seventh author contributed

in analysis and writing parts of the inception of the publication. The candidate conducted significant part of the interviews with the fourth author and had a leading role in the writing process.

In **Publication II**, building on the material from Publication I, the candidate planned, and conducted the interview with the help of the second author. In the analysis and writing process, the candidate held the main responsibility. The third author had contribution in the analysis and writing parts of the article's inception.

In **Publication III**, the candidate planned and carried out the survey with the first author with evenly distributed responsibility. The analysis and writing part were carried out with the help of the third author.

In **Publication IV**, the candidate planned and conducted the interviews and analysed the resulting material together with second, third, fourth and fifth authors. The sixth author contributed in analysis and writing parts of the inception of the publication. The candidate conducted half of the interviews with the third and fourth authors and had a leading role in the writing process.

In **Publication V**, building on Publication IV, the candidate planned the particularizing interview with the other authors. In the writing process, the candidate had a leading role. The third author had a strong contribution in writing the method parts of the article.

In **Publication VI**, an article building on the interview data from Publication I, the interviews were planned and conducted by the first and sixth author along with the candidate. The first author led the analysis contributed significantly to the interviews and to the writing of the article. The candidate had conducted the major part of the interviews and contributed in the analysis of the results. The fourth author contributed in the statistical analysis of the material. The third, fifth and the seventh authors contributed in the writing the article and analysis of the results.

1 Introduction

In the field of software engineering, there has been a growing interest in the practices of Continuous Delivery, and pushing the concept even further to Continuous Deployment. Continuous Delivery refers to “a software development discipline where you build software in such a way that the software can be released to production at any time” [1] with priority on the keeping the software deployable all the time, and Continuous Deployment requires all these changes to be actually deployed immediately and automatically. These practices aim at producing valuable software in short cycles and at ensuring that the software can be reliably released at any time [2]. These approaches boast several benefits, the most obvious one being the shorter time to market. Also, the problems caused by a separate integration and deployment phases in the software process are removed, as integration is constantly verified by automated build and tests [3]. The practice of Continuous Integration has been around for a while already, and is well-known in the field of software engineering. These Continuous Delivery and Deployment practices together enable the software development organizations to add (and even remove) features with as small overhead as possible. From this ability stems also one valuable, but not so evident, benefit – the ability to experimentally determine the value of software components, with real users [4]. Some companies, such as Amazon, Google, and Facebook, utilize this by delivering new versions of the software frequently and in a timely fashion—even several times per day [5].

While these practises have gained prominence only lately, they are not completely new ideas, as fast delivery and Continuous Integration have been in the core of the Agile approaches for several years. Now, the potential release pace has increased to such degree that the whole organization must have a holistic approach to the software development. The development requires much supporting structure to enable the fast pace. Firstly, the quickened release cycle sets requirements to the organization itself and its processes. Secondly, it also requires that the tools and infrastructure are well honed to their purpose of fast software delivery and deployment. This is called as *deployment pipeline* by Humble and Farley [4]. An example of this can be seen in **Figure 1**, where each commit to version control system triggers build and unit tests for this change. The developer gets feedback, and, if successful, the change is promoted to automated acceptance tests. The developer gets the feedback from the tests. If the change is approved in the automated acceptance tests, user acceptance tests follow. If these are also approved, the change is released. Naturally, the developer also gets the feedback on the matter in question also.

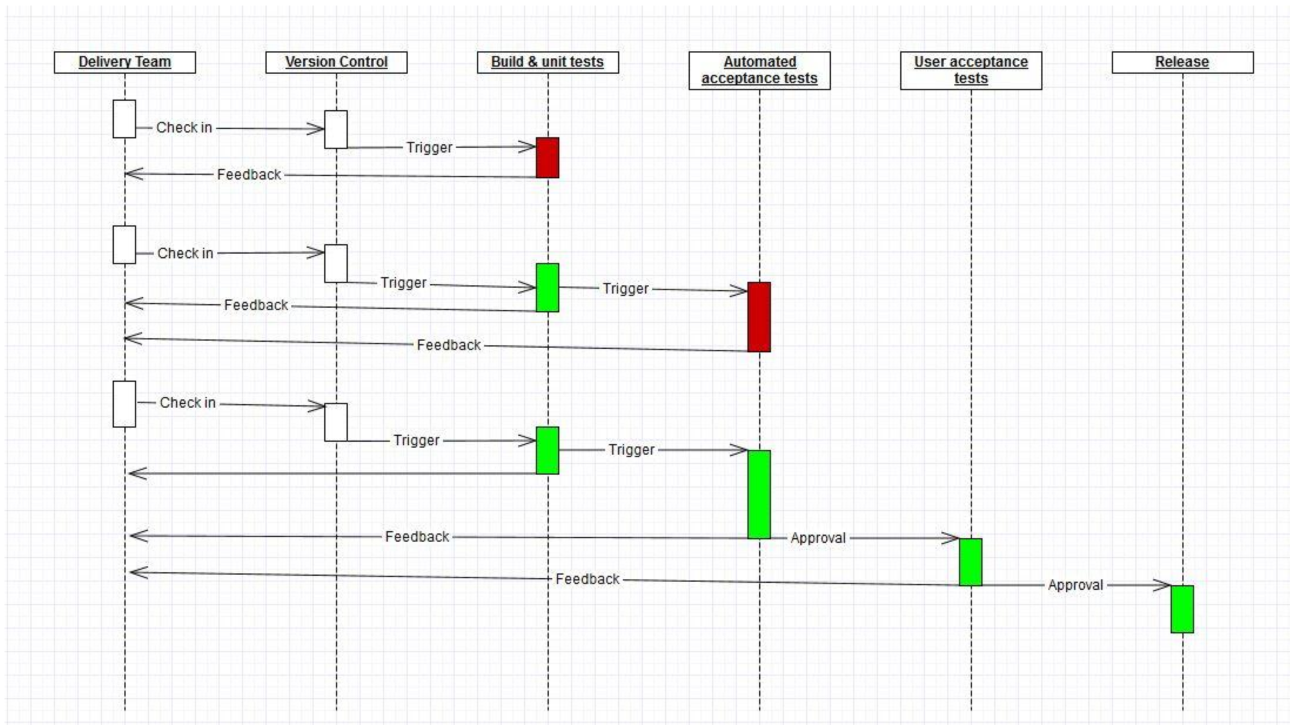


Figure 1 Continuous Delivery process as a pipeline. The developer does three check ins to the version control, gets the feedback from each of them, and the final one gets promoted to a release. Adapted from [4].

These two aspects, processes and infrastructure, are usually seen as an integral part of the popular Development and Operations (DevOps) approach, which is a movement to bridge the gap between development and operations teams [6]. In essence, DevOps promotes “Continuous *” (read, “Continuous star”) with continuous integration, continuous deployment, continuous delivery, continuous testing and so on, even spanning to continuous business side [7]. Taking another perspective, also the software itself has to have certain properties, manifested in architectural decisions, which support rapid software development. For example, the well-known practice of separation of concerns allows the software organization to develop and even deliver smaller parts easily without affecting the rest of the system. [8].

However, many organizations struggle with the adaptation of Continuous Deployment practices. Fast delivery requires that no extra hassle is spent in development itself, and thus more work is needed in these supporting elements; including process, infrastructure, and software architecture. Analogously with other software process improvements, it is easier to have piecemeal improvements than to make a radical change. This resonates with the *kaizen* principle (continuous improvement) familiar from the Lean literature, for example [9], as the preferred method to *kaikaku* (radical change). Even though kaikaku has its applications, it may cause disturbance to the productivity which drops so badly that the organization effectively grinds to the halt. The work spent on honing the infrastructure and the tools also resonate with Lean manufacturing idea of 5S, where the workspace must be well-organized before the production starts, and no work wasted in things that do not bring any value to

the production itself [10]. Because of this required work, the organization cannot just leap to full Continuous Deployment and DevOps, but usually matures parallel on several aspects, reflected by several Continuous Deployment, or DevOps maturity models [11] [12] [13], and to some degree, so called Stairway to Heaven model [14].

Although three aspects of software development, processes, infrastructures, and architecture, are well studied and understood separately in the research community, there is not much literature about how these behave together under the modern rapid software development. This thesis introduces elements that tie Continuous Deployment, software processes, development and deployment infrastructures, and software architecture together by approaching the topic with a series of studies in software companies, who apply rapid software development practices with various degrees of maturity. These studies deal the three aspects in parallel, but delving deeper into each subject sequentially.

1.1 Motivation

This thesis aims at describing how contemporary software development is carried out in the search of delivery speed and quick feedback. After the initial failures with linear-sequential 'waterfall' approaches, modern software is developed in an iterative way. The development is carried out in cycles, where a piece of software has been developed according to some requirements and delivered to the end users. At the same time, there has been a trend to get faster feedback if the produced piece of software really conforms to its requirements. This is done by shortening the cycle time by delivering ever smaller and smaller changes, so-called increments all the way to the end users (**Figure 2**). In the waterfall approach, even with the iterations between steps, the feedback is slow, as the size of the software delivered to the next step is can be large compared to the small increments of Agile. So, with large change sets, the system can be hard to integrate, and found errors may be hard to pinpoint and to correct. All this takes extra time.

In reality, the iterations are not confined to successive steps, so Royce recommended doing the system twice [15]. However, getting the feedback may still take months and relies completely on the only prototype. Instead, with the help of Continuous Integration practice, the popular Scrum framework requires that every sprint produces an increment to the software system. This shortens the feedback cycle to maximum of weeks. However, in the Continuous Delivery and Deployment, it is possible to see the effects of a new functionality as soon as it is done, including the testing, and deployed. However, achieving this degree of delivery speed is not trivial.

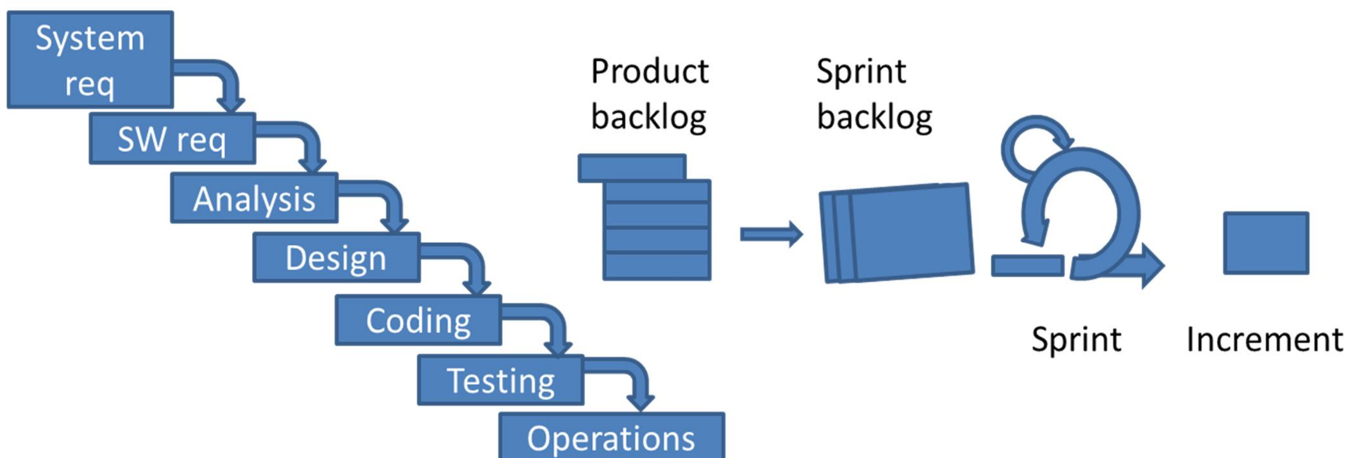


Figure 2 Left side: The waterfall model, adapted from Royce [15]. Right side: the Scrum process, simplified from Scrum Inc. [16].

In this thesis, we study how the fundamental parts of software development affect each other as the delivery speed and feedback is hastened with Continuous * practices. The results will help different stakeholders to understand what to take into account when quickening the delivery cycle and to what aspects of the software development they should focus on in their pursuit for Continuous * practices.

1.2 Approach and Research Questions

The main goal of this thesis is to study how contemporary software development is carried out in the current trend of increasing the delivery speed and to shed light on how the different activities of software development are affected under the increasing speed. Thus, the main research questions of this thesis are:

RQ1: What are the key enablers to increase the delivery speed?

We were interested in how the companies would increase their delivery speed and what obstacles they have to remove or had removed in order to deliver more rapidly. This includes both the experiences gained from the more mature companies and the key issues that the companies have to tackle.

RQ2: What are the ramifications of increasing the delivery speed?

This question was set to find out if increasing the delivery speed causes any other effects on the company than having the requirements from RQ1 in place. This includes any benefits of the increased delivery speed and side-effects on the business and the software itself.

RQ3: What is the state-of-the-practice regarding the speed of delivery practices in the companies?

This question is the place to see how the companies compare to each other in the context of the delivery speed.

Based on the answers to these questions, we embark to build a descriptive model to explain the phenomena related to the Continuous * practices and their effect on the software development under the pressure of rapid deployment pace.

1.3 Research Overview

The research started in 2014, with an effort to chart the state-of-the-art in software development in companies. The focus was on the speed of delivering software to the customer, and what benefits and obstacles the companies encountered while trying to improve their delivery speed with adaptation of Continuous * practices. The timeline of the research is presented in **Figure 3**. The chosen approach was interviews, so we started by selecting interviewees from 15 Finnish software companies. As a result of this first interview round, companies were classified according to their delivery speed, and problems and enablers to CD were identified. These were mostly related to software practices and processes. This analysis was published as Publication I, “The Highways and Country Roads to Continuous Deployment” [PI].

Moreover, one of the companies provided an interesting case to describe a contemporary software development process with rapid delivery, involving the customer company. We conducted an interview with a representative of the software company. This descriptive case study was published as a separate article, Publication II, “Towards Post-Agile Development Practices through Productized Development Infrastructure” [PII]. The study describes an example of so-called “post-agile development” where tools, or the whole development infrastructure, and processes, or practices, are an important part of the development.

After this publication, analysis with the interview data acquired for Publication I was continued with a focus on the tool infrastructure. This new study includes four additional cases, but excludes one previous case because it lacked detailed tooling. Thus, the number of companies interviewed for this study totalled as 18. All this data was analysed regarding the tools used in the software development in these organizations. The interview data formed the basis for Publication VI [PVI], which is currently under review for journal publication.

As a follow-up, we decided to have a look at the software system itself and focused on architectural issues. In a change-focused approach, we chose refactoring as the focal point as it is a way to make changes to the software system architecture while following Agile practices. Constant change without heavy preplanning is enabled by making only small changes to the architecture at time. For this research, we interviewed 12 practitioners on how they see the role and importance of refactoring, and how and when they refactor. Related to the delivery speed, its effects on refactoring practices

were also studied in Publication IV [PIV]. Furthermore, a decision-making framework for refactoring was published as Publication V [PV]. This study was based on the previous data extended with three more in-depth cases. The architectural work is still continued in analysing software systems and their properties that are related to rapid software development.

To get a deeper understanding in the organizational side, a survey was used to examine how software engineering related knowledge is spread in organizations. The 25 responses were analysed on what tools and methods the practitioners felt as important in sharing their knowledge. The results formed the backbone of Publication III [PIII]. The overview of the research timeline with the research activities and the publications is shown in **Figure 3**.

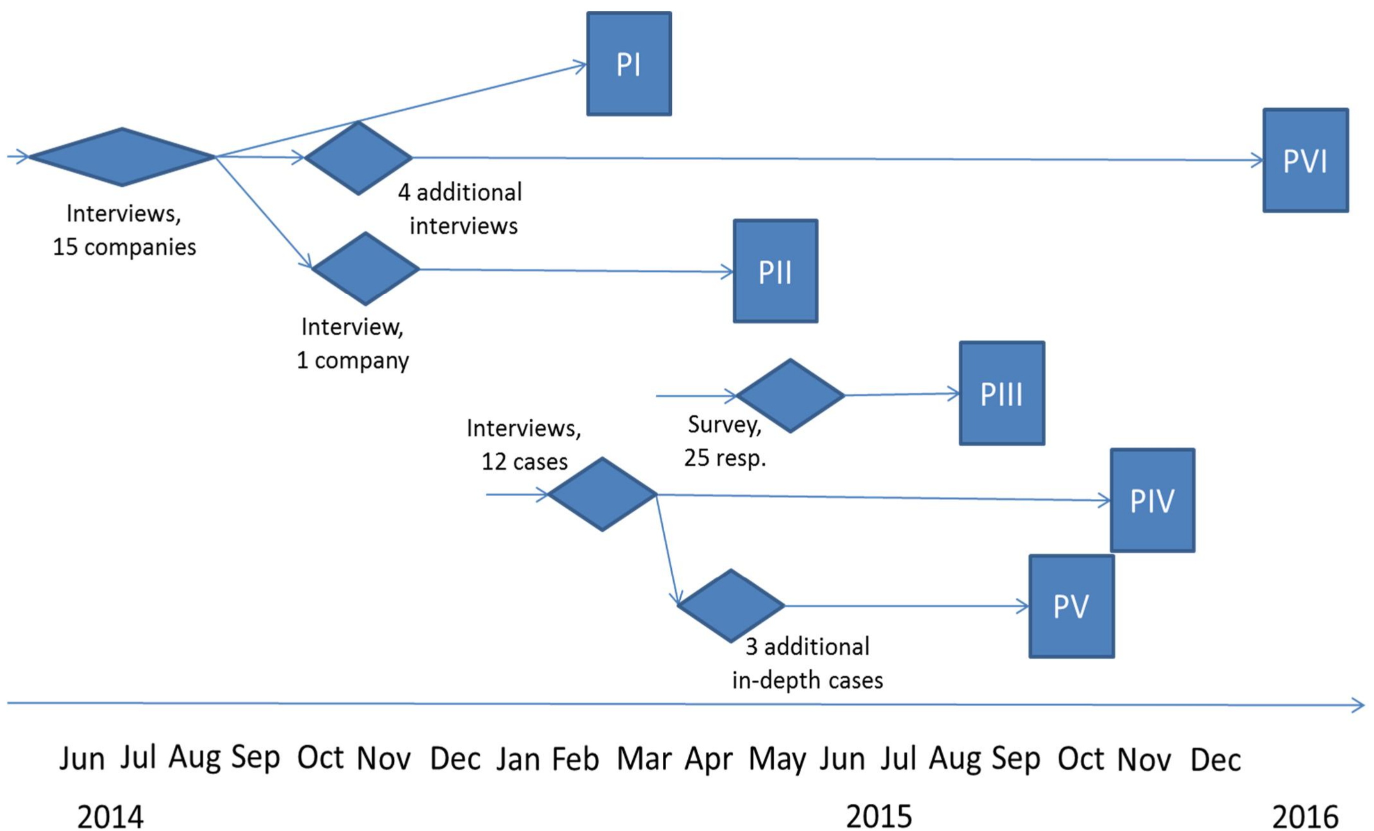


Figure 3 Research timeline. The diamonds represent the research activities, while the boxes represent the publications based on them.

1.4 Research Method

As a starting point for the research design we assessed the current situation of the research related to the research questions posed for this thesis. It was found out that the research topic cross-cuts

several areas of software engineering, and no sufficiently satisfactory model of the interactions between these areas have yet been introduced. As such, the research relies on inductive reasoning on theory-building [17]. Inductive research starts from the specific observations made by researcher, where patterns may be observed. These patterns lead to tentative hypotheses, which then can be further strengthened into a theory.

As the studied phenomena are complex in the nature, it was important to get a holistic view of the situation. From the very beginning, it was clear that there is a vast amount of information available in the companies. This kind of research is at best carried out in the realistic setting, where the researchers have little or no authority to control or affect the studied phenomena. Thus, the selected multiple-case studies as the main research methodology, because of its suitability in studying software engineering related problems. Furthermore, it can be used in building theories from case study research which aims to create constructs, such as models, from case-based, empirical evidence [18].

Most of the studied phenomena are complicated and difficult to quantify, and so interviews were chosen as a research instrument to delve into the rich source of information. As the aim was to explore the current situation regarding the infrastructure in the companies and to provide deep understanding of the studied phenomena, the interviews consisted of open-ended questions. As the guidelines for these case studies was selected the ones published by Runeson and Höst [19]. The case selection was somewhat convenience-based, as the studies were carried out in the context of Need4Speed project (see http://www.n4s.fi/en), which is an industry-led Finnish information and communications technology research program focusing on fast deployment. In addition, other partner companies were selected in order to increase the variety of the company domains involved. From the selected companies, we asked to provide us representatives who had in-depth knowledge of the researched topic. These representatives were then interviewed by the researchers. The interviews were recorded and the researchers also made notes during the interview.

The interviews were transcribed and thematic analysis was used to find the areas of interest and common themes in the answers. Thematic analysis can be used as a tool for data analysis. The main advantage of thematic analysis for our means were its suitability for several researchers and large data sets. In addition, it allowed us to broaden our own comprehension of the studied phenomena [20]. Furthermore, it allowed us to find the relevant categories from the data without any predetermined model or framework [21].

The interviews formed the basis for Publications I and VI, and as such, were meant to be more hypothesis-generating than confirmative and explorative than descriptive. One case turned out to be so interesting that we conducted a semi-structured interview with a company representative to get a descriptive case study published as Publication PII.

As a separate line of research, the architectures of these software systems were studied by conducting semi-structured interviews with the software architects. The rationale for this approach is the

same as with the infrastructures, the phenomenon is complex and an overview on the current state of the art was the desired results. Here the aim was to have as many cases as possible, so collaborating companies from the N4S project were used as the foundation of the sample for the cases. In this way, an easy access to several companies was acquired. To make the sample richer, it was extended with other partner companies known by the researchers. Yet again, thematic analysis was used on the transcribed results of the interviews. The method was to identify common themes from the transcripts and to annotate the respective sections of text with specific keywords. These were analysed further to form a description of the situation in the case companies. The results were reported in Publication IV. To broaden the understanding, and to form a more explanatory view on the phenomenon of refactoring, we decided to select three cases for more in-depth study. In this study, we selected three companies for an interview where we applied the Goals, Operators, Methods, and Selection rules (GOMS) [22] approach to task analysis. The results of the interviews were furthermore strengthened by analysing data from the company version control system. This helped us to triangulate the interview with the data about how the development effort proceeded and how refactoring operations were present in the version control system. The results were published in Publication V.

Furthermore, we set out to study the state-of-practice development set-ups in industry in the context of the concept of social developer. In this research we executed a personal opinion survey extended with some questionnaire-like open ended questions. The idea of a cross sectional survey [23] was to collect data how the developers communicate in the software companies. The survey consisted of background questions and multiple choice questions addressing the communication methods used in the company. In addition, a Likert scale [24] was used to measure how the respondents felt that particular method suited them and an open ended question about improving their communication. The study was disseminated to respondents by email, by direct messaging, and by face to face discussions. These first degree companies were Finnish software intensive companies. However, the companies were allowed to distribute the survey as they saw fit and forward it to their partners. From the acquired 25 responses, the results were analysed and published as Publication III.

So, as the included publications are based on case studies and, mostly, qualitative semi-structured interviews, there are several caveats for the generalizability of the results. In addition, the results are more hypothesis-generating in nature than confirmatory. We are hoping that this line of research will help a relevant theory to emerge, but now, it can only serve as a basis for this kind of future research. However, applicable means against biases have been taken in the studies.

In the selection of the case studies, convenience sampling has been used, as it is challenging to contact uninterested companies for interviews. However, the sheer amount of the companies interviewed and the endeavour to select companies with different domains and different perceived maturity in their software practices should give a more in-depth view on the studied phenomena. In addition, the selection criteria of the cases included an effort to sample companies with different

sizes, domains, and degrees of maturity regarding their delivery speed. Thus, it can be said that the sampling is not random, but theory-based.

The interviews have been designed and carried out by at least two researchers, and the interviews have been recorded and transcribed for the analysis. All thematic analyses have been carried out with at least two researchers, with a third person helping in the ambiguous themes. All this should decrease the amount of researcher bias, as no single prejudice has dominated the study setup.

To summarize the research design for this thesis, we use the structure by Wohlin and Aurum [25] to present the key characteristics of the conducted research.

- Strategy phase
    - The research outcome is basic research instead of applied research, as we aim to understand the problem rather than to propose a solution [26].
    - As the research logic, we use inductive approach instead of deductive logic. As the research field is relatively new, there is no existing theory to test. Rather, the thesis contributes to theory-building.
    - The purpose of research is exploratory, as there is not much existing information available in the research topic. However, in some of the publications, the research questions aim to describe the explored phenomena more deeply, being descriptive research.
    - The used research approach is falls into interpretivist category, as the studied phenomena often involve human activities in a specific situation [27]. Positivist approach would be hard in a setting where identification of quantifiable variables, and construction of testable hypotheses, and the identification of the interferences and causal reasons is difficult.
- Tactical phase
    - The research process can be characterized as a mixed approach, because publications mainly build on qualitative data, but are backed up by quantitative methods (for example, the survey analysis in Publication III and the statistical approach to tooling data in Publication IV).
    - The selected main research methodology is case study to produce thicker data and to understand more deeply individual cases than while using surveys.

- Operational phase
    - o The primary data collection method is interviews, because it is the most feasible option to study multiple cases.
    - o The selected main data analysis method is thematic analysis, as it suits well for multiple researchers and is well suited for large sets of data. However, some publications use also statistical analysis to triangulate the perception acquired from the interviews.

As such, the thesis is aimed only to explore and describe the phenomena related to the studied topic, and does not aim to full theoretical saturation. Thus, the theory building is left on the level of constructing the proposed model of the effects of delivery speed, presenting the patterns found in the data in relation with the constructed model, and comparing these findings with existing literature.

1.5 Thesis Contributions

The thesis examines what effects increasing delivery pace has to software engineering activities.qw The chosen model is to divide the contemporary software engineering activity into three aspects, namely the software processes, the development infrastructure, and the structure of the produced artefact itself, the software architecture. Thus, the model is called a PAI (Process-Architecture-Infrastructure) model.

This PAI model is then used to describe what the prerequisites for increasing the delivery speed of the software are and what the consequences of increasing the speed are. Furthermore, the current state-of-the-practice in the Finnish software companies was studied in the light of their delivery speed.

The main contributions of this thesis thus includes

- A description and an analysis of the common actions and obstacles to increase the delivery speed in software engineering.
- A description and an analysis of the benefits and side effects of increasing the delivery speed.
- A description of the situation in the Finnish software intensive companies regarding the delivery speed of the software.

As an additional contribution, a model helping to understand the relations between the actions, obstacles, benefits, and side effects is constructed. Thus, the contributions of this thesis can be used in companies planning their actions to achieve faster delivery they strive for.

1.6 Structure of This Thesis

This thesis is organized as follows. First, an introduction to the thesis topic matter was given. Next, in chapter 2, the background information on the cornerstones of the research is given. First, general background information is given, and then a more detailed explanation on the software processes, infrastructures and tools, and, finally, software architectures follows. These are followed with a view into Continuous * practices, which acts as the context where the aforementioned aspects of software engineering are studied.

In chapter 3, the main contribution of the thesis is presented. The structure of this chapter follows the structure where every component of software development presented in the previous chapter is covered so that its effect on the other components is discussed in the light of the results of the publications. Finally, the impact of speed is addressed.

In chapter 4, the closure of the thesis, some words about related research are given. In addition, the research questions presented in the introduction are revisited. Also, future work is discussed, and some conclusions regarding the research are presented. Finally, the included publications are introduced briefly.

2 Background

Software development can be seen as a human activity to deliver software artefacts. To achieve this, one needs a working software development organization with functional processes, armed with at least the essential toolset to produce the software. The connection between these has been stated by Fuggetta as

> "A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product." [28]

Fuggetta uses the process as an umbrella term for both organizational and infrastructural parts, but here we will keep them separated, and use the name process only for the organizational parts. In addition to these, the software itself forms an important aspect. These three aspects of the software development are usually seen as orthogonal to each other. However, now, as the benefits of the increasing of the delivery speed of the software, backed with the continuous delivery, have been recognized widely in the industry, the aspects have become parts of a greater system, where they have a profound effect on each other.

To get a better understanding of the phenomena which are related to the effects of increasing the delivery speed, a PAI model presented in **Figure 4** was chosen as the way of dividing and analysing the software engineering under the requirement of extreme delivery speed. The model is called PAI (Process-Architecture-Infrastructure) model after the sides of the triangle in the figure. The different aspects of the software development (Process-Architecture-Infrastructure) form the sides of a triangle, and every aspect has an effect on the other aspects, depicted by the arrows forming the sides. The research hypothesis is that these aspects direct some forces on the other aspects, such as having a well-defined process of managing the architectural changes is helped with having a mandatory review, launched by the Continuous Integration server and supported by a review tool, in the deployment pipeline. Furthermore, it seems that these forces become more significant when the delivery speed is increased. A single delivered piece of software becomes smaller as the speed increases, but the ramifications on the software development as a whole becomes bigger and bigger.

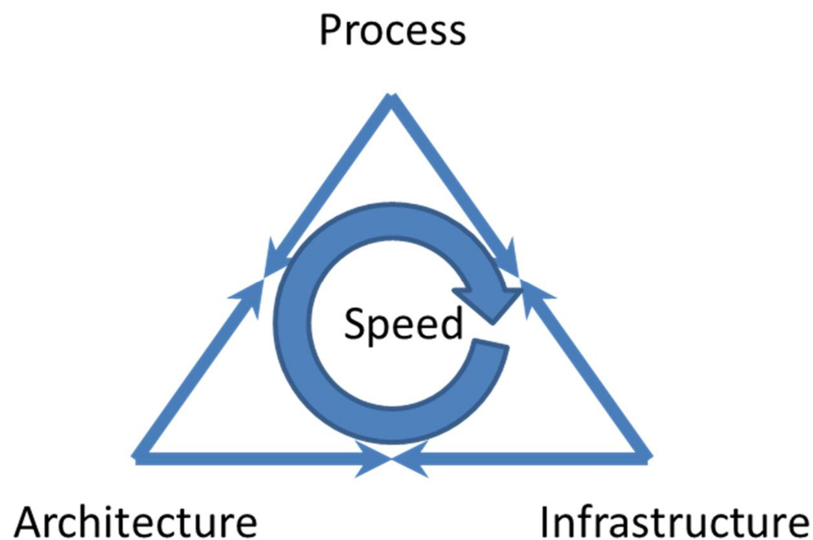


Figure 4 The PAI model—the components of software development. Infrastructure, architecture and process interact with each other related to the speed of development.

In the following sections, a brief introduction on each of these key aspects of the software engineering is presented, along with ideas related to the trend of increasing the delivery speed.

2.1 Software Development Processes

Software business processes build on organization structures, like any other business processes – in so much as any process must be embedded in an organization [29]. Software development as a process aims to deliver a software system, which conforms to a certain set of requirements. These requirements, of course, may be vague, unknown, or may even change as time passes.

In general, the software development has gone through several game-changing innovations since its inception as an established practise in 1968 NATO software engineering conference [30]. First, as software processes started to emerge as a separate activity; a waterfall approach was used as it was seen as the best approximation of an ideal and rational software process [31]. As the size of the systems developed was, and still is, continuously growing, it was soon noticed that heavy design, planning, and documentation led to the situation where the world was changing quicker than the designers could capture the requirements for the software. Thus, more agile approaches mostly superseded the waterfallish, linear model processes. However, the systems developed have grown to always push the boundaries what the development organizations and technologies are able to produce. For the second millennium, these Agile methodologies have gained popularity. They build on a set of principles, published as a part of the Agile manifesto [32].

Most of these principles are present in the Lean software development (LSD), which has been defined as “the application of the principles of the Toyota Product Development System to software development” [33]. In the seven principles of LSD, the focus is on the elimination of waste, making decisions on the last responsible moment, fast delivery, and empowered teams. In addition, amplified learning, integral experience of the customer, and holistic approach are encouraged.

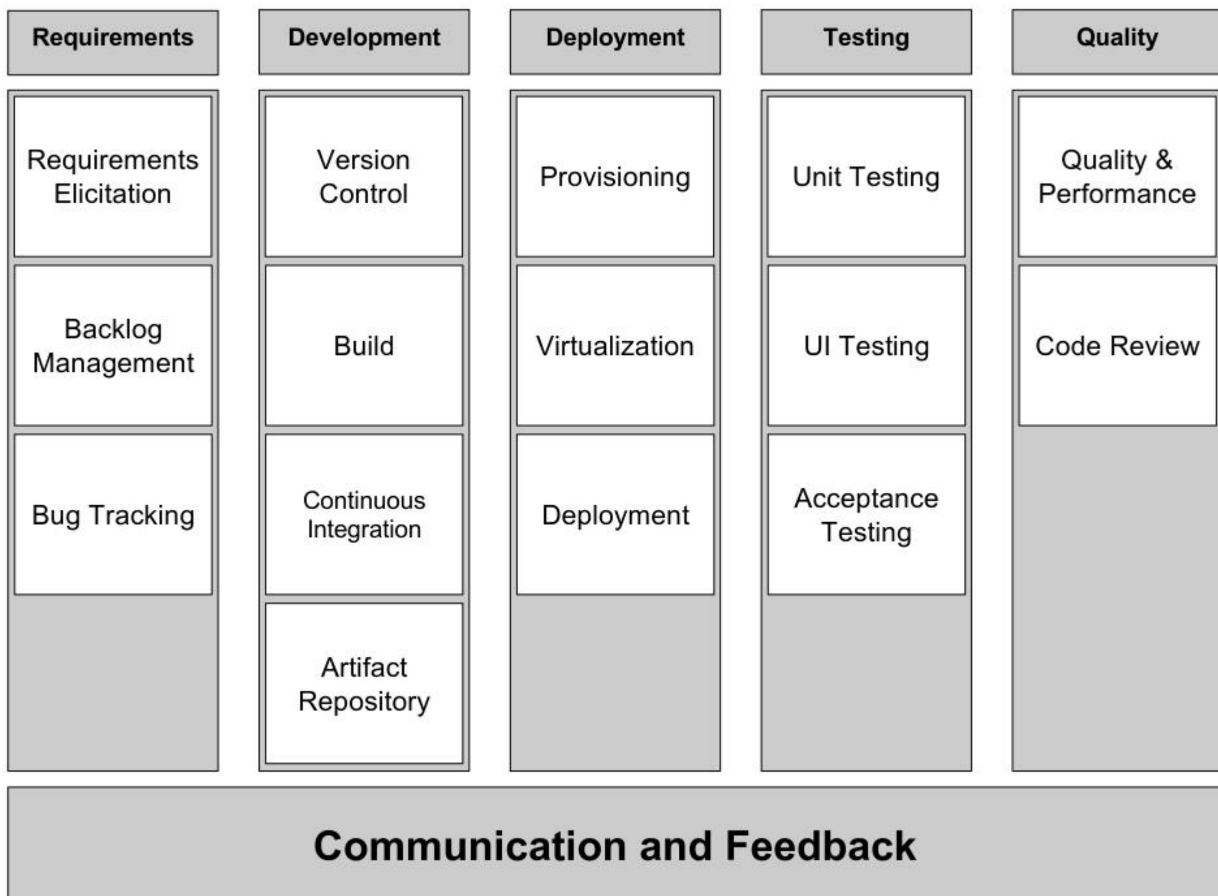


Figure 5 Elements of modern software development, adapted from Publication VI.

As presented in Publication VI, modern software development consists of several activities, see **Figure 5**. These activities may or may not be on the responsibility of separated suborganizations. Some organizations prefer having a separate team for taking care of certain activities. For example, an organization may allocate testing and quality assurance to a specialized team as it can be argued that the developers shouldn’t be the only people testing their own code. Same misconceptions and errors that plague the code are easily transferred to the testware, if the same person does them both. As an example, ISTQB (*International Software Testing Qualifications Board)* recommends avoiding author bias in its syllabus [34] and safety-critical standards usually require the tester to be independent from the developers [35].

Similarly, deployment-related issues may be concern of a separate operations team, which has to ensure that the servers, either virtualized or concrete ones, provide the software the expected environment. As these operations-related activities require specialized skills, it might be wise to utilize the capabilities of the people involved in their fullest, letting them to concentrate in their speciality; the developers in coding, and the system administrators in maintaining the platforms.

Regardless of the organizational structure, there is at least some amount of agreed-upon processes involved. Usually these processes are part of the workflow for the changes made to the code, and can be affected by the selection of the tools for development. An example of this can be that developers ensure quality by invoking a pull request for a review for every bit of code they have checked in to the version control. Good examples of micro-processes that could be applied in any organization involving software development are three fairly well-known branching models for Git[1] version control system. All of have been applied in several industrial settings, as found out in Publication VI; “A successful Git branching model/Git Flow”, “Another Git branching model”, and “A rebase workflow for Git”. Most software development organizations have to come up with a way to use version control system, as it was the ubiquitously found tool from every company that was studied in Publication VI.

It should be noted that Figure 2 only presents the elements present in software development, and does not discuss their mutual order. Some of these elements can be carried out in parallel to each other to hasten up the development pace. For example, Scrum [36], being the most popular software development process framework [37], incorporates sprints, a timebox producing an increment to the software [38]. In a sprint, anything that is developed has to be tested in order to cross the task out as done. Usually unit testing is done right after the implementation, by the developer herself. Any testing beyond that might require some time and computational capacity, so these tests are started automatically after the code has been committed, releasing the developer to produce new features, while the old ones are tested. Simultaneously, the product owner may elicit new requirements and put them into the so-called product backlog. The product owner works as a surrogate customer, who tries to understand the requirements of different stakeholders and put them together as sensible and coherent backlog items. In addition, the product owner organizes the backlog so that the greatest value is added and the most crucial features get implemented first to ensure the customer a good return on investment. After each sprint, there is a potentially shippable product, which can be acceptance tested by the customer, and installed to gain more insight from the end users. This kind of parallelism helps to speed up the development, as there are artefacts in different stages in the development.

---

[1] https://git-scm.com/

2.2 Software Development Tools and Infrastructure

One of the Agile principles tells that

"*Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*" [32]

An important part of the aforementioned environment is the actual tools with which the software is developed. Software is no different from other engineered systems that it is built with tools, ranging from basic stuff as the code editor and a compiler to more advanced automation tools. Also, there are distinct disciplines involved in the development, which all need somewhat different tools. For example, all software parts should be tested, and there are tools for defining the tests, running them, automating them and showing the results to the developers in an easily accessible way.

In addition to the tools for building the actual system, there is the infrastructure that is used for deploying the software system. Naturally, this depends heavily on the domain of the software. On one hand, in embedded domain the deployment can be a cumbersome operation, including physical tools for uploading the software, production stops as the system must be not in use while deploying, and other such blocking work phases. On the other hand, contemporaryb web software can be deployed on virtual servers on cloud leading to very fast deployment. This server environment in the production usually has to be replicated to some degree for running the actual development and tests on the system. So, clones of the virtual machine are run on the developers' desktops, quality assurance servers, acceptance testing servers and so on.

As presented by Humble and Farley, modern software engineering forms so called continuous deployment pipeline from coding to deployment [4]. The role of the pipeline is the automated manifestation of the process how software travels from the version control system to the user, as stated in Publication II. Along the way, there are several tools involved, as presented in Publication VI. The actual choice of tools depends on various factors, such selected technology stack, legacy considerations, developers' personal preferences etc. Thus, the pipeline closely resembles the software architecture, as several design decisions, either consciously made or by pure happenstance, affect the actual tool chain used in a certain project.

Some examples found in Publication VI underline the close relation of the development pipeline to the process, delivery rate and the software itself. For example, as the delivery speed is more pronounced aspect in web system and these systems are prominently made with JavaScript frameworks, several tools for automation in this domain exist. On the other hand, it was found out in Publication I that the testing of user interfaces can be hard to automate with tools due to several reasons, such as lack of tooling, the reactivity of the user interfaces and so on. Thus, several companies relied on manual testing. From the sample used in Publication VI, it seems that a version control system,

mostly Git[2], is always present, and usually some automation tool, such as Jenkins[3], is used if the company adheres Continuous Integration principle in their processes. Also, depending on the processes of the company, additional tools, such as review tool Gerrit[4] is added. Thus, the selected tools are a manifestation of the processes used in the company.

However, one can't deduce the used processes based only on the selected toolset, as the tools may be highly configurable and allow several ways to use them. For example, build systems use build specification files written in a configuration language to build the deliverable software [39]. The configuration, or so called build specifications, can include instructions for compiling code, executing test cases, packaging project files and deploying new software releases [40], or they can be used to manage dependencies of the software, for example to external libraries [39].

2.3 Software Architecture

Software architecture refers to the high level breakdown of the software system to its parts and its description. Thus, it includes the structures of the software and their documentation. In ISO/IEC 42010 architecture description standard [41], software architecture is defined as "fundamental conception of a system in its environment embodied in elements, their relationships to each other and to the environment, and the principles guiding its design and evolution." Compared to the code itself that is meant to produce functionality, the architectural structures should be designed so that the resulting software is able to conform to the non-functional requirements. In fact, Kazman & Bass have stated that the architecture is dictated by quality requirements rather than functional ones [42]. These quality attributes are well known to the software engineering community, and several taxonomies and categories have been published. One of the best-understood quality models is the ISO/IEC 9126-1 quality standard [43]. This ISO/IEC quality standard classifies the quality into a tree-like structure having six main characteristics. The main characteristics are functionality, reliability, usability, efficiency, maintainability and portability. Each of these is then divided into more detailed subdivisions, for example, efficiency can be divided into time behaviour, resource utilization and efficiency compliance.

Another way to divide software quality is to split it dichotomically into internal and external quality [44]. This division relies on the differing viewpoints of the development and the end user, respectively. The architecture is mostly about internal quality, as it is visible in most cases only to the developers.

---

[2] https://git-scm.com/
[3] https://jenkins.io/
[4] https://code.google.com/p/gerrit/

If the internal quality declines, the development becomes slower and more difficult. This phenomenon is also known as technical debt, where time spent on fixing "not-quite-right code" can be seen as paying the interest of making unconsolidated changes to the code in the hope of speedy delivery [45]. To countermeasure this, measures like refactoring and clean coding should be applied [46]. External quality tells about the fitness of the software for its intended purpose and can be mostly measured with functional tests. If external quality gets worse, the software won't satisfy the end users' (or some other stakeholders') needs anymore. When speaking of the impact of the architecture to software development, we are mostly speaking of quality attributes that can be classified as being part of the internal quality.

It is known that any quality attribute cannot be achieved in isolation in real world systems [47], as there are always some trade-offs involved. For example, improving performance by removing extra layers in the architecture may cause a significant loss of modifiability. In some cases, one might sacrifice internal quality for the sake of external quality. This is exactly what happens if one takes deliberate, but prudent technical debt using the terms from the Technical Debt Quadrant by Fowler [48]. The developers make design decisions they know that will hinder their development speed in the future in order to deliver faster now.

The Agile principles do not give much guidelines regarding the software architecture, save for notion of "Continuous attention to technical excellence and good design enhances agility" [32]. This principle has the underlying assumption that the decisions related to the software architecture do not only affect the properties of the software, but the fluency of the development process. Agile proponents usually suggest that big up-front design should be avoided, as so-called analysis paralysis may grind the development to halt, and time is wasted in designing something that may not be ever used [49]. Thus, it is recommended to start small, and to grow your system more organically while the development proceeds. Wrong design decisions may be fixed with the practice of refactoring, where the system architecture is changed to conform for the desired quality attributes, but still preserving the functionality. This kind of practice requires a high degree of automated testing to make sure that the changes do not lose functionality or any faults are not regressing.

It should be remembered that architecture can exist on several levels. Nowadays, as the software systems have grown larger and they—especially after the popularization of so-called microservices [50]—have several integration points with other systems. Microservices consist of small and independent processes, which communicate with each other. This approach makes designing the larger systems easier by promoting a very modular approach. A good microservice is well decoupled from others and highly cohesive focusing only on one small task, following the design principles that have known to work well for a long time [51].

Another example of complex system architecture existing on several levels could be a control system of a moving machine, such as a car. The software systems in modern cars are amongst the largest software systems the humankind has ever made, often topping hundred million lines of code [52].

However, the system is heavily distributed as it is built from several Electronic Control Units (ECUs), which run parts of the software. One way to design the system is to structure the groups of function as Logical Architectural Components (LACs). One LAC could be the locking system of a car, for instance. These LACs can be dispersed over several ECUs, and can communicate with other LACs. LACs consist of Logical Components (LCs), which can be rather small pieces of software, such as the key authorization. Thus, the whole software consists of several layers of architecture, which should be in sync with each other [53].

One other well-known example of decentralized architecture is so-called system of systems (SoS). They are defined in [54] as systems where the components 1) fulfil valid purposes in their own right, and continue to operate to fulfil those purposed if disassembled from the overall system, and 2) are managed mostly for their own purpose, rather than for the purpose of the whole. Furthermore, components are separately acquired and integrated, but maintain a continuing operational existence independent of the whole. Thus, an embedded control system fits nicely in this definition. The properties of the system of systems emerge from the combination of the parts and the system may evolve as the parts are replaced.

Thus, the architecture has varying degrees of granularity, and there might be need of designing the system on several levels of abstraction. In the case of cars, or similar distributed machine control systems, the high level design is done in collaboration with experts from other domains than software. Electronics, hydraulics, and mechanics all can restrict the design of the software architecture. On the other hand, on single control unit, the architecture might consist of several well-known design patterns, such as Variable Manager [55]. Similarly, in a web-based service, such as Spotify, the features may reside as separate and independent feature service servers in the data centers, which can be replicated to achieve redundancy and scaling [56]. On the other hand, simple services may rely on well-known frameworks, such as Angular.js[5], and platforms, such as Node.js[6], to easily implement commonly needed functionalities.

2.4 Continuous *

To keep in pace with the ever-changing world, quick delivery can be a matter of life and death to a software company as it allows them to stay a step ahead of competition [4]. Traditionally, software systems have been delivered as single products, which are installed once and then the customer can reap the value from the investment to the development. However, Agile methods aim at transforming the value of software system to smaller increments. The principle has been stated in the

[5] https://angularjs.org/
[6] https://nodejs.org/en/

Agile manifesto as follows: “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software” [32]. The software that has been developed, but is not yet deployed, is not producing any return on investment. Reflecting the Toyota Production System [57], this kind of undeployed software can be seen as inventory, which is waste that should be minimized.

Based on the above, it is proposed that the software system should be in a potentially shippable state as often as possible. This practice is called Continuous Integration (CI), where every new commit to a code base is integrated with the whole system, and the problems associated with the Big Bang integration are avoided. Big Bang integration refers to the situation where all code is integrated just before the shipment, often leading to a multitude of problems, such as having a large amount of errors, which are hard to isolate and correct, because of the vastness of the software [58]. In Continuous Integration, risks are reduced as errors are found earlier, and motivation is higher as the developers see a working system all the time [59] [60]. Continuous Integration can be achieved through rigorous processes and manual quality control, but most often automation and Continuous Integration tools installed on a CI server help. These tools are triggered by every code commit a developer makes, and the CI server builds and tests the new code automatically.

The practice of Continuous Integration was already seen as a part of the agile methodologies, especially in Extreme Programming (XP) [61]. In these agile methods, fast delivery also allows work to be saved from the requirements elicitation. As producing new versions of the software was easy, it was faster to show the actual software to the customer representative than to try to get the requirements exactly right before starting the implementation. Rapid feedback from every increment helped to establish a system reminiscent of well-known learning cycles, like the PDCA/PDSA (Plan-do-check-act/Plan-do-study-act) method of quality control [62], Build-Measure-Learn cycle [63] championed by lean start-up advocates and the scientific method itself.

However, Continuous Integration is not enough, as it only requires that the developers get feedback if their changes to the code prevent the software from being built. Thus, CI practice omits altogether the parts of the software life cycle after the build. For example, deployment is not considered at all, although a successful deployment of a build may require lots of effort. The practice of Continuous Delivery brings the concept to the next logical level; the software has to be deployable all the time [1]. Now every software change is tested to build with the rest of the system, and the configuration of the system is such that the build could be deployed to its intended environment. As it is now feasible to deliver any commit a developer does, the next level, Continuous Deployment, adds the requirement of automatically passing every clean build to the deployment. Here, every commit to version control system is automatically built on the continuous integration server, tested, and deployed to the production environment successfully.

Advanced continuous deployment often requires virtualization of the deployment environments; development, quality assurance (QA), and production, to name the most common ones. Thus, the development activity is extended beyond the coding and building the product, and requires some

knowledge of these environments. This is traditionally seen as the remit of operations organization, not development organization. Passing the built software artefact from the development to the operations causes a relay race situation, which hinders the speed and agility of the organization as a whole. To make matters worse, in sophisticated continuous deployment the environments should be treated like the code itself. The environment is an indistinguishable part of the software artefact, whose configurations can be stored to the version control system along with the code.

Thus, the developers have to tightly cooperate with system operators, or, even better, have the required skillset and resources to treat the software system and the deployment environments as a whole. This practice lends its name to the DevOps method, where development, quality assurance and operations form a holistic approach [6]. Thus, DevOps requires heavy automation spanning the entire delivery pipeline, with automated tests and virtualization of the environments.

Agile methods were first considered to be only suitable for limited to very small web-based socio-technical systems [64]. Similar critique has been presented to the practice of Continuous Deployment. It has been seen as being fully applicable only to systems that are cloud-based web systems, as this is the environment where CD has been most beneficial. It has also been easiest to implement, as deployment does not involve any physical systems. However, Agile methodologies have been applied nowadays on wide variety of domains, all the way up to embedded systems, even very successfully, so it seems that Continuous Deployment will be widely adopted, even though the degree of feasible automation depends on the domain. For example, an industrial control system may require new hardware or physical changes to the configuration of the devices, so it will be more challenging to automate than purely software-based systems residing on the web. This phenomenon was noticed during the interviews for Publication I, where it was noted that companies in domains which involved physical systems had somewhat slower deployment cycle.

However, it might be argued that in other domains fast delivery is not so crucial, as one of benefits of Continuous Delivery is not only producing faster value to the customer, but to get quick feedback from the end user [65]. This helps to get the right product to the customer, bringing more value by fulfilling the customer needs, and reducing waste [66]. To complete the development pipeline into a fully-connected feedback cycle, end users can also be monitored with analytic tools. The need for requirements elicitation is vastly diminished as real usage data allows to make decisions regarding the future of the software system [67]. In some cases, with proper customer data, new innovations are developed even before the customers have expressed their needs that are addressed by the innovation [68]. Thus, direct communication with customer and the end users loses some of its significance. Tools also can, at least partially, take care of the communication between developers in the daily work, as found in Publication III. Of course, some regular meetings between developers can take place to synchronize tasks and to share common status, but mostly notifications come from the automation tools. As features are deployed instantly after they have passed the tests, only a few interactions between the developers are needed as the automation takes care of triggering all the necessary steps.

The quality of the end product is improved as the organization can quickly produce something tangible on which they can get feedback on and utilize this information in the next iteration. To achieve a rapid development pace, everything in the development has to work together like a well-oiled machine. An example of an organization with a rapid deployment pace is presented in Publication II. There, an industrial case was presented, where rapid software development practices were in everyday use, and Continuous Delivery was the goal (**Figure 6**). Although there was no strict requirement of Continuous Deployment, every team member was invested with the authority to deploy at will, so the deployment management existed purely for coordinating when and what to deploy, so that the customer gets exactly what they want. There was also the additional requirement of quick validation of the business ideas, so it was mandated that every completely new idea could be implemented as proof-of-concept in eight weeks. This proof-of-concept is launched to the end users and a wide array of metrics is collected about the usage of the system. This collected data is then interpreted, and either tweakings are made to improve the original idea, more features are developed, or the whole idea is scrapped altogether. This allows all major stakeholders, namely the business analysts, the product owner, and the developers, to gain a deep insight into the product. Another example of a well-working continuous delivery pipeline has been presented in [2].

.

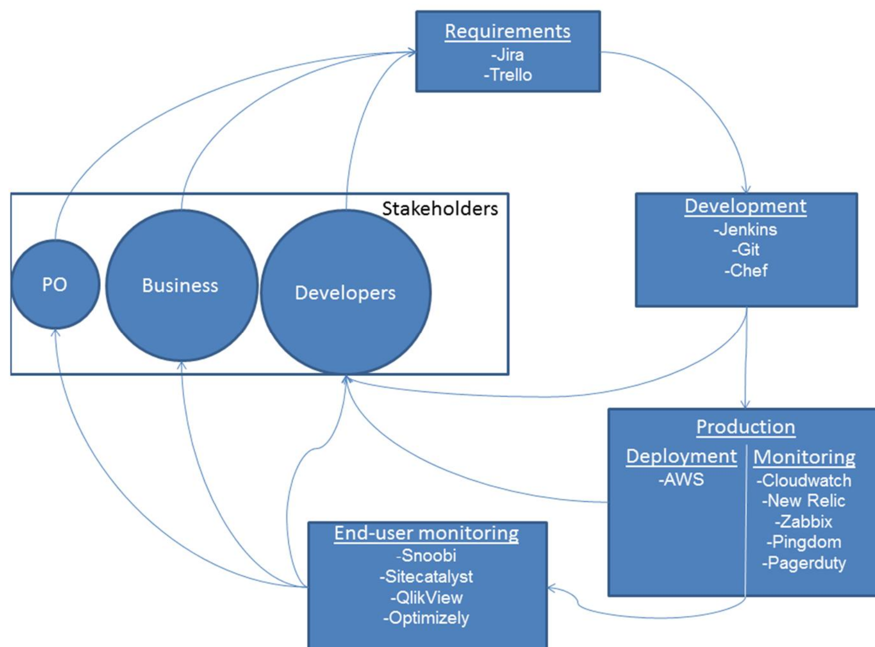


Figure 6 An example of a software process with elements of Continuous delivery and rapid feedback. The tools for each phase are also shown (Publication II).

The aforementioned way of streamlining the deployment pipeline resonates with the lean concept of flow [69]. The software is developed no more so that a certain phase produces a batch of artefacts which then are queued for the next phase. Instead, the artefacts flow seamlessly through the whole pipeline of actions, which all add some value to the processed artefact. Immediately after a new feature is conceived by recognizing the need for it, it starts flowing through the subsequent phases until been deployed to be used by the end user. When this flow is connected as a holistic end-to-end approach to software development by connecting development with business planning and strategy, operations, and continuous improvement and innovation, it has been dubbed as Continuous * by Fitzgerald and Stol [7]. A flawless flow benefits from small batch size, using automation with feed-back in visualised form (jidoka), and kaizen, i.e. continuous improvement.

3 Components of Contemporary Software Development

Today, not all companies achieve such degree of maturity as the company presented in the Publication II. In fact, it seems to be quite uncommon, as found out in Publication I. Companies seem to have a varying degree of maturity, and it seems that companies evolve with piecemeal improvements towards a mature continuous deployment pipeline.

To assess the maturity of an organization, there exist a few continuous deployment maturity models and DevOps maturity models. These evaluate the capability of an organization on several different levels depending on the model. As an example, a well-known maturity model by Rehn et al., published as an InfoQ article [11], categorizes the five key aspects of Continuous Delivery; namely Culture & Organization, Design & Architecture, Build & Deploy, Test & Verification and Information & Reporting. An overview of the model is presented in **Figure 7**. However, these categories are not fully orthogonal to each other. As a basis, Rehn et al. acknowledge that the organizational things are crucial for sustainable CD environment. In this model, Culture & Organization hold things that are related to the software processes and their underlying organizational structures, i.e. testing with development, decentralized decision-making to the team, cross-functional teams and so on. Design & Architecture hold things that are related to the structure of the software system itself. These include, on several levels, for example, having modularity up to the degree to API-separated components which can be deployed individually. On the highest level, the software system is tied with the infrastructure on which it will be deployed. The software system starts to be indistinguishable from the deployment-related technology via virtualization.

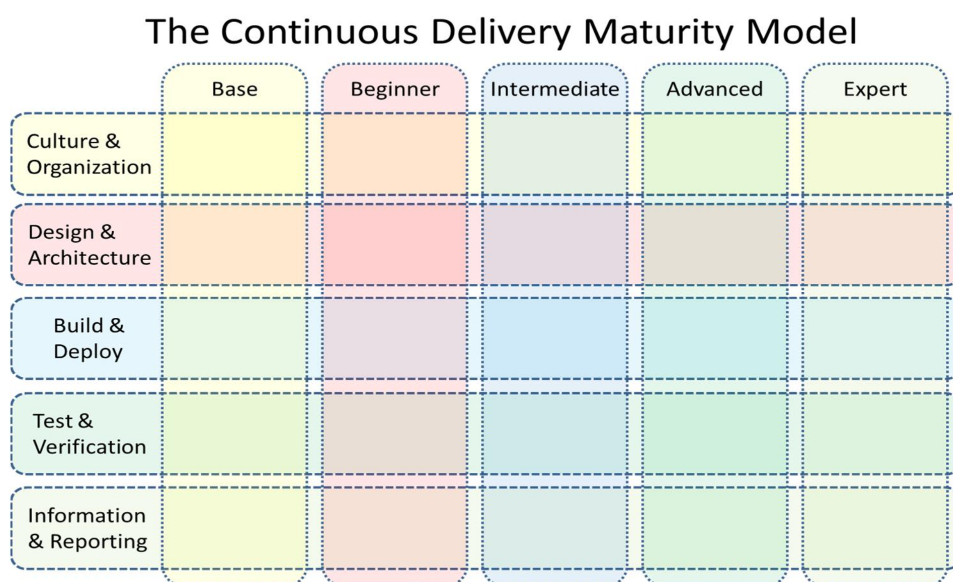


Figure 7 An example of a Continuous Delivery maturity model, redrawn from Rehn et al. [11]. The model consists of five maturity levels on five different aspects of the software development, forming a matrix of 25 different possible combinations of maturity.

The next level, build and deploy, covers things related to the tooling and automation of the deployment pipeline. The selection of right tools aids the developer to advance to the level where every commit is built to be potentially deployable to the production environment, ultimately bundled with the virtual machines. Very closely related to this category is the test and verification, as it also relies heavily on tooling. Information and reporting are basically process related things, but on the advanced levels, most of the inputs are metrics gathered by various tools.

From this, we can see that these levels are entangled with each other. One other way to observe the maturity is the Stairway to Heaven model [14], which puts organizations on five individual steps from traditional waterfall approach through agile to continuous integration, continuous deployment and, finally, being an innovation system for R&D. For example, the organization presented in Publication II, see **Figure 6**, closely resembles the described innovation system, although they do not fully adhere to the principle of continuous deployment. In the Stairway to Heaven model, there is only short descriptions of each level, so assessment is not possible, but it can be deducted that the phenomena described in [11] is also present here. In any case, the model strongly suggests an evolution of an organization towards the innovation system. The consensus seems to be that the software development organization evolves onwards to faster delivery, on several levels, which can be quite arbitrary depending on the units of analysis.

A less studied subject is the effect on these aspects on each other, when the delivery speed is increased. Thus, the PAI model presented in **Figure 4** is used to analyse the effect of release speed on the three selected aspects of the software engineering, namely development processes, the tool infrastructure, and the software architecture. See **Figure 8** for the focus areas of the included publications related to the aspects of the software development. Not only the effect of speed to each of them is studied next based on the research, but the interrelations of each of these on each other. So, when the delivery speed is hastened, processes start to constrain and affect both the tool infrastructure and the software architecture. Similarly, the selected tool infrastructure has its effect on the development process and the software architecture. Finally, the software architecture will have some consequences on the software process and the selected tool infrastructure. These interrelations form a triangle consisting of the aspects as the sides. Changing one aspect will have its effect on all other sides of the triangle, and more importantly, faster the release speed, more profound this effect will be.

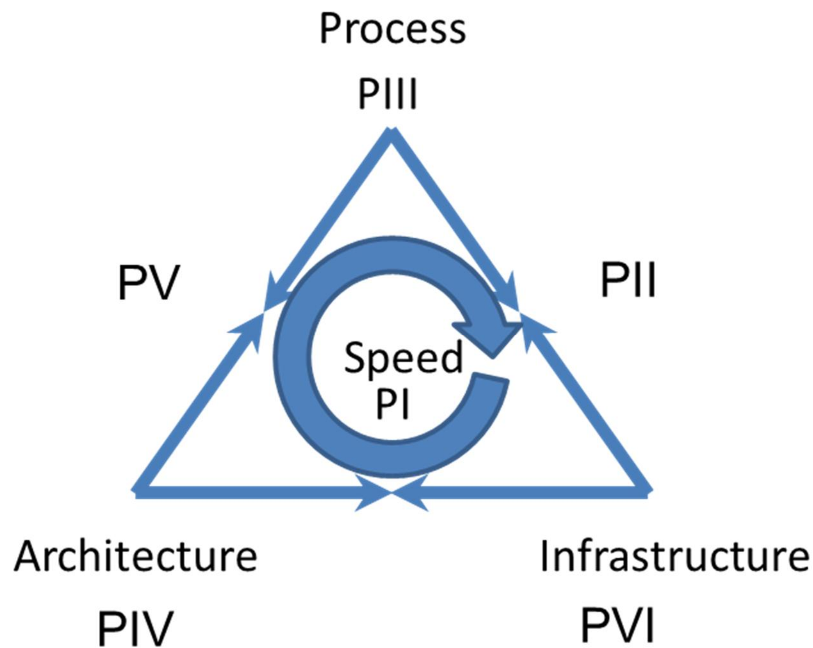


Figure 8 The publications included in this thesis and their rough relation to the elements of software development.

3.1 Architecture-Process relation

Designing successful software architecture requires a lot of knowledge. The architect must know the specifics of the domain, available technologies, potential patterns of design solutions, current requirements for the system, and most probable changes to them. To make matters more complicated, this information has to be gained and shared amongst several stakeholders, such as the customer, end users, product owners, developers and so on. Thus, this knowledge needs to be communicated, and the organizational structure is reflected in the effectiveness of a certain communication connection. Software process can be seen as the way of working in a software organization and as a catalyst or impediment to communication between the relevant parties.

It is a well-known fact that a system and the organization producing the system resemble closely each other: “"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure” [70]. This phenomenon known as the Conway’s law is also reflected in the sociotechnical congruence framework [71]. The law stems from the real-world communication being reflected as interfaces in the system. This relation alone binds the software architecture and the

software processes together tightly. It has been argued that if these two aspects of software are not aligned, the software development is severely hindered [72]. At the same time, good architecture is self-documenting and thus decreases the need for communication.

As the architecture is closely related to the knowledge of the design decisions of the systems, all the information that is not evident from the code falls into realm of knowledge management, which is the process of capturing, developing, sharing, protecting, and using the knowledge efficiently [73]. Thus, architecture knowledge can be managed with processes [74]. The knowledge management processes can be broadly classified to be using either of two strategies, Codification or Personalization [75]. On one hand, Codification means storing the knowledge systematically, and providing an access to it to the involved parties. Personalization, on the other hand, bases on establishing flows of information between individuals, and supporting this peer-to-peer information sharing. For example, all knowledge related to a certain technology would be available to a certain person, and anyone who needs and is entitled to certain information about this technology will know who to ask. Codification usually is more tool-oriented approach, but still calls for processes for storing the information in the first place, and especially to update it, when the software is changed. Codification mostly relies on efficient organizational structure.

One phenomenon related to architecture, but influenced by the process, is the notion of technical debt [45]. It is a metaphor for cutting corners and adding allegorical duct-tape to the system, but can also mean shortcuts and architectural rule violations – all those quick and dirty solutions that are made in the hope of releasing the product more often. Technical debt resembles financial debt in the sense that one might consider debt a resource which helps to cash in on releasing new valuable features faster. Furthermore, as a result of the technical debt, the code becomes harder to maintain, and in the future, it will take more effort to implement new features. This might lead into a vicious circle, and to compensate this slower development speed, more technical debt might be introduced to the software system. Thus, the debt that has not paid back will have some interest rate.

Technical debt can be paid back by refactoring, which means improving the design of existing code [76]. It has also been argued that there cannot be efficient refactoring without good, and preferably automated, tests, being an infrastructural aspect [77]. Good automated tests are required to prevent unintentional changes to the functionality, or to prevent regression of already corrected bugs – in short, to ensure that the changes made did not break any existing functionality. In Publication IV, one of the findings was that several interviewed architects deemed the essential toolset for refactoring to be a good version control system and a good set of automated tests on a continuous integration server.

A major cause for technical debt is schedule pressure [78]. This is in line with Publication IV, where the interviewees nominated the constant rush as the leading cause of refactoring. Other reasons listed in [78] are “carelessness, lack of education, poor processes, unsystematic verification of quality, or basic incompetence.” Thus, the selected process has a definite impact on how architecture

evolves, for good or worse. Also, it has been noted that accumulated technical debt as bad quality code may lead to difficulties in estimating the workload. For example, in Publication IV, company B representative found the code quality to be in a bad shape, and this was reflected in spending several weeks on a feature that was expected to be completed in a few hours.

The currency to pay technical debt back is, of course, hard work. Those “not-so-right” structures have to be corrected by refactoring the code. Refactoring also is the way that Agile methodologies promote getting the architecture right by piecemeal growth, for example, in Crystal Clear a walking skeleton is developed first, and it is evolved in an iterative fashion [79]. Thus, the software development process should allow time for refactoring. Sadly, this is rarely the case, as refactoring is not seen to bring any additional value, but just lose time in non-productive work as found out in Publication IV. This effect is strengthened by the perceived lack of suitable metrics that would tell if there is a need for refactoring. Usually, the metrics for the need of refactoring are so-called code smells which are detected by the tools from the code [80]. However, they can poorly predict certain refactoring decisions as the more advanced problems are usually only identified by experienced evaluators [81]. Thus, the main driver for refactoring decisions is the gut feeling of the developers, as seen in Publication V.

Of course, metrics are usually supported and provided by a tool set, but they drive management decisions, which are in the process domain. Furthermore, there are a plethora of metrics which require disciplined processes to have any meaning. Otherwise, you will get only what you measure [82], in the worst sense of the maxim. For example, in case of duplication metric, people may, just to “please the metric”, start to refactor the code to remove duplications, wasting valuable resources [83]. Based on the above, it might be better to consider metrics based on the measurement of the process itself than try to deduce the need of refactoring only from the software artefact. In Continuous Deployment, the development effort can be made more visible as the changes are smaller and the feedback is instant from the all the phases of the deployment pipeline [84]. Thus, this information could be used in help of detecting the worrisome parts of the code.

It might be even difficult to say code quality really got better after refactoring effort. As stated in Publication V, sometimes, the developers can spend even weeks in a vast refactoring effort, and then notice that the actual quality of the code was worsened and have to revert back. This kind of refactoring failure suggest that the refactoring should be small enough and a part of the daily work, so reverting back would only waste minutes or hours instead of days or even weeks.

Although not found in Publication V, anecdotal evidence in the interviews for Publication IV suggests that accumulated technical debt might lead into a situation where requirements are changed due to the fact that the original idea is too hard to implement without using significant effort on refactoring. Thus, it is easier to come up with an alternative plan to avert the refactoring.

Accumulating technical debt that cannot be corrected within couple of hours of worktime indicates that the development process is not working optimally and the refactoring might prevent Continuous Delivery as the refactoring task is too big to fit nicely into the streamlined process. For example, **Figure 9** depicts how one refactoring related to implementing encrypted cookies to a certain software system was split into version control system. The figure shows that the refactoring was started as a new branch, and the codebase was first refactored to allow the implementation of this new feature. Then the rest of tasks related to this four-day effort are mostly new implementation, interrupted by smaller refactorings. As a result, the development flows smoothly, as the changes related to this new feature are contained in a separate branch, and at the same time the changes are easy to make as the code is first refactored locally to accommodate the required functional changes.

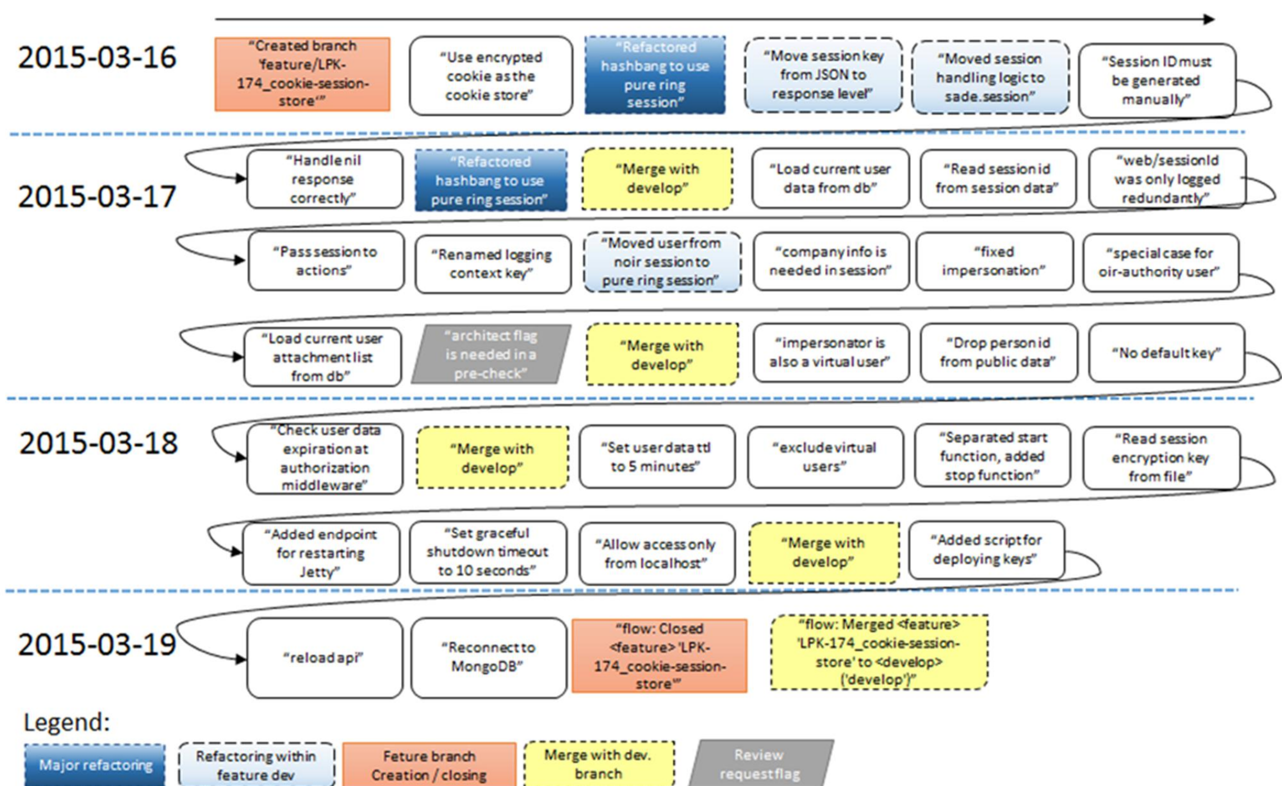


Figure 9 An example from Publication V showing how refactoring is carried out in a software development process.

It is interesting to note that there exists a parallel for technical debt at the organizational level, called social debt [85]. It refers to a situation where the organization is suboptimal in the sense that the right people do not have opportunity to coordinate, collaborate, or cooperate their work. This leads to situation where the productive work is hindered because of organizational problems. This is reminiscent of the aforementioned Conway's law [70]. This is reflected in the findings of Publication III

as the surveyed people desired to improve their communication by having co-located teams and direct customer contact.

The architecture in the light of new ways doing the software, such as DevOps in the context of Continuous Deployment [6] [86], has been argued to be one of the challenges in the software organizations. However, regardless of the gravity of the issue, it has also been a little studied field [87].

3.2 Process-Infrastructure relation

In addition to automating the tasks that someone would have to do manually otherwise, a deployment pipeline infrastructure forms several measurement points, where data and feedback can be collected. Beginning from the Continuous integration servers, the developers get instantaneous feedback from the automated tests, if their recent commits work as expected and no existing functionality has been compromised. This allows the process to be transformed into more agile approach, where tasks have clear completion criteria, or definition-of-dones, as successful tests can be demanded.as a part of a task. In addition to tests, it is possible to add other quality assurance tools to check the code. For example, SonarQube[7] seems to be widely used in the companies, which embrace Continuous * practices, according to the interviews cited in Publication VI. This allows the software development to have processes related to the quality of the end product. However, there is some critique among the developers as they do not find the metric very useful, as honing the metric does not improve the perceived quality and the bad quality code is so self-evident that the metric doesn't help. Blind obedience to a metric may even cause bad design decisions, as found out in Publication IV. Also, code reviews can be a tool-aided part of the process, which also can help in distributing the architecture-related knowledge amongst the developers [88].

In addition to this kind of generated feedback, a good infrastructure allows also collecting the end users' actions [89]. This was a prominent feature in the infrastructure presented in Publication II, where there was status information available to the developers from all stages from the deployment pipeline coupled with the measurement data from the end user actions. This allowed the company to make decisions based on the real user data and the collection of the requirements was also backed up with end user data. In the end, they had a system which allowed them to terminate projects in proof-of-concept stage if the measurements on the end users showed lack of interest on behalf of the users. This indicated low business value and the project could be terminated with low sunken costs. See **Figure 10** for an illustration of the process. The proof-of-concept (PoC) here is the term used in the company, although it is not a technical proof of concept, but closely related to the Minimum Viable Product (MVP) of the Lean movement [90]. The PoC is built when a business

---

[7] http://www.sonarqube.org/

idea is born, and it is formulated as a business hypothesis, such as that people using the service are willing to pay for an additional feature. Then, this business hypothesis is verified by building the PoC/MVP to test the hypothesis. If the hypothesis seems to hold, then the PoC feature will be developed into a full-fledged feature. Otherwise, it is dumped.

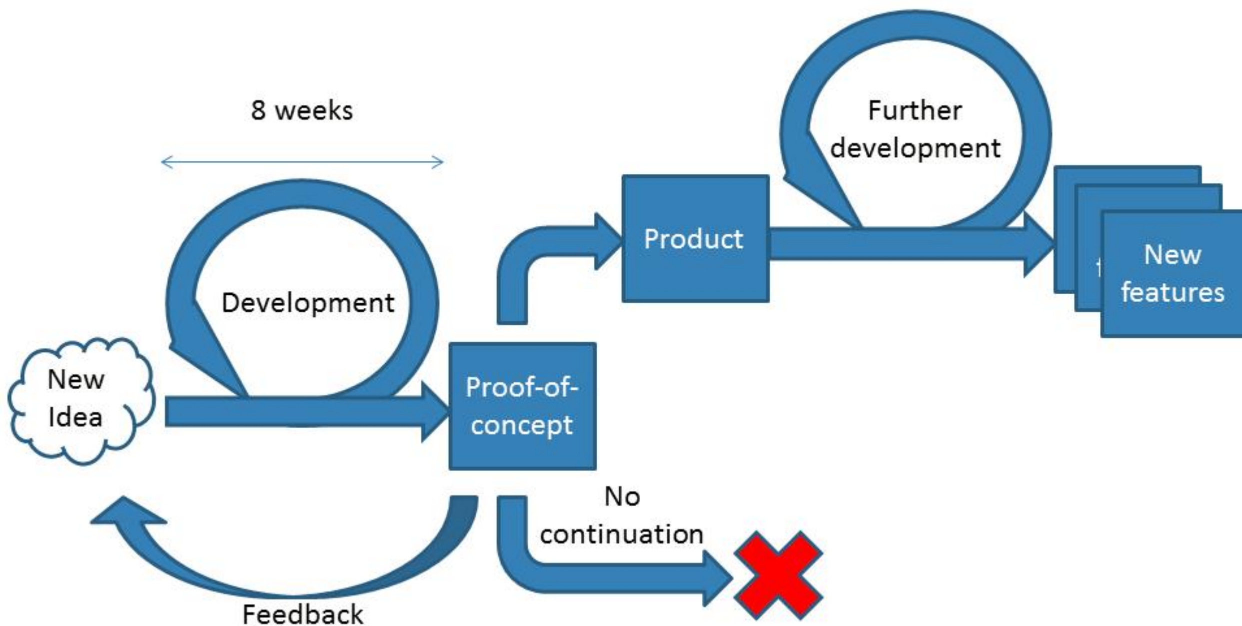


Figure 10 The process presented in Publication II. The company uses proof-of-concept to measure end users if an idea has any business value.

Furthermore, the developers also need to coordinate, cooperate, and collaborate in the development team. The Continuous delivery pipelines often incorporate radiators in creating a way of visual control to facilitate coordination of the work. Problems in the pipeline are immediately visible to all and the state of the system can be easily checked. As found out in publication III, these systems are popular among developers, but have the downside that they only work well with co-located teams. Distributed teams seem to be quite commonplace—for example, 48 percent of the respondents of the survey in Publication III worked in a distributed team—so radiators are not enough. Of course, distribution also hinders communication as face-to-face meetings are more difficult to organize and ad hoc communication must be carried out via tools. Naturally, supportive processes must be in place for efficient work coordination, but ultimately they are backed by tools. So, so called “social developer” depicted in Publication III is an overarching concept, where tools, methods, and processes must be in line with each other.

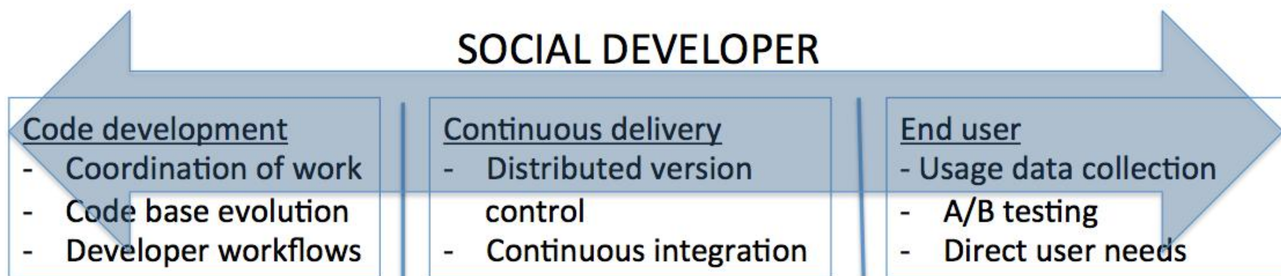


Figure 11 A "social developer" requires a mature set of tools to get feedback and information from all relevant stakeholders and to monitor the development pipeline.

There has also been a growing interest in developing so-called Product development clouds, for example Microsoft DPaaS - development platform as a service provided by Tieto[8], or development infrastructures as a service. There has also been a strong academic interest in such collaborative tools. One example of this is CoRED, which is a browser-based editor for Java web applications allowing collaboration in real time [91]. Other recent development was Amazon's acquisition of Cloud9[9], which is an online browser-based integrated development environment (IDE). Cloud9 adds to Amazon's arsenal of building blocks for modern software development lifecycle by introducing a development environment which is integrated to the platform [92]. In Publication II it was evident that the in-house development infrastructure was crucial for the rapid development of new features. Whenever a new business idea emerged, the company could set up a full development environment in less than one day. However, the original setup had taken a long time, and the required skills were scarce, so only a few persons could handle the setup of the infrastructure. So, it would be a lucrative option to buy this kind of service from another company. However, as the infrastructure has an effect to the process, it might cause some problems as the developers might not find the offered product development infrastructure suitable for them. Thus, they have to be highly customizable, but also it should be very easy to take them into use.

To conclude, it all resonates with the earlier quoted definition of software process by Fuggetta [28] as software development needing the policies, organizational structure, technologies, procedures, and artefacts for the whole life cycle of the product. A tool can require a process to be used efficiently, and the processes can be backed up by tools to produce useful metrics, feedback channels, to automate manual stages, to act as a scaffolding to provide tests to prevent regression of errors and the inadvertent changes to the functionality during refactoring and so on.

[8] https://www.tieto.com/product-development/communications-infrastructure/product-development-excellence-communications-infrastructure/pdcloud
[9] https://c9.io/

3.3 Infrastructure-Architecture relation

As stated by Lianping Chen in [86] the Architecturally Significant Requirements (ASRs) for Continuous Delivery are deployability, security, loggability, modifiability, monitorability, and testability. From these, deployability, monitorability, and testability are ultimately utilized in the development infrastructure. So, if one has devised an architecture that allows one to deploy, monitor, and test the software easily, it means that the corresponding parts of the infrastructure can be utilized without any significant hassle. As a side note, one of the important ASR's, architectural security can be mitigated with continuous deployment as it allows one to close security holes fast as soon as they are noticed. Usually the attacks begin at once when a security hole is published, and there is not too much time to close a known security hole.

Of course, the chosen architectural decisions depend heavily on the chosen tools and technologies. A certain tool can require certain decisions to be made in the software architecture, and if the tool is changed, the decision might need to be revisited, which might lead into refactoring as presented in Publication V. Here the pain zone is reached not because of new requirements by the customer, but by the technical requirements set by the development infrastructure. Additionally, several technology selections are architectural decisions in essence. For example, selecting a certain cloud computing platform, such as Amazon AWS, can affect several other decisions, such as load balancing, the architecture of the databases and so on. Thus, this kind of technical decisions are usually near the root of Kruchten's "trace from" relations in the architectural decision tree. Design decisions can *trace from* technical artefacts upstream: requirements and defects, and *trace to* technical artefacts downstream. In the Kruchten's ontology [93], pericrises i.e. executive decisions, are said to frame or constrain existence (ontrocrises) and property decisions (diacrises). As such, the technology and tool decisions are such. However, in Kruchten's ontology, process-related decisions also classify as pericrises. This type of decision is said to be "driven more by the business environment (financial), and affect the development process (methodological), the people (education and training), the organization, and to a large extend the choices of technologies and tools."

Also, Bellomo et al. [87], in their empirical research on three projects, in the light of continuous integration and continuous delivery practices, studied what key goals the projects had driving their deployment efforts. It was found out that there were several architectural decisions which the projects had made in order to support deployability. So, there is a definite connection between the ability to deploy fast, and the architecture the software system has. For example, from their case studies, it can be seen that there are several decisions such as having an integrated test framework is greatly aided by a *tactic* known as Speciality Access Routines/Interfaces, which is an architectural feature.

Infrastructure as code (IaC) is a recent concept that is utilized in DevOps related approaches [94]. Another term for such concepts is Programmable Infrastructure. It is a continuation of configuration

management, where the actual computing infrastructure is managed and provisioned with configuration files, whose processing can be wholly automated. In a sense, it is writing code, usually in a high abstraction level programming language such as YAML provided by Ansible[10], to manage and automate the parts of the infrastructure. So, there is no need to configure the computing infrastructure separately as it can be handled with the actual application code and becomes a part of the software system itself. Furthermore, with IaC it is possible to use same practices which are already used in the application development; for example, using known good solutions or patterns, version control, testing, and so on. So, in the end the development infrastructure has its own architecture (**Figure 12**), and can utilize the same tools as the software itself.

The system architecture can also provide the points where certain tools may collect their data [95]. As the need for evidence-based software engineering and getting feedback from deployed systems becomes more important, the architecture must assume that data collection from all system levels is required and should be integrated by default [96]. Thus, to have a system presented in Publication II requires architectural support. Furthermore, to support process decisions, such as refactoring, there is a need for architecture to support more advanced analysis by tools than just the code smell detection, which can be unreliable driver for refactoring, as discussed earlier. This would mitigate the perceived lack of metrics so prominently displayed in the interviews for Publication IV.

---

[10] https://www.ansible.com/

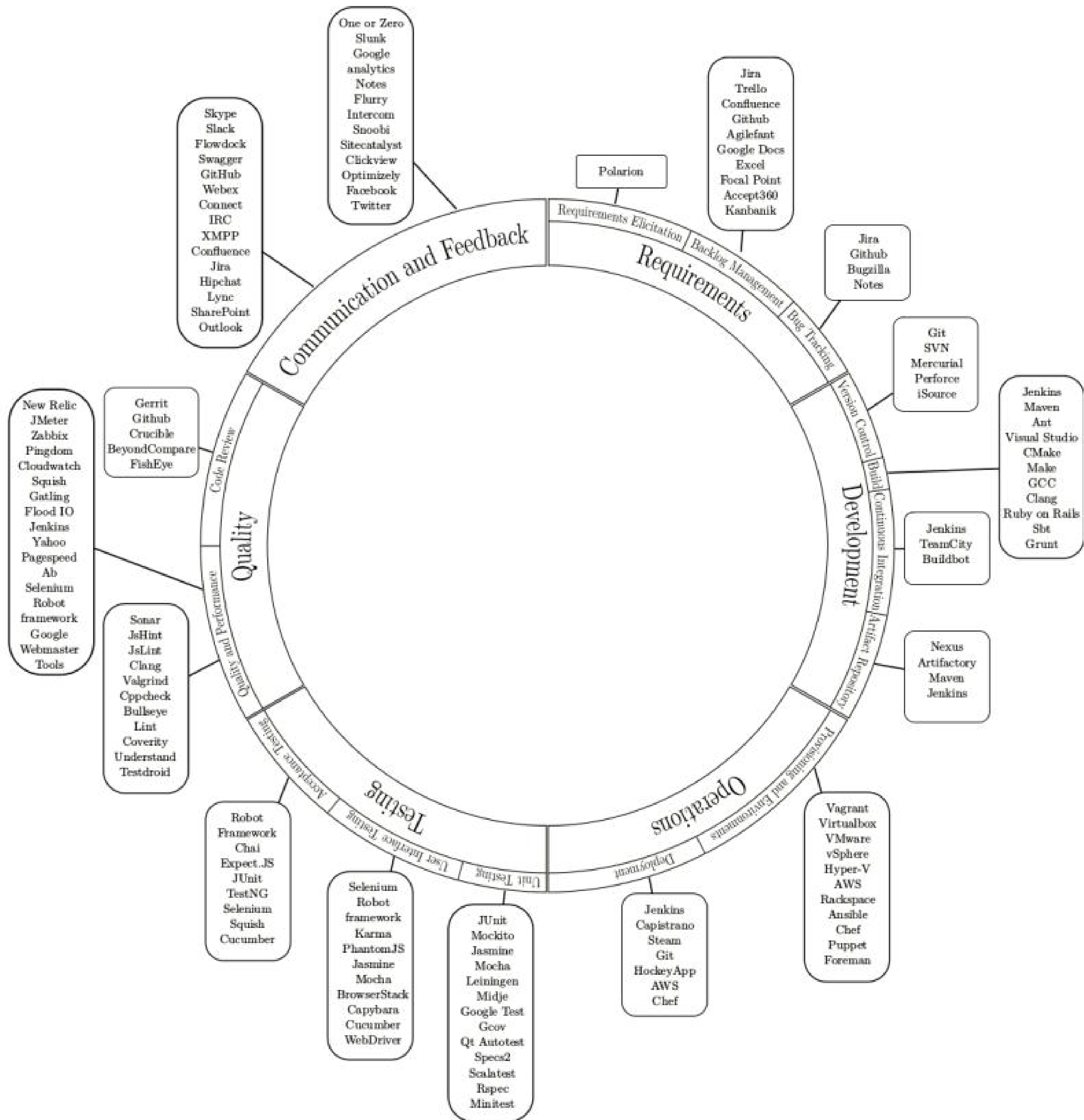


Figure 12 The overview of the tools that were found to be used in Publication VI. The development infrastructure consists of a selection of these tools, forming an architecture of its own.

3.4 Impact of speed

It seems that depending on how mature in their Continuous Deployment practices are, the teams will struggle with different kinds of problems hindering their procession towards a mature development environment and processes. If the team has a fresh start, they will not have any development infrastructure in place, so they have to build it. However, it requires some training and knowledge to build the deployment pipeline. First, the essentials are needed, such as version control system. Then new tools and systems are installed to improve the degree of automation. As presented Publication VI it seems that the deployment tools are something that comes as last improvement to the deployment pipeline. It is quite natural as fast deployment is not essential in many organizations or domains, so improvement effort is usually used in something else as non-automated deployment is something that people can live with or take as granted.

Then, after the basic tools are in the place, the team usually comes to halt as their processes related to the customer, other teams and so on, are not mature enough. This was found out in Publication I, as one of the main obstacles that the interviewees observed in the adaptation of Continuous Deployment was the customer preferences, usually stemming from the customers' inability to cope with the increased release rate. Finally, the system architecture of the system developed might cause problems in squeezing out the final last minutes of the development time. The problems in architecture may include a wide variety of problems, as encountered in the interviews for Publication I. For example, updating single tenant databases may be slow, the system may need a high degree of parallelism to achieve desired response times, system live updates may pose a problem, and, related to that, moving user sessions to new load balancer might be difficult. At the same time, introducing more and more tools into the pipeline will impart its requirements on the software architecture, as testing, deployability, and such details have to be tackled.

High development speed is not enough alone, as the development also has to have a direction. By direction we mean, analogously to the physical concept of velocity, that the developed software must conform to the requirements given by the customer, end user, and other stakeholders. As speed is increased, these requirements can change faster than before, as other systems, business requirements and other related interfaces also change more rapidly. In addition to conforming to certain requirements, the system also has to have satisfactory technical quality. This includes the internal quality of the system, so that the system development speed can be sustained, and the availability of the system to the end users. To ensure this, tighter feedback loops are needed on all levels of the deployment pipeline and from the deployed systems.

4 Closure

In this chapter, first the research questions of the thesis are revisited, and the contribution of the work is presented. Next, we will shortly discuss the related research to this thesis. This section is followed by a short introduction to the included publications. Next, some future work for the line of research is planned. This is followed by a discussion on the limitations of the research presented in the thesis. Finally, some concluding remarks on the thesis is given.

4.1 Research questions revisited

Here the research questions are revisited and answered. Each question will be discussed in the light of the previous chapters and summarized.

RQ1: What are the key enablers to increase the delivery speed?

In a nutshell, the key results based on the publications included in this thesis indicate that one must have all three aspects of PAI model presented in this introductory text in place. First, one should have a *well-automated deployment pipeline*, with as few manual steps as possible, as found out in Publication VI. This deployment pipeline is a part of a greater tool chain, which spans over from basic development tools, such as compiling, building, and deployment, to process tools like *feedback collection* from the various points of the deployment pipeline, deployed software, and work organization. The common tools used in the companies studied in the context of the thesis are presented in Publication VI and an example of such infrastructure with the feedback loops can be seen in Publication II. All this resonates well with literature of known CI practices, such as Automate the Build, Make Your Build Self-Testing, Everyone Can See What Is Happening, and so on [97]. However, it is also interesting to note that not every practice has to be in place to reap at least some of the benefits of CD pipeline.

The aforementioned publications indicate that as the delivery speed increases, the tool chain is usually completed by adding more automation to the end of the pipeline. This is natural, as Continuous Deployment focuses on the automation of the last stages of the pipeline. Similarly, the feedback loops become shorter, and more automation can add fast monitoring to the deployed software. In this way, one does not have to rely solely on the end user reporting on errors.

Secondly, the development pipeline should be *replicable* and its *setup time short*. Ideally, the development infrastructure is a product-like entity which can be quickly harnessed to any new project that would require software development.

In Publication I, the perceived main obstacles to adaptation were investigated. On the process, business and organizational side the main obstacles were resistance to change, the preferences of the customer, constraints imposed by the domain, and the developer's trust and confidence to the tools. In the infrastructure, the main problems were related to having different development and production environments, and the difficulties with manual and non-functional testing. These testing problems are of course also somewhat due to architectural problems, but the main problem seemed to be the lack of tools aiding in the testing of certain non-functional aspects, reactivity of the system, and the graphical user interface. So, in the absence of tools, the companies had to rely on manual testing, and breach the first principle of having fully automated pipeline and no manual steps. Also, time to execute the test suites may be an issue. Thus, *full automated testing* is not a trivial thing, and may require development of novel tools to overcome the problems hindering the adaptation of automated testing.

The different development and production environments can be mitigated with high degree of productization of the development infrastructure. In addition to shortening the set-up times, productization also minimizes the differences in different environments, and removes the need of manual, and error-prone, *configuration of the different environments* when starting up. However, the configuration management of the different environments will still be an important topic. Perhaps in the future, more effort will be seen in productizing the infrastructures. This also resonates with the emerging trend of offering development platforms as a service (DPaaS).

Architecture-wise, the main obstacles were legacy code and large size of the software. These can be mitigated by prudent *refactoring*. However, based on Publication IV, it seemed that the companies somewhat struggled over deciding what to refactor and when. To alleviate this, the interviewed architects hoped to gain the trust of the managers and customers so that they could assert them on the need of refactoring and get the required resources to do this. So, some *agreed-upon and sensible metrics*, which are generated by tools, might help them in this. However, it seemed that at least for now such metrics were rare. Another way to solve this problem would be organizational empowerment of the architects. Now they felt that they only have the responsibility of quick delivery, but no resources to do this. Keeping one accountable for something they are not capable of delivering just creates frustration [98]. The interviewed architects saw many benefits in refactoring, such as the easier future development, increased understandability, possibility for code reuse, improved quality attributes, and even boosting morale and motivation. With all these benefits being possibly lost because of complications in refactoring, it became clear that this is an issue which should be taken care of if striving for quick delivery with a sustainable pace.

Based on the above, it seems that one must follow the well-known practices of *managing an organizational change* to gain the managerial support and mitigate the resistance that might rise when introducing the goal of quickening the deployment pace. Several of these practices are already documented, for example, by Manns and Rising in [99]. Although the interviews did not bring out it as

such, the concept of DevOps is an organizational change, and must be treated as such. Business-wise, the customer might have their own processes which may be hard to adapt for quick delivery of software. For example, the customer might not have allocated time to acceptance testing in such rapid pace as the software development organization could produce new features. These kinds of problems have been discussed in length earlier, for example, in [65].

In addition, the business might set also different types of obstacles. The *business domain* seemed to restrict the development and delivery speed. For example, in the medical systems the safety-related standards caused significant delay in the propagation of the new features to the end users. In the control system domain, there might be downtimes and tight scheduling involved with deploying new features. In mobile applications, there might also be a third party involved in the distribution of the software, dictating some restrictions to the deployment. These might be impossible issues to solve completely, but still many benefits of the Continuous * practices can be reaped even if the pipeline does not extend all the way to the end user, or all changes to the code are not automatically pushed to deployment. The development organization may still work in the mode where they have a deployable software at hand all the time and the actual delivery is automated so, that when it is allowed to happen, the update will be instantaneous. Moreover, there has been some evidence that model-based design and having a plant model at hand may help in removing the need to test with real hardware-in-the-loop systems and to automate the tests to very high degree [100]—thus enabling the software development with Continuous Delivery practices.

Also, the companies craved for *fast feedback mechanisms* in all stages of the development, may it be, for example, communication between individuals or information on the results of the tests on the latest change. The need for quick ways to exchange information was reflected in the preference of instant messaging over face-to-face communication in the most trivial communication. It might be that the advice to avoid unproductive meetings [101] can be extended to, at least in some cases, avoiding a meeting even with two persons, as it may interfere with flow and introduces unnecessary communication, which might be harmful in fast-paced development. Also, the Lean principle of *visual control* [102] was popular in sharing knowledge in those companies which strived for fast development pace. The usual ways to implement this was via radiators, Kanban boards, and similar visual aids, see **Figure 13**. These reduce the amount of needed communication as the state of the software system and the deployment are visible to all developers with just a glance. In addition, automation also can replace communication, as there are less intermediate states where someone is doing manual work on the system. In heavily automated environment, the system has less states where it can be. For example, with manual deployment, there is a significant time when the system is updated with all its libraries, platforms etc. and it may be difficult to roll back all changes if some part of the update fails. However, if the update process is automated with some suitable technology, such as scripts, they can make sure that the system is either fully deployed online with a new version of the software, or previous working version is retained in use.

So to summarize, **Table 1** presents the key enablers for the increasing of the delivery speed. On the right side, the key enablers from the previous paragraphs are presented, and on the left side, the two most important key properties of the enabler are listed.

Key enabler	Key properties
Business domain	Domain-dependent regulations Complexity of the software
Automated deployment pipeline	Replicability Set up time
Fully automated testing	Size of testware Manual tests
Environment management	Similarity of environments Ease of configuration
Software quality	Metrics Legacy code
Organizational support	Additional work for infrastructure and testware Productivity losses due process changes
Feedback collection	Automation Visual control

Table 1 The key enablers of fast software delivery.

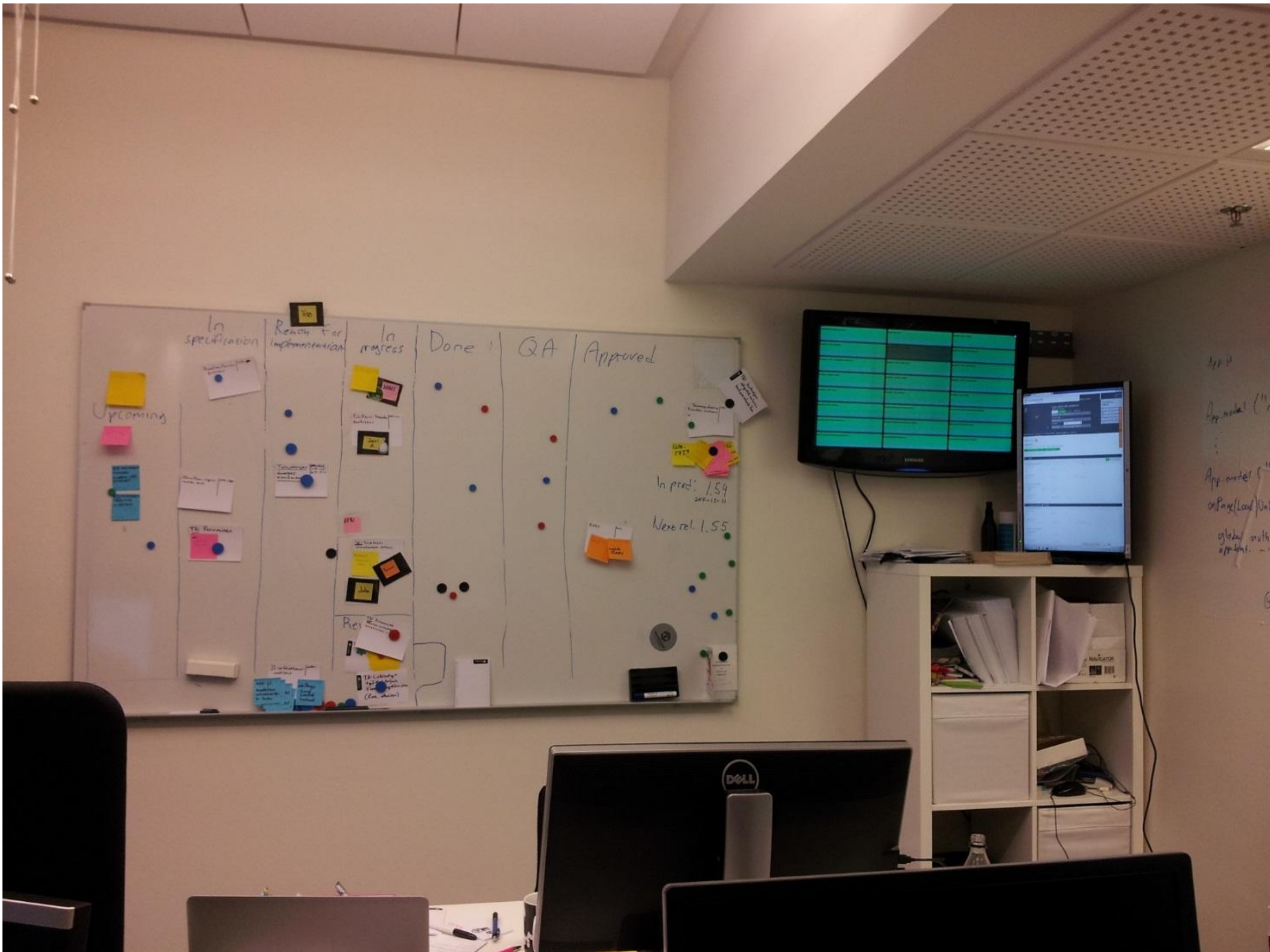


Figure 13 A Kanban board and radiators as means of visual control.

RQ2: What are the ramifications of increasing the delivery speed?

The interviewees mentioned as the benefits of the fast release cycle *rapid feedback*, *more frequent releases*, *improved quality and productivity*, *improved customer satisfaction*, *effort savings*, and, finally, closer *connection between development and operations*. This seemed to be the shared sentiment of all the interviewees, regardless of on what maturity level their company was. Most of the aforementioned effects are business effects, either adding value to the product or decreasing the cost of development due removing waste. Having the system always in shippable state and to promote all changes to production naturally leads to ability to have more frequent releases, which smoothens the flow and reduces the pre-release stress. Furthermore, *small fixes are easy* to incorporate in the running system if the deployment is not much of a hassle, and there are good automated tests. This may lead to better quality products and more satisfied customers. Also, system security can be improved as security holes are quickly blocked. However, as mentioned earlier, not all customers require or are able to accept faster releases. Related to this, if an update requires end user activity, it may interfere with the intended use of the software product, decreasing the satisfaction. Productivity is boosted and work effort saved as manual work is reduced and work is not wasted in

waiting to be deployed. It seems that the relay-race organization in companies where development and operations are separate suborganizations seems to waste time, and this can be mitigated with DevOps -type approach and automation.

Also, fast delivery gives the developers a heightened sense of accomplishment. This and the increased situational awareness because of the quick feedback and visual control improves also developer motivation.

More surprisingly, according to Publication II, fast delivery has profound effects on the how the business decisions can be made. Quick delivery allows *business experimentation* with actual software. This leads to ability to adhere to several of the Lean principles, such as minimum viable product, enabled by the ability to incrementally deploy just the crucial features, having sensible and actionable metrics allows to test different approaches to the end user requirements, and ultimately failing fast if the business idea is not worthy of spending any resources on it anymore.

Architecturally, it seems that technical debt is more visible problem in fast delivery, as the deployment speed is so hectic that there are no natural calm spots where it could be paid back without slowing the development pace. What could be swept under the rug earlier is now showing as the developers are struggling to keep the software releasable all the time. So, there is a definite need of *making technical debt visible for all involved parties*, so that all stakeholders can agree upon the suitable course of action. If it remains invisible, it is unlikely that the customer is willing to pay for refactoring. An often stated problem was that the common metrics were somewhat useless as a tool to justify spending effort on refactoring. This became evident in Publications IV and V, as several of the interviewed architects mentioned having faced problems while communicating the need of refactoring to their managers or the customer, who pays for the spent work. The developers said that their own gut feeling is considerably better indicator of bad code quality. On one hand, obeying the metrics blindly easily would lead to spend effort in the wrong places and, on the other hand, the real problematic parts of the code were self-evident to the experienced architects.

Even though the architects saw refactoring as a beneficial and necessary part of the work, they felt being pulled between producing new features as fast as possible and keeping the code quality so high that future development would be quick and easy. Also, incorporating the changes to the architecture requires *an agreed-upon workflow*. Furthermore, the workflows used in rapid development usually add some reviews as the high degree of automation easily leads to situation where quality assurance is almost fully automatic or on the responsibility of the developer. Thus, reviews add an additional quality gate. Nowadays, when review tools are easily integrated to version control system, every commit can be reviewed in such a short time that it does not hinder the development pace.

Also, there must be *architectural support for deploying very small changes*. As almost any feature will be bigger than just one commit, there is always a possibility of having non-working features in the system if continuous deployment is applied. To prevent this the architecture must have some

mechanism that will not show the work in progress to the end user. It can be achieved through feature switches and similar mechanisms.

Infrastructure as such does not change much due the increasing speed as such, but it naturally has to provide all the tools, environments or platforms, and the measurement points for feedback. One definite effect of this is that *more work must be spent on configuring systems, writing automated tests, setting up environments, and such preparatory work*. Thus, the spent effort becomes more front-loaded. This may lead to higher productization degree of the development infrastructures, as the effort may be saved by using similar and pre-configured development infrastructures where possible. Also, the preparation work must be taken into account when estimating the work effort for a new software project.

Also, when the amount of automation in the deployment pipeline grows, it becomes evident that even *automated tasks take some time*. A characteristic automated task is testing, and the time spent on running all the automated tests could trump the delivery pace from mere minutes to several hours. So, this clearly indicates that there still is a need of research in the field of optimizing the automated tests, even though several improvements have been proposed, such as test case prioritization [103]

RQ3: What is the state-of-the-practice regarding the speed of delivery practices in the companies?

The third research question was set to find out how the companies fare now regarding to their delivery speed. To assess this, the interviews included several metrics and characterizations which were related to the speed of delivery and the feedback. In addition, we used the Stairway to Heaven model as a means of self-assessment to map the maturity of the company. As metrics and characterizations, we used *the fastest possible time for a code change to propagate to production, the cycle time to potentially deployable software, whether the company uses an automatic chain to potentially deployable software, the actual cycle time to deployment,* and *whether the company used an automatic chain to deployment*.

As found out in Publication I, *no company had an automatic pipeline* all the way to deployment in a production environment. Thus, it seems that full-blown continuous deployment is yet to become a common practice. Only one company had even the pipeline automated to the potentially deployable software. It seems that the companies which were fastest based on the metrics and also on their self-assessed scale, shared some properties in common. First, it did not seem to matter if the company was a small or large one, but the operating domain seemed to dictate some restrictions, which hindered the delivery speed in some companies. Subsequently, most of the companies with *fast delivery cycle were operating the Web doma*in. The cycle times had huge differences, as the fastest company was able to potentially deploy several times a day, and the slowest ones could only potentially deploy in timeframes of weeks. Quickest one-line changes could take only minutes, whereas in some companies it could take weeks.

The reasons for long times were the restrictions in business, as discussed earlier. For example, application stores had to review all new versions, safety standard conformance assessments took time and so on. However, it was interesting to notice that these companies rarely strived for faster pace even in the portions which were completely in the company's own hands. It was not felt as necessary as the rest of the process would take so long anyways.

Furthermore, we studied the prevalence of tools in the companies, and it seemed that almost all companies had continuous integration tools, backlog management, build, unit testing, bug tracking, and version control systems in place. At the same time, artefact repositories and acceptance testing tools were uncommon in the company sample. However, these are seen as an important part of continuous delivery [4]. So, it seems that the companies *had tooling for continuous integration*, but only a few companies covered the parts for continuous deployment. Quite naturally, the companies whose toolchains were more automated and contained fewer manual steps were able to deploy software more rapidly. Of course, it is important to remember that domain also plays a role. For example, an embedded system does not benefit so much from automated user interface testing. Also, even though a tool is missing, it does not mean that the activity is completely forgotten. For example, acceptance testing could be done more conventionally by getting feedback from the user. Furthermore, even a very complete toolchain did not guarantee daily deploys. Other restrictions might still cause slower deploy pace.

4.2 Related research

This section presents some related fields of research. These include DevOps, SEMAT framework, post-agility, ChatOps, and Continuous Deployment intersected with architecting process. In addition, the included publications include more detailed related research in the scope of the published research.

4.2.1 DevOps

Nowadays, almost every company claims to be using Agile methods in their development, all way up to 94% of companies surveyed in [37]. However, a set of software development practices called DevOps has been claiming some foothold lately [104]. DevOps is a bit different from Agile methodologies, as it takes a strong stance on the organizational structure. Even the name DevOps suggests that one should not divide the organizational lines across the development and operational staff, but to allow close cooperation between these two. Furthermore, the cooperation is extended to the actual end users. To augment the cooperation, or if there is no easy access to the end users, real usage data can be gathered to elicit the requirements. As the requirements are understood, it is possible for the developers to implement the code conforming to them and to deploy the code all way to the end users. Thus, the software development process is transforming from pushing new installations

to the customer to a continuous and more pull-oriented way, where features are deployed as soon as they are implemented. This requires extensive automation and Continuous Delivery. The role of the pipeline is the automated manifestation of the process how software travels from the version control system to the user.

4.2.2 SEMAT

SEMAT (Software Engineering Method and Theory) is a framework describing software development, presented by Ivar Jacobson et al. [105].The SEMAT framework divides software development to seven core ideas, or kernels; namely opportunity, stakeholders, requirements, software system, work, team and the way of working. This approach is different from approach presented in this thesis as it focuses on the more concrete parts of the software development.

Way of working in the SEMAT framework can be seen as a synonym to the software process, but it also contains parts of the infrastructure. The software system harbours the software architecture. SEMAT can also be used in a certain way of assessing the parts of the software via so-called alphas, where the states of the requirements, software system, and the way of working are checked on six levels. Each state also sets objectives for improvement to get to that state.

4.2.3 Post-Agility

There has been some research on the topic of post-agile information system development. By post-agility, Baskerville et al. mean the next step after Agility, where the problems associated with agility are solved [106]. The problems are caused by the change of the context, where software is developed, and are answered by a change in the process. One huge problem in agility according to Baskerville et al. is the boundary between agile and plan-driven parts of the organization. They speculate that

> ” Instead the focus would shift towards proactively pursuing the dual goal of agility (i.e. fast responses to changing requirements, and frequent releases of valuable, working software) and alignment (of plans, people, and tasks) through a diversity of means, for example through (agile, plan-driven, and other types of) method components, and software tools as well as via new ways of organizing, specializing, communicating, managing relationships, etc.”

In their study, they identified several problems in the case companies. From these problems, they found a new software process. This process involves a solution set of architecture, components/reuse, estimation methods, QA & testing, parallel development, prototyping, and frequent releases. It also involves heavily customer in the process. In the end the result was acquiring both, speed and quality.

Although automated testing is not explicitly mentioned, it can be surmised that automating ever so crucial QA leads into better products in less time. So, it can be said that the all three aspects of software development are seen here as part of the solution to gain more speed and maintaining the

good quality of the software. It is only a natural progression that in the search for speed, automatable parts of the process become automated.

4.2.4 ChatOps

ChatOps [107] is a recent phenomenon, often attributed to GitHub[11]. It is a way to do DevOps as conversation-driven development. The whole development infrastructure is automated, and using standardized interfaces, the tools in the infrastructure are interfaced with a chat bot. Thus, the tool that is used for communication between developers, may it be Slack[12], HipChat[13], Flowdock[14], or something else, is also used as a DevOps tool which empowers the developers to operate all the tools, which allow them to build, deploy, provision, test, and monitor etc. the software. This approach has several benefits. First, the same tool which is used for the developer communication also acts as the tool for doing the operations. Thus, all what has been done is visible for other parties as well. This includes the commands, the warnings, notifications and alarms from the various stages of the development and monitoring infrastructure. This makes the development more transparent, and helps the developers to find more easily the root causes of the problems encountered as chat logs act as history of what has been done in the history. It also helps organizational learning as new developers can easily follow what the more experienced colleagues are doing.

To summarize, ChatOps can be considered an amalgamation of the development process to the development infrastructure, as the steps in the process and the organization become blurred in the chat tool. Basically, everything on the chat is either a command to a certain part of the infrastructure, or a way to transfer knowledge between human participants to facilitate collaboration. However, in ChatOps, the software architecture is not a part of the concept; it is merely a thing that can be discussed over in the chat.

4.2.5 Continuous Deployment and Architecting

A recent line of research has focused on the effects of Continuous Deployment practices to the architecture of the software [108]. This research aims to empirically explore the potential impact of CD practices to architecting process. The results seem to show that architecture plays a significant role in sufficiently and efficiently adopting CD, further strengthening the claims laid out in this thesis. This work may contribute in the future significantly to the body of knowledge that is related to this thesis.

---

[11] https://github.com
[12] https://slack.com/
[13] https://hipchat.com/
[14] https://www.flowdock.com/

4.3 Limitations of the research

The research presented in this thesis is subject to some limitations. These limitations stem from the selected research methods. As the main research method was case studies, the usual criticism for case studies is also applicable for this work. Even though case studies are promoted as being well-suited for software engineering research [19], its weakness can be low generalizability of the results [109]. When constructing a proper case study using interviews, one must carefully select the questions that cover the studied phenomena sufficiently with a wide range of questions. Furthermore, the case studies must be selected so that the representativeness of the sample will be satisfactory. Some selection bias afflicts the case selections in this study as discussed earlier in the section 1.4 about the research methods.

Otherwise, the validity of case studies can be assessed with a classification scheme by Yin [110]. Runeson and Höst [19] also recommend the use of this scheme in the software engineering research. In Yin's classification the validity of a research consists of four aspects: construct validity, internal validity, external validity, and reliability. Next we will examine the validity of this study through these four aspects.

First, construct validity in this reflects if the study really treats the phenomena that the researchers wanted to study. In essence, this is ensured by shared vocabulary and understanding of the context and the goal of the study between researchers and the interviewees. While doing interviews or surveys, it should be made sure that both parties understand the questions essentially in the same way. In the research conducted for this thesis, the threat to construct validity was mitigated with several means. First, the interview questions were constructed in collaboration of several researchers in order to avoid a single interpretation of the questions. Furthermore, questions were piloted to gain insights how interviewees understood them. In the interviews, there were at least two researchers present, so it was possible to give clarifications to most ambivalent questions and these problematic questions could be modified to be more understandable to subsequent interviews. Additionally, the interviews were recorded, and notes were taken by the researchers in order to get evidence and data as faithfully as possible. This helped the researchers to establish a chain of events and to compare multiple sources of evidence with relative ease. The thematic analysis method helped significantly in identifying the common themes and concepts in the interviews.

Regarding the internal validity of this research, it is of utmost importance to understand the real causal relationships of the examined phenomena, i.e. to establish a non-spurious causal relationship. Typically, internal validity is threatened if the researchers assume a causal relationship between two factors and are oblivious to presence of a third factor which affects the relationship, or is the real cause of the effect. As the studied phenomena is complex and involves a plethora of variables, it must be made certain that the control variable is identified in isolation from the other variables. In this research, the most important control variable was the speed of software delivery, and to isolate

this from the other, maybe dependent, variables, such as company size, domain, or other possible factors, the case selection was purposefully made so that as the sample companies would have as diverse backgrounds and domains as possible. Thus, we hoped to be able to distinguish the effects of delivery speed from the other factors. However, all studied companies were Finnish software intensive companies, so this might have affected the results. Nonetheless, the findings are in line with previous studies, so internal validity seems not be threatened by this. Internal validity also affects explanatory research more than the descriptive research, such as the one conducted in this thesis.

Because of the selected method of in-depth interviews and open ended questions, the sample size had to be limited. Depending on the study, the number of interviews varied in between of 11 to 15. This is rather small sample, and the generalizability of the results might be threatened because of this. The possibility of selection bias has been discussed earlier. However, the personal interviews taking from one up to three hours also allowed the researchers and the interviewees to have in-depth discussions on the research topic and gave the researchers possibility to understand the studied phenomena better.

Concerning the generalizability, external validity also covers the ability of others to conduct a similar study and to arrive to similar results in the established domain for generalization. Reliability involves the repeatability of the operations done in the case study. The replicability of the interviews is rather poor, as every interview is a unique event. However, we as researchers, to aid the repeatability have included a portion to every included publication describing how the research was conducted. Thus, it is possible to follow the given protocol and to carry out a similar study. The actual questions (in Finnish) can be acquired from the researchers. Regarding the external validity, the applicability of the results in organizations should be rather high, as the results are presented in rather abstract level. Due to the low number of cases and the fact that no two companies are ever the same, and even the same company evolves as time passes, we have mostly relied in analytical generalization rather than statistical tools. Thus, no concrete solutions can be given, but these results act as a basis which can be adapted to the chosen organization’s own context. Thus, the results how to improve delivery speed and what the ramifications of such improvement will be are more of a description what has happened in other companies, and are a way to reflect what would happen in the readers own company.

4.4 Future work

As said earlier, this thesis contributes to the existing body of knowledge of software engineering, especially when the context is increasing the delivery pace by adopting the practices such as DevOps and Continuous Deployment. The presented PAI model of the software processes, the development infrastructure, and the structure of the produced artefact itself, the software architecture,

is derived from a series of interviews and surveys, and creates a new model to be used as a tool in future theory-forming research.

The logical next step from this research would be combining the qualitative data from the industry with a more quantitative data from the software development. The first steps on this direction have already been taken with the data collection and visualization presented by Mattila et al. [84] and an older article by Lehtonen et al. [111].

There also has been a parallel line of research, which aims at performing systematic literature research on the current state of the art presented in the literature [112]. It would be interesting to combine the interview results forming this thesis with an extensive systematic literature review to see how the research and industrial practice differ from each other.

4.5 Introduction to included publications

In this section, we give a short introduction to the publications included in this thesis. The contributions of the authors have been differentiated in the section List of publications.

Publication I, by Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T., and called "*The Highways and Country Roads to Continuous Deployment*" was published in 2015 in *IEEE Software*, vol.32, no. 2. The published article analyses a set of interview data collected from 15 Finnish software company representatives. The focus of the interviews was in Continuous Deployment practices and their adaptation in the companies. The companies were classified according to their delivery speed using self-assessment by the interviewee. The classification helped us to analyse better what kinds of problems and enablers the companies face in the relation of their current delivery speed. The main findings are that no company had adopted a fully automated deployment pipeline, and, perhaps surprisingly, not all companies see very fast delivery pace possible or even desirable. On one hand, the main perceived benefits of the fast release cycle were faster feedback, more frequent releases, improved quality and productivity, improved customer satisfaction, effort savings, and, finally, closer connection between development and operations. On the other hand, the perceived main obstacles were resistance to change, customer preferences, domain constraints, developer trust and confidence, legacy code, the size of the software, different development and production environments, and the difficulties with manual and non-functional testing.

Next, **Publication II**, by Leppänen, M., Kilamo, T., & Mikkonen, T. "*Towards Post-Agile Development Practices through Productized Development Infrastructure*" published in the Proceedings of 2nd IEEE/ACM International Workshop on Rapid Continuous Software Engineering (RCoSE 2015) describes a single case study. The research revolves around the benefits of a completely continuous delivery workflow. In the paper, one case is described in terms of infrastructure and feedback loops.

Furthermore, one key finding is that the infrastructure needs to have short set-up times to allow business experimentation and adherence to Lean principles –including minimum viable product, sensible and actionable metrics, continuous deployment, and failing fast.

Publication III by Kilamo, T., Leppänen, M., & Mikkonen, T., "*The Social Developer: Now, Then, and Tomorrow*" was published in Proceedings of the 7th International Workshop on Social Software Engineering, 2015. The article focuses on the communication patterns and what kinds of communication mediums are preferred by the software developers. The research was conducted as an online survey. The main finding is that the software developers tend to prefer fast feedback mechanisms, and, quite surprisingly, in some cases, instant messaging over face-to-face communication. Also, making things visible was popular, and asynchronous communication seemed to prevail over synchronous methods.

Next, **Publication IV**, "*Refactoring–a Shot in the Dark?*" by Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte, O. Tuovinen, A.-P., & Männistö, T. and published in *IEEE Software,* vol.32, no.6, in 2015 discusses the refactoring practices as seen by 12 senior software architects or developers, whom we interviewed for the study. In the interviews, it became evident that refactoring as a practice was considered as an invaluable and unavoidable part of the software development. However, the practitioners faced problems while trying to justify it to their managers or the customer, who pays for the spent work. There was a clear sentiment of the developers being caught in a cross-fire, where they have to produce new features as fast as possible, but at the same time, try to keep the code tidy so that the future development would still be easy. The main benefits of the refactoring were, according to the interviewees, easier future development, increased understandability, possibility for code re-use, improved quality attributes, and even boosting morale and motivation. Also, some risks were seen, but deemed small compared to the benefits. These risks included breaking something already working, causing externally visible changes, making the code quality worse regardless of the best intent, and wasting time and effort. One clearly stated problem was that the widely used metrics did not seem to help in justifying refactoring as the developers shared a strong impression that their own gut feeling is considerably better indicator of bad code quality.

Publication V, by Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., & Lehtonen, T., named "*Decision-Making Framework for Refactoring*", and published in IEEE 7th International Workshop on Managing Technical Debt (MTD 2015) handles the concept of refactoring from a slightly different angle. The article investigates how the professionals make refactoring decisions. The data was obtained by interviewing representatives from three companies. In addition, one of the cases was backed up with version control data, as it revealed the actual workflow how a certain refactoring progressed. As a result, a decision-making framework is presented.

Finally, **Publication VI**, "*Improving the Delivery Cycle: A Multiple-Case Study of the Toolchains in Finnish Software Intensive Enterprises*" by Mäkinen, S., Leppänen, M., Kilamo, T., Mattila, A.-L.,

Laukkanen, E., Pagels, M., & Männistö, T. published in Information and Software Technology volume 80, December 2016. The article extends the data and analysis of Publication I by focusing on the reported tool chains in the companies. In the article, we studied what tools are used in the software industry, and if they have any implications on the speed of delivery. One of the key findings was that if the toolchain was highly automated and contained only few manual steps, the deployment speed was significantly faster. Other way round, several manual steps indicates problems in the delivery speed.

4.6 Concluding remarks

To summarize this thesis and the key findings presented in it, we conclude this thesis with some final observations. This thesis presented insights how to improve the delivery speed of software systems, and under what circumstances it can be achieved. Furthermore, we shed light on how the increasing delivery speed affects the organization, its selection of tools and the developed software artefacts.

In addition, the phenomena related to the delivery speed were examined in the context of a model explaining the relationships between the software development processes, the infrastructure of the tools and environments, and the architecture of the software system itself. This PAI model can be used to examine the effects of the delivery speed on these fundamental aspects of software development.

As researchers, we intended the publications included in this thesis to help companies to better understand how to increase their delivery speed and to gain most of it, while avoiding the pitfalls which may be related to this hastening of the development.

Furthermore, the thesis can be used as a basis for the scientific community to continue the research on the themes presented here. Especially, the PAI model can help in theory building related to the effects of fast delivery speed, Continuous * practices, and similar phenomena. It can be compared to the traditional project management triangle, where one decides on the trade-offs between the triple constraints of schedule, scope, and cost and struggles to maintain good quality. In PAI model, one must take care of the three aspects of the software development in order to hasten the delivery pace. The increased delivery speed means that the delivered increments are getting smaller, and this requires that all sides of the triangle to be ready for this. If one of the sides is neglected, then the delivery speed will suffer. This also means that the software development work itself turns into developing new features and improving the quality of the code, but on the same time more work is spent in the building of the scaffolding that is the sides of the PAI triangle.

References

[1] M. Fowler, “ContinuousDelivery,” 30 May 2013. [Online]. Available: http://martinfowler.com/bliki/ContinuousDelivery.html. [Accessed 2016 April 25].

[2] L. Chen, “Continuous Delivery: Huge Benefits, but Challenges Too,” *IEEE Software,* vol. 32, pp. 50-54, March 2015.

[3] M. Fowler, May 2006. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [Accessed 1 December 2015].

[4] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley Professional, 2010.

[5] J. Humble, “The Case for Continuous Delivery,” 13 February 2014. [Online]. Available: https://www.thoughtworks.com/insights/blog/case-continuous-delivery. [Accessed 16 January 2016].

[6] L. Bass, I. Weber and L. Zhu, DevOps: A Software Architect's Perspective, 1st ed., Addison-Wesley Professional, 2015.

[7] B. Fitzgerald and K.-J. Stol, “Continuous software engineering and beyond: trends and challenges,” *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE),* pp. 1-9, 2014.

[8] E. W. Dikjstra, “On the role of scientific thought,” *Selected writings on Computing: A Personal Perspective,* pp. 60-66, 1982.

[9] M. Imail, Kaizen: The Key to Japan's Competitive Success, New York: Random House, 1986.

[10] T. Osada, The 5S's: Five keys to a Total Quality Environment, Asian Productivity Organization, 1995.

[11] A. Rehn, T. Palmborg and P. Boström, "The Continuous Delivery Maturity Model," 6 February 2013. [Online]. Available: http://www.infoq.com/articles/Continuous-Delivery-Maturity-Model. [Accessed 1 December 2015].

[12] Eficode, "DevOps Quick Guides," 2015. [Online]. Available: http://devops-guide.instapage.com/. [Accessed 4 December 2015].

[13] Forrester Research, Inc., "Continuous Delivery: A Maturity Assesment Model," March 2013. [Online]. Available: http://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html. [Accessed 1 December 2015].

[14] H. Holmström Olsson, H. Alahyari and J. Bosch, "Climbing the "Stairway to Heaven" - A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software.," in *38th Euromicro Conference on Software Engineering and Advanced Applications*, Cezme, 2012.

[15] W. W. Royce, "Managing the Development of Large Software Systems," in *Technical Papers of Western Electronic Show and Convention (WesCon)*, Los Angeles, USA, 1970.

[16] Scrum Inc., "The Scrum Framework," [Online]. Available: https://www.scruminc.com/scrum-framework/. [Haettu 10 January 2016].

[17] W. M. Trochim, The Research Methods Knowledge Base, 2nd Edition, Cincinnati, OH: Atomic Dog Publishing, 2000.

[18] K. M. Eisenhardt, "Building theories from case study research," *The Academy of Management Review ,* vol. 14, no. 4, pp. 532-550, 1989.

[19] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empirical Software Engineering,* vol. 14, no. 2, pp. 131-, 2009.

[20] G. Guest, Applied thematic analysis, Thousand Oaks, California: Sage, 2012.

[21] J. Saldana, The coding manual for qualitative researchers, Thousand Oaks, CA: Sage, 2009.

[22] S. K. Card, T. P. Moran and A. Newell, The Psychology of Human-Computer Interaction, Hillsdale, NJ, USA: Lawrence Erlbaum, 1983.

[23] B. A. Kitchenham and S. L. Pfleeger, “Personal opinion surveys,” in *Guide to Advanced Empirical Software Engineering*, Springer, 2008, pp. 63-92.

[24] R. Likert, “A Technique for the Measurement of Attitudes,” *Archives of Psychology,* vol. 140, p. 1–55, 1932.

[25] C. Wohlin and A. Aurum, "Towards a decision-making structure for selecting a research design in empirical software engineering," *Empirical Software Engineering,* vol. 20, no. 6, pp. 1427-1455, 2015.

[26] J. Collis and R. Hussey, Business Research: A Practical Guide for Undergraduate and Postgraduate Students, Palgrave Macmillan, 2009.

[27] H. K. Klein and M. D. Myers, "A set of principles for conducting and evaluating interpretive field studies in information systems," *MIS Quarterly - Special issue on intensive research in information systems,* vol. 23, no. 1, pp. 67-93, 1999.

[28] A. Fuggetta, "Software process: a roadmap," in *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, 2000.

[29] H. J. Johansson, P. McHugh, A. J. Pendlebury and W. A. Wheeler, Business Process Reengineering: Breakpoint Strategies for Market Dominance, Wiley, 1993.

[30] P. Naur and B. Randell, "Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, Oct 1968," Scientific Affairs Division, NATO, Brussels, Belgium, 1969.

[31] D. L. Parnas and P. C. Clements, “A rational design process: How and why to fake it,” *IEEE Transactions on Software Engineering,* vol. 12, no. 2, pp. 251 - 257, 1986.

[32] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, “Principles behind the Agile Manifesto,” 2001. [Online]. Available: http://agilemanifesto.org/.

[33] M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003.

[34] International Software Testing Qualifications Board, “Certified Tester Foundation Level Syllabus,” 2011.

[35] Internal Electrotechnical Commission, ”IEC 61508-:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General requirements,” 2010.

[36] K. Schwaber, “Scrum Development Process,” in *Proceedings of the 10th annual ACM conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1995.

[37] VersionOne, “9th Annual State of Agile Survey,” 2015. [Online]. Available: https://www.versionone.com/pdf/state-of-agile-development-survey-ninth.pdf. [Accessed 19 November 2015].

[38] K. Schwaber and M. Beedle, Agile Software Development with Scrum, Upper Saddle River, NJ: Prentice-Hall, 2001.

[39] S. McIntosh, B. Adams and A. E. Hassan, “The evolution of Java build systems,” *Journal of Empirical Software Engineering,* vol. 17, no. 4-5, pp. 578-608, 2012.

[40] C. Prasad and W. Schulte, “Taking control of your engineering tools,” *Computer,* vol. 46, no. 11, pp. 63-66, 2013.

[41] ISO, *ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description,* 2011.

[42] R. Kazman and L. Bass, Toward Deriving Software Architectures From Quality Attributes, Pittsburgh: Software Engineering Institute, Carnegie-Mellon University, 1994.

[43] ISO / IEC, “Software engineering -- Product quality -- Part 1: Quality model,” 2001.

[44] S. McConnell, Code Complete, Microsoft Press, 1993.

[45] W. Cunningham, “The WyCash portfolio management system,” in *OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, New York, NY, USA, 1992.

[46] M. Fowler, “Internal And External Quality,” [Online]. Available: http://c2.com/cgi/wiki?InternalAndExternalQuality. [Accessed 24 December 2015].

[47] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, Addison-Wesley Professional, 2012.

[48] M. Fowler, “TechnicalDebtQuadrant,” 14 October 2009. [Online]. Available: http://martinfowler.com/bliki/TechnicalDebtQuadrant.html. [Accessed 24 December 2015].

[49] P. Abrahamsson, M. A. Babar and P. Kruchten, “Agility and architecture: Can they coexist?,” *IEEE Software,* vol. 27, no. 2, pp. 16-22, 2010.

[50] S. Newman, Building Microservices: Designing Fine-grained Systems, O'Reilly Media, 2015.

[51] W. P. Stevens, G. J. Myers and L. L. Constantine, “Structured Design,” *IBM Systems Journal,* vol. 13, no. 2, pp. 115-139, 1974.

[52] R. N. Charette, "This Car Runs on Code," *IEEE Spectrum,* 2009.

[53] U. Eliasson, A. Martini, R. Kaufmann and S. Odeh, “Identifying and Visualizing Architectural Debt and Its Efficiency Interest in the Automotive Domain: A Case Study,” in *Seventh International Workshop on Managing Technical Debt (MTD2015)*, Bremen, Germany, 2015.

[54] M. W. Maier and E. Rechtin, The Art of Systems Architecting, CRC Press, 2009.

[55] V.-P. Eloranta, J. Koskinen, M. Leppänen and V. Reijonen, Designing Distributed Control Systems: A Pattern Language Approach, Wiley, 2014.

[56] G. Forsum, “Backend infrastructure at Spotify,” 15 march3 2013. [Online]. Available: https://labs.spotify.com/2013/03/15/backend-infrastructure-at-spotify/. [Accessed 6 July 2016].

[57] T. Ohno, Toyota Production System: Beyond Large-Scale Production, Cambridge, MA: Productivity Press, 1988.

[58] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw-Hill Education, 2014.

[59] M. A. Cusumano and R. W. Selby, “How Microsoft Builds Software,” *Communications of the ACM,* vol. 40, no. 6, pp. 53-61, June 1997.

[60] K. Olsson and E.-A. Karlsson, "Daily Build - The Best of Both Worlds: Rapid Development and Control," Swedish Engineering Industries, 1999.

[61] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional, 2000.

[62] W. E. Deming, The New Economics for Industry, Government, Education, Cambridge, MA.: MIT Press, 1993.

[63] A. Maurya, Running Lean: Iterate from Plan A to a Plan That Works, O'Reilly Media, Inc, 2012.

[64] P. Kruchten, “Software Architecture and Agile Software Development—A Clash of Two Cultures?,” in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering*, 2010.

[65] G. G. Claps, R. B. Svensson and A. Aurum, “On the journey to continuous deployment: technical and social challenges along the way,” *Information and Software Technology ,* vol. 57, pp. 21-31, 2015.

[66] T. Dingsøyr and C. Lassenius, "Emerging themes in agile software development: Introduction to the special section on continuous value delivery," *Information and Software Technology,* vol. 77, pp. 56-60, 2016.

[67] H. Holmström Olsson and J. Bosch, "From Requirements To Continuous Re-priorization of Hypotheses," in *2016 International Workshop on Continuous Software Evolution and Delivery*, 2016.

[68] P. Bosch-Sijtsema and J. Bosch, "User Involvement throughout the Innovation Process in High-Tech Industries," *Product Innovation Management,* vol. 32, no. 5, pp. 793-807, 2014.

[69] J. P. Womack and D. T. Jones, Lean Thinking: Banish Waste and Create Wealth in Your Corporation, Productivity Press, 2003.

[70] M. E. Conway, “How Do Committees Invent?,” *Datamation,* 1968.

[71] M. Cataldo, J. D. Herbsleb and K. M. Carley, “Socio-technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity,” in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, Kaiserslautern, Germany, 2008.

[72] J. O. Coplien and N. B. Harrison, Organizational Patterns of Agile Software Development, Prentice Hall, 2004.

[73] T. H. Davenport, "Saving IT's Soul: Human-Centered Information Management," *Harvard business review,* vol. 72, 1994.

[74] T. Dingsøyr and H. van Vliet, "Introduction to Software Architecture and Knowledge Management," in *Software Architecture Knowledge Management*, Springer, 2009, pp. 1-17.

[75] M. T. Hansen, N. Nohria and T. J. Tierney, "What’s Your Strategy for Managing Knowledge?," *Harvard Business Review,* vol. 77, no. 2, pp. 106-116, 1999.

[76] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[77] P. Bell, "Refactoring Without Good Tests," Code Climate, 5 December 2013. [Online]. Available: http://blog.codeclimate.com/blog/2013/12/05/refactoring-without-good-tests/. [Accessed 20 July 2016].

[78] P. Kruchten, R. L. Nord and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” *IEEE Software,* vol. 29, no. 06, pp. 18-21, Nov.-Dec. 2012.

[79] A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley Professional, 2004.

[80] M. V. Mäntylä ja C. Lassenius, ”Subjective evaluation of software evolvability using code smells: An empirical study,” *Journal of Empirical Software Engineering,* osa/vuosik. 11, nro 3, pp. 395-431, 2006.

[81] M. V. Mäntylä and C. Lassenius, "Drivers for software refactoring decisions," in *ISESE '06 Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, 2006.

[82] H. T. Johnson, "Lean Dilemma: Choose System Principles or Management Accounting Controls—Not Both," in *Lean Accounting: Best Practices for Sustainable Integration*, Hoboken, NJ, John Wiley & Sons, 2012.

[83] E. Bouwers, J. Visser and A. van Deursen, "Getting What You Measure: Four common pitfalls in using software metrics for project management," *Communications of the ACM,* vol. 55, no. 7, pp. 54-59, 2012.

[84] A.-L. Mattila, T. Lehtonen, H. Terho, T. Mikkonen and K. Systä, "Mashing Up Software Issue Management, Development, and Usage Data," in *2nd IEEE/ACM}International Workshop on Rapid Continuous Software (RCoSE 2015)*, Florence, Italy, 2015.

[85] D. A. Tamburri, P. Kruchten, P. Lago and H. van Vliet, "What is social debt in software engineering?," in *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on* , San Francisco, CA, 2013.

[86] L. Chen, "Towards Architecting for Continuous Delivery," in *12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Montreal, QC, Canada, 2015.

[87] S. Bellomo, N. Ernst, R. Nord and R. Kazman, “Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail,” in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Atlanta, GA, 2014.

[88] M. Coram and S. Bohner, "The Impact of Agile Methods on software project management," in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)* , 2005.

[89] T. Lehtonen, S. Suonsyrjä, T. Kilamo and T. Mikkonen, “Defining Metrics for Continuous Delivery and Deployment Pipeline,” in *Proceedings of the 14th Symposium on Programming Languages and Software Tools*, Tampere, 2015.

[90] SyncDev, “Minimum Viable Product,” SyncDev. [Online]. [Accessed 1 February 2016].

[91] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen and M. Englund, "CoRED: browser-based Collaborative Real-time Editor for Java web

applications," in *CSCW '12 Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, New York, NY, USA, 2012.

[92] J. MSV, "The Master Plan Behind Amazon's Acquisition of Cloud9 IDE," Forbes, 18 July 2016. [Online]. Available: http://www.forbes.com/sites/janakirammsv/2016/07/18/the-master-plan-behind-amazons-acquisition-of-cloud9-ide/#83ee62228dba. [Accessed 11 August 2016].

[93] P. Kruchten, P. Lago and H. van Vliet, "Building Up and Reasoning About Architectural Knowledge," in *QoSA'06 Proceedings of the Second international conference on Quality of Software Architectures*, 2006.

[94] K. Morris, Infrastructure as Code: Managing Servers in the Cloud, O'Reilly Media, 2016.

[95] S. Suonsyrjä and T. Mikkonen, "Designing an Unobtrusive Analytics Framework for Monitoring Java Applications," in *25th International Workshop on Software Measurement and 10th International Conference on Software Process and Product Measurement, IWSM-Mensura 2015, Kraków, Poland, October 5–7, 2015, Proceedings*, 2015.

[96] J. Bosch, "Speed, Data, and Ecosystems: The Future of Software Engineering," *IEEE Software,* vol. 33, no. 1, pp. 82-88, 2016.

[97] M. Fowler, "Continuous Integration," 10th September 2000. [Online]. Available: http://martinfowler.com/articles/continuousIntegration.html. [Accessed 21 December 2016].

[98] P. Bregman, "The Right Way to Hold People Accountable," *Harvard Business Review,* 2016.

[99] M. L. Manns ja L. Rising, Fearless Change: Patterns for Introducing New Ideas, Addison-Wesley, 2004, p. 273.

[100] U. Eliasson, R. Heldal, J. Lantz and C. Berger, "Agile Model-Driven Engineering in Mechatronic Systems - An Industrial Case Study," in *Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014*, Valencia, Spain, 2014.

[101] M. C. Mankins, "Stop Wasting Valuable Time," *Harvard Business Review,* September 2004.

[102] J. K. Liker, The Toyota Way: 14 management principles from the world's greatest manufacturer, McGraw-Hill Professional, 2004.

[103] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality,* vol. 21, no. 3, pp. 445-478, 2013.

[104] P. Debois, "Devops: A Software Revolution in the Making?," *Cutter IT Journal,* vol. 24, no. 8, pp. 3-5, 2011.

[105] I. Jacobson, P. W. Ng, P. McMahon, I. Spence and S. Lidman, "The Essence of Software Engineering: The SEMAT Kernel," *Queue,* vol. 10, no. 10, 2012.

[106] R. Baskerville, J. Pries-Heje and S. Madsen, "Post-agility: What follows a decade of agility?," *Information and Software Technology,* vol. 53, no. 5, pp. 543-555 , 2011.

[107] T. Waits, "ChatOps in the DevOps Team," Software Engineering Institute, 29 January 2015. [Online]. Available: https://insights.sei.cmu.edu/devops/2015/01/chatops-in-the-devops-team.html. [Accessed 21 April 2016].

[108] M. Shahin, M. A. Babar and L. Zhu, "The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives," in *ESEM'16*, Ciudad Real, Spain, 2016.

[109] B. Flyvbjerg, "Five Misunderstandings About Case-Study Research," *Qualitative Inquiry,* vol. 12, no. 2, pp. 219-245, 2006.

[110] R. K. Yin, Case Study Research: Design and Methods, 5th ed., SAGE Publications, Inc, 2013.

[111] T. Lehtonen, V.-P. Eloranta, M. Leppänen and E. Isohanni, "Visualizations as a Basis for Agile Software Process Improvement," in *20th Asia-Pacific Software Engineering Conference, APSEC 2013,* , Ratchathewi, Bangkok, Thailand, 2013.

[112] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner and M. Oivo, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software,* 2016.

ORIGINAL PAPERS

I

THE HIGHWAYS AND COUNTRY ROADS TO CONTINUOUS DEPLOYMENT

By

Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T., 2015

IEEE Software vol. 32, no. 2, p. 64-72

© 2015 IEEE. Reprinted with permission.

II

TOWARDS POST-AGILE DEVELOPMENT PRACTICES THROUGH PRODUCTIZED DEVELOPMENT INFRASTRUCTURE

By

Leppänen, M., Kilamo, T., & Mikkonen, T., 2015

Proceedings of 2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering (RCoSE), p. 34-40

© 2015 IEEE. Reprinted with permission.

THE SOCIAL DEVELOPER: NOW, THEN, AND TOMORROW

By

Kilamo, T., Leppänen, M., & Mikkonen, T., 2015

Proceedings of the 7th International Workshop on Social Software Engineering 2015, (SSE 2015) p. 41-48

Reproduced with a kind permission by ACM.

IV

REFACTORING–A SHOT IN THE DARK?

By

Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte, O. Tuovinen, A.-P. & Männistö, T., 2015

IEEE Software vol. 32, no. 6, p. 62-70

© 2015 IEEE. Reprinted with permission.

V

DECISION-MAKING FRAMEWORK FOR REFACTORING

By

Leppänen, M., Lahtinen, S., Kuusinen, K., Mäkinen, S., Männistö, T., Itkonen, J., Yli-Huumo, J., & Lehtonen, T., 2015

Journal of IEEE 7th International Workshop on Managing Technical Debt (MTD 2015), p. 61-68

© 2015 IEEE. Reprinted with permission.

VI

IMPROVING THE DELIVERY CYCLE: A MULTIPLE-CASE STUDY OF THE TOOLCHAINS IN FINNISH SOFTWARE INTENSIVE ENTERPRISES

By

Mäkinen, S., Leppänen, M., Kilamo, T., Mattila, A.-L., Laukkanen, E., Pagels, M., & Männistö, T., 2016

Information and Software Technology vol. 80, p. 1339-1351

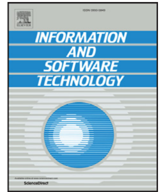
Reproduced with a kind permission by Elsevier.

Information and Software Technology 80 (2016) 175–194

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Improving the delivery cycle: A multiple-case study of the toolchains in Finnish software intensive enterprises

Simo Mäkinen[a,*], Marko Leppänen[b], Terhi Kilamo[b], Anna-Liisa Mattila[b], Eero Laukkanen[c], Max Pagels[a], Tomi Männistö[a]

[a] Department of Computer Science, University of Helsinki, P.O. 68 (Gustaf Hällströmin katu 2b), FI-00014 University of Helsinki, Finland
[b] Department of Pervasive Computing, Tampere University of Technology, Korkeakoulunkatu 1, FI-33720 Tampere, Finland
[c] Department of Computer Science and Engineering, Aalto University School of Science, P.O. Box 19210, FI-00076 Aalto, Finland

ARTICLE INFO

Article history:
Received 16 December 2015
Revised 12 August 2016
Accepted 7 September 2016
Available online 8 September 2016

Keywords:
Continuous deployment
Continuous delivery
Software development tools
Deployment pipeline
Agile software development

ABSTRACT

Context: Software companies seek to gain benefit from agile development approaches in order to meet evolving market needs without losing their innovative edge. Agile practices emphasize frequent releases with the help of an automated toolchain from code to delivery.

Objective: We investigate, which tools are used in software delivery, what are the reasons omitting certain parts of the toolchain and what implications toolchains have on how rapidly software gets delivered to customers.

Method: We present a multiple-case study of the toolchains currently in use in Finnish software-intensive organizations interested in improving their delivery frequency. We conducted qualitative semi-structured interviews in 18 case organizations from various software domains. The interviewees were key representatives of their organization, considering delivery activities.

Results: Commodity tools, such as version control and continuous integration, were used in almost every organization. Modestly used tools, such as UI testing and performance testing, were more distinctly missing from some organizations. Uncommon tools, such as artifact repository and acceptance testing, were used only in a minority of the organizations. Tool usage is affected by the state of current workflows, manual work and relevancy of tools. Organizations whose toolchains were more automated and contained fewer manual steps were able to deploy software more rapidly.

Conclusions: There is variety in the need for tool support in different development steps as there are domain-specific differences in the goals of the case organizations. Still, a well-founded toolchain supports speedy delivery of new software.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Today, software development has evolved into a high-paced business tackling challenges of demanding contexts, changing customer requirements and the increased need for short development cycles and fast time-to-market. At the same time, software development organizations should be able to produce high-quality software while maintaining an innovative edge.

In a quest to answer these challenges, software development organizations have turned to agile practices such as Kanban [1], Scrum [2] and Extreme Programming (XP) [3] due to their promise of speed of delivery, ability to adapt to changing requirements and improved software quality. The leading principle of agile is to "satisfy the customer through early and continuous delivery of valuable software" [4] . Some companies, such as Amazon, Facebook and Google [5] being some of the most famous examples, have adopted agile practices to the extent where software is released to the customers continuously in a rapid pace. Still, many are yet to adopt this kind of agility to its fullest.

Continuous deployment refers to the practice of deploying new software and features to the customers as it gets developed instead of making less frequent timed releases [6]. The one mandatory aspect of continuous deployment is a pipeline without any interrup-

* Corresponding author.
E-mail addresses: simo.v.makinen@helsinki.fi, simo.makinen@cs.helsinki.fi (S. Mäkinen), marko.leppanen@tut.fi (M. Leppänen), terhi.kilamo@tut.fi (T. Kilamo), anna-liisa.mattila@tut.fi (A.-L. Mattila), eero.laukkanen@aalto.fi (E. Laukkanen), max.pagels@alumni.helsinki.fi (M. Pagels), tomi.mannisto@cs.helsinki.fi (T. Männistö).

http://dx.doi.org/10.1016/j.infsof.2016.09.001
0950-5849/© 2016 Elsevier B.V. All rights reserved.

tions all the way from code to delivery. Hence, in order to achieve continuous deployment, a software company must set up an automated toolchain and establish tool-related workflows [7]. A properly documented and easy to set up toolchain enables a company to get development going rapidly and makes speedy deployment of new software to the customers possible.

In this paper, we investigate the current state of the practice in Finnish software organizations regarding their workflows and toolchains. We focus on the current toolchains organizations have in place and how automated the different stages of development are. In addition, we investigate why tooling is not used for some phases and does the missing tooling have effect on the speed of delivery. We conducted interviews with the representatives of 18 Finnish software organizations. The organizations were of different sizes and represented several business domains to get a comprehensive view on the phenomenon.

The rest of the paper is structured as follows: Section 2 discusses the background and related work. Section 3 introduces the research methodology. Section 4 describes the modern software development toolchain and its elements, constructed in this study. Section 5 gives the results from the conducted interviews. Section 6 draws comparisons with the maturity of the toolchain and the deployment times. The results introduced in Sections 5 and 6 are then discussed in Section 7 where the validity threats for the study are also explored together with future directions. Section 8 summarizes the findings and concludes the paper.

2. Background

During the past two decades software development has transformed from a rigid and documentation-heavy waterfall approach to lean and agile development models. Lately, the cross-functional teams and the ability to rapidly deliver new software to end-users have been noted as especially important practices. The concept of DevOps, where developers and operations work closely together, is fused together with continuous deployment into the day-to-day software work. This section presents the lean principles from which the modern-day software development practices have been partly derived from, describes the terminology related to the continuous forms of software engineering, and positions the article against the backdrop of related work.

2.1. Lean and DevOps

Modern software development relies on a versatile set of agile practices which in turn have been heavily influenced by lean development principles. Lean itself originates from manufacturing [8] while the concept of lean software development was introduced by Poppendieck and Poppendieck [9]. The Poppendiecks' base lean software development on seven key principles: (1) Eliminate waste, (2) Amplify learning, (3) Decide as late as possible, (4) Deliver as fast as possible, (5) Empower the team, (6) Build integrity in, and (7) See the whole. Continuous software engineering [10] is an emerging trend taking a holistic approach on continuous development – incorporating these seven lean principles to the development.

One development approach that relies on teams' ability to deploy features continuously in an automated manner, and is gaining increasing popularity, is DevOps [11,12] – a portmanteau of developers (Dev) and operations (Ops). In DevOps, developers and information technology professionals (operations) work together as a team instead of being separated in so-called 'information silos'. DevOps relies on practices that help to shorten the time between committing a change and its deployment to normal production use while maintaining high software quality. Thus, it shares many of the goals as continuous software engineering in general – improved communication, more intensive collaboration, and faster delivery of high quality software – as well as builds on top of a solid toolchain from code to production.

2.2. Continuous integration, delivery and deployment

At first, a few words on terminology. There is a multitude of terms and concepts involved in continuous software engineering [10]. For the purpose of this paper, the distinction between continuous integration, continuous delivery and continuous deployment is crucial. In continuous integration [13], developers send their changes frequently to version control repositories which trigger tests on continuous integration servers whereas in continuous delivery [14] the steps include additional, automated, actions to prepare a release-ready software version. Typically, the actual release to the production system is not fully automated and requires manual intervention with continuous delivery but in continuous deployment [6] the last mile is also automated.

One of the principles of lean development is "deliver as fast as possible" [15]. In this spirit, the typical agile development process incorporates the concept of continuous integration (CI) [13]. To enable continuous integration, the developers in the team integrate their work frequently under version control. Continuous integration also includes the practice of automated tests in order to get frequent, fast feedback to the development team. Even though there are different levels of automation and flavors to what gets included into continuous integration, the concept is still relatively well understood and defined.

The difference between continuous delivery and continuous deployment is less well defined. In his blog, Fowler [14] defines continuous delivery "as a software development discipline where you build software in such a way that the software can be released to production at any time". Neely and Stolt [16] define continuous delivery on the same lines as "the ability to release software whenever we want". Thus, in continuous delivery, a new software version is not necessarily deployed automatically to the users but it is kept deployable all the time during development of new features. Continuous deployment requires continuous delivery, but takes it a step further and deploys every commit that passes the tests must be delivered automatically to production [6]. For the scope of this paper we will use the definition: continuous deployment is the practice of always deploying new software to production as it gets built.

Humble and Farley [17] describe that continuous delivery is achieved with a stage-gate process called deployment pipeline, consisting of *commit stage, acceptance test stage* and *other testing stages*. This pipeline starts similarly to traditional continuous integration. Developers make changes to source code and each change committed to the version control system will trigger the commit stage of the pipeline. In this stage, software is typically compiled, unit tested and a build artifact is produced. Often, static code analysis and code metric collection is also done. The commit stage should be fast and able to catch the most severe errors. Next, the acceptance test stage is executed. It is still automatic, but executed in a more production-like environment and can take a longer time to complete. The purpose of this stage is to validate that the software fulfills its acceptance criteria and could be released to the users. After passing the acceptance test stage, the software can go through other testing stages which can be manual, such as user acceptance testing or capacity testing. Manual testing stages should, however, be cut down to the bare minimum because they stop or slow the flow in the deployment pipeline. After passing all the testing stages, the software can be released to production.

Based on the recommendations of Humble and Farley, the deployment pipeline should ideally be automated as much as can be, or at the very least, semi-automated. For example, even if deployment would not happen automatically for each successful integration, it should be possible to execute the deployment with a single push of a button. Thus, the recommendation is that all manual work is minimized, leading to the logical conclusion that proper toolchains are essential to continuous delivery.

Ståhl and Bosch studied what effects does continuous integration have [18] and how their differences can be explained with differences between continuous integration implementations. Later, they developed a meta-model to describe differences in continuous integration implementations [19]. They have discovered that (1) continuous integration practices vary between companies and even between projects and (2) what companies call continuous integration is not as continuous as usually understood [20]. Because of (2), Ståhl and Bosch suggest calling the integration practices "automated integration flows".

This study is similar to the study of Ståhl and Bosch in the sense that in both studies, toolchains for automated software development are studied. However, the focus of Ståhl and Bosch is on continuous integration practice, whereas in our study, the focus is on the whole, continuous deployment.

Leppänen et al. [21] focused on the potential benefits and challenges of continuous deployment in a multiple-case study of 15 industry companies. They found out from the interviews that although not all companies aimed for fully automated continuous deployment, frequent releases could potentially help the companies to build better products for the customers and be more reactive in development. Social factors within companies or attitudes in the field could prevent the adoption of continuous deployment practices. Technical matters such as the size and complexity of projects, and the need to perform manual testing stages were seen also to be in the way of continuous deployment. The study presented in this article is based partly on the data from the study of Leppänen et al. [21] but the emphasis here is on the overall development process utilized in the companies and the underlying tool infrastructure which were not covered in the original study.

Claps et al. [22] studied continuous deployment with a single exploratory case study strategy. Through interviews, they discovered a total of 20 technical and social continuous deployment adoption challenges. To mitigate these, they also proposed strategies which were adopted by the case company. Multiple challenges mentioned by Claps et al., such as hardware and software infrastructure, deploying the product, seamless upgrades, continuous integration process, testing, source code control, shorten customer feedback and team coordination may need tool support to be overcome properly. Thus, their results support the argument about the importance of toolchain.

3. Research methodology

In this section, we formalize the research methodology and characteristics of the multiple-case study, describe the study design, detail the case companies, describe how data was collected through interviews and analyzed with thematic analysis, and finally synthesized into the research results.

3.1. Research methodology and study design

Our multiple-case study [23] is an exploratory study of the toolchains in Finnish software-intensive organizations interested in transitioning towards a more rapid deployment rate – up to continuous deployment. Our main aim is to ascertain the current state of the practice with respect to continuous deployment in the Finnish IT field. For this study, our formal research questions are:

- **RQ1**: Which toolchains are used to support software delivery?
- **RQ2**: What are the reasons behind the gaps in the toolchains?
- **RQ3**: What possible implications do toolchains have on the speed of software delivery?

In order to help researchers consider the research process as a whole, we use the framework by Wohlin and Aurum [24] to characterize our research:

- Our research outcome is *basic research* instead of applied research, because we investigate a problem instead of proposing a solution.
- We use *inductive* research logic instead of deductive, because we do not have an existing theory to test. Rather, our contribution can be used for theory-building.
- Our research purpose is *exploratory*, as our research topic is quite new and there are not many existing studies.
- Our research approach is *interpretivist*, because especially RQ2 and RQ3 would be difficult to study under the positivist lens.
- Our research process is *mixed approach*, because RQ1 and RQ2 are based on qualitative data, whereas RQ3 uses quantitative approach.
- Our research methodology is *case study*, because it allows deeper investigation of the phenomenon than surveys.
- Our data collection method is *interviews*, because it is the most feasible option to study multiple cases, especially for questions RQ2 and RQ3.
- Our data analysis method is *thematic analysis*, because it is well suited for RQ1 and RQ2. For RQ3, we also used descriptive quantitative measures.

To answer the research questions, we followed the research process depicted in Fig. 1. In the following sections, we describe the process more specifically.

3.2. Company and case selection

The case selection of the study was made with a twofold selection criteria. Our larger population of interest comprises Finnish software-intensive companies and enterprises interested in the benefits afforded by faster software delivery. Inside this population our selection comprises companies (i) participating in Need for Speed (N4S) [25], a Finnish research program focusing on rapid value delivery, and (ii) other companies known to researchers as interested in the subject. The Need for Speed research program is a four year program running from 2014 to 2017 with around 40 program partners. Majority of the program partners, thirty or so, are industrial organizations and the rest are universities or research institutes. Partners in the program engage in joint industrial research and development endeavors in the area of rapid value delivery and this article is a result of such work. By joining the program, the industry partners have shown interest and commitment to improve their capabilities towards more real-time operations.

We approached prospective program partners and other known companies in person and via e-mail, asking of their interest to participate in the study. The companies could freely choose whether to participate in the study. As such, selection at the organization level can be considered self selection or convenience selection. However, within this context, companies were purposefully chosen to ensure variability in terms of domain and organization. Almost the same selection of companies was used in a previous study [21]. Our study includes four additional cases (C6, C12, C17, C18) but excludes one case because it lacked detailed tooling information (Company O in [21]).

Most of the selected companies were small or medium sized enterprises with less than two hundred employees. Several well-established companies in the study were substantially larger, employing over ten thousand workers each. Also, the industry sectors

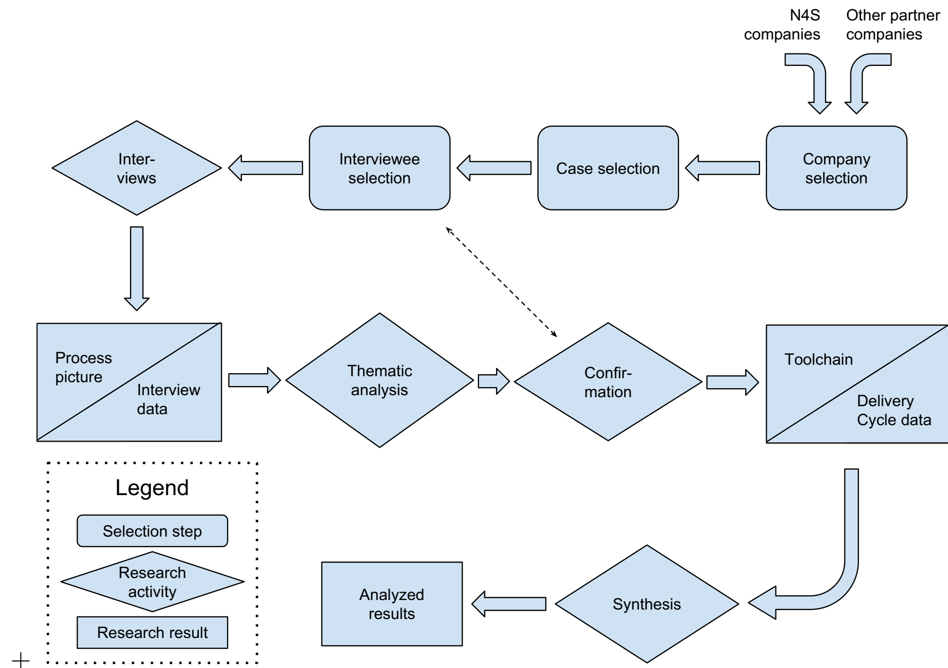


Fig. 1. Research process of the study. Different steps of the process are explained in Sections 3.2–3.5.

Table 1
Background information of the selected case organizations.

Case	Domain	Personnel	Team Size	Platform	Primary Technologies
C1	Consulting	80	8	Web	Java
C2	Consulting	80	7	Web	Java, Javascript
C3	Consulting	250	<10	Web	Clojure, Javascript
C4	Consulting	180	3	Web	Javascript, Scala
C5	Consulting	100	10	Embedded	C++
C6	Embedded Systems	500	30	Embedded	Proprietary
C7	Industrial Automation	15,000	50	Embedded	C++
C8	Mobile Games	7	7	PlayStation Portable, Apple iOS, Google Android, PC	Platform specific
C9	Mobile Games	9	4	Apple iOS	Unity, Scala
C10	Mobile Games	9	3	Apple iOS, Google Android, Windows Phone	C++ (cross-compiled)
C11	Telecom	>10,000	100–300	Network	Java
C12	Telecom	>10,000	50–100	Network	C++
C13	Telecom	>10,000	350	Network	C++
C14	UI Framework	80	7	Web	Java
C15	UI Framework	1000	200	Cross-platform	C++
C16	Web Services	7	7	Web	Ruby on Rails, JavaScript
C17	Web Services	8	3	Web	Ruby on Rails
C18	Web Services	90	30–50	Web	Java, Scala

of the companies differed. The consultancies were engaged with contracted customer projects where the technologies varied per project whereas some case companies had a more specific focus. Companies in the mobile game industry and telecommunications formed clearly identifiable groups. The web was a popular choice as a platform, but some companies also targeted embedded and mobile platforms. These included embedded Linux and consumer-oriented mobile operating systems like Apple iOS or Google Android. Software in these specific cases was typically programmed in Java, C++ or Scala. In the web domain, Javascript and Ruby on Rails were mentioned as was the functional programming language Clojure. Characteristics of the companies can be seen from Table 1. It should be noted that the cases C11 and C12 were from the same company, but from different products. That is why we refer to the cases as organizations instead of companies.

From this group of selected organizations, we made a selection of interviewees. We requested to interview a technically well-versed representative of the development organization that was their most advanced in terms of adoption of rapid delivery practices. Organization representatives were asked to evaluate whether other teams in the organization were using similar software processes and methods, and the rationale for potentially differing practices but the relative experience or expertise of the team was not judged further. Heterogeneous practices are possible especially in large organizations so to make the responses concrete, the unit of analysis was chosen to be the projects or products engineered by the team of which the interviewee had direct experience. The interviewees were software developers, architects, quality assurance personnel and team leaders who worked mostly in small developer teams of less than ten people. The teams in the telecom organizations were quite a bit larger than the more typical teams and, in addition, were geographically distributed across multiple sites.

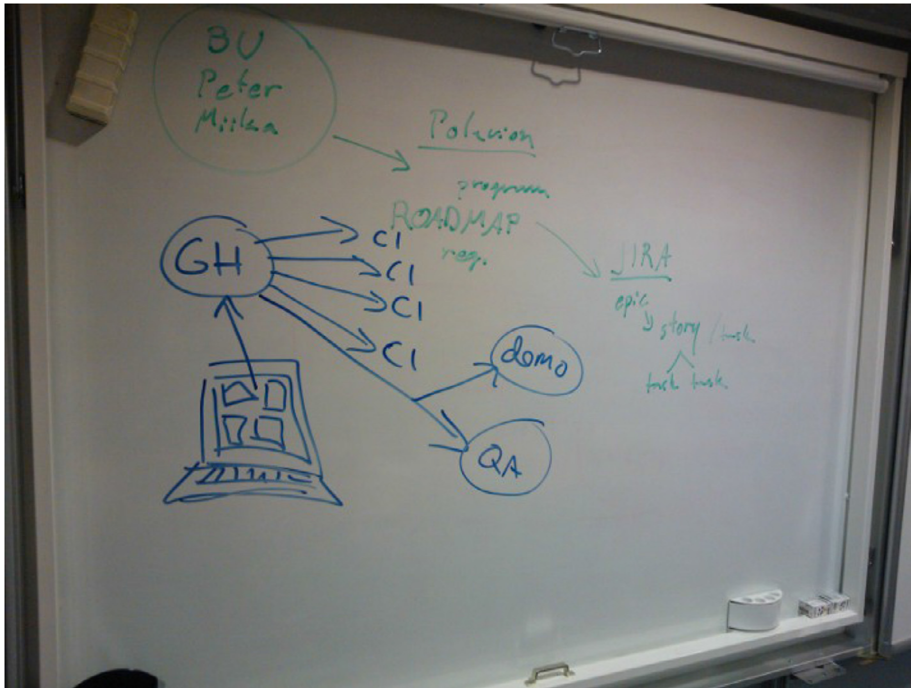


Fig. 2. A typical process diagram drawn by an interviewee.

3.3. Interviews

This study utilizes the interviews conducted in a previous study [21]. Same interview records and notes were analyzed, but four additional interviews were done for this study alone. Also, the analysis in this continuation study focuses on toolchains, whereas the previous article discussed the motivations, benefits and obstacles of continuous deployment practice. Overall, there were a total of around 50 questions grouped into 9 subtopics in the original interview protocol and the same questions were used in the additional interviews. Questions about the toolchains in the organizations were part of the broader original study design so the datasets between the studies were compatible without further need for adjustment. The relevant subset of questions for this study is highlighted in Appendix A, while other topics are included there too for completeness to illustrate the other themes covered in the interviews but which were not pertinent to the continuation study.

In order to answer the research questions, we chose surveys in the form of open-ended, semi-structured interviews. Interviews were carried out in Finnish, with two researchers present and primarily one case representative. Mostly, one researcher presented the questions following the interview outline while the other acted as a scribe and took notes. Three separate research teams from the participating universities were involved in conducting the interviews. In addition to basic closed-ended demographic questions, interviewees were asked a variety of open-ended and closed-ended questions on software delivery, see Appendix A. These questions were grouped into subtopics; the data presented in this article pertains to the topic of toolchains.

Interviewees were asked to describe the tools they use in daily project activities, and to denote the stage or stages in which tools were used as part of their development workflow. This was conducted by asking the interviewee to draw a picture of their delivery process, see Fig. 2 for an example. Qualitative data from the interviews was gathered primarily by taking notes during the interviews. Using audio recordings as support, the notes were revised after each interview in order to ensure completeness. The revised notes were circulated with the researchers not present in the interview.

3.4. Thematic analysis

The primary analysis method for the data collected in our study is thematic analysis [26]. In thematic analysis, collected data is first examined in order to extract relevant items such as results or other key findings from the data. As the analysis and synthesis progresses, extracted data is coded and assigned to common categories for grouping the coded entities. Similar codes and categories are later associated with more abstract themes which are seen to represent the underlying concepts. Thematic analysis allowed us to approach the data analysis from the ground up by identifying tools and categorizing them into common categories based on their purpose.

Using the revised interview notes, tools used in each organization were coded into themes based on the primary purpose of the tool (i.e. the scope and situations for which it is intended to be used). If the notes were not comprehensive enough, audio recordings were listened again to acquire the needed information. Documentation and public websites of tools were consulted in cases where the purpose of the tool was unknown to researchers. Thematic analysis of qualitative data is subject to a number validity threats, the specifics of which are described in detail in Section 7.5.

Following theme identification, each theme was given a name to reflect a particular software development activity. For each identified tool, there is one or more associated primary activities, as a single tool could belong to multiple activities. The constructed activities are described in the next Section 4. After assigning the tools to the primary activities for each organization, the representation of the toolchain with named tools for activities was sent to the organizations for verification. Data for the toolchains was adjusted and new tools added based on the responses. All but two case organizations (C13, C14) responded to the verification request.

Table 2
Metrics analyzed in this study.

Metric	Interview questions (Appendix A)
Actual releasable software cycle	2.2, 2.3, 7.1, 7.2 and 7.3
Actual release cycle	2.2 and 2.3

3.5. Synthesis

Synthesis based on the research data commenced once the data had been coded and confirmed. Oral descriptions of the development process, process illustrations drawn by the respondents and the codified data of tool usage in different development stages were used to construct a representation of the development pipeline for each organization. The constructed development pipeline representations and information about the toolchain allowed identifying pipeline sections and activities where work was done without tools. Qualitative analysis of the interviews was used to further analyze and describe the reasons behind the selection of the set of tools in an organization, their usefulness and the rationale for the current workflow.

The deployment and delivery frequencies of the organizations were analyzed by coding the verbal responses of the frequency-related questions in the interviews as numbers. These numbers represented the deployment and delivery capability of the organizations. The deployment and delivery capability of the organizations was compared against the amount of tool-aided phases in each case for which the data was extracted from the toolchain representation. The definition for tool usage was binary: either the organization had tooling support for the activity or it did not.

Two frequency metrics were extracted from the interview data (Table 2):

- *Actual releasable software cycle* means an actual cycle during which the development organization produces an artifact that could be, in principle, released. However, factors out of control of the development organization can prevent realizing this release capability. There was no distinct question to acquire this metric, instead the metric was extracted from multiple answers. The interview protocol allowed such flexibility.
- *Actual release cycle* means how often the software is actually released. This takes into account other factors of the release.

Results yielded from counting the tool-aided phases were subject to quantitative analysis which allowed grouping of the organization based on their deployment and delivery capabilities, and the amount of tool aided phases. This process is further described in Section 6.

4. Modern software development toolchain

In this section, we introduce a modern software development toolchain which was constructed based on the interviews conducted and the results of the thematic analysis in this study. Each element of the toolchain is described individually, accompanied by references from the literature. The main elements covered in the section include *requirements* as the first link in the toolchain, *development* as the core element for software development, *operations* as an element interfacing between various environments, *testing* as the element for verification and validation, *quality* as an additional element ensuring the non-functional quality properties, and finally the overarching *communications and feedback* which provides a common link between the elements.

A modern software development is backed up by a toolchain that consists of automated tools from code all the way to delivery of new software. The role of these tools is to ensure each stakeholder can develop, test and deploy the software when they need to and to make sure suitable feedback for from all stages of the development process is produced. Fig. 3 illustrates the elements of modern software development that might benefit from tool support.

4.1. Requirements

Deploying software constantly, even several times a day, requires that software development teams have clear requirements for features to work with. Without a steady flow of requirements in the beginning of the deployment pipeline, the development team would not be able to keep up a constant pace. Regardless of the pace, keeping track of all the data associated with requirements is an essential part of software development. Here, *requirements elicitation* and *backlog management* tools assist in the process.

The classic user requirements workshop organized just once in the beginning of a project is no longer seen as the optimal solution due to the chance of miscommunicating requirements, and the rate of change in the requirements during the actual development. Thus, there is need for continuous feedback systems as detailed by Maalej et al. [27]. The modern feedback systems work both ways, pull and push. User requirements can be obtained not only by asking users directly through explicit pull mechanisms, such as workshops and interviews, but also by a push mechanism straight from the users, like feature and enhancement requests. The users can also be experimented on without their knowledge by implicit and tacit field experiments.

In modern-day software development, prioritizing work and knowing what needs to be done relies on maintaining a backlog, which is essentially a cumulative collection of work to be done. According to agile experts, backlogs should be ordered lists of well-defined work items. There can be several backlogs with items of varying granularity. For example, Scrum advocates two backlog artifacts: a product backlog, with feature-scale items, and a sprint backlog, with tasks that one person can do in a couple of hours. *Bug tracking* is also closely related to backlog management as a bug to fix is a specialized case of a development task for the developer – a work item among others. However, in some cases, bugs are also reported by customers or end-users, and a specialized tool may be needed to facilitate communication.

4.2. Development

Version Control Systems (VCS) exist to manage changes in various documents, such as source code and web pages, and other information that might change during the development of the software. A VCS stores information to retrieve any revision of the documents, so that the user may return to an earlier version of the modified document. Contemporary version control systems are usually distributed, as opposed to older centralized revision control systems. See Otte [28] for comparison of different VCS types.

Build systems are elementary for the development of software. Fundamentally, build systems construct deliverables based on build specification files written in a configuration language [29]. Build specifications can include instructions for compiling code, executing test cases, packaging project files and deploying new software releases [30]. Build tools which can interpret the specifications also help the developers in managing the internal and external dependencies of software, for instance with the help of proprietary mechanisms for resolving used external libraries [29].

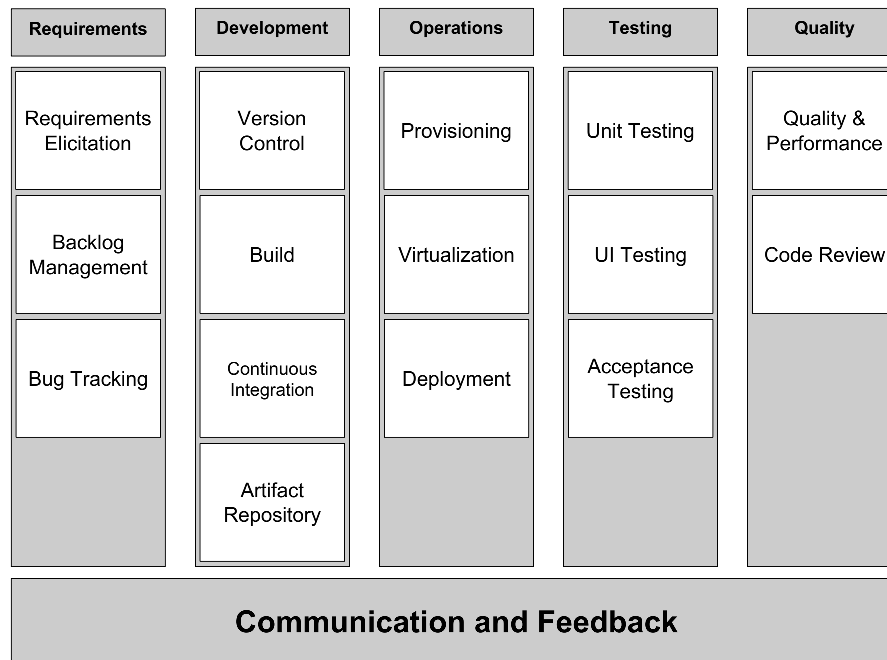


Fig. 3. Elements of modern software development toolchain.

One of the challenges with build systems is that when software systems grow in size and complexity, so do the build specification files, requiring much effort to maintain [29]. Build systems running large software projects can also face resource shortages when a great number of build artifacts needs to be created over and over again, an issue which can be alleviated with the help of incremental builds or cloud-based parallel execution of tasks [30].

As an additional challenge to the size of the system, a software project might not only have internal dependencies to its own components but to external components developed by others as well. These can be libraries created by other teams within the organization or individual libraries, modules and packages developed elsewhere, containing features utilized by the project in question. When a multitude of such dependencies exist, it becomes increasingly difficult to manage the different versions of each dependency and the interdependencies between the dependencies themselves [31].

Automated builds bring in the principle of *continuous integration*, which in turn is a prerequisite for continuous delivery and continuous deployment. While CI can be implemented with an old computer and a rubber chicken [32], that is with modest investments and suitable practices using the chicken as a passable token, in modern software development continuous integration tools form a backbone for any continuous deployment pipeline. The task of continuous integration tools is to monitor version control systems for changes that consequently trigger the execution of the various steps configured on the continuous integration server [33]. The steps can include the compilation and packaging of code in order to build an identifiable software version, and execution of test suites with the aim to inform the developers whether their recent changes were successfully integrated or not.

Together with build systems, *artifact repositories* offer a solution to developers for managing dependencies and versioning different libraries. It could also be considered that artifact repositories help with the issue of incremental builds [30] since it is possible to store already compiled and tested, known to work, versions into the artifact repository.

4.3. Operations

Software can run on environments ranging from purely physical to purely virtual. *Provisioning, virtualization* and cloud environments make possible to set up instances of the software's running environment. Typically different environments, such as staging and production, are used in parallel during development. Those should however be such that the same build can be deployed in any of them. When it comes to continuous *deployment*, it should thus be relatively effortless to spin up new environment instances so that the build could pass from development to production without many obstacles or problems originating from the differences in execution environments.

Humble and Farley introduce common antipatterns related to delivering software. These are (1) Deploying software manually, (2) Deploying to a production-like environment only after completing development and (3) Manually configuring the production environments [17]. Instead, the aim should be in delivering software as automatically as possible. To avoid challenges related with changing configurations the development environments should also be as close as possible to the production. The same process should also be used for deploying to all environments.

In continuous delivery and deployment, the deployment infrastructure can be treated as code, and it may also be versioned. Ultimately, the system can be reproduced wholly from the VCS. One approach for release management to maximize continuity and similarity of environments is blue-green deployment [17,34]. In it, two identical versions of the production environment are maintained – a blue one and a green one. These two take turns in being the production. Each new deployment is done to the one not acting as production at the time. Moving to the new software version is done by switching the roles of the environments. The approach also makes a roll-back to the previous release possible and smooth, if something seems not to work in the new version.

4.4. Testing

Even though practitioners usually refer testing a separate activity, through test-driven development (TDD) and the introduction

of continuous integration, contemporary testing is very close to the actual development activity. This is especially true with *unit testing*. Continuous integration tools with automated tests have to ensure that changes made in development do not introduce new bugs. The goal is to maintain a build that is tested against a thorough set of unit and component tests at each integration. Maintaining a more complete set of automated tests facilitates regression testing as the tests can be re-executed in subsequent iterations. For the purpose of continuous deployment, it is still sensible to handle testing as an element of its own right, especially when it comes to acceptance and UI testing that are more difficult to fully automate.

User interface (UI) testing is a testing task where the graphical user interface is tested. A relatively small program may have a considerable amount of UI test cases and different sequence of tests may cause differing response from the software. Furthermore, a huge redesign of the UI may cause all user interface related tests to fail even though the actual functionality of the application may not have been changed at all. Some areas of UI testing, such as usability or user experience testing, fall close to the domain of quality assurance as they are not purely functional testing and may be hard to automate.

Acceptance testing is a special case of testing, where it is determined if the software system conforms to the requirements, either functional or non-functional ones. From the point of view of continuous deployment, acceptance tests should be run in an environment like the production. Humble and Farley [17] argue that acceptance testing should be automated and such that each build gets tested against acceptance tests provided it passes the earlier tests.

Traditionally, acceptance tests have required customer involvement in the execution of the tests. However, this is not typically possible if acceptance tests are run for each change to the software. Therefore, in this paper, we usually mean *automated* acceptance tests when we use the term "acceptance test". For automated acceptance tests, customer involvement is required only when constructing the automated test case, not when executing it.

4.5. Quality

Functional testing alone is not enough to ensure the quality of software products. However, not only continuous deployment, but agile practices in general, promote high software quality also on the code level and when it comes to non-functional requirements. Thus, additional care must be taken to take into account internal quality issues and other aspects of quality, such as performance and security.

One apparent way to assure *quality* is to measure certain metrics of the actual software system. For example, *performance* is usually difficult to assure without measuring it from the software execution. Thus, certain artificial load and stress tools may be used to ensure the functionality of software under even hardest circumstances.

Several aspects of quality may be checked even without having the software system built at all. One way to achieve this is to have static program analysis tools, such as Lint, to check the code for a certain patterns, which may indicate bugs, vulnerable parts, suspicious code or just bad coding conventions. In certain domains, such as the medical software domain, their use is commonplace to improve the quality of safety-critical code.

Another way to improve code level quality is to have other human beings to review the code — even in the form of pair programming [3]. In the continuous deployment mindset, *code reviews* represent a way to introduce some human intervention to the otherwise automated pipeline. Some aspects of software quality may be hard to inspect automatically, so reviews can act as an additional safety guard. Code reviews are an old practice, but nowadays formal review sessions taking up to hours and covering large parts of code have fallen out of popularity. Instead, it is commonplace to integrate a code review tool to version control system so that every committed change may be reviewed by one or several persons. Persons may have various roles, for example certain people may have to give a permission for the commit to be integrated in the code base after they have reviewed it.

4.6. Communication and feedback

Communication and collaboration are key elements in software development. Developers in software teams share knowledge through various mechanisms, which not only can help the team to build a social identity as a team but to also share task-related project knowledge between members [35]. Collaborative frameworks have use beyond the developer organization itself as software users can be integrated into software development with collaborative technologies [35], creating continuous push-and-pull feedback system [27]. A continuous deployment pipeline can be fully functional in a technical sense but without handling feedback efficiently from the developers and users alike, the development team might miss important cues in which direction to head next. Optimally, feedback from users and developers is directly associated with traceable software requirements and eventually the tasks which are put in the continuous deployment pipeline.

Feedback is an element that is crosscutting all stages of software development. One of the reasons behind the modern development toolchain is to make timely feedback reachable for all stakeholders. Timely feedback and short feedback cycles are a cornerstone of lean and agile. In continuous deployment, feedback is propagated from each integration.

5. Interview results

In this section, we describe the tool usage in the case organizations. First, we overview the results and then we follow the elements described in Section 4 for the description of the results. Results for the general areas of requirements, development, operations, testing, quality, and communication and feedback are each covered in turn.

5.1. Overview

The overview of the results for each software development stage is illustrated in Fig. 4. In the figure, grouped by domain, tool support for every case has been plotted based on the information retrieved from the interviews. Rows in the figure represent the results of a single interview. Entries marked with black mean that a specific tool was **used** in the organization for the development stage and white means that any tool for this stage was **not used**. In certain application areas, some of the tools were considered **irrelevant** by the interviewees (e.g. user interface testing for embedded systems): these entries are marked with light gray. When there was not enough information to determine whether an organization used a tool in an activity or not, we categorized the entries to the **not mentioned** category indicated by the dark gray color. The results were later confirmed from the case organizations, although confirmation was not received from two cases (C13, C14).

In total, results from 18 cases are shown in the figure. The results indicate that while handling requirements has shifted towards a more agile way of working by de-emphasizing the usage of formal requirements engineering tools and applying flexible management methods with ticketing boards, organizations have the first half of the toolchain quite well covered. Development is

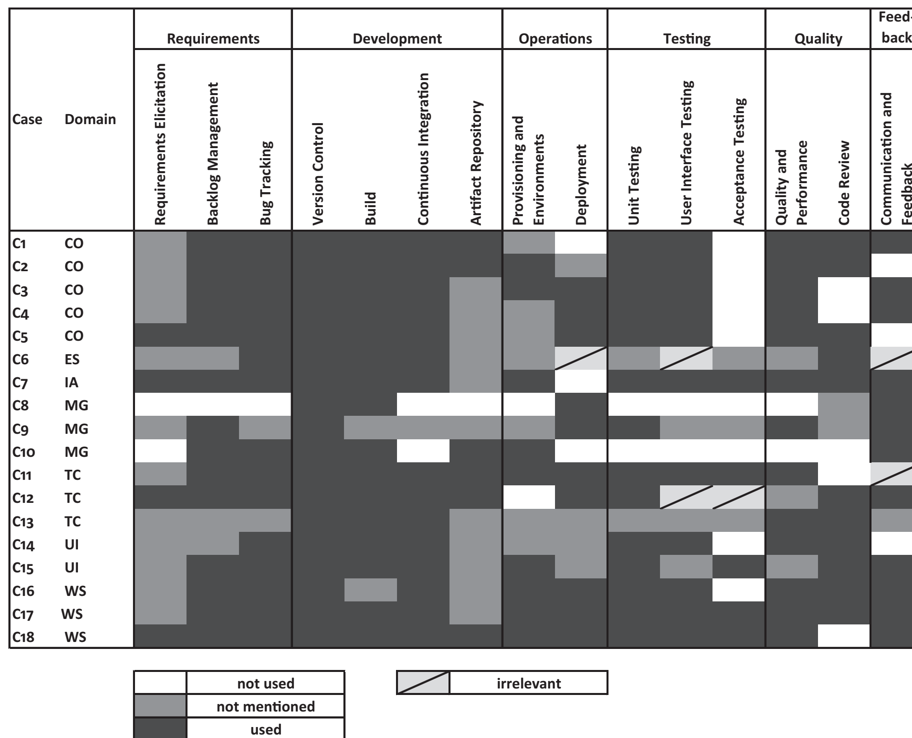


Fig. 4. Summary of the tools used in the case development organizations.
Domains: **CO** = Consulting, **ES** = Embedded Systems, **IA** = Industrial Automation, **MG** = Mobile Games, **TC** = Telecom, **UI** = UI Framework, **WS** = Web Service.

Table 3
An overview summarizing reported tool usage in different stages of development.

Dev. Stage	Overview
Requirements	Backlog management tools were common and had replaced the rarely utilized specific tools for requirements engineering. Physical equipment like whiteboards can replace the need for backlog tools. Defects are handled with tools in most organizations.
Development	Version control practices had permeated all the organizations but branching strategies differed. Automated build systems for optimizing the build process were almost as common. Continuous integration systems were common except in the mobile games sector. Artifact repositories for storing precompiled libraries were not mentioned often.
Operations	Provisioning tools to configure and construct production-like environments were modestly used. Deployment tools had similar popularity although deployment was manual in many cases.
Testing	Programming language specific unit testing tools were commonplace. User interface testing was modestly tool assisted but hard to automate. Acceptance testing relied mostly on human judgment.
Quality	Besides separate testing procedures, quality was assured by various methods. Code quality was kept in check by occasionally used static analysis tools and code review protocols. Non-functional qualities were assured by tools for real-time monitoring especially in the web domain.
Communication and Feedback	Internal communication tools, sometimes associated with build and testing systems for rapid developer feedback, were used for development. The gap between developers and third parties was not often bridged with external communication tools. Customer data analytics were used sparingly.

done through shared code repositories with good continuous integration support. Testing tools are used but user interface testing and acceptance testing are somewhat overlooked. Continuous deployment to production environments is not fully supported by existing tools in the organizations although certain deployment solutions are in use. Internal feedback tools are abundant but tools for customer feedback have less support. Table 3 shortly summarizes the observations of tool usage from each development stage. There seem to be certain domain-specific differences in tool utilization which are further discussed in the following sections.

A well-functioning continuous deployment pipeline requires that the output from each stage moves effortlessly to the next development stage. Comprehensive single-vendor software suites where development stages are interconnected by proprietary solutions were nonexistent. Instead, the organizations relied on best-of-breed tools, some of which had common interfaces that made them compatible with each other. When it comes to local workstations, developers were at times given leeway in selecting suitable development environments they felt most comfortable with as long as they followed the generic development process in the organization.

Version control systems and continuous integration servers were the integration hotspots that typically triggered workflows and tools in other development stages. In an ordinary situation when the degree of automation was high enough in an organization, the developer would commit work from the local workstation to a particular branch in a remote version control system which in turn triggered various testing activities through continuous integration servers, possibly provisioning the release to various environments closer to the production system. Similarly, tools for code review and version control were in some cases connected so that quality checks by peer developer reviews could be initiated after version control commits. Integration between development activities and tools was not entirely seamless and many tasks were still done manually, especially in the last stages prior to production releases, but at least in several cases work

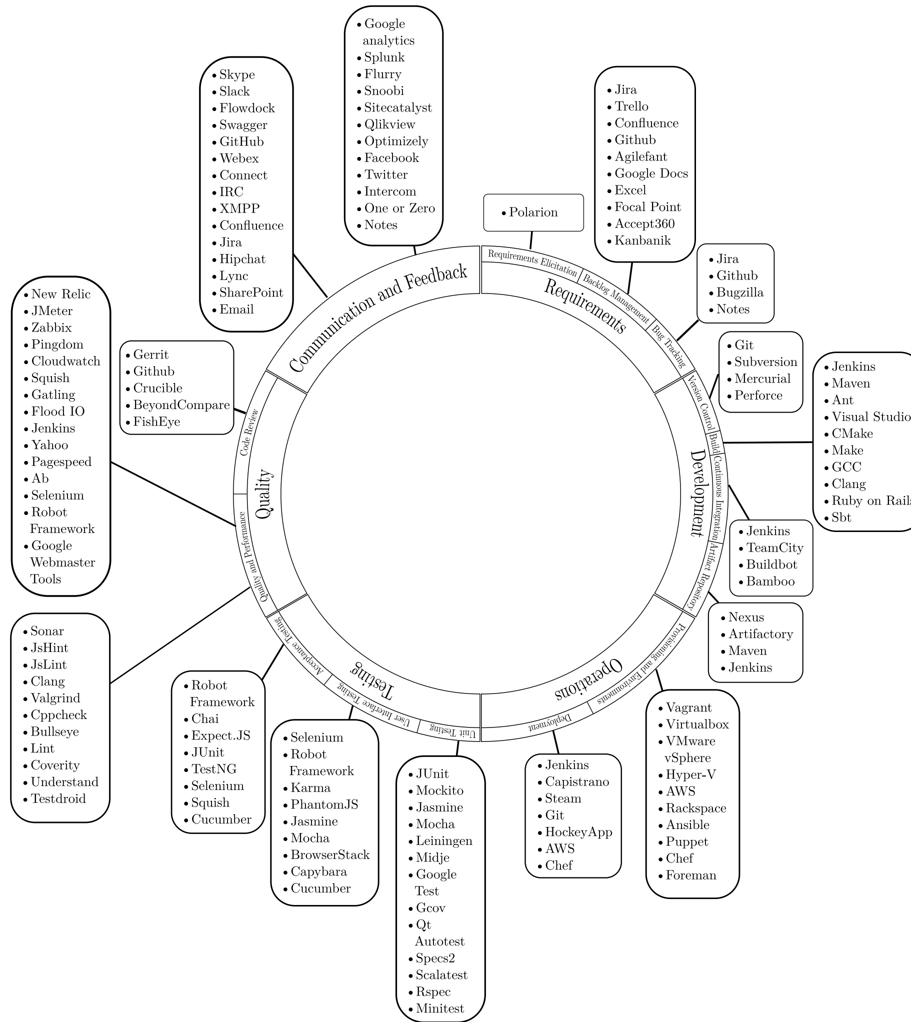


Fig. 5. Overview of used tools in each development stage. For more information about the tools, see the tool appendix [36].

was being done to remove the bottlenecks and to reduce manual labor.

The concrete tools used by the organizations in each development stage reflected the development practices used in the organizations. While many of the tools can be considered independent of the development platforms and frameworks, the availability of tools and the current culture of tool usage in the field contributes to the choice of tools. In this article, we refer to the tools by name. For more information about the tools, see the tool appendix [36].

Diversity of tools used in each development stage varied accordingly as depicted in Fig. 5. Certain tools had universal applicability regardless of domain and context whereas others were more specific to platforms and frameworks. Tools for version control (Git), continuous integration (Jenkins) and code review (Gerrit) could be considered universal. Testing tools were more closely associated with development platforms. Performance testing and monitoring tools such as New Relic had stronger support in the web domain where availability can be a greater concern.

5.2. Requirements

In a modern software development pipeline, the access to clear requirements is ensured through *requirements elicitation, backlog management* and *bug tracking*. The results show that requirements elicitation tools were used rarely and had been replaced by the more commonly used backlog management tools such as Jira. Tracking bugs with tools was common and alike for backlog management, Jira was the choice in many cases.

5.2.1. Requirements elicitation

The results for our study indicate that organizations did not have many tools in use specifically for requirements elicitation. Closer to a hundred requirements elicitation tools have been identified to exist [37] but a specialized tool for requirements management and elicitation was used in only a single case where they utilized Polarion Requirements.

The rest of the interviewed representatives, however, did not explicitly mention tools for requirements elicitation and managing requirements belonging to their existing toolchain. One organization, however, indicated that they use Atlassian Jira and Trello for requirements elicitation. Jira was also used for backlog management, so it seems that this organization effectively transforms requirements into backlog items on the fly.

To illustrate the problems in the organizations, there is an example from one of the interviews. An interviewee mentioned that they were struggling to maintain a continuous flow of requirements for their development team. Their requirements came in bursts so at times they were sitting idle, enjoying leisure time at their office, and then shortly after swarmed with new requirements when a person in the requirements engineering role had sufficient time to write them down.

Clearly, requirements engineering and elicitation tools were not commonly used. That is not to say that the organizations would not have processes in place for requirements engineering and elicitation. Perhaps one of the reasons is that in the contemporary agile world, even large organizations are shifting towards more continuous forms of planning, where gradually prepared requirements through such instruments as user stories are seen to improve the development process [38]. Agile backlog management tools are thus seen more relevant than traditional tools for requirement elicitation.

5.2.2. Backlog management

The usual tool for backlog management is Jira (50% of the cases), but Trello (17%) and Confluence (11%) also have a strong support. Jira has also been documented as the most popular tool in previous studies [39], especially for bug tracking. In a sense, bugs can be considered specialized occurrences of backlog tasks.

Tracking issues and directing the flow of development in a project can also be done without the help of software tools. Whiteboards and other physical tools were in some cases just as important as software in backlog management. Matter-of-factly, one interviewee commented that “the whiteboard was predominantly the most important tool in their project” as it could be used in tying up information from version control, backlog management, code review, bug tracking, acceptance testing and deployment. Thus, it was also a means of communication as project members could use it to convey important information. It seems that a whiteboard is easy to use and gives transparency to information as it can be checked by just giving a quick glance.

5.2.3. Bug tracking

Bug tracking is closely related to backlog management since a bug fix is a certain type of a development task for the developer. However, in some cases, the bugs are also reported by the customer or the end-user, and a specialized tool may be needed to facilitate communication. Thus, it is not surprising that Jira is the most popular tool for the bug tracking as several organizations already utilized it as an internal tool for backlog management. Other tools, also used for backlog management, were Trello and Github. Both of these was encountered once in the interviewed organizations. Three organizations used unnamed proprietary tools for bug tracking.

5.3. Development

Development work is supported in the pipeline with *version control, build* systems, *continuous integration* and *artifact repositories*. Results for development suggest that the usage of version control tools is pervasive and Git has a strong foothold in the category. Build tools are almost as crucial to development as version control but the range of options for building is broader. Usage of continuous integration servers, mainly Jenkins, is frequent but there are exceptions to the rule. Artifact repositories were not often mentioned.

5.3.1. Version control

Almost all organizations used distributed version control systems, such as Mercurial and Git. Hybrid solutions and centralized systems were only seen in three organizations. It is interesting to note that even though the majority of organizations use Git (almost every third), their practices using it vary. At least three fairly well-known branching models have been applied; “A successful Git branching model/Git Flow” [40], “Another Git branching model” [41] and “A rebase workflow for Git” [42]. In addition to the mentioned version control systems, some proprietary variations exist. Further research is needed to analyze reasons and rationales behind different choices and their impact on the actual development process.

The existence of a VCS was ubiquitous to all organizations we interviewed. It was the only tool class to achieve this status. Thus, it underlines the need for this kind of a tool and its place as a commodity for a software developer. This is interesting to note, as a decade ago there were reports of organizations who did not use version control [43]. We can only speculate if the tools have achieved a certain degree of familiarity and maturity to be have such widespread use.

5.3.2. Build

Based on our interviews, organizations were using several different systems for building. Here, it is good to distinguish between build systems which run locally, and systems which run remotely on a server such as a continuous integration server where additional build steps can be executed, possibly triggered by changes in version control.

Around thirty percent of the respondents mentioned that they used Jenkins as their build system, which is also used as a continuous integration server. Java-based build systems Maven and Ant were almost as popular. Jenkins was mostly used as a remote build system that is used to launch the execution of a series of scripts containing actions for build steps. Although Jenkins utilizes build systems like Maven and Ant that can be run locally, in several cases build steps in Jenkins contained more complex workflows such as the orchestration and creation of virtual machines for testing and transferring content between machines. Build systems dealing with makefiles were used as well: Make and CMake were quite common choices although makefiles can contain overly complex structures due to their design [29].

Several interviewees said they used proprietary scripts as build systems while others commented that they used their integrated development environments for building. The choice for a build system can also be tied to the programming language. Certain systems can be built best with native tools for that environment. Furthermore, dynamic programming languages such as Ruby do not require compilation. Without having code to compile, the build steps in a Ruby on Rails application can include other actions such as precompiling Javascript assets used in web development or preparing databases and test fixtures for testing stages.

Given their elementary role, build systems and their smooth operation can be essential to continuous delivery and deployment. A badly working build system creates lag in the development process. An interviewee detailed that several years ago it took over two hours for the build system to complete building. They subsequently improved their system by buying new hardware, utilizing parallelism and incremental builds. Each of the improvements gradually improved their development capability by reducing build times and currently the build takes only three minutes to complete. That is quite a drastic change in performance.

It can be considered that build activities can generally take a long time to complete. Similarly to the previous case, a representative from another organization working in the embedded domain mentioned that build times for them range from ten minutes to an hour depending on the environment. He also said that roughly half of their working time is spent on waiting for the test runs to finish. An efficient build system which compiles the code fast, reuses already compiled artifacts and executes the tests in a reasonable time would likely be on the checklist for any organization aiming to do continuous deployment.

5.3.3. Continuous integration

The interview results show that continuous integration servers are frequently used in organizations as part of their develop-

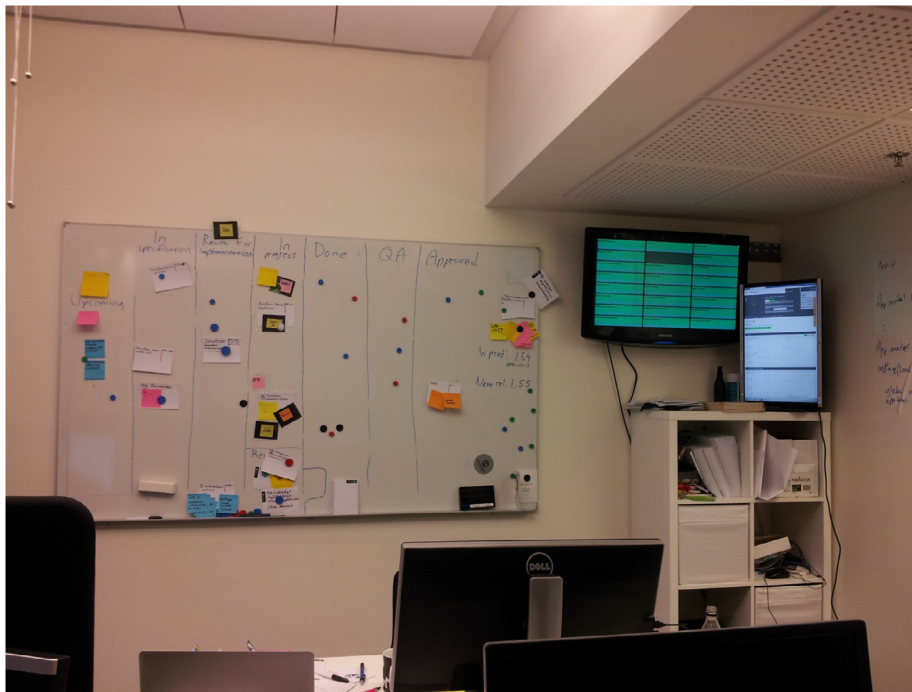


Fig. 6. Radiators used in software developers' work spaces.

ment pipeline. A vast majority of the organizations mentioned that they used Jenkins as their continuous integration solution whereas other solutions such as TeamCity, Circle CI, Buildbot or Atlassian Bamboo were seldom in use. It seems that Jenkins has a firm foothold in the toolchain of organizations which might be due to its nature as an open source alternative to the other proprietary options.

The continuous integration workflow of the organizations matched the description of Meyer [33] quite well. Continuous integration environments had been set up so that processes on the continuous integration server were triggered by the changes in a version control system which was being monitored. Continuous integration servers were used to compile and build a software version with all the necessary dependencies which could then be put to test. Actual test execution was in some cases configured to run in external quality assurance environments where the continuous integration server pushed the freshly built software package to. However, pushing the latest version to different testing environments was not always automatic and required some manual work.

Feedback from the continuous integration servers was considered important, too. Certain interviewees regarded the continuous integration server as a safeguard for detecting abnormalities, catching the defects early on. Developers could observe the output of the continuous integration servers, and see whether their recent code changes passed the bar. Continuous integration servers may even show green status signifying success on the developers' screens or on the information radiators found in the physical space. An example developer space with two radiators from one of the cases is shown in Fig. 6. The feedback loop back to the developer can be quite fast as a developer said that they can get the information from the continuous integration server in just a few minutes.

Passing the automated tests on a continuous integration server can be considered crucial when it comes to advancing to the next phase in the continuous deployment pipeline. There is little confidence in the software version if it does not pass the assigned tests it is supposed to pass. In cases where the build did not automatically progress to the next phase, respondents explained that they have to see that all tests pass before they can make the decision to trigger processes that promote the version in the end, i.e. accepting that the version in question is good to go.

Interestingly enough, three organizations reported not having any specific continuous integration environment. All three shared a common trait beside their usage of tools: they were small enterprises operating in the mobile games sector.

Why would a mobile game organization choose to omit automated builds and testing through continuous integration servers? An interviewee from a games organization stated that they did not have need for continuous integration since the developer could quickly, in less than a minute, run all the test locally without the help of external environments.

Still, mobile games – why not games in general – have properties which constrain the usefulness of automated testing which might not be present in other domains as such. In another games organization, the representative explained that there are only few things that could be automated. Testing of a game requires subjective understanding of the elements which eventually make the game fun to play. The same ideal was reflected in the opinions of the third organization: they had a minimal set of tools for testing and they preferred testing of their product on real users. Hence, under special circumstances, continuous integration tools are not always considered essential, especially so if circumstantial factors prevent the fully streamlined flow of a continuous deployment pipeline.

5.3.4. Artifact repository

Artifact repositories were not so commonly used in the organizations. It seems that declaring the dependencies in local project files and downloading the libraries straight from external, public, repositories was more common than actually using an artifact repository within the organization.

An internal artifact repository such as Artifactory was used to store compiled binaries in some cases. Others reported using Sonatype Nexus, and there were some custom solutions for artifact repositories, too.

5.4. Operations

Managing technical infrastructure and deploying software requires attention to be paid to *provisioning and environments* and *deployment* tools. Infrastructure tools for providing environments were modestly used but organizations took advantage of virtualization technologies. Deployment tools for automating the last mile to production systems were uncommon.

5.4.1. Provisioning and environments

Interviewees had a few shared concerns when talking about their infrastructure. Replicating development environments was seen essential by several developers who wished to avoid mistakes originating from incompatibilities between the local environments of developers and those used in later stages such as testing and in production. For that purpose they mentioned to use a tool called Vagrant that can be used with virtualization technologies to clone exact environments from scratch during the build stages. This way, there should be no difference in environments. In one organization a separate product, Ansible, was used to orchestrate the creation and provisioning of environments. VMware and Virtualbox were popular choices for running virtual machines. For configuration management, the organizations preferred to use Chef and Puppet.

In fact, around half of the organizations used some kind of virtualization technologies as part of their workflow. If the products are being hosted in a public or private cloud, it could be considered essential to have tools available for creating, configuring and provisioning the remote servers. Otherwise this work would have to be done partly manually. The organizations reported to use common cloud service providers and their services in various ways.

In one organization they found it useful to first provision the new software versions to completely new virtual machines running in the cloud, waiting the servers to update while still running the old servers with the old version. Only when provisioning of enough machines was complete, the switch to the new version was finally made. By using the virtual machines this way, they did not have to keep their servers down for very long after an update. This is an example how having scalable external resources available can minimize the downtime and the negative environment-related effects of version updates.

5.4.2. Deployment

Deployment tools were not too common and the last mile of deployment involved varying amounts of manual work. Those who were the closest of being able to deploy automatically used a technique called blue-green deployments [17] where a new, identical environment to production called blue is set-up alongside the currently running green one. When the so called smoke tests pass in the blue environment, a switchover between the blue and green environments can be done with little downtime. Jenkins, the general purpose continuous integration server, was one of the tools mentioned to be used in this context. Jenkins was also used in other organizations where deployment was half-manual so that the build job to production or other environments could progress only once approved by chosen individuals. Automated deployment jobs supported by Jenkins were utilized to deliver tested builds from one staged development or testing environment to another, or even to production. Scheduled deployment jobs in Jenkins helped specifically when the deployment had to be done outside office hours.

Capistrano was a choice for some organizations that had deployment tools; this tool was used together with software written in Ruby although it could be used with other programming languages as well. While not strictly related to production environment deployment, one organization used a staging framework called HockeyApp to create a production-like environment in which they could test out their mobile applications before deploying them to production environment. Several organizations reported to have an in-house developed tool for deployment.

5.5. Testing

Testing divides into *unit testing, UI testing* and *acceptance testing*. While common, there were differences in the usage of unit testing tools and practices across industry fields. Difficulties in testing user interfaces showed in the responses although there were some tools used for the task. Automated acceptance testing tools were uncommon.

5.5.1. Unit testing

Most organizations used some kind of unit testing framework which is in line with common development practices. All used unit testing frameworks were open source and depended on the technology stack of the organization. For example, organizations working mostly with Java used JUnit (seven cases), those working on C++ used Google Test or Qt Autotests (three cases) and organizations involved with web development stacks and Javascript used Jasmine or Mocha (two cases).

Unit testing was not mentioned or used in four cases. In two of the cases, specific tools for unit testing were not highlighted in the interviews although unit testing practices were discussed in general. In the other two cases from the mobile game sector, unit testing frameworks were explicitly stated not to exist and in one there were no plans to incorporate such frameworks. These mobile gaming organizations considered unit testing challenging and not too valuable because games rely much on player interaction. Games are also heavily tested manually because graphical and interaction bugs are only detectable by the human eye. Therefore, bugs that unit tests would discover should anyway be discovered when manual testing is done.

5.5.2. UI testing

Several interviewees, especially all from the mobile game domain, said that testing user interfaces is a hard task to automate. Thus, the lack of automated UI testing is one of the biggest challenges stopping the organizations from using a fully continuous deployment pipeline.

Even though these organizations had tools to aid them in UI testing, some additional manual testing was usually involved in full tests. One notable class of test tools include browser automation and cross-browser testing tools such as Browserstack, Selenium and PhantomJS.

5.5.3. Acceptance testing

Acceptance testing is special testing where it is determined if the software system conforms to the requirements, either functional or non-functional ones. However, in the scope of continuous deployment, acceptance testing is usually seen as the last tests to decide if the system can be deployed. As such, it seems that the trust for tools to handle this kind of activity is low. Only five organizations had automatic tools for acceptance testing. All other organizations had either team decisions, product owner decisions or manual testing as the means of acceptance testing. One organization used behavior-driven development tool, called Cucumber, to define acceptance tests.

5.6. Quality

Regarding quality there are overall issues with *quality and performance* as well as *code reviews* to consider. Assuring non-functional properties of products and systems was modestly tool-

assisted with performance being the primary quality attribute being targeted. Code reviews assisted by the tools Gerrit or Github were a relatively common practice.

5.6.1. Quality and performance

Static analysis tools for code quality were not very common in general, but in embedded domains, such as medical and telecommunications, their use was widespread. One interviewee indicated that – as the quality requirements are quite strict in this domain and the deployment process relatively slow – they want to invest the effort to find out quality problems before deployment.

In web service domain, performance was the most important quality aspect. As weak performance may result in bad user experience or denial of service, it is natural that the organizations pay attention to this. New Relic was the most popular tool, but a wide variety of service provider tools and stress tools were also used to ensure performance under heavy loads. These tools include, but were not limited to, Google Webmaster Tools, Google PageSpeed, JMeter, Gatling and Flood. Other group of tools were used to ensure network availability in cloud services. Zabbix, Pingdom and Cloudwatch are examples of such tools reported by the organizations.

Yet again, one interesting group was the mobile game organizations. As their product will be used on multiple platforms, there is a remarkable amount of testing involved in ensuring that the games will run as intended on these. On the other hand, the organizations are rather small, so they face a proportionally huge task whenever changes, which may be platform dependent, are made. All three interviewed mobile organizations had solved this by using subcontracting in testing.

5.6.2. Code review

In practice, tool aided code reviews were almost uniformly conducted via Github pull requests or using the Gerrit tool. Both ways seemed to share similar popularity. Some organizations relied on other tools also, such as FishEye or BeyondCompare. Fully manual reviewing – usually by pair programming – was also used.

The practices vary greatly; some organizations enforce reviewing for all commits to the version control systems, some organizations do not see any additional value for mandatory reviews. However, even those cases might review some parts of the code, if they consider it to be critical for the quality. The interviewees also emphasized the effect of reviews in making the code more easier to understand. This is in line with the findings of Siy and Votta [44].

5.7. Communication and feedback

Based on our interview results, instant messaging tools are quite widespread: Skype, Internet Relay Chat (IRC), XMPP clients, WebEx, Hipchat and Lync were all used for instant messaging. Two organizations used a special purpose instant messaging application – Flowdock – for developer communications. This tool is designed for software development and it has hooks for other development and testing tools. For example, one of these organizations used Flowdock to communicate continuous integration server test failures to developers. Another web development organization used a tool called Slack for similar purposes.

E-mail was also used for communication and for notifications in several development stages. A traditional face-to-face discussion was seen as an important way of conveying a message, too. Tools do not always have to be electronic: one interviewee proclaimed later that a whiteboard and a marker are their most important communication tools they use on a daily basis.

Besides managing internal communication channels, the developers have to deal with external users and their feedback about a software system. Feedback from the users can originate from explicit push communication channels [27] that allow users to explicitly express their wishes for the system. Our interview data suggests that organizations did not have many customer-facing explicit push channels: respondents said that they mainly answered to e-mails, phone calls or messages from contact forms. In one case it was mentioned that a Helpdesk service called One or Zero was used for client communication. In yet another interview it came up that an organization had previously been using a tool called Intercom for handling requests but this real-time system was too taxing for the developers to use in the end as they had to be constantly on call.

When developers take initiative and actively seek information from users, the communication mechanism can be referred to as pull communication [27]. These can be either explicit methods where developers and users collaborate such as typical workshops or interviews, or more implicit where product or service usage data is analyzed. There was some evidence of pull mechanisms and tools being used in our case organizations. Organizations mentioned that they do surveys or personally meet with their users to acquire feedback. Tools for implicit pull communication had limited support in web development as organizations were tracking their users with Google Analytics, Site Catalyst, Flurry or Snoobi and using supporting tools to visualize and understand data gathered from users.

6. Comparing tooling and deployment capability

Next, we focus on studying the connection between tool usage and deployment speed. From the interviews of each case we have collected data about the time it takes to have a minimum deployable software, and how often actual release is done. These metrics respectively correspond to the metrics of *actual releasable software cycle* and *actual release cycle* mentioned in the study synthesis description of Section 3.5. Based on this information, we created a visualization which shows the differences between the release cycle and deployment capability of different cases. We did not have enough information about deployment times for one of the cases (case C17) so it is left out from the observations.

In Fig. 7 the cases are set to timeline by the release time. In the visualization, the deployment potential, meaning the time to build a minimum deployable software, is visualized with black and time spent to actual release is visualized with grey. From the figure it can be observed that the longest release cycle is 1.5 years (case C5) where the shortest cycle is 20 minutes (case C16). We can also see that in many cases the deployment capability is much less than the actual length of the release cycle. For example in case C5 the deployment capability is one month but the release is done only once per 1.5 years. This shows well that the release time is not connected to deployment capability which is the case for many of the cases presented.

As the release time itself is not a good metric in determining how capable the organizations are in deployment, we made another visualization that discards the release time and shows only the time to minimum deployable software. This visualization is shown in Fig. 8.

In Fig. 8 the time unit is now days whereas it was months in the Fig. 7. From the figure we can observe that case C13 has the weakest deployment capability, being 4 weeks, where the case C16 has the best deployment capability, being 20 min.

6.1. Case groups

We can divide the cases into four different groups based on the deployment capability. In the first group, containing the slowest cases, the deployment capability is approximately one month. This

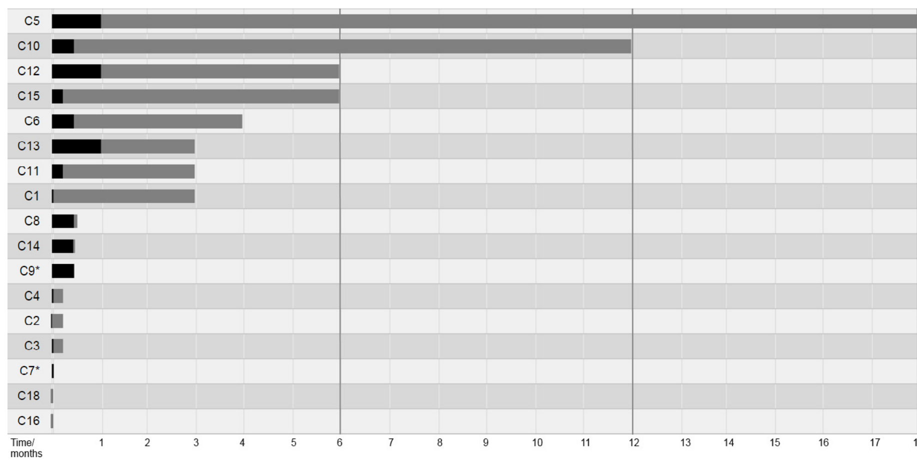


Fig. 7. Release time of cases presented on timeline. Time unit in the figure is months. Black color presents minimum time to deployable software (deployment capability) and grey presents the actual release time.

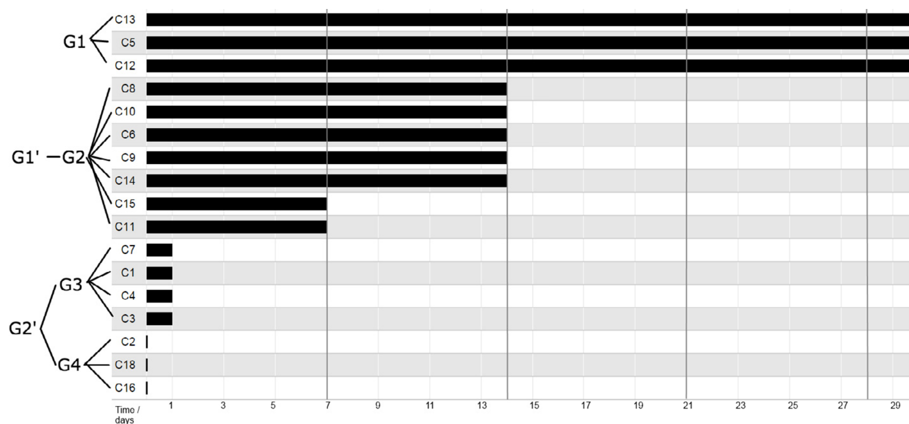


Fig. 8. Deployment capability of cases presented on timeline. Time unit in the figure is days.

group (G1) includes the cases C13, C5 and C12. The second group (G2) includes cases C8, C10, C6, C9, C14, C15 and C11. In this group, the deployment capability is from a week to two weeks. The third group (G3) includes cases which deployment capability is around one day. The cases in the group three are C7, C1, C4 and C3. The last group (G4) contains cases C2, C18 and C16 which deployment capability is less than an hour.

To see if there is any possible connection in tooling and deployment capability we constructed a reordered version of the toolchain matrix presented in Fig. 4. We ordered the matrix based on the deployment capability (to the same order as in Fig. 8) and we divided the matrix into four groups based on the categories presented above. The reordered matrix is presented in Fig. 9. From the toolchain matrix we tried to find similarities within the groups and also compared the groups to see if the groups with better deployment capability have more tooling in use than the groups that have weaker deployment capability.

Looking at the reordered toolchain matrix, we can notice that for the development part of the pipeline, except artifact repository, tooling is present for most of the cases. However, we also observe that the cases with the fewest confirmed tools belong to groups G1 and G2 which are the slowest groups in terms of deployment capability.

We also calculated the amounts of tool aided phases and phases where no tools are used in the toolchain for the organizations. In Table 4, the medians and arithmetic means for both types of phases are presented for the organization groups. These numbers act as indicators for the level of tooling in a certain group. We can see from Table 4 that the means and medians of groups G3 and G4 are close to each other. However, the G1 group has a median similar to G3 and G4, but a mean similar to G2. This observation refers to that the three cases in group G1 do not have high similarity in level of tooling. If we look at the individual cases in G1 we can observe that in cases C5 and C12 the use of tooling is overall in the same level as in the cases in group G3. However, the use of tooling in case C13 is closer to cases in G2 since in the case of C13, tools have been mentioned only for five phases, being closer to the mean of G2 than to G1.

Table 4
Level of tooling between groups. The level of tooling is calculated by counting the number of phases which are tool aided and the number of phases which are not tool aided for each case. The level of tooling for a group is then calculated by arithmetic mean from the level of tooling of the cases in the group, also medians are presented.

Group	mean and median amount of phases			Group	mean and median amount of phases		
G1	no tool used	6	4				
	tool aided	9	11				
G2	no tool used	7.29	7	G1'	no tool used	7.29	7
	tool aided	7.71	8		tool aided	7.71	8
G3	no tool used	3.75	4				
	tool aided	11.25	11	G2'	no tool used	3.43	4
G4	no tool used	3	4		tool aided	11.57	11
	tool aided	12	11				

Fig. 9. Toolchain matrix presented in Fig. 4 reordered by deployment capability.
Domains: **CO** = Consulting, **ES** = Embedded Systems, **IA** = Industrial Automation, **MG** = Mobile Games, **TC** = Telecom, **UI** = UI Framework, **WS** = Web Service *Groups:* Cases are grouped by the deployment capability. **G1'** = slow deployment capability, **G2'** = fast deployment capability.

Based on the low similarity between cases in group G1, we discarded this group from further observations. Now group G2 can be renamed into group G1'. Furthermore, groups G3 and G4 can be combined into a new group named G2' based on the similarity. The means and medians for these new combined groups are also presented in the Table 4. Based on the data we can state that there is clearly more tool support in the group G2' which also have overall better deployment capability than group G1'.

When comparing the groups G1' and G2' we can find differences in tooling in many areas. In group G1' four cases out of seven uses unit test tools where in group G2' all cases use unit test tools. User interface testing is tool aided for all cases in G2' where in group G1' only two cases have stated to use tools for user interface testing. Differences can also be found from backlog management and quality and performance testing. In group G2' all cases have stated that they use tooling for backlog management where in group G1' four cases out of seven have tooling for backlog management. For quality and performance testing tooling is used in three cases in group G1'. Conversely, in group G2' all use tooling for quality and performance testing. For both groups, acceptance testing and code review were the areas where least tooling were used.

The observations refer to possible relation between tooling and deployment capability and seconds the statement that firm automated tool support is needed to be able to deploy fast. However, in the original group division group G1, which contains the three cases with the weakest deployment capability in terms of time, the use of tooling was somewhat in the same level as in G2' (the group with the best deployment capability). Based on this observation we can state that the degree of tooling is not a sole guarantee of fast deployment capability.

7. Discussion

In this section, the research questions of the study are answered and compared to related work. Limitations to the validity of the study are also discussed and possible directions for future work explored towards the end of the section.

7.1. RQ1: which toolchains are used to support software delivery?

Several observations can be made from the interview data. Looking at the interview results in Fig. 4, some of the tool categories are used in almost all organizations. These *commodity tools* include *version control, backlog management, build, continuous integration, unit testing* and *bug tracking*. They are used in over two thirds (12 cases) of the development organizations. These tools can be seen as an industry standard, and almost every software development project should incorporate them in the development activities. However, at least with unit testing there seems to be some domain-related controversy. In the mobile game industry, unit testing is mentioned in only one organization which indicates that it is not a necessary component in every domain.

Some of the tool categories are not used as much. These *modestly used tools* are *provisioning and environments, UI testing, performance testing, code review, deployment,* and *customer feedback*. They are used in a third or two thirds of the development organizations (from six to twelve cases). It might be that some of these categories are not so useful in every context and domain. For example, UI testing is naturally unnecessary if the software does not have a UI. In addition, some categories can be performed well without any software tooling, such as code reviews and communication.

Some tools are used only in a third or fewer (fewer than six cases) of the development organizations. These *uncommon tools* include *requirements elicitation, artifact repository, provisioning and environments, testing of quality attributes* and *acceptance testing*. Again, requirements elicitation, for instance, can be done without tooling. Likewise, many organizations stated that acceptance testing is done manually. However, automating IT infrastructure, acceptance tests and having an artifact repository are considered to be important parts of continuous delivery [17].

Meyer [33] listed required tools for continuous integration. He mentions that use of version control, build, continuous integration, deployment and monitoring tools are necessary for the practice of continuous integration. In our cases, tools other than deployment and monitoring tools are used in the majority of the organizations. Deployment and monitoring (feedback category) tools are

also used in over half of the development organizations (10 cases for deployment, 12 cases for feedback), but not in as many organizations as the other tools mentioned by Meyer. Thus, overall the case organizations seem to have comprehensive tooling for continuous integration, but the later parts of the continuous deployment pipeline are not covered in all organizations.

Based on the domain of the cases, it seems that mobile gaming industry does not rely on tooling as much as other domains. This might be because mobile games rely on heavy manual testing and games are typically so small in size as applications that it is feasible to test them manually.

When looking at the cases when interviewees have explicitly said *not* using a tool category, acceptance testing was most often mentioned as a missing component from the toolchain (8 out of 18 cases). This implies that automated acceptance tests are still quite rare and thus, continuous deployment is not feasible for most of the organizations.

7.2. RQ2: what are the reasons behind the gaps in the toolchains?

Based on the interviews of organization representatives, a number of gaps were identified in the toolchains of the organizations, and as noted, some of the tools for specific software engineering activities were more prevalent than others. Several reasons for the absence of tools in these areas surfaced in the analysis.

When a specific tool is not being used for an activity as part of the toolchain in an organization, it does not necessarily mean that the activity is completely disregarded. For instance, several of the organizations that were not using unit testing tools still relied on external testing services where they sent their products to be checked. While conserving internal resources, outsourcing testing might cause a notable delay in the flow of the continuous deployment pipeline. An activity can also be performed completely without tools. Code reviews are examples of activities where a tool can be helpful but not entirely necessary as peer reviews can be made more or less manually.

Since parts of the continuous deployment pipeline are interconnected, the tools have contingencies and are partly dependent on the existence of other tools. Those organizations that did not have unit testing tools also did not have user interface testing tools or continuous integration servers running. If there are no tests to execute, having a continuous integration server might not be worthwhile.

There might also be a gap in the toolchain of an organization because the tool is not seen as relevant. If an organization is mostly operating in the embedded system domain, it does not have a strong incentive to use tools for user interface testing. In this sense, a gap cannot always be seen as a flaw in the continuous deployment pipeline of a specific organization: the context and domain play an important role as to which tools are seen integral and which are not.

Widespread usage of a tool might be hindered by insufficient knowledge about the tool and the value it might provide to the organization. For instance, artifact repositories were only confirmed to be used by a handful of organizations and perhaps not many had seen the benefits of incremental builds or proper internal management and versioning of in-house libraries. Continuous feedback systems shared a similar fate to artifact repositories in a way. There were numerous internal communication tools used but far fewer were utilized to interact directly with the customers and end users or used implicitly to observe user behavior. Constant feedback can put a strain on the developers and thus keeping feedback loops closed can be a conscious choice.

Concerning the final push of the software version and deployment of the application to the production environment, organizations did seem to have a certain number of associated tools but the workflow was not fully automated. Deployments could in some cases be done with one-button pushes but deployment was preceded by manual verification and validation activities in a vast number of the cases. It is plausible that organizations might not see fully automated deployment tools attractive due to their manual activities in this stage of development. The various manual steps and checks performed before deployment might also explain why the adoption of acceptance testing tools was low: the organizations were somewhat unaccustomed to capture user scenarios as acceptance tests and were faced with the fact that user acceptance had to be obtained through more conventional means.

7.3. RQ3: what possible implications do toolchains have on the speed of software delivery?

Connection between toolchains and speed of software deployment time was studied based on the data gathered from the interviews. We noticed that the actual deployment time did not have much relevance to deployment capability which we defined as the minimum time to deployable software. We also observed that for the fastest cases the deployment capability and also the deployment time can be measured in minutes whereas for the slowest cases the deployment capability is over a month and the actual deployment time can even be well beyond a year.

As the focus was to study if there is connection between tooling and deployment capability, we divided the cases into groups based on their deployment capabilities and reordered the toolchain matrix presented in Fig. 4 to match the order of cases in deployment capability timeline. The reordered toolchain matrix is presented in Fig. 9. We observed that the toolchain is more complete in cases that can deploy every day or more often. Based on this observation we state that the toolchain should be as complete as possible to enable daily deployments. However, we also observed that even a good toolchain is not a guarantee for fast deployment. In some cases the deployment was done seldom but the toolchain was on the same level with the cases that deploy fast. In addition, cases in group G2 lacked many tools, but the organizations were still able to deploy every two weeks.

7.4. Future work

In this section, we propose future research topics based on our observations. One of the research discoveries was that organizations tend to have a better capability to deliver software than they actually do in practice. This raises the question why there is such a great difference between the capability to deliver and the actual release cycle? Domain differences explain this to some extent, as in some domains customers are more reluctant to acquire new releases, but there could be other factors at play here. If the delivery capability of an organization is a lot shorter than the actual release cycle, does it still make sense to do rapid and small releases internally? For the organizations which have already set a steady and relatively rapid pace, what are the benefits of having daily releases as opposed to bi-weekly releases?

In this study, we asked the interviewees to describe their development process and name the tools they used while developing software but we did not ask them to rate the tools for usefulness. Which tools in which development stages are the most critical when it comes to rapid delivery and what tools do developers themselves find the *most* useful? Are the same tools needed if an organization only strives for a bi-weekly release cycle rather than a daily release cycle?

Besides seemingly having an effect on the capability to release software faster, toolchains could also have an impact, positive or negative, on product or process qualities such as defect rates. A strong support for test automation tools could affect perceived

product failures as could tools in other development stages like operations and deployment; reduced manual configuration could potentially result in fewer mistakes being made. An increased reliance on tools and automation could mean, however, that certain types of bugs would go unnoticed if less human attention is paid to each release. The overall relationship between toolchains and product qualities remain somewhat unclear and this could be yet another avenue for future research.

7.5. Threats to validity

Case studies may have a number of threats concerning their validity. These threats can be classified into four themes: construct validity, internal validity, external validity and reliability [45]. In this chapter, we will address each of these four themes with respect the results of our study. In addition, we address threats to validity concerning open-ended interviews as a data collection method.

7.5.1. Construct validity

Construct validity refers to how well operational measures represent what researchers intended them to represent in the study in question. For an interview study such as ours, this means ascertaining the extent to which interview questions were interpreted the same way by both interviewers and interviewees. Considering toolchains, we find there is little room to interpret direct questions regarding their use differently. In addition, initial results from our interviews were sent to interviewees for inspection and approval, further strengthening the construct validity. Construct validity is to an extent threatened by the ability of the interviewees to recall their development process and related tools as other sources of evidence were not used besides the interviews. Having the interviewees illustrating their development process by drawing and asking them to add the tools to the depicted phases later reduces this threat to construct validity.

Considering frequency metrics, there is a certain threat to construct validity. Interview responses indicated that constructs like release, deployment and delivery might not be interpreted in the same way by all stakeholders. In order to mitigate the threat, the metrics were defined more precisely after the interviews and the interview responses were checked to match the definitions.

7.5.2. Internal validity

Internal validity concerns studies in which causal relationships are examined. In particular, internal validity concerns efforts made to ensure that possible confounding factors are identified and alleviated. Threats to internal validity in the study arise from the implications of the amount of tool-aided software development phases in the organizations.

In the analysis and discussion of data it is implied that the degree of tool-aided phases has an effect on the release frequency of the organizations. The possibility of rival explanations for the speed of delivery cannot be ruled out as there are confounding factors which might affect the internal and external release cycles of an organization. The pattern-matching logic held in most cases, but not all, and confounding factors such as the domain of the organizations could already be identified. No strong causal claims are made due to the descriptive nature of the study but the implied validity threat is acknowledged.

7.5.3. External validity

External validity concerns the extent to which results of a case study can be generalized. In most cases, results of a case study cannot be wholly generalized [23]. The multiple-case design and the replication logic with cross-case analysis increases the external validity of this study. Limited generalization for similar contexts may be possible although there were not many cases for each domain, which poses a threat to external validity. While not guaranteed, a practitioner operating in a particular domain may gain insight to the development of their toolchain by comparing the study context and their own environment, and applying the findings.

7.5.4. Reliability

Reliability concerns the extent to which results are dependent on particular researchers. Ideally, a study should be able to be repeated by other researchers, achieving the same results [45]. We cannot guarantee that the coding of data in our thematic analysis is entirely void of bias; the reliability of any thematic analysis is inherently subject to differences in interpretations by researchers [46], although we mitigated this bias by sending the results back to the organizations for verification. The confidence in the results of the thematic analysis is increased by the amount of researchers involved in the study as the results and findings were cross-checked by multiple researchers.

8. Conclusions

Continuous deployment requires seamless collaboration between people working on various aspects of software engineering and a streamlined product flow from one stage to another. Using a sufficient amount of tools can help in achieving a state of continuous deployment but it is not always straightforward to select the right tool for the job.

The results of the interview study carried out in Finnish software development organizations reveal the prominence of tools in different development stages and that some tools can be considered more important in terms of continuous deployment than others. However, the relative importance of a tool is specific to an organization and the domain the organization operates in as the need for the speed of deployment can differ.

While there are activities, such as code review, that can be done without specialized tools, having proper tool support is essential in certain stages of development. Version control systems are so prevalent in the industry that they were found in all of the interviewed organizations. Although having somewhat lower support overall, tools in the areas of backlog management, building and testing had high adoption rates in the organizations as well. Support for other tools varied more and the results seem to indicate that the completeness of the whole toolchain had an effect on the delivery and deployment capabilities of the organizations but there were exceptions to the rule.

All organizations might not even be looking to fully achieve the state of continuous deployment which has a direct impact on their toolchain. Without apparent need to deliver software faster to the end users, there is no subsequent need for tools in the toolchain that enable continuous deployment.

Automating stages and the application of tools especially in the later stages of development, including system testing and deployment, might not be restricted solely by the availability of appropriate tools. Prevailing company culture and human nature can increase reluctance in adopting tools as there seems to be a tendency to rely on manual inspection rather than automated tests to spot latent errors and to verify the readiness of a release. Good reasons for retaining control over releases exist but practitioners should be aware that the reasons might as well be cultural or related to having adequate trust on the automated toolchain.

It can be concluded that a well-established tool infrastructure supports shipping software releases continuously. Tools ranging from facilitating management of requirements, easing daily development activities, ensuring quality, speeding deployment, to handling rapid user feedback all may improve the delivery cycle. Organizations employing such tools in practice might just get the extra

boost to development that would allow them to shorten release cycles and speed the delivery of new software to the ever-evolving markets.

Acknowledgments

We would like to thank the organizations for describing their development processes and the participants to the interviews for their time and effort. This article was supported by TEKES [47] as part of the N4S Program [25] of DIGILE (Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

Appendix A. Interview outline

Table A.1
Interview topics. Questions used in this study are shown under the topics.

ID	Related Research Questions	Type	Question
1			**Background Information**
1.1	–	Closed-ended	How large is your organization?
1.2	–	Open-ended	What is the domain of the project?
1.3	–	Closed-ended	What is the size of the team you represent?
2			**Methods**
2.1	–	Open-ended	What software engineering methods are you using?
2.2	RQ1, RQ2	Open-ended	Describe and draw your process from requirements to release?
2.3	RQ3	Open-ended	How often do you release?
3			**Testing**
3.1	RQ1, RQ2	Open-ended	How is the product tested during the development process?
3.2	RQ1, RQ2	Open-ended	What testing tools and frameworks are you using?
4			**Operations**
5			**Toolchain**
5.1	RQ1	Open-ended	Add names of the tools used in the process picture drawn at question 2.2.
5.2	RQ2	Open-ended	Are there manual steps in the delivery pipeline? How long do they take?
5.3	RQ1	Open-ended	Do you use life cycle management systems?
6			**Life Cycle Management**
7			**Feedback**
7.1	RQ3	Closed-ended	What is your lead time from a customer bug report to release?
7.2	RQ3	Closed-ended	What is your lead time from the event when testing is completed to release?
7.3	RQ3	Closed-ended	How fast can you release a change of one source code line?
8			**Delivery Challenges**
9			**Delivery Benefits**

References

[1] D.J. Anderson, Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results, Prentice Hall Professional, 2003.
[2] K. Schwaber, Scrum development process, in: Business Object Design and Implementation, Springer, 1997, pp. 117–134.
[3] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Professional, 2000.
[4] K. Beck, M. Beedle, A.v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, Manifesto for Agile Software Development, 2001, Retrieved: August 2016. URL http://agilemanifesto.org.
[5] J. Humble, The Case for Continuous Delivery, (http://www.thoughtworks.com/insights/blog/case-continuous-delivery). Retrieved: August, 2016.
[6] T. Fitz, Continuous Deployment, 2009, Retrieved: November 2015. URL http://timothyfitz.com/2009/02/08/continuous-deployment/.
[7] M. Leppänen, T. Kilamo, T. Mikkonen, Towards post-agile development practices through productized development infrastructure, in: Rapid Continuous Software Engineering (RCoSE), 2015 IEEE/ACM 2nd International Workshop on, 2015, pp. 34–40.
[8] Y. Monden, Toyota production system, J. Oper. Res. Soc. 46 (5) (1995) 669–670.
[9] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003.
[10] B. Fitzgerald, K.-J. Stol, Continuous software engineering and beyond: trends and challenges, in: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering, in: RCoSE 2014, ACM, New York, NY, USA, 2014, pp. 1–9.
[11] P. Debois, Devops: A software revolution in the making, J. Inf. Technol. Manage. 24 (8) (2001) 3–5.
[12] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.
[13] M. Fowler, M. Foemmel, Continuous Integration, 2005. http://www.martinfowler.com/articles/continuousIntegration.html, Retrieved: August 2016.
[14] M. Fowler, ContinuousDelivery, 2013, Retrieved: August 2016. URL http://martinfowler.com/bliki/ContinuousDelivery.html.
[15] M. Poppendieck, T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[16] S. Neely, S. Stolt, Continuous delivery? easy! just change everything (well, maybe it is not that easy)., in: AGILE, IEEE Computer Society, 2013, pp. 121–128.
[17] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st, Addison-Wesley Professional, 2010.
[18] D. Ståhl, J. Bosch, Experienced benefits of continuous integration in industry software product development: a case study, in: IASTED Multiconferences - Proceedings of the IASTED International Conference on Software Engineering, SE 2013, ACTAPRESS, 2013, pp. 736–743.
[19] D. Ståhl, J. Bosch, Modeling continuous integration practice differences in industry software development, J. Syst. Softw. 87 (2014) 48–59.
[20] D. Ståhl, J. Bosch, Automated software integration flows in industry: A multiple-case study, in: Companion Proceedings of the 36th International Conference on Software Engineering, in: ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 54–63.
[21] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M.V. Mäntylä, T. Mannistö, The highways and country roads to continuous deployment, Softw. IEEE 32 (2) (2015) 64–72.
[22] G.G. Claps, R.B. Svensson, A. Aurum, On the journey to continuous deployment: Technical and social challenges along the way, Inf. Softw. Technol. 57 (0) (2015) 21–31.
[23] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Softw.Eng. 14 (2) (2009) 131–164.
[24] C. Wohlin, A. Aurum, Towards a decision-making structure for selecting a research design in empirical software engineering, Empirical Softw. Eng. (2014) 1–29.
[25] Digile N4S, 2015, Retrieved: August 2016. URL http://www.n4s.fi/en.
[26] D. Cruzes, T. Dyba, Recommended steps for thematic synthesis in software engineering, in: International Symposium on Empirical Software Engineering and Measurement (ESEM), 2011, pp. 275–284.
[27] W. Maalej, H.-J. Happel, A. 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, ACM, New York, NY, USA, 2009, pp. 981–990.
[28] S. Otte, Version control systems, Proseminar Technische Informatik, Computer Systems and Telematics Institute of Computer Science Freie Universität Berlin, Germany, 2009.
[29] S. McIntosh, B. Adams, A.E. Hassan, The evolution of Java build systems, Empirical Softw. Eng. 17 (4–5) (2012) 578–608.
[30] C. Prasad, W. Schulte, Taking control of your engineering tools, Computer 46 (11) (2013) 63–66.
[31] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, SIGPLAN Not. 47 (6) (2012) 263–274.
[32] J. Shore, Continuous Integration on ADollar ADay, 2006, Retrieved: August 2016. URL http://www.jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html.
[33] M. Meyer, Continuous integration and its tools, IEEE Softw. 31 (3) (2014) 14–16.
[34] M. Fowler, Bluegreendeployment, 2016. (http://martinfowler.com/bliki/BlueGreenDeployment.html), Retrieved: August.
[35] S. Ghobadi, What drives knowledge sharing in software development teams: a literature review and classification framework, Inf. Manage. 52 (1) (2015) 82–97.
[36] Tool Appendix, 2016, Retrieved: August. URL https://goo.gl/qr6KU5.
[37] J.M.C. de Gea, J. Nicolas, J.L.F. Aleman, A. Toval, C. Ebert, A. Vizcaino, Requirements engineering tools, IEEE Softw. 28 (4) (2011) 86–91.
[38] E. Bjarnason, K. Wnuk, B. Regnell, A case study on benefits and side-effects of agile practices in large-scale requirements engineering, in: Proceedings of the 1st Workshop on Agile Requirements Engineering, AREW '11, ACM, New York, NY, USA, 2011, pp. 3:1–3:5.

[39] M. Dubakov, P. Stevens, Agile Tools. The Good, the Bad and the Ugly., Technical Report, TargetProcess, 2008. Retrieved: August 2016, URL http://www.targetprocess.com/download/whitepaper/agiletools.pdf.
[40] V. Driessen, A Successful Git Branching Model, 2015. Retrieved: August 2016. URL http://nvie.com/posts/a-successful-git-branching-model/.
[41] A. Pelletier, Another Git Branching Model, 2016. Retrieved: August 2016. URL http://blogpro.toutantic.net/2012/01/02/another-git-branching-model/.
[42] R. Fay, A Rebase Workflow for Git, 2016. Retrieved: August 2016. URL http://randyfay.com/content/rebase-workflow-git.
[43] D. Spinellis, Version control systems, Softw. IEEE 22 (5) (2005) 108–109.
[44] H. Siy, L. Votta, Does the modern code inspection have value? in: Software Maintenance, 2001. Proceedings. IEEE International Conference on, 2001, pp. 281–289.
[45] R.K. Yin, Case Study Research: Design and Methods, Sage publications, 2014.
[46] G. Guest, K.M. MacQueen, E.E. Namey, Applied Thematic Analysis, Sage, 2011.
[47] Tekes — The Finnish Funding Agency for Innovation, 2016. Retrieved: August. URL http://www.tekes.fi/en/tekes/.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3898-8
ISSN 1459-2045