An Empirical Study of Architecting and Organizing for DevOps

Author: **Mojtaba Shahin**

Principle Supervisor: Prof. Dr. Muhammad Ali Babar
Co-supervisor: Prof. Dr. Liming Zhu

A thesis submitted for the degree of Doctoral of Philosophy

in

Centre for Research on Engineering Software Technologies (CREST)
School of Computer Science
Faculty of Engineering, Computer and Mathematical Sciences
The University of Adelaide

August 2018

Contents

List of Figures

List of Tables

Abstract

An Empirical Study of Architecting and Organizing for DevOps

by Mojtaba Shahin

Attracted by increasing the need of being able to improve business competitiveness and performance, many organizations have started to optimize themselves to develop and deliver high-quality values more quickly and reliably. Development and Operations (DevOps) is emerging as a promising approach in the software industry to help organizations to realize this goal. However, establishing DevOps practices, specifically continuous delivery and continuous deployment practices, in the industry is a challenge as it requires new organizational capabilities and novel techniques, methods and tools for application design, testing and deployment.

Most research on DevOps focuses on tooling support, improving automation in testing and deployment, improving performance and integrating security into the deployment process to initiate and implement DevOps. To date, little is known about the impact of continuous delivery and deployment as two main DevOps practices on organizational structure (i.e., team structure) and the architecture of a system, those that are supposed to be fundamental limitations to adopt these practices.

This thesis aims at filling this gap by conducting a set of empirical studies. We first design and conduct a systematic literature review to gain a comprehensive understanding of the concept of continuous delivery and deployment and the current state of research in this regard. Second, we design, implement and analyze a large-scale mixed-methods empirical study, consisting of 21 interviews and 98 survey responses. Finally, we conduct an in-depth industrial case study with two teams in a case company to explore the role of software architecture in DevOps transition. The empirical studies contribute to (1) provide detailed insights into the specifics of challenges moving from continuous delivery to continuous deployment; (2) find how teams are organized in software industry for adopting continuous delivery and deployment; and (3) develop evidence-based guidelines on how to (re-) architect an application to enable and support continuous delivery and deployment.

Declaration

I, **Mojtaba Shahin**, certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint award of this degree.

I give consent to this copy of my thesis when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

I acknowledge that the copyright of published works contained within this thesis resides with the copyright holder(s) of those works.

I give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

I am also truly thankful to Data61, a business unit of CSIRO, that has partially supported my PhD.

Date: 2/08/2018

Signature

Acknowledgments

First and above all, I wish to thank God, the Almighty, for all the gifts that I have had in my life: health, strength, and a great family.

I would like to express my deepest gratitude to my principal supervisor, Prof. Dr. M. Ali Babar. I really thank Ali for giving me the opportunity to conduct the research on DevOps. It would not have been possible to finish this thesis without his patience, wisdom, endless support, constructive criticism, insightful comments, and motivations along the journey of my PhD.

A heartfelt thanks to Prof. Dr. Liming Zhu, my co-supervisor, who has played an informative role in my PhD. Liming helped me to improve the quality of my papers by providing the detailed reviews and excellent comments.

I wish to thank my collaborator, Dr. Mansooreh Zahedi, for her excellent comments and insights which contributed to this thesis.

I would like to thank my examination committee members, Prof. Dr. Filippo Lanubile and Dr. Jingyue Li. I greatly appreciate your time and valuable comments on this thesis.

I must thank all the members of CREST: Faheem Ullah, Bakheet Aljedaani, Chadni Islam, Jamal El Hachem, Nguyen Khoi Tran, Matthew Thyer, Benjamin Ramsey, Victor Prokhorenko, and Hao Chen. It was a pleasure to work with you all. A special thanks should go to Faheem for his feedback and comments on the early versions of my papers and presentations.

I should give credit to Prof. Peng Liang, who has been playing a significant role in my research career since I was a master student. Peng introduced me to the world of software engineering research and has always been supporting me as a good friend and a mentor.

I am also truly thankful to all the participants and companies involved in this research. Thank you for your time and sharing your knowledge and experiences on DevOps.

I want also to thank all my teachers in the school and the university, who have helped and supported me to achieve the PhD.

I must thank my extended family members. I cannot name all of you, but I would like to express my deepest gratitude to all of you, especially my uncles, Abbas, Shahram, Shahpoor, and Farshid, my aunts, Golchin, Esmat, Eshrat, Shiva, and Atefeh, and all my cousins. You have always been motivating me to pursue my education.

The last but not the least, I would like to thank all members of my family. It is absolutely impossible to find the right words to express my feeling about them. A great thanks should go to my sister, Farzaneh, for being a kind, compassionate, inspirational, and supportive sister all the time. I must thank my brother-in-law, Yaser, for his continuous support. Special thanks to my lovely niece, Taraneh, for being the biggest source of happiness for me and my family. I am immensely grateful to my brothers, Farzan and Fardin. Thank you so much for your love and supporting our family and taking all responsibilities instead of me. I am forever grateful to my parents, Avazali and Zomorod, for all their kindness, sacrifices, love, and support. My father is my role model in any aspect of my life. For me, he is truly the icon of a family man as he has sacrificed all his life to provide us with opportunities to be successful. I should express my deepest gratitude and affection to my mother, who has always been the front-line supporter in all aspects of my life with unconditional love. Without you guys, I could not have finished my PhD.

Mojtaba Shahin

2/08/2018

Dedication

To my family

Chapter 1

Introduction

1.1 Background

To succeed in a world where the requirements of customers and technologies change at the speed of light, software organizations need new paradigms for software development, in which they outpace their competitors by delivering changes to customers faster. Development and Operations (DevOps) is a new movement in the software industry to solve the disconnect between development and operations teams by promoting collaboration, communication, and integration between them [6]. Put another way, DevOps is characterized by treating operations staff as first-class stakeholders of the software development process by bringing them close to the software development team right from the beginning [7]. The main idea behind integrating development and operations stuff is to reduce the time between committing a change and deploying the change into production without quality degradation [7]. A set of technical and social-technical practices are associated with DevOps including continuous integration, continuous delivery, continuous deployment, and infrastructure as code [8, 9]. Through continuous integration, continuous delivery, continuous deployment, IT organizations are enabled to accelerate delivering high-quality value to customers.

Continuous Integration (CI) is a widely established development practice in software development industry [10], in which members of a team integrate and merge development work (e.g., code) frequently, for example, multiple times per day. Continuous integration enables software companies to have shorter and frequent release cycle, improve software quality, and increase their teams' productivity [10]. This practice includes automated build and testing [11].

Continuous delivery is a software engineering approach that aims at keeping software releasable all the time after successfully passing automated tests and quality checks [12, 13]. Continuous delivery employs a set of practices e.g., continuous integration, and deployment automation to deliver software automatically to a production-like environment [9]. According to [13, 14], this practice offers several benefits such as reduced deployment risk, lower costs, and faster user feedback. Figure[1] 1.1 indicates that having continuous delivery practice requires continuous integration practice.

Continuous deployment practice goes a step further in order to automatically and continuously deploy new changes to production or customer environments [12, 15]. There is a robust debate in academic and industrial circles about defining and distinguishing between continuous delivery and continuous deployment [10, 12, 13]. It is mainly because continuous delivery and deployment are highly correlated and intertwined, and their meanings highly depend on how a given organization interprets and employs them [16, 17]. Continuous deployment is a push-based approach and should not include any manual steps. It means that as soon as developers commit a change, the change is released to a production environment through a pipeline [18]. In contrast, continuous delivery is a pull-based approach for which a business decides *what* and *when* to release [18]. In other words, the scope of continuous delivery does not include frequent and automated release, and continuous deployment is consequently a continuation of continuous delivery. Whilst continuous delivery practice can be applied to all types of systems and organizations, continuous deployment practice

[1] Note that the icons that are used in the figures of this thesis are taken from **freepik.com** and **thenounproject.com**

may only be suitable for certain types of organizations or systems [13, 18, 19]. Figure 1.1 shows the relationship between these practices. We also present how an application is deployed to different environments [20]. Whilst a production environment is where applications or services are available for end users, a staging environment aims at simulating a production environment as closely as possible.

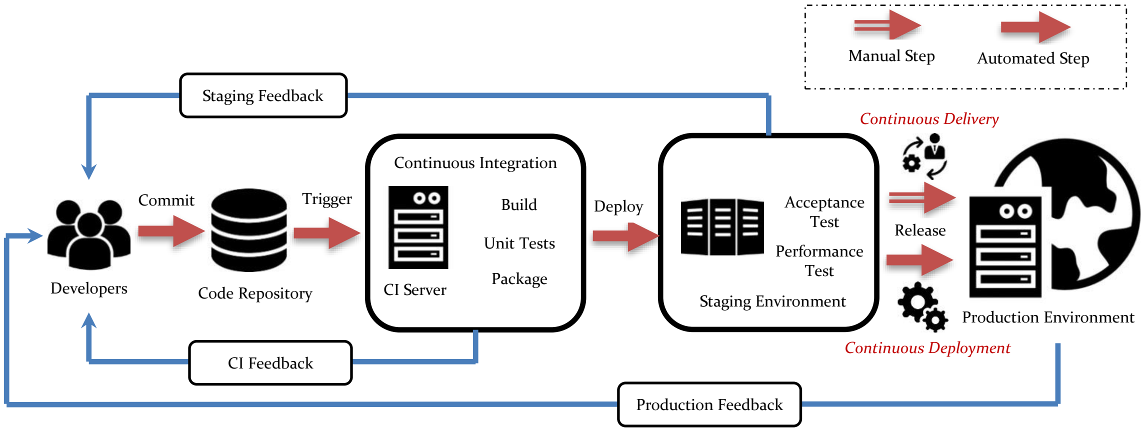


Figure 1.1 The relationship between continuous integration, delivery, and deployment

1.2 Research Objectives and Questions

DevOps promises many benefits such as improvement in the business competitiveness and performance, faster development and deployment of new changes, and faster failures detection [7, 9, 21]. For example, it has enabled many highly innovative organizations such as Facebook, Netflix, and Etsy to significantly reduce time to market as they release software changes to their customers multiple times a day [22]. Continuous Delivery and Deployment (CD) are the key DevOps practices to suitably realize the promises of DevOps [7]. However, implementing CD practices might be a challenging task for IT organizations because they may need to change and/or augment organizational processes, practices, and tools, which may not be ready to support the highly complex and challenging nature of these practices.

Software engineering community has recently started a notable investigation of different aspects of CD. Existing studies have mostly focused on reporting the challenges of CD adoption [11, 23, 24], tooling support [25-27], improving automation in testing and deployment [28, 29], improving performance [30, 31] and integration security [32] into the deployment process to initiate and implement CD. On the other hand, it has been recently proclaimed that the fundamental limitations to adopting these practices are deeply ingrained in the organizational structure (i.e., team structure) [33] and the architecture of a system [7, 9, 34] and fixing these limitations often requires making alignment between organizational structure and software architecture [35]. It is mainly because according to Conway's Law "*organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations*" [36]. Whilst, academic, and industry communities have called a new line of research to fill in this gap [9, 37-40], there are a few studies on this subject and almost all of them are not across multiple projects, organizations, and contexts. Overall, the goal of this thesis is to empirically investigate how organizational structure and software architecture being impacted by or impacting CD practices [41]. This PhD thesis provides guidelines in this regard with empirical evidence for research and practice and thereby increasing the success of adopting CD and suggesting recommendations for better practices and tools development.

Problem Statement: A deeper understanding of continuous delivery and deployment practices and their impact on architecting and organizational structures is needed as it is expected to provide an evidence-based body of knowledge for practitioners and researchers in order to support further development and adoption of these practices.

To realize the goals of this thesis, three high-level research questions, along with their sub-research questions are defined, which would be the focus of this thesis (See Table 1.1). Moreover, Table 1.1 shows which research methods are used to answer the research questions and indicates corresponding chapters where the research questions are addressed.

Table 1.1 An overview of research questions and research methods used to answer them

High-level Research Questions	Sub Research Questions	Chapter #	Research Method
RQ1. What is the state of art of continuous integration, delivery and deployment research?	**RQ1.1** What approaches and associated tools are available to support and facilitate continuous integration, delivery, and deployment?	Chapter 3	Systematic Literature Review
	RQ1.2 Which tools have been employed to design and implement deployment pipelines (i.e., modern release pipeline)?		
	RQ1.3 What challenges have been reported for adopting continuous integration, delivery, and deployment?		
	RQ1.4 What practices have been reported to successfully implement continuous integration, delivery, and deployment?		
RQ2. What are the organizational impacts of CD?	**RQ2.1** What factors do limit or demotivate moving from continuous delivery to continuous deployment?	Chapter 4	Mixed-methods Study
	RQ2.2 How are development and operations teams organized to initiate and adopt continuous delivery and deployment?	Chapter 5	
	RQ2.3 How is collaboration among teams and team members improved for adopting continuous delivery and deployment?		
	RQ2.4 How does adoption of continuous delivery and deployment impact on team members' responsibility?		
RQ3. What are the architectural impacts of CD?	**RQ3.1** How should an application be (re-) architect to enable and support continuous delivery and deployment?	Chapter 6	
	RQ3.2 What key architectural decisions are made by a case company to adopt DevOps?	Chapter 7	Case Study

1.3 Thesis Contributions

The key contributions of this thesis can be categorized into six areas:

1. Establishing a solid background knowledge of three key practices of DevOps, namely continuous integration, delivery and deployment (Chapter 3)

    - A taxonomy of the approaches, associated tools, and challenges and practices of continuous integration, delivery, and deployment in an easily accessible format
    - A set of factors that a given organization should carefully consider when implementing continuous integration, delivery and deployment practices

2. Designing and conducting the largest empirical study, to date, concerning the state of the practice on architecture and organizational aspects of continuous delivery and deployment (Chapter 2).
    - Publicly available survey instrument enables other researchers to replicate our study in different organizations and contexts
    - A publicly available dataset of 98 anonymized survey responses
3. Understanding why continuous delivery has been adopted more than continuous deployment in the industry (Chapter 4)
    - Understanding the structure (e.g., phases) and limitations of the deployment pipelines in the industry
    - Identifying confounding factors that influence moving to continuous deployment from continuous delivery
4. Understanding the impact of continuous delivery and deployment on teams (Chapter 5).
    - Identifying four distinct working styles of organizing development and operations teams
    - Providing an evidence-based understanding of collaboration strategies for CD
    - Identifying new/required responsibilities and skills for succeeding in CD
5. Understanding practitioners' perspectives on software architecture and design aspects of continuous delivery and deployment (Chapter 6)
    - A better understanding of practicing CD within monoliths
    - Characterizing "small and independent deployment units" principle attempted by the practitioners to ease a CD journey
    - Identifying a set of CD-driven quality attributes
    - A conceptual framework to (re-) architect for CD
6. Exploring a DevOps journey from software architecture perspective in a case company (Chapter 7)
    - Identifying, documenting and analyzing the architectural decisions and their implications made by the case company to implement DevOps

1.4 Outline of Thesis and Publications

The core chapters of this thesis are derived from the publications which have been previously published or are currently under submission. Figure 1.2 shows an overview of the thesis scope and outline. The rest of this thesis is organized as follows:

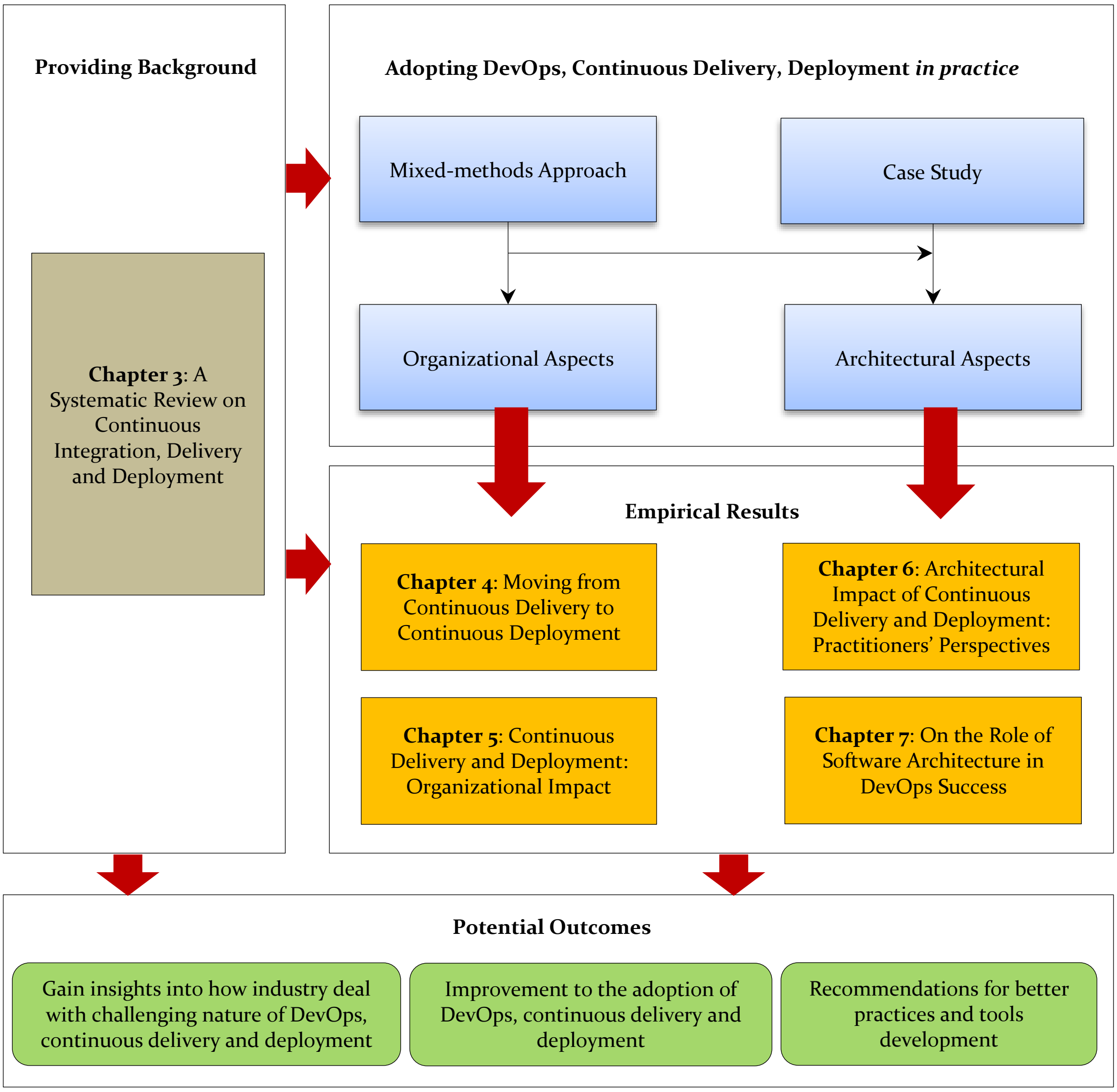

Providing Background
Chapter 3: A Systematic Review on Continuous Integration, Delivery and Deployment
Adopting DevOps, Continuous Delivery, Deployment in practice
Mixed-methods Approach
Case Study
Organizational Aspects
Architectural Aspects
Empirical Results
Chapter 4: Moving from Continuous Delivery to Continuous Deployment
Chapter 6: Architectural Impact of Continuous Delivery and Deployment: Practitioners' Perspectives
Chapter 5: Continuous Delivery and Deployment: Organizational Impact
Chapter 7: On the Role of Software Architecture in DevOps Success
Potential Outcomes
Gain insights into how industry deal with challenging nature of DevOps, continuous delivery and deployment
Improvement to the adoption of DevOps, continuous delivery and deployment
Recommendations for better practices and tools development

Figure 1.2 An overview of the thesis scope and organization

Chapter 1 – This chapter describes the motivations behind this thesis, the research questions, the contributions of this research, and the organization of this thesis. Parts of this chapter have appeared in:

❶ **Mojtaba Shahin**, *Architecting for DevOps and Continuous Deployment*, In Proceedings of 24th Australasian Software Engineering Conference (ASWEC), Doctoral Symposium Track, Vol II, Pages: 147-148 Adelaide, Australia, 2015, ACM.

Chapter 2 – This thesis uses three different research methods to answer the research questions presented in Table 1.1. Furthermore, the results of Chapters 4, 5, and 6 are based on one large-scale mixed-methods empirical study. Therefore, this chapter presents a detailed description of each research method including the reasons behind the chosen research method, the challenges faced in using the research method and strategies adopted to overcome those challenges.

Chapter 3 – This chapter addresses the research question **RQ1** and presents the results of a systematic literature review to classify the approaches, tools, challenges, and practices of continuous integration, delivery, and deployment reported in the literature. This chapter has been previously published as:

❷ **Mojtaba Shahin**, Muhammad Ali Babar and Liming Zhu, *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*, IEEE Access, 5 (99), Pages: 3909-3943, 2017, IEEE. [Impact Factor (2017): **3.55**, SJR rating: **Q1**]

Chapter 4 – This chapter answers the research question **RQ2.1** by conducting a mixed-methods empirical study. First, it presents the current state of automation support in continuous delivery and deployment. Second, it identifies a set of confounding factors that limit or demotivate organizations to have the automatic and continuous deployment. This chapter has appeared in:

❸ **Mojtaba Shahin**, Muhammad Ali Babar, Mansooreh Zahedi and Liming Zhu, *Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges*, In Proceedings of 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Pages: 111-120, Toronto, Canada, 2017, IEEE. [Core rating: **rank A**, acceptance rate: **19%** (21/109)]

Chapter 5 – This chapter empirically investigates the impact of adopting continuous delivery and deployment on team structures (**RQ2.2**), collaboration (**RQ2.3**) and team members' responsibilities (**RQ2.4**). This chapter has been published as:

❹ **Mojtaba Shahin**, Mansooreh Zahedi, Muhammad Ali Babar and Liming Zhu, *Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities*, In Proceedings of 21st International Conference on Evaluation and Assessment in Software Engineering (EASE), Pages 384-393, Karlskrona, Sweden, 2017, ACM. [Core rating: **rank A**, acceptance rate: **37.5%** (27/72)]

Chapter 6 – This chapter introduces a catalogue of findings of architecting for continuous delivery and deployment to address the research question **RQ3.1**, which is expected to make a significant contribution to the growing body of evidential knowledge in this regard. This chapter includes the following papers:

❺ **Mojtaba Shahin**, Muhammad Ali Babar and Liming Zhu, *The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives*, In Proceedings of 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Pages: 44:1-44:10, Ciudad Real, Spain, 2016, IEEE. [Core rating: **rank A**, acceptance rate: **22%** (27/122)]

❻ **Mojtaba Shahin**, Mansooreh Zahedi, Muhammad Ali Babar and Liming Zhu, *An Empirical Study of Architecting for Continuous Delivery and Deployment*, (subject to minor revision): Journal of Empirical Software Engineering (EMSE), 2018, Springer. [Impact factor (2107): **2.93**, SJR rating: **Q2**]

Chapter 7 – This chapter reports on the results of an industrial, in-depth case study. This chapter collects, reports and analyzes a set of significant architectural decisions (e.g., architectural tactics) and their implications made by two teams in a case company to smooth the DevOps transition path. This contribution answers the research question **RQ3.2**. The results presented in this chapter will be submitted to a CORE A-ranked software engineering conference.

Chapter 8 – The last chapter concludes the thesis. It closes the thesis with suggestions for future work.

1.5 Statement of Contribution

I as the author of this thesis was the primarily responsible for the inception, plan, design, data collection, analysis and write-up of all the activities and publications presented in this thesis.

1.6 Writing Style

It is argued that constructing knowledge is a community-based activity [42, 43]. Although I was the main responsible for all activities in this PhD thesis, most of them have been conducted in collaboration with my PhD supervisors. Hence, the first person plural ***we*** is used in this thesis to refer to collaborative efforts [43]. For places that are needed to explicitly distinguish the roles of the researchers involved in this thesis, for example when describing the analyzing process of the mixed-methods study, ***the present author*** is used to refer to the author of the thesis and ***other researchers*** (or ***persons***) are used to refer to others.

1.7 Other Publications

In parallel to the research presented in this thesis, I have collaborated in the following publications as first author or co-author, which are not used in this thesis:

❼ **Mojtaba Shahin** and Muhammad Ali Babar, *Improving the Quality of Architecture Design through Peer-reviews and Recombination*, In Proceedings of 9th European Conference on Software Architecture (ECSA 2015), Pages: 70-86, Dubrovnik/Cavtat, Croatia, 2015, Springer. [Core rating: **rank A**, acceptance rate (full papers): **12%** (12/100)]

❽ Mansooreh Zahedi, **Mojtaba Shahin** and Muhammad Ali Babar, *A Systematic Review of Knowledge Sharing Challenges and Practices in Global Software Development*, International Journal of Information Management, Elsevier, 36 (6), Part A, Pages 995–1019, 2016, Elsevier. [Impact factor (2017): **4.51**, SJR rating: **Q1**]

❾ Faheem Ullah, Adam Johannes Raft, **Mojtaba Shahin**, Mansooreh Zahedi and Muhammad Ali Babar, *Security Support in Continuous Deployment Pipeline*, In Proceedings of 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Pages: 57-68, Porto, Portugal, 2017. [Core rating: **rank B**]

Chapter 2

Research Design

This chapter presents the research approaches pursued in this thesis. It starts with a brief explanation of the systematic literature review and its goal, followed by describing the mixed-methods research approach adopted for Chapters 4, 5 and 6. Finally, we describe the design of an exploratory, holistic, two-case study, which is used for Chapter 7.

2.1 Systematic Literature Review

In order to establish a solid background knowledge on DevOps, continuous integration, delivery and deployment practices, we used Systematic Literature Review (SLR) that is one of the most widely used research methods in Evidence-Based Software Engineering (EBSE) [44]. SLR aims at providing a well-defined process for identifying, evaluating and interpreting all available evidence relevant to a particular research question or topic [45]. An SLR evaluates existing studies on a specific phenomenon fairly and creditably. This research method involves three main phases: defining a review protocol, conducting a review, and reporting a review. We discuss the review protocol and the results of this SLR in Chapter 3.

2.2 Mixed-methods Empirical Approach

It is argued that the research method should be selected based on the nature and objectives of the studied problem [46]. Considering the exploratory nature of the research questions RQ2.1, RQ2.2, RQ2.3, RQ2.4 and RQ3.1 introduced in Chapter 1, we chose a mixed-methods approach with a *sequential exploratory strategy* [46] to answer them. The mixed-methods approach makes use of both qualitative and quantitative research methods for data collection and analysis [46] in order to enable researchers to compensate for the weaknesses of both methods [47]. The mixed-methods approach consisted of interviews and a survey to find the answer to these research questions. Gathering data from different sources (i.e., data triangulation) increases the accuracy and reliability of the results [46, 48].

As shown in Figure 2.1, after creating an interview guide, we conducted 21 interviews with software practitioners to gather a wide range of perspectives and to gain a deep understanding of how adopting CD practices may impact on the organizational and software architecting processes. The interviews had a qualitative focus and mainly dealt with "How" and "What" questions (See Appendix A). At the next step, we ran a survey to reach out to large populations of software industrial practitioners. The survey aimed at quantifying, augmenting, and generalizing the findings obtained from the interviews. Although the survey mostly focused on quantitative aspects, we had some open-ended questions to gain further thoughts and opinions from the respondents. Both the interviews and survey have been conducted under the ethics approval obtained from the Human Research Ethics Committee at the University of Adelaide (See Appendix E). We developed a research protocol using appropriate guidelines [49, 50] and meticulously followed it whilst conducting this study. Table 2.1 summarizes the mixed-methods research method. Chapters 4, 5 and 6 report the findings of this mixed-methods research. The following sections deal with the commonalities (e.g., recruitment process and threats to validity) between Chapters 4, 5 and 6 in terms of research method. However, there might be slight differences, for example, each chapter targets specific interview and survey questions. We will describe these differences in the corresponding chapters. Hence, Chapters 4, 5 and 6 would have a very short research method section and we mostly refer to Chapter 2 when their research methods are introduced. Due to the confidentiality purpose, the anonymity of the interviewees, the survey participants, and their respective organizations have been strictly maintained when presenting the findings in each chapter.

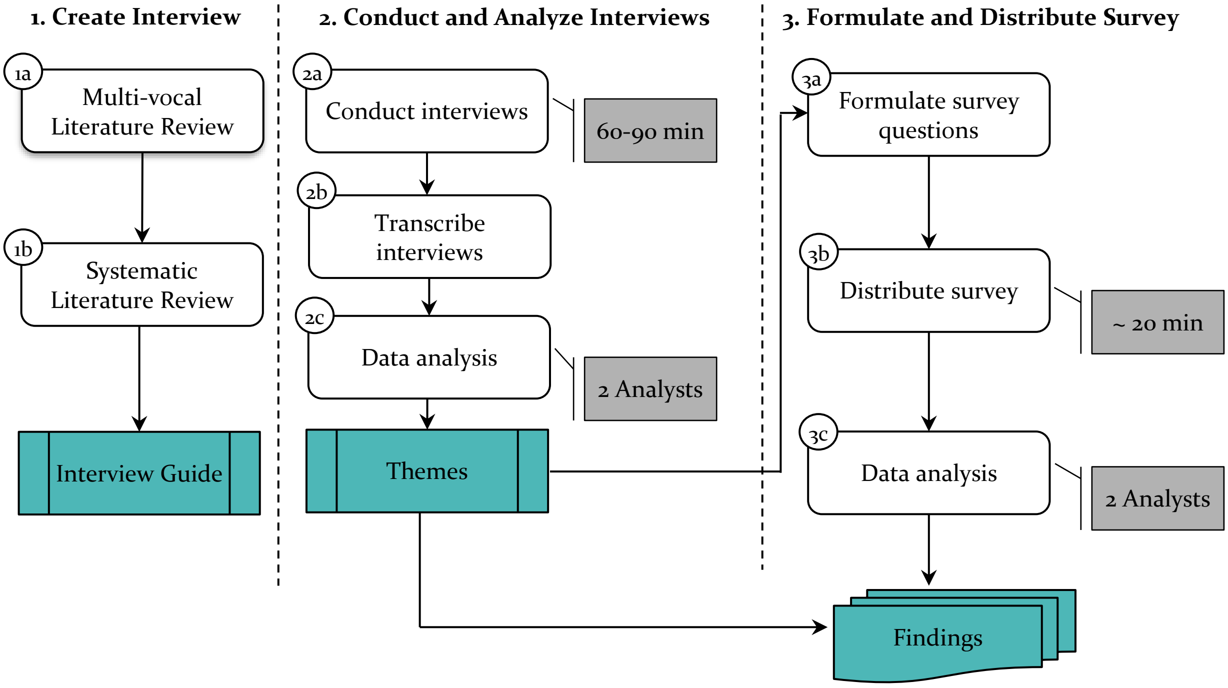


Figure 2.1 Mixed-methods research method steps

Table 2.1 Mixed-methods research summary

	Goal	Participants	Protocol	Analysis
Interviews	Exploring the impact of continuous delivery and deployment on architecting and organizational aspects	21 practitioners (P1-P21) from 19 organizations	Semi-structured; 60-90min; 19 Skype- and 2 email-based interviews	Thematic Analysis-NVivo
Survey	Quantifying and generalizing the interviews' findings	98 practitioners (R1-R98)	Approx. 20 min	Descriptive Statistics + Thematic Analysis-NVivo

2.2.1 Interviews

2.2.1.1 Protocol

The interview guide involved 40 open-ended questions that were designed to support a natural conversation between the interviewer and the interviewee. The interview questions enabled the participants to freely and openly share their experiences and thoughts. We carried out 21 semi-structured, in-depth interviews with participants from 19 organizations. Most of the interviews (i.e., 19) were carried out via Skype. Given the time and geographical restrictions, 2 of the participants opted for sharing their responses via email. After getting practitioners' consent for participation in the interviews, we shared the questions with them before conducting the interviews. This practice enabled the participants to become familiar with the objectives of the interview's questions before participating in the interview [49].

As the study progressed, we modified some of the questions based on the feedback from the interviewees [48]. This included rephrasing three questions, dropping four questions (e.g., the question related to development methodology) to shorten the interview time, and adding two questions. Although the interview questions were only stable after the fifth interview, we believe this did not affect our findings as the findings reported in this thesis were confirmed by multiple interviewees. It is worth mentioning that the interviews were exploratory in nature and generalizability was not the main goal. The interviews were conducted by one person (i.e., the present author), each of which lasted from 60 to 90 minutes. With the interviewee' consent, we audio-recorded the interviews and fully transcribed them for an in-depth analysis. For readability purposes, we fixed the grammatical issues in the transcripts of the interviews. Figure 2.1 shows that the interview guide was created based on the systematic review of literature reported in Chapter 3 [1] and multi-vocal literature review (i.e., non-peer-reviewed sources of information such as blogs [51]).

The interview instrument had four main parts. First part briefed the high-level goals of the research to the interviewees. Second, the interviewees were asked demographic questions (e.g., role and number of years of experience). Third, we asked them to select at least one project from their organization or client that had adopted or was adopting Continuous delivery or/and Deployment (CD) practices. In fact, CD had to be the major focus in the project to be used as a reference point for each of the interviews. Next, the interviewees were asked to share how adopting CD practices have (positively/negatively) influenced their organizational and architecting processes and the principles, strategies, and practices that they employed to succeed in. At the end of each interview, we asked the interviewees to share any other comments and potential issues regarding the questions. The interviews produced a total of almost 26 hours of audio and over 90000 words of transcriptions. The interview guide is included in Appendix A of this thesis.

2.2.1.2 Participants

We aimed to recruit participants using purposive sampling for this study [52]. We approached software practitioners who either had worked for the organizations adopting DevOps/CD practices (e.g., were DevOps engineers) or were involved in DevOps consulting organizations (e.g., were DevOps consultants). The interviewees were identified in multiple ways: (i) we identified the potential participants, for example, through our personal networks, and by exploring the list of speakers and attendees of industry-driven conferences on DevOps, CD, or software architecture. Furthermore, we ran the following search terms on the Google search engine to find highly relevant practitioners in this regard: "*architecting for DevOps*", "*architecting for continuous delivery/deployment*", and "*microservices and continuous delivery/deployment*". (ii) We strictly analyzed their profiles to understand whether they had the right kind of experiences and expertise to participate in our study. The targeted population included people who worked in the organizations adopting DevOps/CD practices, people who provided consulting services, and people who were regular speakers at industry conferences in these areas. (iii) Then, we sent an invitation email to them directly. Apart from rigorously analyzing the potential participants' profiles to understand whether they had the right types of competences for this research, as discussed in Section 2.2.1.1, the interview questions were sent in advance to the potential participants so that they could decide whether or not they were suitable participants.

We motivated the interviewees by giving them a free copy of a book (i.e., "DevOps: A Software Architect's Perspective" [7]) after the study. It should be noted that we interviewed practitioners with different levels of seniority, different project roles, and different types of experiences in order to achieve a broad and rich understanding and characterization of the implications of CD. We also used "snowballing technique" to ask participants to introduce suitable candidates for participation [53]. Out of 21 interviewees, 5 were identified through our personal contacts, 2 using the snowballing technique and 14 by googling and browsing their profiles.

Interviewees characteristics: In total, 21 practitioners (i.e., indicated by **P1** to **P21** in Chapters 4, 5 and 6) participated in the interviews. All the participants were male and came from 19 organizations in 9 countries. Table 2.2 presents a summary of the demographic details of the participants and their projects.

Technical role: As shown in Figure 2.2, 7 out of 21 the interviewees were currently in the role of an architect, followed by consultants (4 out 21, %19). The rest of them were executives (e.g., CTO, 2), team leads (2), program managers (2), a developer (1), DevOps engineer (1), operations engineer (1), and software engineer (1).

Experiences: Regarding the interviewees' experiences in software development, 14 interviewees had more than 10 years, five had 6-10 years and two had 1-5 years of experience.

Organization sizes and domains: The interviewees' organizations were from different domains including consulting and IT services (8), financial (2), telecommunication (2), games (2). Among the 21 interviewees, 9 worked in large organizations (>1000 staff), 7 in medium-sized (100-1000 staff) and 5 in small ones (<100 staff).

Table 2.2 Summary of the interviewees' details in the mixed-methods study

ID	Role	Country	Project Domain	Type	Team Size
P1	Architect	Australia	Project 1: Cloud-based system	Greenfield	10
			Project 2: Data integration system	Maintenance	7-8
P2	Developer	China	Network monitoring system	Greenfield	7
P3	Senior DevOps Consultant	US	Financial system	Greenfield	20
P4	Program Manager	Australia	N/A	Greenfield	25
P5	Director of Engineering	India	Supply chain system	Maintenance	40
P6	Vice President of Development	US	Commercial credit software	N/A	15
P7	Manager/Chief Architect	US	Network access management system	Maintenance	14
P8	DevOps Engineer	Australia	Telecommunication system	Greenfield	10-15
P9	IT Consultant/ IT Architect	Netherlands	Project 1: Transport system	Maintenance	15 -20
			Project 2: Banking system	Greenfield	5
			Project 3: Financial system	Maintenance	6 teams with about 7 members

P10	Continuous Delivery Consultant	Netherlands	Insurance website	Maintenance	Development team (21) DevOps/CD team (4)
P11	Architect	Germany	N/A	Maintenance	10
P12	Technical Lead/Architect	England	Content management system	Greenfield	50 in 4-5 different teams
P13	Architect	US	Financial system	Maintenance	25
P14	Architect/ Independent Consultant	Latvia	Financial system	Greenfield	3 distributed teams each 50 members.
P15	Architect/ Consultant	Latvia	Financial system	Greenfield	15
P16	Operations Engineer	Finland	Website application for selling game	Greenfield	7
P17	Technical Lead	USA	Software game	Maintenance	150
P18	Consultant	UK	Scientific software	Maintenance	100
P19	Solution Architect	USA	N/A	N/A	N/A
P20	Software Engineer	USA	Email software	Maintenance	15
P21	Solution Architect	India	Word processing software	Greenfield	5

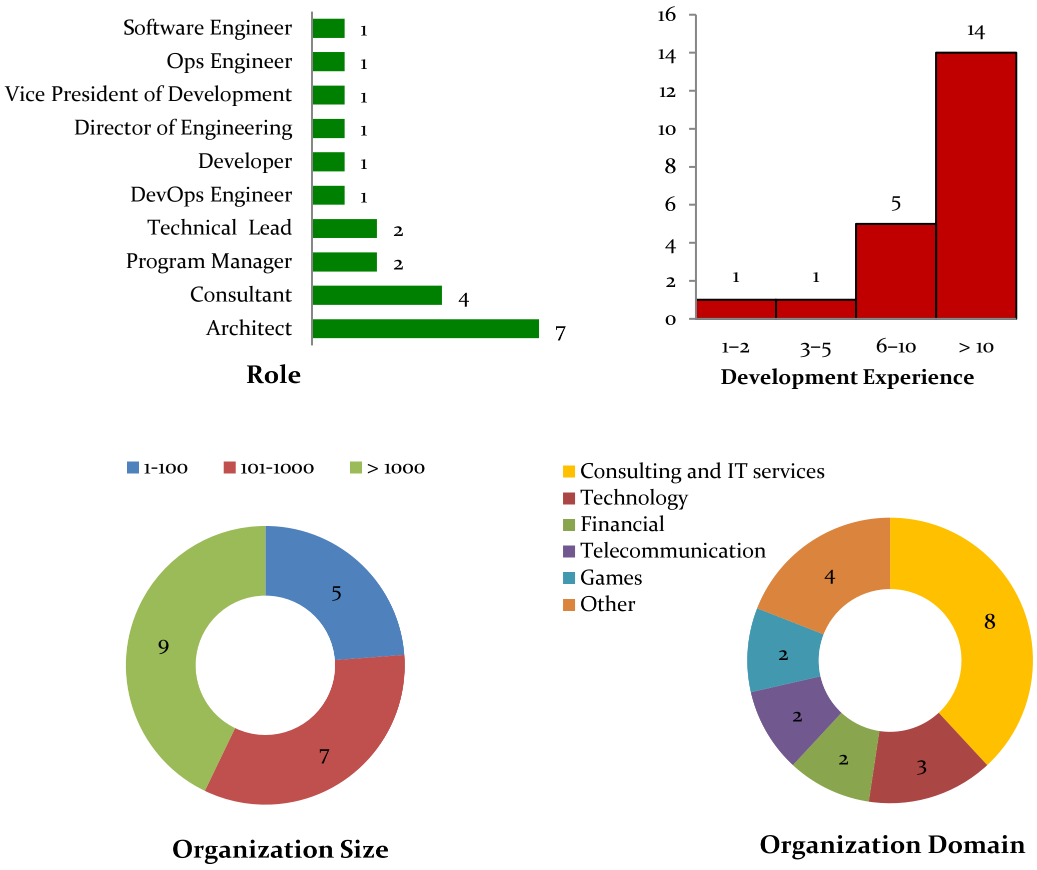
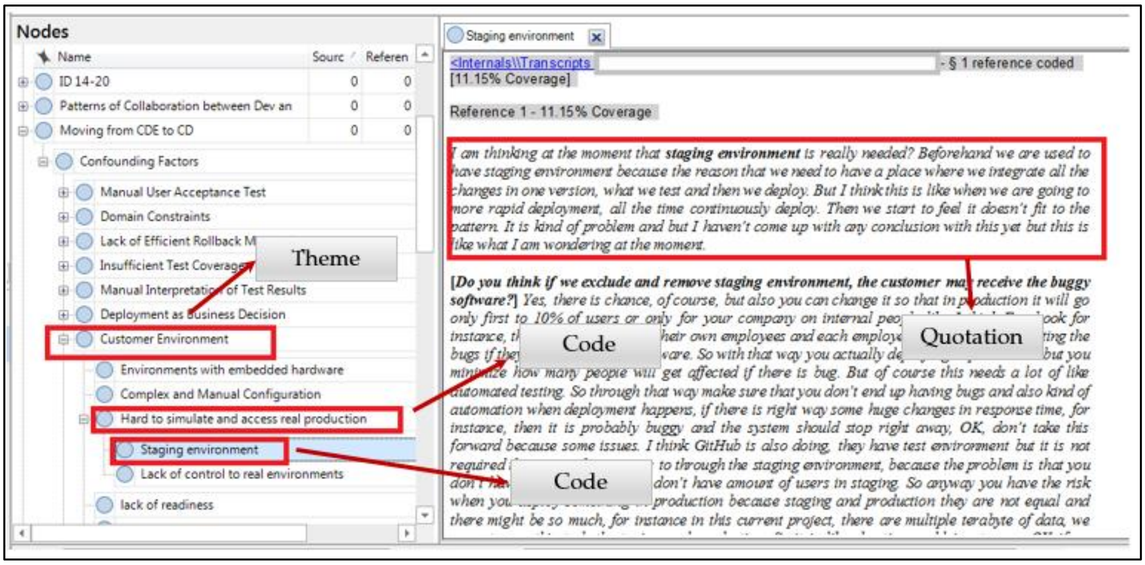


Figure 2.2 Demographics of the interviewees in the mixed method study

2.2.1.3 Analysis

We performed a qualitative analysis of the interviews' data using a conceptualized thematic analysis technique in software engineering [54]. Given the large volume of data, we decided to use a qualitative data analysis tool called NVivo[2]. This allowed a systematic and more convenient analysis and comparison of emerging themes. Our data analysis process started after the third interview, indicating both data collection and analysis proceeded in parallel [55]. While the analysis process was performed by one person (i.e., the present author), all extracted themes were examined by another researcher to confirm the themes and identify any other potential themes. A screenshot of a concrete use of NVivo tool in data analysis process is shown in Figure 2.3.



Nodes

Name	Sourc	Referen
ID 14-20	0	0
Patterns of Collaboration between Dev an	0	0
Moving from CDE to CD	0	0
Confounding Factors		
Manual User Acceptance Test		
Domain Constraints		
Lack of Efficient Rollback M		
Insufficient Test Coverage		
Manual Interpretation of Test Results		
Deployment as Business Decision		
Customer Environment		
Environments with embedded hardware		
Complex and Manual Configuration		
Hard to simulate and access real production		
Staging environment		
Lack of control to real environments		
lack of readiness		

Staging environment

<Internals\\Transcripts - § 1 reference coded [11.15% Coverage]

Reference 1 - 11.15% Coverage

*I am thinking at the moment that **staging environment** is really needed? Beforehand we are used to have staging environment because the reason that we need to have a place where we integrate all the changes in one version, what we test and then we deploy. But I think this is like when we are going to more rapid deployment, all the time continuously deploy. Then we start to feel it doesn't fit to the pattern. It is kind of problem and but I haven't come up with any conclusion with this yet but this is like what I am wondering at the moment.*

[Do you think if we exclude and remove staging environment, the customer may receive the buggy software?] *Yes, there is chance, of course, but also you can change it so that in production it will go only first to 10% of users or only for your company on internal peo ook for instance, th heir own employees and each employe ting the bugs if they ware. So with that way you actually de but you minimize how many people will get affected if there is bug. But of course this needs a lot of like automated testing. So through that way make sure that you don't end up having bugs and also kind of automation when deployment happens, if there is right way some huge changes in response time, for instance, then it is probably buggy and the system should stop right away, OK, don't take this forward because some issues. I think GitHub is also doing, they have test environment but it is not required to through the staging environment, because the problem is that you don't hav don't have amount of users in staging. So anyway you have the risk when you production because staging and production they are not equal and there might be so much, for instance in this current project, there are multiple terabyte of data, we*

Figure 2.3 Qualitative analysis process using NVivo tool in the mixed-methods study

The five steps of the conceptualized thematic analysis method were conducted as follows:

(1) **Extracting data:** data analysis began with reading and examining the transcripts of the interviews line-by-line to extract the key points of each interview and transferred them to the NVivo tool.

(2) **Coding data:** at this step of the analysis, the initial codes were constructed. Our interview mostly targeted "How" questions, which were answered in detail. This enabled us to extract the initial codes for later analysis (See Figures 2.4.A and 2.5.A). Making use of NVivo enabled us to move back and forth between the codes easily and review all the extracted data under a particular code.

(3) **Translating codes into themes:** for each interview transcript, the codes identified in the last step were clustered into potential themes (See Figures 2.4.B and 2.5.B).

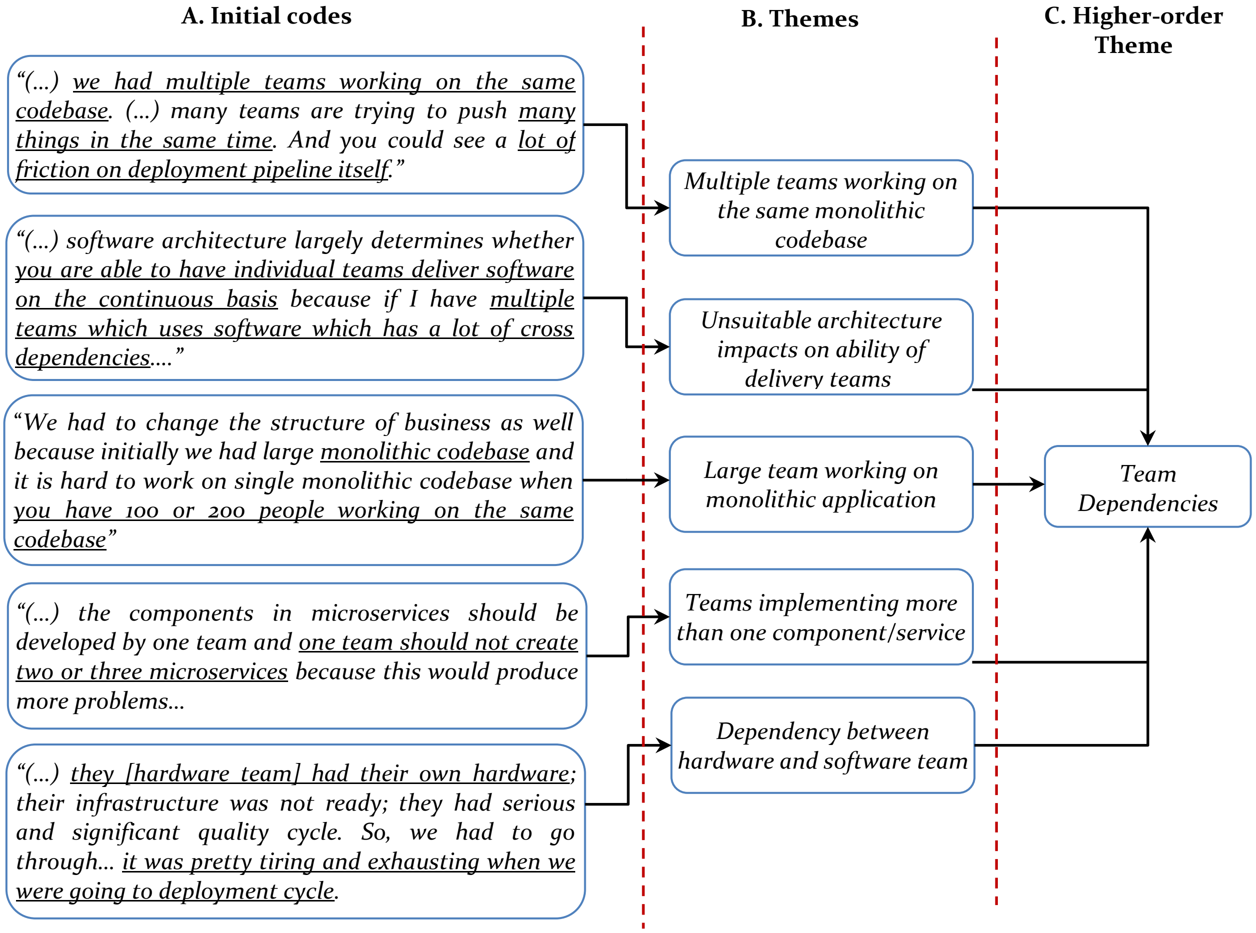
(4) **Creating a model of higher-order themes:** this step involved re-evaluating the extracted themes against each other to merge presumably related themes or exclude the themes with low evidence support [56]. At the end of this step, we generated a higher-order model of themes (See Figures 2.4.C and 2.5.C).

(5) **Assessing the trustworthiness of the synthesis:** through this step, we first assessed the trustworthiness of the interpretations from which core themes emerged [54]. In this step, we established arguments for the extracted themes, for example in terms of credibility, are the

[2] http://www.qsrinternational.com

claimed core themes supported by the evidence of the thematic synthesis? For confirmability purposes, is there any consensus among the researchers on the coded data? Then, each core theme was given a clear and precise name.

Figures 2.4 and 2.5 show the application of the conceptualized thematic method on some of the interview transcripts to identify the "*team dependences*" and "*no visible Ops team*" themes respectively.


A. Initial codes
B. Themes
C. Higher-order Theme
Multiple teams working on the same monolithic codebase
Unsuitable architecture impacts on ability of delivery teams
Large team working on monolithic application
Teams implementing more than one component/service
Dependency between hardware and software team
Team Dependencies

"(...) we had multiple teams working on the same codebase. (...) many teams are trying to push many things in the same time. And you could see a lot of friction on deployment pipeline itself."

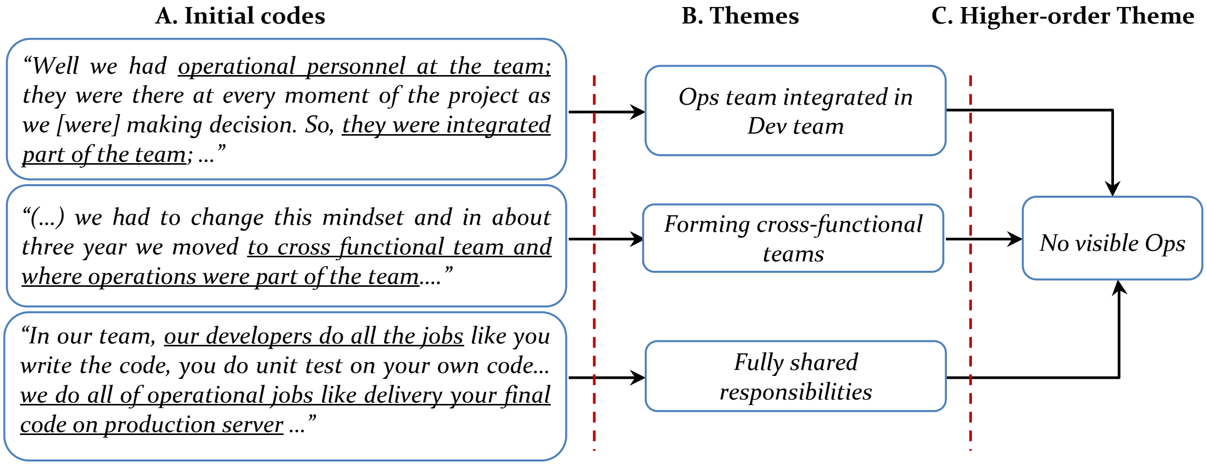
"(...) software architecture largely determines whether you are able to have individual teams deliver software on the continuous basis because if I have multiple teams which uses software which has a lot of cross dependencies...."

"We had to change the structure of business as well because initially we had large monolithic codebase and it is hard to work on single monolithic codebase when you have 100 or 200 people working on the same codebase"

"(...) the components in microservices should be developed by one team and one team should not create two or three microservices because this would produce more problems...

"(...) they [hardware team] had their own hardware; their infrastructure was not ready; they had serious and significant quality cycle. So, we had to go through... it was pretty tiring and exhausting when we were going to deployment cycle.

Figure 2.4 Steps of applying conceptualized thematic analysis leading to "*team dependencies*" theme


A. Initial codes
B. Themes
C. Higher-order Theme
Ops team integrated in Dev team
Forming cross-functional teams
Fully shared responsibilities
No visible Ops

"Well we had operational personnel at the team; they were there at every moment of the project as we [were] making decision. So, they were integrated part of the team; ..."

"(...) we had to change this mindset and in about three year we moved to cross functional team and where operations were part of the team...."

"In our team, our developers do all the jobs like you write the code, you do unit test on your own code... we do all of operational jobs like delivery your final code on production server ..."

Figure 2.5 Steps of applying conceptualized thematic analysis leading "no visible Ops" theme

2.2.2 Survey

2.2.2.1 Protocol

The online survey was designed based on the guidelines suggested by Kitchenham and Pfleeger [50] and hosted on Google Forms. The survey questions were formulated based on the interviews' findings to augment and generalize the findings with a larger sample size. In the survey preamble, we briefly explained the study's goals and the eligibility requirements of the potential participants. We also clearly defined the architecting process, continuous delivery, continuous deployment, and deployability terminologies. This information was necessary to ensure that all the survey participants understood and used those terminologies consistently.

Apart from demographic questions (6 questions), the survey had 46 questions including five-point Likert-scale (31 questions), multiple-choice (3 questions), single-choice (4 questions) and open-ended (8 questions) questions. All questions were mandatory. For multiple- and single-choice questions, an "Other" field was added to collect further perspectives and thoughts form the participants [57]. Likert-scale questions asked the participants to rate five types of statements: (1) how they agreed or disagreed with the statements (i.e., from *strongly agree* to *strongly disagree*); (2) how important (i.e., from *very important* to *unimportant*) the statements or the challenges reported in the statements were; (3) how frequently the statements occurred (i.e., from *almost always* to *never*); (4) how likely they experienced the statements (i.e., from *not at all* to *very much*); and (5) how they scored the statements (i.e., from *1* to *5*). At the end of the survey, we included an optional open-ended question to collect any general comments about the questionnaire. Feedbacks provided by the participants through this question helped us to rephrase five questions (e.g., removing ambiguity in a question's wording) and add three questions (i.e., Q19, Q28, and Q43 in Appendix B) in the middle of running the survey to cover more aspects of CD. It is worth noting that the survey questions were not reworded any further after receiving the tenth response. The survey was in English and took about 20 minutes to complete. The complete list of the survey questions is shown in Appendix B of this thesis.

2.2.2.2 Participants

We employed three recruitment methods for our survey. Initially, we publicly advertised the survey to several groups interested in the topics related to DevOps, CD, and microservices on LinkedIn. Secondly, an invitation letter was sent to 4050 GitHub users via email and invited them to complete the survey. In the email invitation and survey preamble, we asked the participants to forward the survey to any colleague eligible to participate. We incentivized the participation in the survey by offering five copies of a DevOps book (i.e., "*DevOps: A Software Architect's Perspective*") to five randomly selected respondents, who would have wished to be considered for the draw. However, we were not successful in recruiting practitioners using the first two methods as fewer than 10% of all responses came from these recruitment approaches. We believe that it is mainly because our survey needed an advanced level of knowledge and expertise in both software architecture and CD. Murphy-Hill et al. [58] also revealed that posting the survey on social networks may not encourage a large number of practitioners to participate.

Although we only used the email addresses that were publicly available on GitHub to invite the GitHub users, this approach raised minor issues, e.g., a few numbers of them complained about why their email addresses being harvested. Therefore, we approached the highly relevant practitioners by following the process used to recruit the interviewees: finding highly relevant practitioners (e.g., speakers and attendees of industry-driven conferences on CD, DevOps, and SA), thoroughly analyzing their background and expertise, and contacting them directly via email. Overall, we emailed the survey to 487 highly relevant practitioners. In the end, we received 103 responses from all the three recruitment methods. All 103 responses were examined to identify careless responses [59]. We found 5 invalid responses by analyzing outliers, examining inconsistencies in response to two related questions (e.g., Q45 and Q47 in Appendix B), and recognizing the same responses to consecutive questions (e.g., Q29 to Q32 in Appendix B) [59]. It should be noted that we abstained from measuring a response rate for our survey due to having a heterogeneous target population (e.g., practitioners might be members of multiple LinkedIn groups).

Survey participants characteristics: 98 software practitioners (i.e. indicated by **R1** to **R98** in Chapters 4, 5 and 6) completed the survey.

Technical role: As shown in Figure 2.6, the majority of the survey participants were architects (40), followed by DevOps engineers (12), consultants (10), and team leads (8). The rest were developers (7), software engineers (6), executives (e.g., director, 3), operations engineers (3), and others (9).

Experiences: 75.5% of the participants had more than 10 years of experience in the software industry, 14.3% 6-10 years, 7.1% 3-5 years, and 3.1% 1-2 years.

Organization sizes and domains: Similar to the interviewees, the survey participants came from very diverse organizations in terms of the domain including consulting and IT services (36), financial (10), e-commerce (10), and telecommunication (6). 39 practitioners from large, 31 from medium-sized and 28 from small organizations completed the survey.

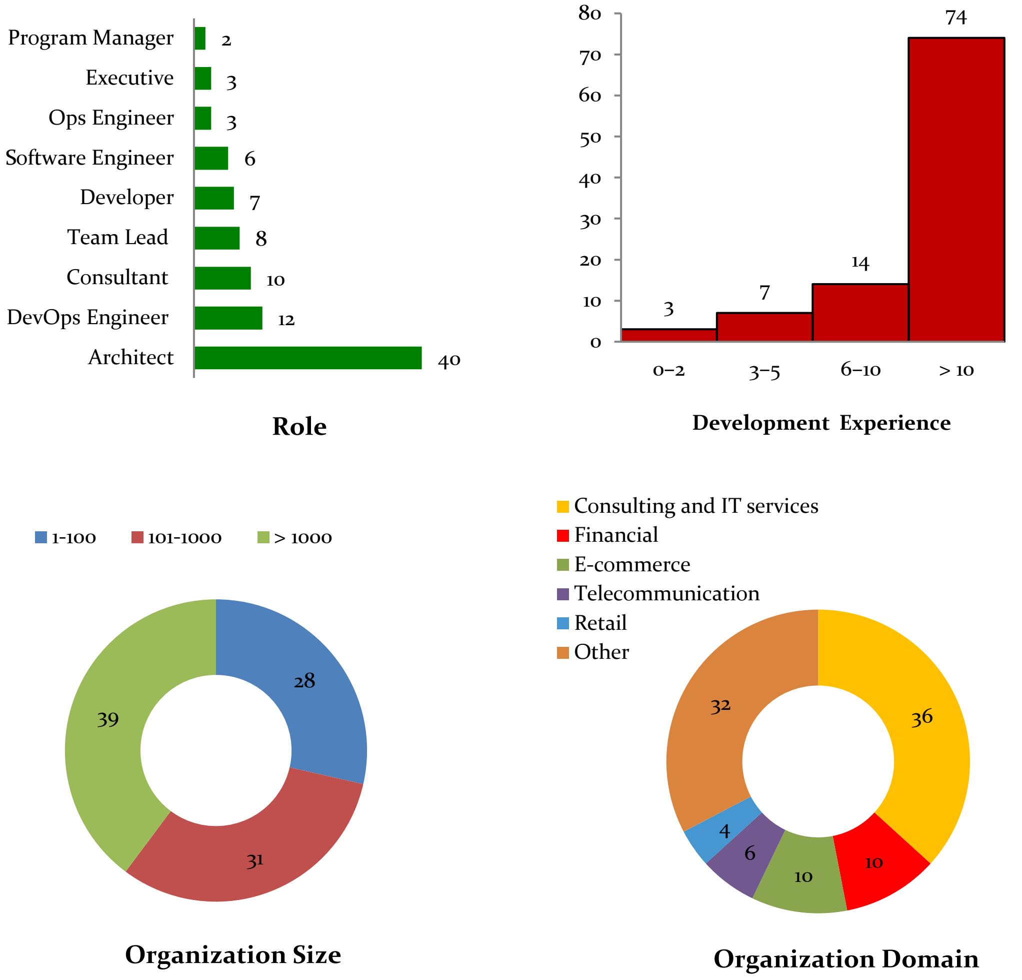


Figure 2.6 Demographics of the survey participants in the mixed-methods study

2.2.2.3 Analysis

We applied descriptive statistics to analyze the data gathered from the closed-ended questions (e.g., Likert-scale questions) [58]. To analyze the open-ended questions, we followed the conceptualized thematic analysis method described in Section 2.2.1.3. Similarly, the present author conducted the analysis process and then other researcher examined all the extracted themes.

2.2.3 Threats to Validity

Whilst we followed strictly the guidelines reported in [50, 60] to conduct this study, similar to other empirical studies, there are some threats that may have affected the findings of this study.

Internal validity: One of the threats that may occur in any empirical study concerns the sampling method. As we described in Sections 2.2.1.2 and 2.2.2.2, we purposively recruited the participants (e.g., analyzing the profiles of potential practitioners). It was possible to select the practitioners who did not have the right kind of experience and expertise in order to take part in the study. To address this issue, we applied strict criteria (e.g., seeking for potential practitioners and rigorously reviewing their public profiles) for selecting participants for both parts of this study. Additionally, we explicitly added the characteristics of the target practitioners in the survey preamble. We also gathered the level of experience in DevOps/CD adoption. We are confident that most of the interviewees and the survey participants had the right experience and expertise to participate in our study.

Our results may have been affected by one specific role bias (e.g., DevOps engineer). We avoided this threat by targeting the participants holding different roles in software development. In the retrospective studies (e.g., interviews), the participants may not have been able to remember all the details during interviews [58]. We adopted two strategies to alleviate the memory bias: first, we sent the interview questions in advance to the interviewees. This helps them to refresh all the relevant details and implicit decisions. We also asked them to share their experiences from their most recent projects or clients. Another threat that may have influenced the participants' answers was social desirability bias [61, 62], in which a participant tries to answer the questions in a manner that s/he perceives a researcher would want. We limited this bias by informing the participants at the beginning of the interviews and survey that personal details are not to be divulged and all the collected data would be anonymized [57].

Researcher bias can be another potential threat to the validity of the findings in a qualitative study. A large part of the data analysis step was conducted by one person (i.e., the present author). In order to minimize this threat, a second person investigated all the extracted themes. In case of any doubt, continuous discussions were organized to maintain the accuracy of the analysis process, which was also guided by the pre-defined research protocol described in Sections 2.2.1 to 2.2.2. This study used the triangulation technique to collect data from two sources to minimize any researcher bias. The findings of the interviews and formulation of the survey questions heavily relied on the interviewees' statements, which might be subjective, and can negatively impact on the findings of this study. To alleviate this threat, we have only reported those findings that were confirmed by multiple participants (e.g., at least two participants). In addition, we provided a precise description of the terminologies used in the interview and survey questions to the participants. We are confident that this strategy helped both the researchers and the participants to have a common understanding of the terminologies used.

Construct validity: Appropriateness and comprehensibility of the questions and answer options used in both the interviews and the survey can be another source of threat in our study [63, 64]. In order to deal with this threat, the interviews' questions were designed based on the systematic review presented in Chapter 3 [1] and multi-vocal review, with seeking feedback and validation from the other researchers and a few industrial practitioners. The feedback collected at the end of the interviews and the survey was valuable as it helped us to fine-tune some questions (e.g., changing questions wording) that were confusing or unclear. During the interviews, we mostly used open-ended questions, and extensively encouraged the interviewees to provide as detailed answers as possible. The survey questions originated from the interviews' findings. Wherever required we included open-ended questions or an "Other" field in questions responses to collect additional information. It should be noted that we did not find too much additional information through the open-ended questions and "Other" fields, suggesting the interviews successfully identified the significant findings. We are confident that our questions covered the important aspects of CD.

External validity: Similarly, to other empirical studies, generalizability is a potential threat to the findings of our study. For the interviews, the participants with a wide range of backgrounds (e.g., different roles) were knowingly selected and invited from very diverse types of organizations in terms of size, domain, and the way of working in several countries. We believe our sampling technique largely improved the reliability of our analysis and the generalizability of the findings [65]. Additionally, we augmented and generalized the findings of the interviews through the survey.

2.3 Industrial Case Study

Given DevOps is a relatively new phenomenon and the exploratory nature of RQ3.2 introduced in Chapter 1, we applied a case study approach to gain a deep understanding of the role of software architecture in DevOps transition in the context of a company (i.e., it is referred to the case company in this thesis) who develops Big Data solutions [66]. A case study is "an empirical method aims at investigating contemporary phenomena in their context" [67]. Our case study was an exploratory, holistic, two-case study as we studied two teams from the same company [66, 68]. Informed by the established guidelines for conducting a case study [66, 67], a research protocol was developed in advance and was strictly followed when performing the case study (See Appendix C)

Table 2.3 Overview of the investigated projects, teams and interviewees in the case study

Team	Project Domain	Project Type	Team Size	Characteristics of Interviewees			
				ID	Role	Years of experience in role	Experience in IT
TeamA	Social Media Platform	Greenfield	8	PA1	Software Engineer	2.5	10
				PA2	Solution Architect	5	15
				PA3	Software Architect	15	20
TeamB	Social Media Platform	Greenfield	Engineering team: 5 Data science team: 4	PB1	Senior Software Engineer	2	6
				PB2	System Architect	1	12
				PB3	Software Engineer	2	2

2.3.1 Context

2.3.1.1 The Case Company

The case company is a research and development organization, which develops and delivers robust Big Data solutions and technologies (e.g., tools). By providing such Big Data capability, the customers and end users of the case company are enabled to make critical decisions faster and more accurately. The case company is made up of several teams working on various research and development programs. Each team includes a variety of roles such as software engineers, software architects, and data scientists. In this study, we studied two teams: **TeamA** and **TeamB**.

2.3.1.2 TeamA

TeamA develops a social media monitoring platform which collects the available online multimedia data (text, image, and video) and tries to make them digestible to security analysts. This can enable the analysts to quickly extract and identify intelligence and unforeseen insights. Facebook and Twitter are the main data sources for this platform. This project is to descriptively summarize the social media by applying image processing and natural language processing approaches. TeamA consists of 8 members including software engineers, developers, and software architects in a cross-functional team. The team started by 4 members for about 18 months, but by growing the project, more people were added. The platform is a greenfield project. TeamA started with microservices architecture style, but they changed the architecture of the platform to a monolith.

2.3.1.3 TeamB

TeamB is another team in the case company who works on a greenfield platform. The platform aims at identifying and tracking the potential social event trends. The platform ingests a large amount of publicly available data from a diverse range of social media websites (e.g., Facebook). The goal is to automatically and accurately predict and track society level events such as protest, celebration and disease outbreak. The predictions will then be used by the data science team. TeamB has two teams: one engineering team and one data science team. The work style is that the data engineering team is the customer for the engineering team and data science team has its own customers (e.g., security analysts). The engineering team is composed of 5 members including system architect and software engineers. The team had recently re-architected the platform and converted it from a monolith to a new architecture (i.e., the team refers to it as microarchitecture), to more rapidly introduce new data sources into the platform.

2.3.2 Data Collection

The data collection process initially started by an **informal meeting** with CTO of the case company and one key informant from each team. That meeting enabled us to get a basic understanding of the case company's structure and domain and helped us to find the projects adopting DevOps to be used as a reference point for further steps of our case study. Furthermore, the team members who were suitable for interviews (e.g., those who had a broad view of the software development process such as software architects and senior software engineers) were identified during the meeting. Finally, that meeting helped us to understand what documents in the case company should be investigated.

Face-to-face, semi-structured interviews were the main tool of data collection. We conducted 6 interviews in total, 3 interviews with each team. From TeamA, one software engineer, one solution architect, and one software architect, with an average of 15 years of experience in the IT industry participated in the interviews (See Table 2.3). We also interviewed two (senior) software engineers and one system architect in TeamB, who had an average of 6.6 years of experience in IT industry (See Table 2.3). Each interview had 30 open-ended questions, but we asked follow-up questions based on the participants' responses. The initial questions in the interviews were demographic questions (e.g., participants' experiences in their current role). Next questions asked about team organization and the characteristics of the projects (e.g., domain, deployment frequency, tools, and technologies used for deployment pipeline). Later, we primarily focused on the challenges facing by each team, and the practices, decisions, and tools used at the architecture level for adopting DevOps. Last part of the interviews investigated architectural decision-making process in the DevOps context. However, following semi-structured interview, the participants were allowed to openly discuss any significant DevOps related experiences and insights they had during their respective project, not limited to the architecture [49]. The complete list of the interview questions is available in Appendix C[3].

It is important to mention that we shared the interview guide with the participants before conducting the interviews. This helped them to be prepared for answering the questions and engaging in discussions [49]. The interviews lasted from 40 minutes to one hour and were conducted at the interviewees' workplaces. All 6 interviews were recorded with the participants' permission and then transcribed, resulting in approximately 40 pages of transcripts.

Besides the interview data, we used the internal documents provided by the case company and publicly available organizational data (e.g., the case company's newsletters). This enabled us to triangulate our findings and increase the validity of our findings [66]. Particularly, we had access to project documents (e.g., project plan) and architecture documents stored on an internal wiki.

It should be noted that when we refer to data from the interviews with TeamA and TeamB, we use **PAX** and **PBX** notations respectively. For instance, PA1 refers to the interviewee 1 in TeamA (See Table 2.3). The excerpts taken from the documentation are marked as **D**.

2.3.3 Data Analysis

We analyzed the interviews data and documentation using the core qualitative data analysis techniques of Grounded Theory (GT) including open coding and constant comparison [69, 70]. We also used NVivo[4] to support qualitative coding and analyzing. Data analysis process began with performing open coding over multiple iterations in parallel with data collection to thoroughly analyze the data. This resulted in capturing key points in our data and assigning a label (i.e., code) to each key point. Figure 2.7 depicts an example of applying open coding on a portion of an interview transcript.

Then, the constant comparison was performed to compare the codes identified in the same interview against each other, and to the codes from other interviews and the excerpts taken from the documentation [71]. We then iteratively grouped these emergent codes to generate higher levels of abstraction, called *concepts* and *categories* in the Grounded Theory [70], which became the architectural decisions presented in the Result section in Chapter 7. As data analysis progressed, we constructed relationships among the categories. Figure 2.8 shows how the category, "application should capture and report status properly" was built from six concepts.

[3] Some of the interview's questions are inspired by/taken from "2017 State of DevOps Report" [35] "2017 State of DevOps Report, Available at: goo.gl/Y6sm13 [Last accessed: 10 November 2017]." 2017..

[4] http://www.qsrinternational.com

Raw data: "*We're trying to make sure everything [to be] more substitutable, which allows to do mocking if we need it. We're trying to keep everything independent as you can just test that set of function; that succeeded in unit tests*".

Key point: "*Independent stuff as can be mocked and can be independently tested*"

Code: Independent units for test

Figure 2.7 Constructing codes from the interview transcripts (case study)

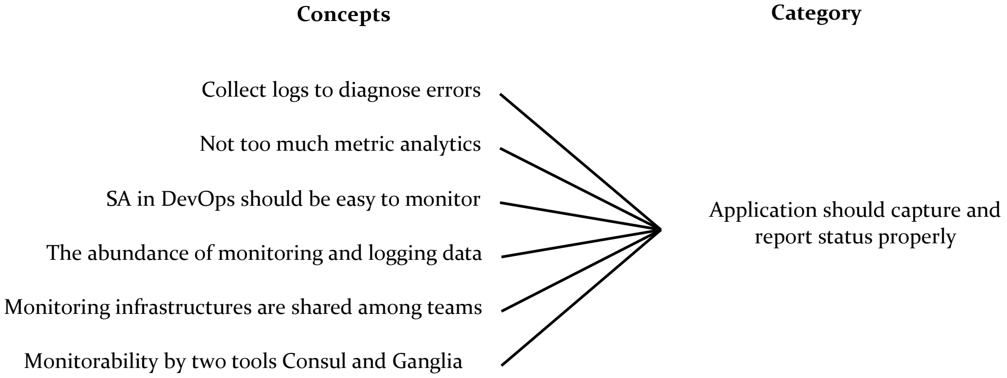

Concepts
Category
Collect logs to diagnose errors
Not too much metric analytics
SA in DevOps should be easy to monitor
The abundance of monitoring and logging data
Monitoring infrastructures are shared among teams
Monitorability by two tools Consul and Ganglia
Application should capture and report status properly

Figure 2.8 Building a category by applying open coding and constant comparison (case study)

2.3.4 Threats to Validity

Our research method and findings in this case study may have few limitations which we discuss below from qualitative research perspective [72].

Transferability: The main limitation of this study is that the findings (e.g., architectural decisions and design challenges) are entirely based on one organization in a particular domain. Whilst we have tried to minimize the potential validity impact of this limitation by studying two independent teams working on two different projects in one organization, our findings may not be (statistically) generalizable to other organizations or domains. However, it should be noted that the focus of this study was to provide a deep understanding of architectural decisions, tactics, and problems within the case company in transition to DevOps. Moreover, this study aimed at discussing important lessons learned through rigorous data collection and analysis [66], so that other organizations and practitioners can benefit from that. Finally, it is worth mentioning that the studies involving a single organization are regarded as valuable contributions in the software engineering community as they contribute to scientific development [73, 74].

Credibility: Using two different data sources (i.e., interview and documentation) and investigating two different teams ensured that the obtained findings to a large extent are plausible. Selection of the participants can be another threat to the credibility of our findings. To recruit motivated participants, we ensured that personal details, opinions, and thoughts would be regarded as strictly confidential and will not be divulged to the researchers and other team members by making a confidential disclosure agreement. After discussing the objectives of this study with the CTO and two key informants at the case company, suitable persons from each team for the interviews were introduced to us. This gives us confidence that the team members who chose to participate were likely more willing and had the right types of competences to provide an unbiased opinion.

Another possible limitation is about the interview questions. Our interview questions were primarily designed based on a comprehensive systematic review (See Chapter 3) [1] and the existing empirical studies on software architecture and DevOps [2, 35, 38, 75]. In addition, we tried to fine-tune the questions and ask appropriate follow-up questions according to the participants' responses and their projects.

Confirmability: Data analysis was conducted by one researcher (i.e., the present author). Whilst this helped to obtain consistency in the results [76], it can be a potential threat to the validity of the findings. This was mitigated to some extent by organizing internal discussions to review and verify the findings and solicit feedback. Furthermore, other research also indicated that data triangulation strategy, which previously described for improving credibility, can be used to establish confirmability as it can reduce the subjectivity of researcher's understanding and judgment [77].

Chapter 3

A Systematic Review on Continuous Integration, Delivery and Deployment

Related publication:

This chapter is based on IEEE Access paper "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices" [1].

Continuous practices, i.e., continuous integration, delivery, and deployment, are the software development industry practices that enable organizations to frequently and reliably release new features and products. With the increasing interest in and literature on continuous practices, it is important to systematically review and synthesize the approaches, tools, challenges, and practices reported for adopting and implementing continuous practices. This chapter aims at systematically reviewing the state of the art of continuous practices to classify approaches and tools, identify challenges and practices in this regard, and identify the gaps for future research. To realize this goal, we conducted a systematic literature review (SLR) method for reviewing 69 peer-reviewed papers on continuous practices published between 2004 and 1st June 2016. Whilst the reported approaches, tools, and practices are addressing a wide range of challenges, this chapter also identifies several open challenges and gaps (e.g., architecting for continuous delivery and deployment) which require further research work.

3.1 Introduction

With increasing competition in the software market, organizations pay significant attention and allocate resources to develop and deliver high-quality software at a much accelerated pace [78]. Continuous Integration (CI), Continuous DElivery (CDE), and Continuous Deployment (CD), called continuous practices for this chapter, are some of the practices aimed at helping organizations to accelerate their development and delivery of software features without compromising quality [28]. Whilst CI advocates integrating work-in-progress multiple times per day, CDE and CD are about the ability to quickly and reliably release values to customers by bringing automation support as much as possible [10, 79].

As described in Chapter 1, continuous practices are expected to provide several benefits such as: (1) getting more and quick feedback from the software development process and customers; (2) having frequent and reliable releases, which lead to improved customer satisfaction and product quality; (3) through CD, the connection between development and operations teams is strengthened and manual tasks can be eliminated [11, 14]. A growing number of industrial cases indicate that the continuous practices are making inroad in software development industrial practices across various domains and sizes of organizations [11, 80, 81].

Due to the growing importance of continuous practices, an increasing amount of literature describing approaches, tools, practices, and challenges has been published through diverse venues. An evidence for this trend is the existence of five secondary studies on CI, rapid release, CDE and CD [33, 34, 82-84]. These practices are highly correlated and intertwined, which distinguishing these practices are sometimes hard and their meanings highly depend on how a given organization interprets and employs them [16]. Whilst CI is considered the first step towards adopting CDE practice [9], truly implementing CDE practice is necessary to support deploying software changes automatically and continuously to production or customer environments (i.e., CD practice). We noticed that there was no dedicated effort to systematically analyze and rigorously synthesize the literature on continuous practices in an integrated manner. By integrated manner we mean simultaneously investigating approaches, tools, challenges, and practices of CI, CDE, and CD, which aims to explore and understand the relationship between them and what steps should be followed to successfully and smoothly move from one practice to another. This chapter aimed at filling that gap by conducting a Systematic Literature Review (SLR) of the approaches, tools, challenges, and practices for adopting and implementing continuous practices.

This chapter provides an in-depth understanding of the challenges of adopting continuous practices and the strategies (e.g., tools) used to address the challenges. Such an understanding is expected to help identify the areas where methodological and tool support to be improved. This increases the efficacy of continuous practices for different types of organizations and software-intensive applications. Moreover, the findings are expected to be used as guidelines for practitioners to become more aware of the approaches, tools, challenges and implement appropriate practices that suit their industrial arrangements. For this review, we have systematically identified and rigorously reviewed 69 relevant papers and synthesized the data extracted from those papers in order to answer a set of research questions that motivated this review. The significant contributions of this work are:

1. A classification of the reported approaches, associated tools, and challenges and practices of continuous practices in an easily accessible format.
2. A list of critical factors that should be carefully considered when implementing continuous practices in both software development and customer organizations.
3. An evidence-based guide to select appropriate approaches, tools, and practices based on the required suitability for different contexts.
4. A list of researchable issues to direct the future research efforts for advancing the state-of-the-art of continuous practices.

Chapter organization: In Section 3.2, we outline the existing research gap. Section 3.3 describes the systematic literature review process with the review protocol. The quantitative and qualitative results of the research questions are described in Section 3.4. Section 3.5 reports a discussion of findings. The threats to validity are discussed in Section 3.6. Finally, we present our conclusions in Section 3.7.

3.2 Research Gap

3.2.1 Existing literature reviews

During this review, we also found five papers that have reported reviews on different aspects of continuous software engineering - two studies have investigated continuous integration in the literature [82, 84], two papers have explored continuous delivery [33] and deployment [34], and one study has targeted rapid release [83] (See Table 3.1). We summarize the key aspects of these studies.

Table 3.1 Comparison of this SLR with existing secondary studies

Study	Focus	# included papers	Search date
Stahl and Bosch [82]	CI	46	October 2012
Eck et al. [84]	CI	43	N/A
Mäntylä et al. [83]	Rapid release	24	N/A
Rodríguez et al. [34]	CD	50	27 June 2014
Laukkanen et al. [33]	CI and CDE	30	February 2015
This work	CI, CDE, and CD	69	1 June 2016

Stahl and Bosch [82] have presented an SLR on different attributes or characteristics of CI practice. That review has explored the disparity in implementations of CI practice in the literature. Based on 46 primary studies, the study had extracted 22 clusters of descriptive statements for implementing CI. The clusters have been classified into two groups: (a) *culled clusters* (e.g., fault frequency) which either came from one unique source or the literature interpreted and implemented them the same; and (b) *preserved clusters* (e.g., build duration) were described as statements that there is contention on them in the published literature. The paper proposed a descriptive model (i.e., the main contribution of the paper) to address the variation points in the preserved clusters.

Eck et al. [84] conducted a concept-centric literature review to study the organizational implications of continuous integration assimilation in 43 primary studies. The review revealed that organizations require implementing numerous changes when adopting CI. The study proposed a conceptual framework of 14 organizational implications (e.g., providing CI at project start) of continuous integration. The authors also conducted a case study of five software companies to understand the organizational implications of CI. Mäntylä et al. [83] performed a semi-systematic literature review to study benefits, enablers, and problems of rapid release (including CI and CD) in 24 primary studies. The review did not comply with several of the mandatory aspects of an SLR's guidelines reported in [45] (e.g., lack of doing data extraction and analysis rigorously, including papers that were not found through applying search string). The review revealed that rapid releases are prevalent industrial practices that are utilized in several domains and software development paradigms (e.g., open source). It has been concluded that the evidence of the claimed advantages and disadvantages of rapid release is scarce. Rodríguez et al. [34] reported a systematic mapping study on continuous deployment to identify benefits and challenges related to CD and to understand the factors that define CD practice. Based on 50 primary studies, it has been revealed that moving towards CD necessitates significant changes in a given organization, for example, team mindsets, organization's way of working, and quality assurance activities are subject to change. The authors also found that not all customers are happy to receive new functionality on a continuous basis and applying CD in the context of embedded systems is a challenge. However, the main contribution of this mapping study lies in the identified 10 factors that define CD practice. For example, (a) fast and frequent release; (b) continuous testing and quality assurance; (c) CI; (d) deployment, delivery, and release processes and configuration of deployment environments.

We found that the work done by Laukkanen et al. [33] is the closest work to our study in this chapter. They conducted a systematic review of 30 primary studies to identify the problems that hinder adopting CDE practice. The authors also reported the root causes of and solutions to the problems. The study grouped the problems and solutions into seven categories: build design, system design, integration, testing, release, human and organizational, and resource. The review [33] only focused on CDE practice rather than CD, in which the authors investigated CDE as a development practice where software is kept production-ready (i.e., CDE practice), but not necessarily deployed continuously and automatically (i.e., CD practice). Laukkanen et al. [33] also revealed that the work by Rodríguez et al. [34] used the term CD, while it actually referred to CDE practice. Furthermore, the SLR done by Laukkanen et al. [33] indicated that whilst it is interesting to study CD, but it failed to find highly relevant literature on CD.

It is worth noting that it is common in software engineering to conduct several SLRs on a particular concept or phenomenon. To exemplify, there are four reviews (i.e., SLR or systematic mapping study) on technical debt [85]. What differentiates SLRs on a particular subject from each other is having different high-level objectives, research questions, included studies and results. Having done a thorough analysis of the related reviews, we observed the following differences between our SLR and the existing reviews:

Search string, inclusion and exclusion criteria: Our search string, inclusion, and exclusion criteria were significantly different with [33, 34, 82-84] for selecting the primary studies. Our work was aimed at reviewing papers that included empirical studies (e.g., case studies and experiments); we excluded the papers with less than 6 pages, which were included in [33, 82, 84]. It is important to note that the previous reviews except [33] used only automatic search, but we used both automated searches and snowballing for finding the relevant papers. Due to the aforementioned reasons, there is a significant difference in the papers reviewed by our SLR with the included papers in other SLRs. Out of 69 papers in our SLR, there were only 10, 2, 7, and 12 common papers with [33, 34, 82, 84] respectively.

Research questions and results: regarding **RQ3.1** and **RQ3.2** and their respective goals (i.e., presented in Table 3.2), there are no similar questions in other reviews. Both goals and results of **RQ3.4** are different to *RQ1* in [82, 84]. Whilst the objective of our research question (**RQ3.4**) was to comprehensively identify and analyze practices, guidelines, lessons learned and authors' shared experiences for successfully adopting and implementing each continuous practice, the given statements for implementing CI in [82] were not sufficiently abstracted and generalized and were not reported as practices for adopting and implementing CI. In fact, the main goal was to indicate there is a lack of consensus on implementing CI in practice. The focus of the review reported in [84] is on the organizational aspects of assimilating CI practice rather than individual software projects. Furthermore, for both reviews [82, 84], the main contributions are model, conceptual framework, and empirical study rather than systematically summarizing, analyzing, and classifying the literature on CI. It is worth noting that due to having different coding schemes, level of details and emergence of categories, it was not easy to make a one-to-one comparison of the identified challenges and practices between our SLR and [33]. However, our study identified a more comprehensive list of challenges, practices, guidelines, lessons learned and the authors' shared experiences. Our findings show that we only have 5 common practices with [33].

Regarding **RQ3.3**, there is a partial overlap between our SLR and the *RQ1* and *RQ4* in [33, 34] respectively. Whilst the goal of the questions has some overlaps with together, closely looking at the result from each study, it clearly indicates a complementary relationship between them. Some of the major differences in the identified challenges are *lack of awareness and transparency*, *general resistance to change*, *distributed organization*, *team dependencies*, *customer environment, dependencies with hardware and other (legacy) applications*, which were not reported in the previous reviews [33, 34].

Analyzing CI, CDE, and CD practices in an integrated manner: As discussed earlier, CI, CDE, and CD practices are highly correlated and intertwined concepts, in which there is no consensus on the definitions of these practices [17]. In our understanding to obtain a clear understanding of the approaches, tools, challenges, and practices, it is essential to broadly study and cover CI, CDE and CD practices across its different dimensions, such as approaches, tools, contextual factors, practices, and challenges simultaneously in an integrated manner.

3.2.2 Motivation for this SLR on continuous practices

According to [10], continuous software engineering includes a number of continuous activities such as continuous integration, delivery, and continuous deployment. It is asserted that CI is a foundation for CDE, in which implementing reliable and stable CI practice and environment should be the first and highest priority for a given organization to successfully adopt CDE practice. We have mentioned that CDE and CD practices are frequently confused together and used interchangeably in the literature and practitioners' blogs. It is sometimes hard to distinguish these correlated and intertwined practices. The meanings of these practices highly depend on who uses them [16, 17]. Since the main objective of this study is to systematically collect, analyze and classify approaches, tools, challenges, and practices of continuous practices, we believe these practices, particularly CDE and CD practices, should be investigated together. Analysing CI, CDE, and CD practices in an integrated manner provides an opportunity to understand what challenges prevent adopting each continuous practice, how they are related to each other, and what approaches, associated tools, and practices exist for supporting and facilitating each continuous practice. Furthermore, this helps software organizations to adopt continuous practices step by step and smoothly move from one practice to another. We could not find any systematic review, which has studied these intertwined practices (i.e., integration, delivery, and deployment) together. The abovementioned reasons indicate the need for conducting a literature review tailored to the scope of the continuous integration, delivery, and deployment in an integrated manner.

3.3 Research Method

As we described in Chapter 2, we used Systematic Literature Review (SLR) to obtain a solid background knowledge on continuous integration, delivery and deployment practices [44]. Following the SLR guidelines reported in [45], our review protocol consisted of (i) research questions, (ii) search strategy, (iii) inclusion and exclusion criteria, (iv) study selection, and (v) data extraction and synthesis. We discuss these steps in the following subsections:

3.3.1 Research questions

Our study in this chapter aimed at summarizing the current research on "***continuous integration, continuous delivery and continuous deployment practices in software development***". We formulated a set of research questions (RQs) to be answered through this chapter. Table 3.2 summarizes the research questions as well as the motivations for them. The answers to these research questions can be directly linked to the objective of this SLR: an understanding of the available approaches and tools in the literature to support and facilitate CI, CDE, and CD practices (RQ3.1, RQ3.2), challenges (RQ3.3) and practices (RQ3.4) reported by empirical studies during adopting each continuous practice. The results of these research questions would enable researchers to identify the missing gaps in this area and practitioners to consider the evidence-based information about continuous practices before deciding their use in their respective contexts.

It is worth noting that we distinguish between approaches and practices in this SLR. Cambridge's and Longman's dictionaries define the *approach*, *method*, and *technique* similarly as the following "*a [special/planned/particular] way of doing something*"; however, *practice* is defined as "*the act of doing something regularly or repeatedly*" [86, 87]. In this SLR, we define *approach*, *method*, and *technique* as a technical and formalized approach to facilitate and support continuous practices [88]. For simplicity purpose, the approaches, methods, techniques, algorithms, and frameworks, along with the tools to support them, that are developed and reported in the literature for this purpose, are classified as an *approach* rather than *practice*. On the other hand, software *practice* is a social practice [89] and is defined as shared norms and regulated rules and activities, which can be supported and improved by an approach [88, 90].

Table 3.2 Research questions of this SLR

Research Question	**Motivation**
RQ3.1 What approaches and associated tools are available to support and facilitate continuous integration, delivery, and deployment?	To gain a comprehensive understanding of approaches (e.g., methods, algorithms, frameworks, techniques) and associated tools to facilitate implementation of continuous practices and to develop a classification of the approaches and tools.
RQ3.2 Which tools have been employed to design and implement deployment pipelines (i.e., modern release pipeline)?	The deployment pipeline is significantly important to move towards continuous practices (in particular CD/CDE). The idea of this question is to understand how researchers form deployment pipelines and which tools are employed to implement the deployment pipelines. It should be noted that the tools identified in **RQ3.1** can also be covered by this question provided that they are integrated and implemented in the deployment pipeline.
RQ3.3 What challenges have been reported for adopting continuous practices?	There might be obstacles and conflicts when adopting and implementing continuous practices in software provider and customer organizations. So, the idea of this question is to get an overview of different types of technical and organizational challenges, problems, and constraints that the organizations might experience in the transition to continuous practices.
RQ3.4 What practices have been reported to successfully implement continuous practices?	To identify good practices, guidelines, lessons learned and shared experiences when adopting and implementing CI, CDE, and CD.

3.3.2 Search strategy

In order to retrieve as many relevant studies as possible, we defined a search strategy [45, 91]. The search strategy used for this review is designed to consist of the following elements:

3.3.2.1 Search method

We used automatic search method to retrieve studies in six digital libraries (i.e., IEEE Xplore, ACM Digital Library, SpringerLink, Wiley Online Library, ScienceDirect, and Scopus) using the search terms introduced in Section 3.3.2.2. We complemented the automatic search with snowballing technique [92].

3.3.2.2 Search terms

We formulated our search terms based on guidelines provided in [45]. The resulting search terms were composed of the synonyms and related terms about “continuous” AND “software”. After running a series of pilot searches and verifying the inclusion of the papers that we were aware of, we utilized the final search string as presented in the following. It should be noted that the search terms were used to match paper titles, keywords, and abstracts in the digital libraries (except SpringerLink) during the automatic search. The reason we included the “software” and its related terms in the search string was that continuous delivery and continuous deployment terminologies are also used in other disciplines (e.g., medicine). Therefore, we were able to avoid retrieving a large number of irrelevant papers.

TITLE-ABS-KEY *((“continuous integration”* ***OR*** *“rapid integration”* ***OR*** *“fast integration”* ***OR*** *“quick integration”* ***OR*** *“frequent integration”* ***OR*** *“continuous delivery”* ***OR*** *“rapid delivery”* ***OR*** *“fast delivery”* ***OR*** *“quick delivery”* ***OR*** *“frequent delivery”* ***OR*** *“continuous deployment”* ***OR*** *“rapid deployment”* ***OR*** *“fast deployment”* ***OR*** *“quick deployment”* ***OR*** *“frequent deployment”* ***OR*** *“continuous release”* ***OR*** *“rapid release”* ***OR*** *“fast release”* ***OR*** *“quick release”* ***OR*** *“frequent release”* ***OR*** *“deployability”* ***OR*** *“continuous build”* ***OR*** *“rapid build”* ***OR*** *“fast build”* ***OR*** *“frequent build”* ***OR*** *“quick build”)* ***AND*** *(“software”* ***OR*** *“information system”* ***OR*** *“information technology”* ***OR*** *“cloud*”* ***OR*** *“service engineering”))*

3.3.2.3 Data sources

We queried six digital libraries, namely IEEE Xplore, ACM Digital Library, SpringerLink, Wiley Online Library, ScienceDirect, and Scopus for retrieving the relevant papers. According to [93], these are the primary sources of literature for potentially relevant studies on software and software engineering. For all these libraries, except SpringerLink, we ran our search terms based on title, keywords and abstract. It is important to note that currently, SpringerLink search engine does not provide any facility for searching on the title, abstract and keywords [94]. We were forced to either restrict our search on the title only or apply search terms on the full text of the articles. While the former resulted in a quite few numbers of papers, the latter strategy returned more than 11700 papers. In order to address this situation, we followed the strategy adopted in [94]; we examined only the first 1000 papers retrieved by the search on the full text. However, we believe that Scopus was a complement to SpringerLink as Scopus indexes a large number of journals and conferences in software engineering and computer science [95, 96]. It is worth noting that Google Scholar was not selected as the data source because of having the low precision of search results and generating many irrelevant results [93].

3.3.3 Inclusion and exclusion criteria

Table 3.3 presents the inclusion and exclusion criteria, which were applied to all studies retrieved from digital libraries. We did not choose a specific time as the starting point of the search period. Only peer-reviewed papers were included, and we excluded editorials, position papers, keynotes, reviews, tutorial summaries, panel discussions and non-English studies. Papers with less than 6 pages were excluded. We selected only those papers that have reported the approaches, tools, and practices using empirical research methods such as case study, experience report, and experiment. In cases where we found two papers addressing the same topic and have been published in different venues (e.g., in a conference and a journal), the less mature one was excluded. We eliminated duplicate studies retrieved from different digital libraries.

Table 3.3 Inclusion and exclusion criteria of this SLR

Inclusion Criteria	
I1	A study that is peer-reviewed and available in full-text.
I2	A study that presents approaches (e.g., methods, techniques, frameworks, and algorithms) and associated tools to facilitate continuous practices or reports practices and challenges in adopting continuous practices.
I3	Empirical study: a study that evaluates, validates, or investigates the proposed approaches, tools and practices through empirical research methods such as case studies, survey, and experiments.

Exclusion Criteria

E1 Non-peer-reviewed papers such as editorials, position papers, keynotes, reviews, tutorial summaries, and panel discussions.

E2 Short papers (i.e., less than 6 pages).

E3 A study that is not written in English.

E4 Non-empirical studies (e.g., tool demo)

3.3.4 Study selection

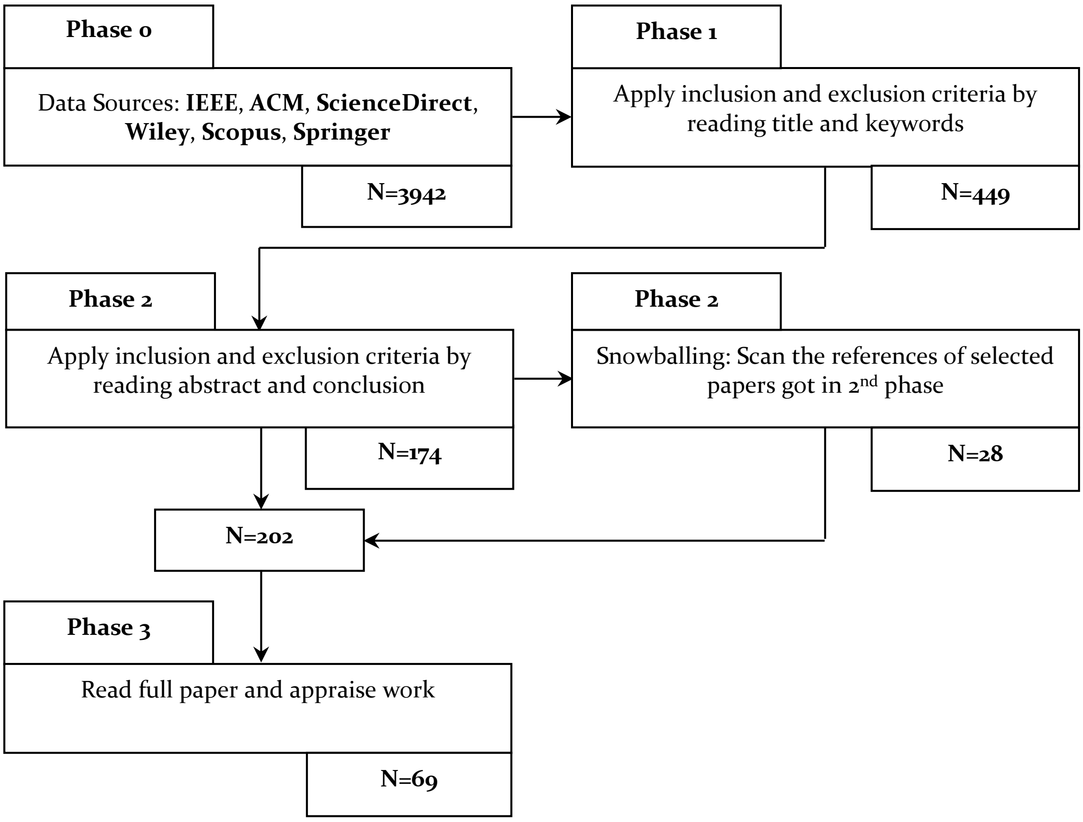

Phase 0
Data Sources: IEEE, ACM, ScienceDirect, Wiley, Scopus, Springer
N=3942
Phase 1
Apply inclusion and exclusion criteria by reading title and keywords
N=449
Phase 2
Apply inclusion and exclusion criteria by reading abstract and conclusion
N=174
Phase 2
Snowballing: Scan the references of selected papers got in 2nd phase
N=28
N=202
Phase 3
Read full paper and appraise work
N=69

Figure 3.1 Phases of the search process

Figure 3.1 shows the number of studies retrieved at each stage of this SLR. The inclusion and exclusion criteria were used to filter the papers in the following way:

Phase 0: We ran the search string on the six digital libraries and retrieved 14723 papers. Considering only first 1000 results from SpringerLink, we finally found 3942 potential papers.

Phase 1: We filtered the papers by reading title and keywords. When there were any doubts about the retrieved papers and it was not possible to determine the papers by reading the titles and keywords, these papers were transferred to the next round of selection for further investigation. At the end of this phase, 449 papers had been selected.

Phase 2: We looked at the abstracts and conclusions of the retrieved articles to ensure that all of them were related to the objective of our SLR. We applied the snowballing technique [92] to scan the references of the selected papers in the second phase. We found 51 potentially relevant papers by title from the references of these 174 papers.

Inclusion and exclusion criteria were applied to the abstracts and conclusions of those 51 potentially relevant papers and we finally selected 28 papers for the next phase. It is important to mention that the main reason for conducting snowballing in this phase rather than applying it in the third phase, was to find as many relevant studies as possible.

Phase 3: In the last (third) selection round, we read the full text of the selected studies from the second phase and if a paper met all the inclusion criteria, this paper was selected for inclusion in this SLR. We excluded the papers that were shorter than 6 pages, irrelevant, or whose full texts were not available. Furthermore, we critically examined the quality of primary studies to exclude those had low quality e.g., low reputation venues. We found four types of papers on continuous practices:

- Papers that present approaches (e.g., methods, techniques, frameworks, and algorithms) and associated tools to facilitate each continuous practice (RQ3.1).
- The second group consists of experience report papers which either present the challenges, problems, and confounding factors in adopting and implementing continuous practices (RQ3.3) or discusses practices, guidelines and lessons learned for this purpose (RQ3.4).
- A group of papers reporting surveys of the usage and importance of agile practices (e.g., continuous integration and delivery) in software development organizations.
- The papers in the fourth group used the concepts of continuous integration, delivery, and deployment on developing and deploying an application, for example, applying CI practice on robotic systems, and most reported the potential benefits obtained by these concepts.

Since most papers in third and fourth groups did not meet any of the research questions and were out of the objectives of this review, we excluded a large number of the papers in those groups. Finally, we selected 69 papers for this review. In each phase, we recorded the reasons of inclusion or exclusion decision for each of the papers, which were used for further discussions and reassessment whether a paper had to be included or not. A cross-check using a random number of the selected papers for each step was performed by another researcher.

3.3.5 Data extraction and synthesis

3.3.5.1 Data extraction

We extracted the relevant information from the selected papers based on the data items presented in Table 3.4 in order to answer the research questions of this SLR. It shows the research question(s) (described in Section 3.3.1) that were supposed to be answered using different pieces of the extracted data. The extracted information was stored in MS Excel Spreadsheet for further analysis.

3.3.5.2 Synthesis

We divided the data extraction form into a) demographic and contextual attributes, b) approaches, tools, challenges, practices and critical factors of continuous practices. We used descriptive statistics to analyze the data items D1 to D10. In order to identify the research types (i.e., data item D7) reported in the selected papers, we classified them into six well-known research types: validation research, evaluation research, solution proposal, philosophical paper, opinion paper, and experience report [97]. The second set of data items (i.e., D11, D12, D13, and D14) were analyzed using qualitative analysis method, namely thematic analysis [56]. We followed the five steps of the thematic analysis method [56] as detailed below:

(1) **Familiarizing with data**: we tried to read and examine the extracted data items, e.g., D11 (approaches and tools), D12 (challenges), D13 (practices) and D14 (critical factors) to form the initial ideas for analysis.

(2) **Generating initial codes**: in the second step, we extracted the initial lists of challenges, practices, and factors for each continuous practice. It should be noted that in some cases, we had to recheck the papers.

(3) **Searching for themes**: for each data item, we tried to combine different initial codes generated from the second step into potential themes.

(4) **Reviewing and refining themes**: the challenges, practices and critical factors identified from the third step were checked against each other to understand what themes had to be merged with others or dropped (e.g., lack of enough evidence).

(5) **Defining and naming themes**: through this step, we defined clear and concise names for each challenge, practice, and critical factor.

Table 3.4 Data items extracted from each study and related research questions

#	Data item	Description	RQs (Section 3.3.1)
D1	Author(s)	The author(s) of the paper.	
D2	Year	The year of the publication of the paper	Demographic data
D3	Title	The title of the paper	
D4	Publication type	The type of publication (e.g., journal paper)	Demographic data
D5	Venue	The name of the publication venue	Demographic data
D6	Data analysis type	Qualitative, quantitative or mixed.	Demographic data
D7	Research type	The type of research i.e., validation research, evaluation research, solution proposal, philosophical paper, opinion paper, and experience report.	Demographic data
D8	Study context	The study contexts are categorized in industry and non-industry (e.g. student) cases.	Demographic data
D9	Project type	It recodes the type of project e.g., greenfield or maintenance.	Demographic data
D10	Application domain	The type of application used for reporting challenges as well as for validating proposed techniques, tools, and practices.	Demographic data
D11	Techniques and tools	The techniques and tools that facilitate the continuous integration, delivery and deployment (i.e., continuous practices).	RQ3.1, RQ3.2
D12	Challenges	It documents the challenges and barriers that have been reported to adopt continuous practices in software development and customer's organizations.	RQ3.3
D13	Practices	It records lessons learned, authors' experiences and good practices to successfully implement continuous practices.	RQ3.4
D14	Critical factors	Factors to be considered when introducing and adopting continuous practices.	Discussion

3.4 Results

Following subsections report the results from analyzing and synthesizing the data extracted from the reviewed papers to answer the research questions. The results are based on synthesizing the data directly collected from the reviewed papers with our minimal interpretations. We interpret and reflect upon the results in the discussion section.

3.4.1 Demographic attributes

This subsection reports the demographic and research design attributes information: studies distribution, research types, study context and data analysis type, and application domains and project types. All of the included papers are listed in Appendix D.

3.4.1.1 Studies distribution

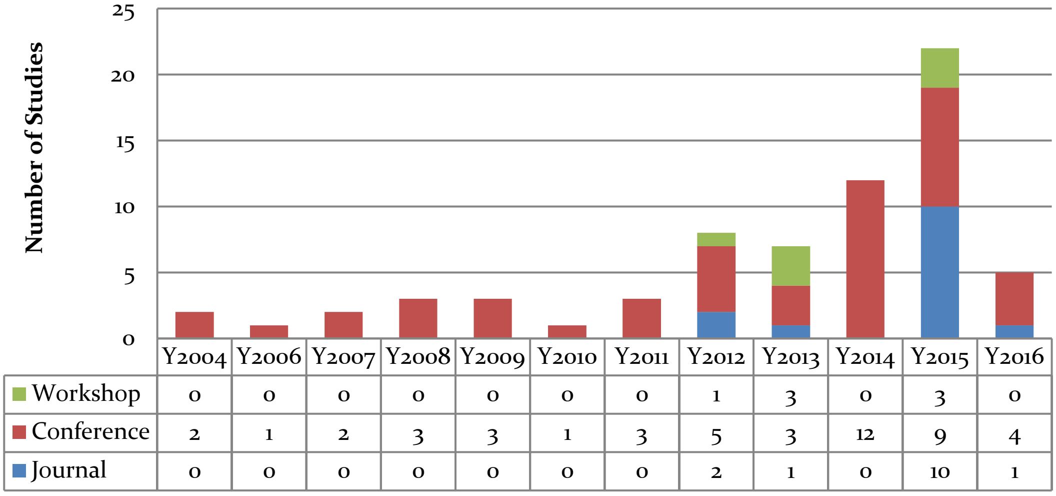
It is argued that reporting demographic information on the types and venues of the reviewed papers on a particular research topic is useful for new researchers who are interested in conducting research on that topic. Therefore, the demographic information is considered one of the important pieces of information in an SLR. Figure 3.2 summarizes how 69 primary papers are distributed along the years and the different types of venues. The selected papers were published from 2004 to 2016. Note that the review only covers the papers published before 1st June 2016. In spite of continuous practices, in particular, continuous integration and delivery are considered as the main practices proposed by agile methodologies (e.g., eXtreme Programming) introduced in early 2000, we were unable to find many relevant papers to our SLR before 2010.

We found a couple of papers that conducted surveys on the usage and importance of agile practices (e.g., continuous integration and delivery) in software development organizations before 2010, but those papers have been excluded as they did not report any approach, practice and challenge regarding CI and CDE. It is argued that CDE and CD practices have recently been known and studied in academia (i.e., last 5 years) [98]. Figure 3.2 indicates a steady upward trend in the number of papers on continuous practices in the last decade. We noticed that 39 papers (56.5%) were published during the last 3 years, suggesting that researchers and practitioners are paying more attention to continuous practices. It is clear from Figure 3.2 that conference was the most popular publication type with 48 papers (i.e., 69.5%), followed by journal (14 papers, 20.2%), while only 7 papers [S15, S23, S28, S62, S63, S64, S65] came from workshops.

There are 11 out of 14 journal papers that have been published in 2015 and 2016, which indicates that the research in the area is becoming mature. Table 3.5 summarizes that the reviewed papers were published in 47 venues, in which *IEEE Software* and *International Conference on Agile Software Development (XP)* are the leading venues for publishing work on continuous practices research as they have published 10.1% (7 papers) and 8.6% (6 papers) of the reviewed papers. The *International Conference on Software Engineering* (i.e., 5 papers) and Agile Conference (e.g., 4 papers) maintained the subsequent positions. There are two venues (i.e., ITNG and RCoSE) with only two papers each. We note that more than half of the papers (40 out of 69, 57.9%) were published in 40 different venues. Some of the publication venues are not directly related to software engineering topics such as Robotic; it indicates that the research on continuous practices is being adopted by researchers in several areas that require software development.

Table 3.5 Distribution of the selected studies on publication venues

Pub. Venue	#	%
IEEE Software	**7**	**10.1**
International Conference on Agile Software Development (XP)	**6**	**8.6**
International Conference on Software Engineering (ICSE)	**5**	**7.2**
Agile Conference	**4**	**5.7**
Information and Software Technology (IST)	**3**	**4.3**
International Workshop on Rapid Continuous Software Engineering (RCoSE)	**2**	**2.8**
International Conference on Information Technology: New Generations (ITNG)	**2**	**2.8**
Others	**40**	**57.9**



	Y2004	Y2006	Y2007	Y2008	Y2009	Y2010	Y2011	Y2012	Y2013	Y2014	Y2015	Y2016
🟩 Workshop	0	0	0	0	0	0	0	1	3	0	3	0
🟥 Conference	2	1	2	3	3	1	3	5	3	12	9	4
🟦 Journal	0	0	0	0	0	0	0	2	1	0	10	1

Figure 3.2 Number of selected studies published per year and their distribution over types of venues

3.4.1.2 Research types

This section summarizes the results from analyzing the data item D7 about research types. Table 3.6 shows that a large majority (49 out of 69, 70.9%) of the papers were reporting evaluation or validation research, in which they each correspond to 36.2% (25 papers) and 34.7% (24 papers) of the selected papers respectively. The high percentage of the evaluation research was not surprising because a noticeable number of the reviewed papers investigated and extracted challenges and practices of CI, CDE, and CD in the industry through case studies with the interview as data collection method (e.g., [S4]). That is why a vast majority of the papers in this category had used qualitative research approaches. Since prominent research methods of the validation papers are the simulation, experiments, and mathematical analysis [97], 22 out of 25 papers in this category employed quantitative research methods. We also categorized 15 (21.7%) papers as personal experience papers, in which practitioners had reported their experiences from introducing and implementing one of the continuous practices. Solution proposal (5 papers) maintained the subsequent position. To give an example, [S9] collected the opinions of three release engineers through interviews on continuous delivery's benefits and limitations, the required job skills, and the required changes in education. The reviewed papers were not fallen in the philosophical and opinion categories because we only included empirical studies.

3.4.1.3 Study context and data analysis type

We classified the reviewed papers into industry and academic cases. The industrial studies were carried out with industry or used real-world software-intensive systems to validate the proposed approach and tool; whilst the academic category refers to those studies, which were performed in an academic setting. Our review reveals that a large majority of the reviewed papers (64 out of 69, 92.7%) are situated in the industry category, whilst only 6 [S1, S2, S16, S20, S22, S40] papers were conducted in academic settings.

It shall be noted that one paper [S40] has been placed into both categories as it conducted two case studies in academic and industry. The high percentage of the industry papers indicates a significant level of relevance and practicality of the results reported in this SLR. According to Table 3.6, there was the same number of reviewed papers that used qualitative and quantitative (26 out of 69, 37.6% each) research approaches. We also found 14 papers (20.2%), which employed both qualitative and quantitative research approaches for data analysis. It was not possible for us to specify the data analysis method of three studies [S15, S30, S47] based on the provided information.

Table 3.6 Number and percentage of papers associated with each research type and data analysis type

		Data Analysis Type				
		Qualitative	Quantitative	Mixed	Unclear	Total
Research Type	Evaluation Research	S5, S6, S7, S9, S10, S12, S13, S31, S36, S43, S45, S46, S56, S60, S62, S63 **(16)**	S18, S28, S51 (3)	S4, S11, S33, S41, S44 (5)	S30 (1)	**25 (36.2%)**
	Validation Research	S22, S67 (2)	S1, S2, S3, S8, S21, S23, S24, S27, S32, S34, S38, S40, S53, S54, S55, S61, S69 (17)	S16, S20, S25, S29, S64 (5)	(0)	**24 (34.7%)**
	Experience Report	S26, S37, S42, S49, S50, S52, S65 (7)	S39, S48 (2)	S14, S17, S57, S58 (4)	S15, S47 (2)	**15 (21.7%)**
	Solution Proposal	S35 (1)	S19, S59, S66, 68 (4)	(0)	(0)	**5 (7.2%)**
	Opinion Paper	(0)	(0)	(0)	(0)	**0 (0%)**
	Philosophical Paper	(0)	(0)	(0)	(0)	**0 (0%)**
	Total	**26 (37.6%)**	**26 (37.6%)**	**14 (20.2%)**	**3 (4.3%)**	

3.4.1.4 Application domains and project types

We analyzed the data items D9 and D10 in Table 3.4 in order to provide potentially useful information for practitioners who are interested in project types and the domain-specific aspects of the approaches, tools, challenges and practices reported for CI, CDE, and CD. Table 3.7 shows the application domains in which the reviewed approaches, practices, and challenges can be placed. Regarding the application domain, not all the reviewed papers provided this information, which resulted in categorizing 38 studies under "unclear" category. For those papers that reported the application domains, we classified them into 13 application domains. The approaches, tools, and practices introduced in one study can be applied in more than one application domain with several cases. For example, the continuous integration testing approach reported in [S40] has been applied in two different domains including communication software and information management system. If one study uses more than one system as a case study, then we count this study N (number of systems) times in Table 3.7. The work reported in [S34] uses two utility software as case studies, and x represents the number of cases in S34(x).

It becomes clear from Table 3.7 that the "software/web development framework" domain has gained the most attention for continuous practices, followed by "utility software" and "data management software". We investigated the type of project (i.e., greenfield and maintenance) that continuous practices have been applied to. Our analysis of the data item D10 revealed that the greenfield and maintenance projects were reported in 17 and 16 papers respectively. However, there are 36 papers without any information about the types of projects for which the proposed continuous approaches, tools and practices had been applied.

Table 3.7 Distribution of application domains of the selected studies

Application domain	No. of Cases	Cases
Unclear	39	S1, S4, S5, S6, S9, S10, S12, S13, S16, S17, S18, S22, S23, S27, S28, S31, S33, S37, S38, S42, S44, S46, S47, S48, S49, S50, S55, S57, S58, S60, S61, S62, S63, S64, S65, S66, S67, S68
Software/Web development framework	13	S3(2), S21, S24(4), S25, S32, S34, S36, S59(2)
Utility software	9	S3(3), S24(3), S30, S34(2)
Data management software	8	S3(2), S24(4), S25, S43
Financial Software	5	S7(2), S41, S43, S45
General software library	5	S2, S3(2), S24, S25
Embedded system	5	S11, S19, S26, S52, S56
Information management system	4	S8, S24, S40, S53
Web server	3	S3(2), S24
Communication software	3	S3, S24, S40
Military Software	3	S11(2), S39
Distributed system	2	S14, S54
Web browser	2	S29, S69
Other domains	11	S3 (4), S7, S15, S20, S51, S24, S34, S35

3.4.2 Approaches and associated tools to facilitate continuous integration, delivery, and deployment? (RQ3.1)

We found 29 papers (42%) that reported approaches and associated tools to support and facilitate continuous integration, delivery or deployment practices. Table 3.8 lists all approaches and associated tools presented in the reviewed papers. The *Description* column provides a summary of the proposed approaches and associated tools. The third column indicates the proposed approaches and tools have been mainly used and applied to facilitate what continuous practices. We classified the available approaches and associated tools into six groups depending on their features and/or the areas in which they were used as the followings. Apparently, the six categories are not mutually exclusive, as there were several approaches and tools fallen in more than one category. For brevity purpose, we only elaborate a small subset of the studies as examples.

3.4.2.1 Reduce build and test time in CI

The approaches and tools in this category aim at minimizing the total time in the build process and test phase, which consequently improves performance and efficiency of continuous integration practice [S3, S19, S23, S25, S34, S55, S64, S67]. Since slow build process can be an obstacle to practicing continuous integration, Bell et al. [S3] proposed two approaches namely VMVM (Virtual Machine in a Virtual Machine) and VMVMVM (Virtual Machine in a Virtual Machine on a Virtual Machine) to isolate in-memory and external dependencies among test cases respectively. Whilst eliminating in-memory dependencies between tests enables running each test in its own process, which significantly reduces the overhead of dependencies among short test cases, VMVMVM approach executes the long-running test cases in parallel. The combination of VMVM and VMVMVM accelerates the total build time, which can relieve a deployment pipeline from long-running builds.

A number of papers [S34, S55, S64] in this category developed approaches that reduce the time of test execution by selecting a set of tests cases and prioritizing them, in which developers are enabled to receive the results early in the testing process. To give an example, Elbaum et al. [S55] proposed CRTS (Continuous Regression Test Selection) and CTSP (Continuous Test Suite Prioritization) approaches to effectively run regression tests within continuous integration development environments. The proposed approaches use test suite execution history data to improve the cost-effectiveness of pre-submit testing (i.e., tests performed by developers before committing code to the repository) and reduce test case execution costs.

McIntosh et al. [S25] revealed that in C and C++ applications there might be header files that not only increase the time of the rebuild process but also due to frequent maintenance requires significant effort. Thus, these header files, called hotspots, are a bottleneck to continuous integration build process. Through analysis of the Build Dependency Graph (BDG) and the change history of a system, the proposed approach in [S25] enables the team to identify the header files that should be optimized first to improve build performance. Hence, the team members only can focus on header files with added value.

3.4.2.2 Increase visibility and awareness on build and test results in CI

As the frequency of code integration increases, the information (build and test results) produced during practicing CI would increase exponentially. This may considerably slow down the feedback in CI. Therefore, it is critical to collect and represent the information in a timely manner to help stakeholders to gain better and easier understanding and interpretation of the results. Several studies [S1, S2, S13, S22, S24, S33, S38, S52, S64, S67] have reported approaches and associated tools for improving developers' understanding of their projects' status when implementing CI practice. The authors of [S2] found that stand-alone CI tools (e.g., Jenkins) produce a huge amount of data that may not be easily utilized by stakeholders (e.g., developers and testers). They developed a framework and platform called SQA-Mashup to integrate and visualize the information produced in CI-toolchain using two views: (1) dynamic view, which is a visualization view for developers and testers and (2) time view, which indicates a chronological view on events (i.e., failure event) happened in CI-toolchain. It was found that the interpretation of the proposed views is time-consuming and should be performed by professionals (e.g., tester). Brandtner et al. [S24] proposed a rule-based approach, named SQA-Profile, to classify stakeholders based on their activities in the CI environment. The project-independent SQA-Profile enables tailoring and dynamic composition of scattered data in the CI system. Nilsson et al. [S13] have found that companies need to describe and arrange testing activities and efforts before moving to CI. CIViT (Continuous Integration Visualization Technique) aims at visualizing the end-to-end process of testing activities. CIViT enables team members to avoid duplicate testing efforts and visually understand the status (i.e., time and extent) of testing of quality attributes.

3.4.2.3 Support (semi-) automated continuous testing

There are 7 papers that have proposed approaches and tools for (semi-) automating tests in deployment pipelines [S19, S32, S38, S40, S52, S53, S54]. Two papers [S40, S53] have provided frameworks to support Continuous Integration Testing (CIT) in SOA systems. Whilst the work reported in [S40] partly automates test case generation in CIT using sequence diagrams as input, Surrogate, the simulation framework proposed by [S53], enables CIT for partial implementation. Through this framework, bugs can be identified when some components or even all components are still unavailable. Kim et al. [S38] proposed NHN Test Automation Framework (NAFT) as an integrator for existing CI servers to facilitate CI practices through automating repetitive and error-prone processes for testing. In addition, tests and test environments are visualized using tables and communication among stakeholders would be improved.

3.4.2.4 Detect violations, flaws and faults in CI

Addressing the failures and violations in continuous integration systems, particularly at the early stage of development are the targets of several papers [S16, S21, S25, S32, S33, S34, S42, S52, S53, S54, S55]. For example, one study [S16] reported an approach and associated tool called WECODE to automatically and continuously detect software merge conflicts earlier than a version control system is used by developers. The tool enables developers to detect the conflicts in uncommitted code that version control systems are not able to detect. In [S21], the authors developed a method includes incremental integration with simple and true backtracking in order to reduce the impacts of broken builds in the context of component-based software development. In the normal situation, a failure in the build process of a component stops the integration process. The failure should be resolved, and the component needs to be rebuilt. But the incremental integration method addresses this issue by building components using the earlier build results of the same components. This approach leads the integration process becomes more resilient against build failures.

3.4.2.5 Address security and scalability issues in the deployment pipeline

Our literature review has identified only two papers dealing with the security issue in deployment pipelines [S27, S66]. Gruhn et al. argued that continuous integration systems are vulnerable to security attacks and misconfiguration [S27]. Having proposed a secure build server, they encapsulated build jobs using virtualization environment with snapshot capability to prevent one project's security attacks from infecting other projects' build jobs in multitenant CI systems. In [S66], it has been discussed that the security of a deployment pipeline may be threatened by malicious code being deployed through the pipeline and direct communication between components in the testing and production environments. Rimba et al. [S66] proposed an approach, which integrates security design fragments (i.e., security patterns) through four compassion primitives namely connect tactic, disconnect tactic, create tactic, and delete tactic to secure deployment pipelines. For a large-scale software project, the full build can take hours as it includes compilation, unit testing, and acceptance testing. Roberts [S47] has extended normal continuous integration process and proposed Enterprise Continuous Integration (ECI) approach to split up a project into several modules using binary dependencies. Despite every module has its own CI, ECI provides the feedback that a single-project CI provides. ECI addresses the scalability issue in normal CI and enables small teams to continuously integrate with the binary dependencies developed by other teams.

3.4.2.6 Improve dependability and reliability of deployment process

Some papers [S8, S59, S68] dealt with the deployment process of applications that have adopted continuous delivery or deployment practices. The work reported in [S8] investigated the reliability issue in high-frequency releases of Cloud applications. It has been argued that two major contributing factors i.e., cloud-infrastructure APIs (EC2 API) and deployment-tool (i.e., OpsWorks[5] and Chef[6]) can affect the reliability of cloud applications when they adopt continuous delivery and deployment. Four error-handling approaches have been implemented on rolling upgrade tool to deal with reliability issues and facilitate continuous delivery. Increasing the frequency of deployment (e.g., by adopting CD practice) would make error diagnosis harder during sporadic operations [S68]. An approach, called Process Oriented Dependability (POD), has been proposed to improve the dependability of the deployment process in cloud-based systems. The POD approach models the sporadic operations as processes through collecting metrics and logs in order to alleviate the difficulty of error diagnosis process in deploying cloud-based systems on a continuous basis.

Table 3.8 A classification of approaches and associated tools to facilitate continuous integration, delivery and deployment: ❶ (reduce the build and test time in CI); ❷ (increase visibility and awareness of build and test results in CI); ❸ (support (semi-) automated continuous testing); ❹ (detect violations, faults, and flaws in CI); ❺ (address security and scalability issues in deployment pipeline); ❻ (improve dependability and reliability of deployment process)

Description of Approaches and Tools	Category	Apply to
Wallboard technique [S1]: It indicates the current integration and delivery status of all branches within a project.	❷	All
SQA-Mashup [S2]: It can integrate and visualize data produced in CI environments.	❷	CI
VMVM/VMVMVM [S3]: It is used to isolate in-memory and external dependencies among test cases.	❶	CI
Error-handling approaches on rolling upgrade [S8]: A set of error-handling approaches to deal with reliability issues which are inherent to cloud environments.	❻	CDE/CD
CIViT [S13]: It is used to visualize the end-to-end process of testing activities in the transformation to continuous integration. By visualizing the end-to-end process of testing activities, team members are enabled to reduce testing efforts.	❷	CI
WECODE [S16]: It automatically and continually detects software merge conflicts earlier than a version control system is used by developers as well as detects conflicts in uncommitted code.	❹	CI
uBuild [S19]: It provides continuous testing making reproducible and deterministic tests in order to achieve automated build.	❶❸	CI
Backtracking Incremental Continuous Integration [S21]: Through simple and true backtracking approaches, this approach increases the resilience of build process against failures and ensures that a working version is available at all times.	❹	CI
BuildBot Robot [S22]: It notifies who is responsible for test failure in the CI environment in a friendly and funny way. It makes continuous integration	❷	CI

[5] https://aws.amazon.com/opsworks/
[6] https://www.chef.io/chef/

environment visible to all developers.		
Hydra [S23]: It is Nix-based continuous build tool, which automatically produces a build environment for projects. Therefore, it can reduce the efforts to maintain a continuous integration environment.	❶	CI
SQA-Profile [S24]: Through a set of rules, it can provide a dynamic composition of CI dashboards based on stakeholder activities in tools of a CI environment.	❷	CI
Hotspot Approach [S25]: In order to have a fast build system in continuous integration infrastructure, this approach identifies header files that are bottlenecks for the build process.	❶❹	CI
Secure Build Server [S27]: It extends the default build server in a CI environment using encapsulating infected build jobs and prevents spreading infection to other build jobs in multitenant CI systems.	❺	CI
Automatic and agile testing of product lines based on combinational interaction testing [S32]: It makes automatic testing as an integrated part of continuous integration framework and enables developers of software product lines to identify potential interaction faults in the build process.	❸❹	CI
Ambient awareness-based approach [S33]: It enhances build status awareness among team members, which results in decreasing the number of broken builds and a strong sense of responsibility towards failures in the build process.	❷❹	CI
Integrating fault localization and test case prioritization technique in CI [S34]: The fault location and test case prioritization approaches are combined to support commit built in continuous integration, which consequently improves efficiency (time) and effectiveness of the whole CI process.	❶❹	CI
NHN Test Automation Framework [S38]: It supports CI practices through automating repetitive and error-prone processes for testing in a continuous integration environment. It aids communication among various stakeholders using tables to represent tests and test environments.	❷❸	CI
Continuous Integration Testing for SOA [S40]: A Unified Test Framework (UTF) for Continuous Integration Testing (CIT) of SOA, which would partly automate test case generation in CIT using sequence diagrams as input.	❸	CI
User-defined Script [S42]: It supports enforcement at commit time by establishing a pre-commit step (i.e., Subversion pre-commit hook) to force developers into fixing the violation in code commit. It does not allow committing codes that comply with conventions (e.g., as-build architecture).	❹	CI
Enterprise Continuous Integration [S47]: A modified version of the normal continuous integration process to split up the project into several modules using binary dependencies. Despite every module has its own CI, ECI provides the feedback that single-project CI provides.	❺	CI
Tinderbox [S52]: It is a continuous integration and automated testing system, which helps find integration problems earlier in the development cycle, reduce the cost to fix the integration problems and improve visibility and awareness among team members.	❷❸❹	CI
Surrogate [S53]: A simulation framework to implement continuous integration testing for SOA systems when some components or even all components are still unavailable. With this, bugs are identified at the early stage of development.	❸❹	CI
CiCUTS [S54]: It integrates CUTS, system modeling executing tool, with continuous integration environments. Through this approach, developers and testers are enabled to continuously perform system integration testing on target environments using emulation approaches and identify performance problems before components are completely implemented.	❸❹	CI

Continuous Regression Test Selection (CRTS) [S55]: It enables running effectively regression testing within continuous integration development environments. The technique improves the cost-effectiveness of pre-submit testing (i.e., tests performed by developers before committing code to the repository) and reduce test case execution costs.	❶	CI
Continuous Test Suite Prioritization (CTSP) [S55]: It can reduce delays in fault detection during post-submit testing (i.e., all tests that are performed after code submitted to the repository). In overall, it can improve the cost-effectiveness of the continuous integration process.	❶❹	CI
Rondo [S59]: Adopting continuous deployment in dynamic environments such as pervasive computing environments is associated with a number of challenges. Deployment process in such environments should be reproduced in different sites, support customizability, and should be equipped with custom rollback mechanism. Rondo, an automation tool, satisfies all above-mentioned requirements to facilitate adopting continuous deployment practice in dynamic environments.	❻	CDE/CD
Code-Churn Based Test Selection (CCTS) [S64]: This technique analyses code churns and test execution results to select an optimal subset of test suites on the system level. So, it helps large-scale software development organizations speed up CI. It enables team members to gain a better understanding of the number of test failures.	❶❷	CI
Enhancing the security design of a deployment pipeline [S66]: This approach integrates four security design fragments (i.e., security patterns) at the design level to secure deployment pipelines. The security of the pipeline is ensured through not allowing malicious code is deployed through the pipeline and preventing direct communication between components in the testing and production environments.	❺	CDE/CD
Morpheus [S67]: It facilitates CI practice through improving the quality of feedback, in which each developer only receives the test results of his own changed code (i.e., easy interpretation of test results). Additionally, in order to minimize build and test time, it executes automated tests in the environment that is similar to the production environment.	❶❷	CI
Process Oriented Dependability (POD) [S68]: An approach to improve the dependability of the deployment process in cloud-based systems. This approach models the sporadic operations as processes in order to alleviate the difficulty of error diagnosis during sporadic operations when CD practice is adopted and implemented.	❻	CDE/CD

3.4.3 Tools used to design and implement deployment pipelines? (RQ3.2)

This section presents the findings to answer to RQ3.2 Deploying software on a continuous basis to end users has increased the importance of deployment pipelines [98]; the success of adopting continuous practices in enterprises heavily relies on deployment pipelines [78]. Hence, the choice of appropriate tools and infrastructures to make up such pipeline can also help mitigate some of the challenges in adopting and implementing continuous integration, delivery and deployment practices. We have investigated the deployment toolchain reported in the literature and the tools for implementing deployment pipelines. Since continuous delivery and deployment might be used interchangeably, we used the term deployment pipeline, which is equal to the modern release engineering pipeline [98], instead of continuous integration infrastructure, or continuous delivery or deployment pipeline.

A deployment pipeline should include explicit stages (e.g., build and packaging) to transfer code from the code repository to the production environment [7, 78]. Automation is a critical practice in the deployment pipeline; however, sometimes manual tasks (e.g., quality assurance tasks) are unavoidable in the pipeline. It is worth noting that there is no standard or single pipeline [78]. Our literature reveals that only 25 out of 69 studies (36.2%) discussed how different tools were integrated to implement toolchain to effectively adopt continuous practices. It should be noted that the tools reported in this section are mostly existing open sources and commercial tools, which aim to form and implement a deployment pipeline. However, the tools discussed in Section 3.4.2 are intended to facilitate the implementation of continuous practices. These tools can be also used as part of the deployment pipeline implementation provided that they are integrated and evaluated in the pipeline. As shown in Figure 3.3, we divided the deployment pipeline into 7 stages: (i) version control system; (ii) code management and analysis tool; (iii) build tool; (v) continuous integration server; (vi) testing tool; (vii) configuration and provisioning; and (viii) continuous delivery or deployment server. It should be noted that not all stages are compulsory as well as we could not find any primary study among the 25 studies that had implemented a pipeline involving all stages mentioned in Figure 3.3. At the first stage, developers continually push code to code repository. The most popular version control systems used in deployment pipelines are *Subversion*[7] and *Git/GitHub*[8] as each has been reported in 6 papers. We found 7 papers [S2, S14, S18, S20, S42, S52, S62], which used code management and analysis tools as part of the deployment pipeline to augment the build process. The work reported in [S20] integrated *SonarQube*[9] into *Jenkins*[10] CI server for gathering metric data such as test code coverage and coding standard violations and visualized them to developers. Continuous integration servers check the code repository for changes and use automated build tool [99]. Through CI servers, it is possible to automatically trigger the build process and run unit tests. *Jenkins* [S2, S14, S17, S20, S26, S27, S30, S35, S62, S63, S66] has gained the most attention among existing CI servers in the literature. It should be noted that some CI servers (e.g., *Jenkins*, *Bamboo*[11], and *Hudson*[12]) are also able to deploy software to staging or production environment [100]. A study reported in [S30] used *Jenkins* as continuous delivery/deployment server.

Bamboo and *CruiseControl* maintained the subsequent positions. [S39, S58] used TeamCity as CI server in the pipeline and other CI servers have been reported in one paper each. The next step of the deployment pipeline is to run a set of tests in various environments. There are only four papers [S18, S35, S54, S62], which integrated testing tools as part of the deployment pipeline. Two papers [S35, S18] employed *JUnit*[13] and *NUnit*[14] for the unit test in the pipeline respectively, while one paper [S35] also used a test runner called *Athena* to execute test suites and store the results in a format that can be used by Jenkins. Furthermore, *TestLink*[15] as a test management framework has been employed to store the results of acceptance tests run on different sites. The work reported in [S54] combined *CUTS* as a system modeling executing tool to *CruiseControl*[16] to enable developers and testers to continuously run system integration tests at the early stages of the software lifecycle (i.e., before complete system integration time) of component-based distributed real-time and embedded systems. The tool can capture performance metrics of executing systems such as execution time, throughput, and the number of received events. It is asserted that providing automated configuration of servers and virtual machines is one of the innovations in deployment pipelines [98]. That can be the reason why we observed only two studies [S58, S63] that used configuration management tools as an integrated part of the deployment pipeline to streamline the configuration and provisioning tasks. One study [S1] used *HockeyApp*[17] as a continuous delivery server to distinguish external release from internal one as well as it enables to deliver a build as a release to customers. The cases reported in [S17, S62] respectively used a Ruby-based software deployment called *deoloyr* and *Web Deploy* tool to automatically deploy code to production.

[7] https://subversion.apache.org/
[8] https://github.com/git/git
[9] www.sonarqube.org/
[10] https://jenkins-ci.org/
[11] https://www.atlassian.com/software/bamboo/
[12] hudson-ci.org/
[13] junit.org/
[14] http://www.nunit.org/
[15] testlink.org/
[16] http://cruisecontrol.sourceforge.net/
[17] http://hockeyapp.net/features/

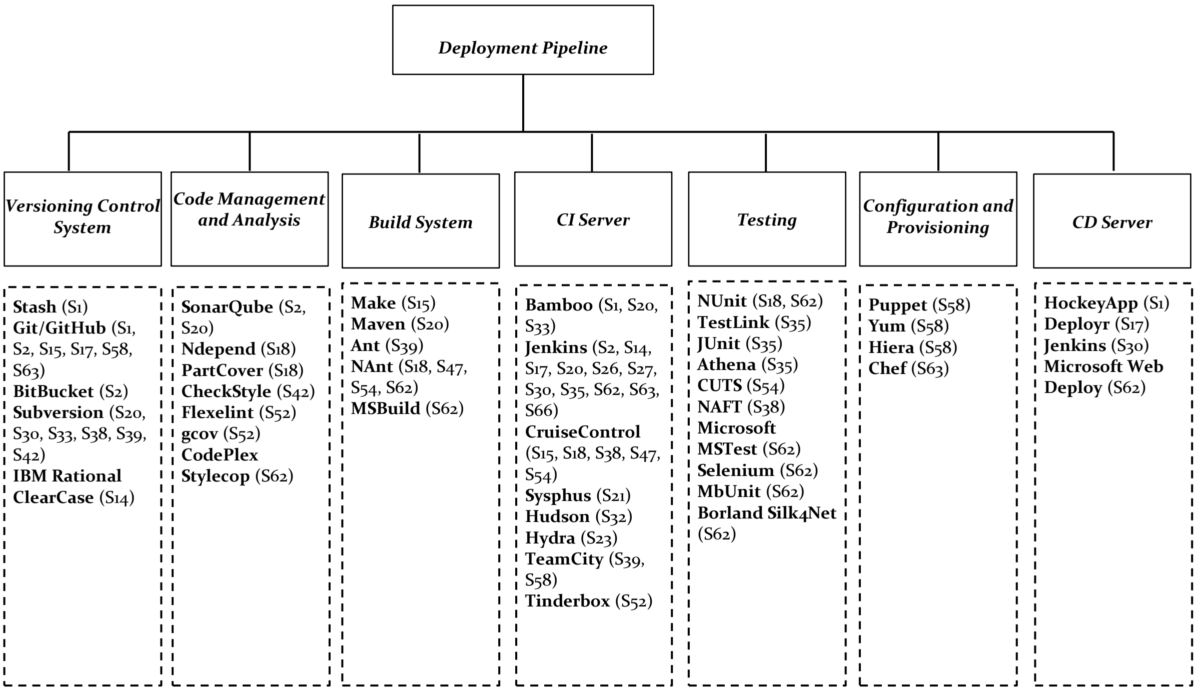


Figure 3.3 An overview of tools used to form deployment pipeline

3.4.4 Challenges of adopting continuous practices (RQ3.3)

This section summarizes the results of RQ3.3, *"What challenges have been reported for adopting continuous practices?"* As discussed in Section 3.3.5.2, we analyzed the data item D12 using the thematic analysis method [56] for identifying and synthesizing the challenges for moving to and adopting CI, CDE, and CD. Our analysis resulted in the identification of 20 challenges, which are shown in Table 3.9. We provide detailed descriptions of the identified challenges as a follow:

3.4.4.1 Common challenges for adopting CI, CDE and CD practices

Under this category, we list the challenges of implementing all continuous integration, delivery and deployment practices together. Most of the challenges are usually associated with introducing any new technologies or phenomena in a given organization.

A) Team Awareness and Communication

Lack of awareness and transparency: Our review has identified several papers that report a lack of sufficient awareness among team members may break down transition towards continuous practices [S6, S10, S31, S43, S45, S50, S56, S62]. Espinosa et al. [101] defined "awareness" as short-term knowledge about a team and its tasks. Continuous delivery process should be designed in a way that the status of a project, number of errors, the quality of features, and the time when features are finished are visible and transparent for all team members [S10, S31, S43, S50]. The work reported in [S31] asserted a lack of sufficient knowledge about the changes made in the main branch during developing work packages by self-organized teams resulted in an increased number of merge conflicts in delivery.

Coordination and collaboration challenges: Some of the reviewed studies also reported that successfully implementing continuous practices requires more collaborations and coordination between all team members [S4, S6, S10, S41, S45, S56, S62]. For example, compared to less frequent release, deploying software on a continuous basis requires more communication to and coordination with operations teams [S4]. Gmeiner et al. [S62] argued that the real benefits of the deployment pipeline can be obtained by having a common understanding and collective responsibilities among all stakeholders. Another study [S41] noted that there is a need for strong coordination and communication between the release manager and other team members (e.g., testers) to improve the release process. Laukkanen et al. [S45] also reported coordination and collaboration challenges as part of adopting continuous integration in distributed teams.

B) Lack of Investment

***Cost*:** Cost and investment play an important role in embracing continuous practices in both customer and software development organizations. Several of the reviewed studies [S4, S6, S12, S27, S37, S43, S45, S49, S57, S62] reported that practicing efficiently each of the continuous integration, continuous delivery or deployment is associated with the high cost that can be attributed to many factors. For example, a study [S37] reported that a major resource upgrade was needed to support CI practice. Gruhn et al. [S27] observed that adopting continuous integration in Free, Libre and Open Source Software (FLOSS) requires extra computation, bandwidth, and memory resources. CI systems are required to perform build jobs, which include downloading patch files, compiling new versions of code, and running a large set of unit and acceptance tests.

The work reported in [S43] revealed that performing automated acceptance tests in the deployment pipeline requires a significant amount of resources from customers. Two studies [S57, S62] observed that building, improving, and maintaining infrastructures (e.g., deployment pipeline) for continuous deployment practice needed a significant amount of time, money and training. There was also cost associated with training and coaching team members to adopt continuous practices [S57].

***Lack of expertise and skill*:** Several papers [S4, S5, S6, S12, S45, S49, S57] reported a significant gap in the required skills when implementing continuous practices. This is mainly because most of the practices (e.g., test and deployment automation) associated with CI, CDE, and CD demand new technical and soft (e.g., communication and coordination) skills and qualifications. Several studies [S4, S6, S57] indicated the needs of highly skilled developers for practicing CD.

***More pressure and workload for team members*:** It has been reported that building high-quality applications that are supposed to be frequently released to customers may cause some team members to face more stress and extra efforts [S4, S5, S6, S45, S49, S58]. Callanan and Spillane [S58] discussed that the operations team was under more pressure to deliver software on a continuous basis. The study reported in [S49] has found that transforming a six-month release into continuous release noticeability increased the workload of the developers and the release team. Whilst the transition forced developers to more analyze their codes in order to thoroughly identify the negative side effects of their codes, the release team experienced difficulties to find issues in the release process. One reason for this pressure could be that team members are directly responsible for affecting their customers' experiences.

Lack of suitable tools and technologies: According to eleven studies [S5, S6, S8, S10, S27, S43, S49, S56, S57, S60, S66], the limitations of existing tools and technologies are inhibitors to achieving the goals of continuous practices. Researchers pointed out [S5, S10] that the existing tools are inefficient in reviewing code and providing feedbacks from test activities in continuous integration. They emphasized that test automation is not sufficiently provided by current infrastructure. Other studies [S8, S27] highlighted the build and deployment tools employed in the deployment pipeline are vulnerable to security and reliability issues. Analyzing the reliability issue in high-frequency releases of Cloud applications revealed that using external resources and cloud-based tools in a deployment pipeline leads to increased errors and delays, which consequently hinders continuous delivery practice [S8].

Olsson et al. [S10] indicated that the high-frequency changes in tools and the need for learning new tools are the major barriers to adjust to continuous integration. Three papers [S56, S57, S60] revealed that the current tools and technologies either have limited functionalities or cannot enable all organization to truly adopt CD practice. To exemplify, a study [S56] reported that a lack of appropriate technologies hindered automatically and continuously deploying applications in embedded system domain with customer-specific environments.

C) Change Resistance

***General resistance to change*:** Whilst employees generally resist to change, people may embrace changes provided that there are convincing reasons for those changes [102]. Introducing continuous practices may necessitate adopting a new way of working for some team members (e.g., accepting more responsibilities by developers). The reviewed studies reported that objections to change were a barrier to move towards and successfully implement continuous practices [S4, S5, S6, S12, S56, S57, S62]. A study [S62] found that establishing the necessary mindset required by a continuous delivery was a time-consuming process; another study [S5] concluded that changing the old habits of developers was problematic when introducing CI. Our investigation revealed that the team members were unwilling to change their ways of working due to lack of trust and rapport on the benefits of continuous practices, fear of exposing low-quality code, and suffering more stresses and pressures.

***Skepticism and distrust on continuous practices*:** Six papers [S4, S5, S6, S12, S45, S49] referred to lack of trust and skepticism about the added values that may bring by adopting continuous practices as potential risks for moving towards these practices. To give an example, the experience reported in [S49] revealed that the release team was worried about allowing several concurrent releases. This is mainly because continuous release might bring side effects to them and make them unable to identify which release was causing which problem. In addition, another study [S12] reported that lack of trust in application's quality may reduce the confidence of team members to move from CI to CD and deploy the application to production on a continuous basis.

D) Organizational Processes, Structure, and Policies

***Difficulty to change established organizational policies and cultures*:** According to [103], the organizational culture is a set of habits, behaviors, attitudes, values and management practices adopted by an organization. Two studies [S10, S12] discussed the difficulties in changing organizational cultures for aligning with the principles of continuous practices. Based on a study, Olsson et al. [S10] reported that being traditionally a hardware-oriented company was an obstacle in the transition towards CI practices, however, [S12] highlighted this issue as the case company used to have a six-month release cycle. Both papers revealed the lack of suitable and agile business model in organizations resulted in negative consequences for continuous practices. Rissanen and Münch [S43] found that practicing the short-lived feature branching, which is regarded as one of the best practices in continuous delivery is not easy to apply in a company with long-established practices.

***Distributed organization*:** It has been reported that practicing continuous integration and deployment in distributed development teams can be associated with a number of challenges (i.e., lack of visibility) [S12, S37, S45]. In both cases [S12, S45], the authors argued that introducing CI practice in distributed development model was challenging. That is mainly because it would prohibit having consistent perceptions among distributed teams and decrease the visibility of development sites. In an experience reported by Sutherland and Frohman [S37], it has been asserted that the distributed development model adopted by Scrum team was a barrier to CI practice. It is mainly because allocating a dedicated and private integration server environment to each individual Scrum team led to detecting integration issues that have been postponed to a very large extent. As a result, the team was forced to put all teams into a single server environment.

Table 3.9 A classification of challenges in adopting CI, CDE, and CD practices

		Challenges	Key Points and Included Papers	#
Common Challenges for Adopting CI, CDE, and CD	Team Awareness and Communication	**Ch1**. Lack of awareness and transparency	▪ Lack of awareness and transparency in the delivery process [S6, S10, S31, S43, S45, S50, S56, S62] ▪ Lack of understanding about the status of the project increases the number of merge conflicts [S31]	8
		Ch2. Coordination and collaboration challenges	▪ Practicing CI, CDE, CD needs more and effective coordination and communication between team members [S4, S6, S10, S41, S45, S56, S62]	7
	Lack of Investment	**Ch3**. Cost	▪ Major upgrade in infrastructures and resources [S4, S6, S12, S27, S37, S43, S45, S49] ▪ Training and coaching [S57, S62]	10
		Ch4. Lack of experience and skill	▪ CI, CDE, and CD demand new technical and soft skills [S4, S5, S6, S12, S45, S49, S57] ▪ Need highly skilled developers [S4, S6, S57]	7
		Ch5. More pressure and workload for team members	▪ More stress for developers and operations team [S4, S5, S6, S45, S49, S58] ▪ More responsibilities for developers [S49]	6
		Ch6. Lack of suitable tools and technologies	▪ Lack of mature tools for automating tests and reviewing code in CI [S5, S6, S10, S43, S49] ▪ Frequency changes in tools [S10] ▪ Security and reliability issues in build and deployment tools [S8, S27, S66] ▪ Current tools don't fit all organizations [S56, S57, S60]	11
	Change resistance	**Ch7**. General resistance to change	▪ Changing the old habits of team members [S4, S5, S6, S12, S56, S57, S62] ▪ A time-consuming process to change team mindset [S62]	7
		Ch8. Skepticism and distrust on continuous practices	▪ Lack of trust on benefits of CI, CDE, CD [S4, S5, S6, S12, S45, S49]	6
	Organizational processes, structure and policies	**Ch9**. Difficulty to change established organizational policies and cultures	[S4, S6, S10, S12, S43] ▪ Lack of agile and suitable business model [S10, S12] ▪ Changing long-lived feature branching to short-lived one in an established company [S43]	5
		Ch10. Distributed organization	▪ Distributed team model [S12, S37, S45] ▪ Inconsistent perceptions among team members [S12, S45]	3

Challenges for Adopting CI	Testing	**Ch11**. Lack of proper test strategy	▪ Lack of fully automated testing [S4, S5, S12, S36, S41, S43, S45] ▪ Lack of test-driven development [S12]	7
		Ch12. Poor test quality	▪ Instable tests [S4, S5, S6, S41, S45, S50, S62] ▪ Low test coverage [S56] ▪ Low-quality test data [S6] ▪ Long-running tests [S4, S5, S45, S50] ▪ Test dependencies [S5, S41]	8
	Merging conflicts	**Ch13.** Merging conflicts	[S4, S6, S21, S31, S41, S45] ▪ Third party components [S45] ▪ Incompatibly among dependent components [S31] ▪ Lack of understanding about changed components [S31]	5
Challenges for Adopting CDE	Lack of suitable architecture	**Ch14.** Dependencies in design and code	[S4, S5, S6, S10, S31, S41, S57, S60] ▪ Highly coupled architectures [S60] ▪ Difficulty to find autonomous requirements for frequent integrations [S5]	8
		Ch15. Database schema change	▪ Frequent changes in database schema [S6, S57, S58, S62]	4
	Team dependencies	**Ch16**. Team dependencies	▪ Cross-team dependencies [S6, S31, S45, S50, S56, S57] ▪ Ripple effects of changes on multiple teams [S50] ▪ Dependency between feature team and module team in embedded system domain [S56]	6
Challenges for Adopting CDE	Customer challenges	**Ch17**. Customer environment	▪ Lack of access to customer environment [S56, S60] ▪ Complex and manual configuration [S10, S62] ▪ Diversity and complexity of customer sites [S4, S6, S10, S29, S43] ▪ Difficulty to stimulate production-like environment [S56, S60]	8
		Ch18. Dependencies with hardware and other (legacy) applications	▪ Releasing an application on a continuous basis requires deploying all dependent applications in customer site [S6, S10, S29, S43, S56, S62] ▪ Hardware and network dependencies [S56]	6
		Ch19. Customer preference	▪ Not all customers happy with the frequent release [S6, S29, S43] ▪ Customer organization policy may affect practicing CD [S57]	4
	Domain constraints	**Ch20**. Domain constraints	▪ Some domains don't allow or cause difficulties to truly adopt and implement CD [S4, S5, S6, S9, S10, S24, S31, S41, S44, S48, S56, S57, S60, S65]	14

3.4.4.2 Challenges for adopting CI practice

A) Testing

Lack of proper test strategy: One of the most prominent roadblocks to adopting continuous integration reported by several studies was the challenges associated with the testing phase. Whilst it is asserted that automated test is one of the most important parts of successfully implementing CI, the case organizations studied in [S4, S5, S12, S36, S41, S43, S45] were unable to automate all types of tests. Lack of fully automated testing may stem from different reasons such as poor infrastructure for automating tests [S12], time-consuming and laborious process for automating manual tests [S43] and dependencies between hardware and software [S5]. Whilst lack of test-driven development (TDD) practice has been reported in [S12] as a barrier to establishing CI practice, Debbiche et al. [S5] have revealed that regardless of TDD being practiced or not, a huge dependency between code and its corresponding tests made integration step very complicated. The work reported in [S36] revealed that although automating Graphic User Interface (GUI) testing through applying a set of GUI testing tools could partially alleviate the challenges of rapid release, but due to reliability concerns, the quality assurance (QA) members were needed to manually check the system during running automatic test.

Poor test quality: The next challenge in testing phase during CI adoption is about low-test quality. This includes having unreliable tests (i.e., frequent test failures) [S4, S5, S6, S41, S45, S50, S62], the high number of test cases [S50], low test coverage [S56] and long-running tests [S4, S5, S45, S50]. These issues not only can impede the deployment pipeline but also can reduce the confidence of software development organizations to automatically deploy software on a continuous basis. Rogers [S50] observed that the number of tests grows in large-codebase and they run slowly. Therefore, developers are not able to receive the feedback from tests quickly and practicing CI starts to break down. To give another example, the author of [S62] found that it is hard to stabilize tests at the user interface level.

B) Merging Conflicts

Our review has revealed that conflicts during code integration cause bottlenecks for practicing CI [S4, S6, S21, S31, S41, S45]. There are several reasons for these conflicts that can occur when integrating code: one study [S45] reported that third-party components caused severe difficulty to practice CI. Sekitoleko et al. [S31] observed that incompatibility among dependent components and lack of knowledge about changed components caused teams facing extra effort to rewrite their solutions. It is asserted that merge conflicts are mainly attributed to the highly coupled design [S31, S41].

3.4.4.3 Challenges for adopting CDE practice

A) Lack of Suitable Architecture

We found several studies discussing that unstable application architectures create hurdles in the transition towards continuous delivery and deployment practices.

***Dependencies in design and code*:** Some authors [S4, S5, S6, S10, S31, S41, S57, S60] asserted that inappropriately handling dependencies between components and code cause challenges in adopting continuous integration and in particular continuous delivery and deployment practices. The work reported in [S10] argued that the existence of huge dependency between components and the dependency between components interfaces resulted in highly dependent development teams and the ripple effect of changes. It has been concluded that highly coupled architectures can cause a severe challenge for CDE practice because changes are spanned across multiple teams with poor communications between them [S57, S60]. There was only one paper [S5], which considered software requirements as a challenge for CI as the interviewees reported that (i) finding the right size of requirements for being tested separately when broken down is challenging; (ii) it is not easy to understand whether small changes that do not directly add value to a feature are worth integrating or not.

Database schema changes: Technical problem relating to database schema changes should be effectively managed in the deployment pipeline. Four reviewed studies [S6, S57, S58, S62] revealed that frequent changes in database schema as a technical problem when moving to continuous delivery. One study [S6] in this category highlighted that small changes in code resulted in constant changes in database schemas. Another study [S62] argued that a large part of concern in the configuration of the automated test environment involved setting up databases. The study reported in [S57] discussed that one of the studied case companies did not put extra effort to streamline its database schema changes, which resulted in severe bottlenecks in its deployment process.

B) Team Dependencies

Team structures and interactions among multiple teams working on the same codebase system play an important role in successfully implementing CDE and CD practices. Several of the reviewed studies [S6, S31, S45, S50, S56, S57] reported that high cross-team dependencies prohibited development teams to develop, evolve and deploy applications or components and services into production independently of each other. This issue also has a major impact on practicing CI as a small build break or test failure may have ripple effects on different teams [S50]. The author of [S56] argued that feature and module (hardware) teams developing embedded domain systems were highly dependent, in which each feature was compiled, tested and built by a combination of both teams. This required a strong and proper communication and coordination among them. Two studies [S50, S57] in this group also discussed that nonexistence of a suitable architecture can increase the cross-team dependency.

3.4.4.4 Challenges for adopting CD practice

It has been noted that CD practice may not be suitable for any organizations or systems. We discuss the challenges and barriers that can limit or demotivate organizations from adopting CD practice.

A) Customer Challenges

Customer environment: A set of papers discussed that diversity and complexity of customers' sites [S4, S6, S10, S29, S43], manual configuration [S10, S62], and lack of access to customer environment [S56, S60] may cause challenges for team members when transferring software to customers through CD practice. According to [S4, S43], continuously releasing software product to multiple customers with diverse environments was quite difficult as it was needed to establish different deployment configurations for each customer's environment and component's version. Two papers [S56, S60] reported that it was not easy, if possible, to provide production-like test environment. Lwakatare et al. [S56] also observed that lack of access to and insufficient view on customer environment complicated simulating the production environment. The aforementioned issues caused organizations challenges in providing fully automated provisioning and automated user acceptance test.

Dependencies with hardware and other (legacy) applications: Our analysis has revealed that albeit an application might be production-ready, dependencies between the application with other applications or hardware may be roadblocks to transition from CDE to CD practices (i.e., deploying the application on a continuous basis) [S6, S10, S29, S43, S56, S62]. It means it is needed to ensure that there is no integration problem when deploying an application to production. For example, a study [S10] reported that an increased number of upgrades and new features made the networks highly complex with the potential of becoming incompatible with legacy systems. The authors of [S56] found that dependency with hardware and compatibility with multiple versions as a challenge for steady and automatically deploying software into the customer environment.

Customer preference: Some studies considered the preference of customers and their policies as important factors which should be carefully considered to move towards CD practice. It was revealed that not always customers are pleased with the continuous release due to frequent update notifications, broken plug-in compatibility and increased bugs in software [S6, S29, S43]. Customer organization's policy and process may not allow truly implementing CD, as in an experience report Savor et al. [S57] reported that banks did not allow them to continuously push updates into their infrastructures.

B) Domain Constraints

A software system's domain is a significant factor that should be considered when adopting continuous deployment practice [S4, S5, S6, S9, S10, S24, S31, S41, S44, S48, S56, S57, S60, S65]. A qualitative study by Leppänen et al. [S4] indicated domain constraints could change the frequency of deploying software to customers as well as the adoption of deployment method (e.g., calendar-based deployment). Compared with telecommunication and medical systems, web applications more frequently embrace the frequent deployment. In [S24], it has been reported that despite continuous integration practice was successfully adopted by a case company, it was not possible to fully apply continuous deployment practice on safety-critical systems. We found two studies discussing the challenges of adopting CD in embedded systems [S56] and pervasive systems [S65].

3.4.5 Practices reported to implement continuous practices (RQ3.4)

This section reports the findings from analysis of the data extracted (i.e., D13) to answer RQ3.4, "*What practices have been reported to successfully implement continuous practices?*" Similar to RQ3.3, we first provide a high-level classification of practices to understand which practices can be applied to each CI, CDE, CD and which practices are common for all CI, CDE, and CD. Table 3.10 presents 13 practices and lessons learned reported in the reviewed papers.

3.4.5.1 Common practices for implementing CI, CDE, and CD

A) Improve Team Awareness and Communication

In Section 3.4.2.2, we discussed how approaches and associated tools can increase a project's visibility and transparency for adopting continuous practices. This section reports the analysis of a few papers [S6, S31, S37, S43, S44, S47, S49] that provided practices for increasing team awareness and communication. Robert [S47] observed that appropriately labelling the latest version of client source and keep updating the server version in client-server application enabled developers to understand when everything is working together. To make changes visible for customers, a study [S44] in this category suggested recording the changed features in a change log to enable customers to track what and when features have changed. Marschall [S49] suggested that team members be regularly informed (e.g., by email) about branches that are completely out-dated. We found four papers [S6, S31, S37, S44] that argued knowledge sharing practice should be consolidated among team members as enablers for adopting CI [S31, S37] and improvement for rapid release [S44].

B) Investment

Planning and documentation: It is argued that establishing continuous practices in a given organization necessitate planned and structured steps for clearly defining and documenting all the business goals and development activities [S28, S31, S36]. This is considered helpful to minimize the challenges associated with continuously releasing software features [S28, S31, S36]. Bellomo et al. [S28] observed that weaving requirements and designs through prototyping at the beginning of a release planning cycle enabled the studied team to smooth continuous delivery process. The release level prototyping with quality attributes focus enabled product owner and architect to work closely for quickly responding to prototype feedback. The case organization studied in [S58] developed a standard release path (i.e., a set of rules) for application packaging and deployment for which all the steps and activities to production are determined. This enabled the organization to easily embrace CD and release frequently and with confidence. Adopting CD should be slow with preparing, understanding and documenting engineering processes. For example, one of the case companies studied in [S57] spent 2 years to institutionalize CD practice. Five studies [S6, S11, S17, S37, S43] emphasized the importance of documentation when adopting continuous practices. It has been suggested that continuous activities (build, test, and packaging) should be well documented to help different stakeholders to understand the history of the activities in deployment pipeline. For example, Ståhl and Bosch [S11] proposed a descriptive *Integration Flow Model* for enabling team members to describe and record integration flow implementations in software development companies. The model consists of "input" (e.g., binary repository), "activity" (e.g., packaging) and "external triggering factors (e.g., scheduling)" elements.

Promote team mindset: As discussed earlier, the lack of a positive mindset about continuous practices is a confounding factor in the adoption of these practices. Two papers [S5, S45] reported that organizational management organized CI events, which were run by the team who built the CI infrastructure to spread the positive mindset about CI. In order to encourage new developers to commit code several times per day, Facebook runs a six-week boot camp [S48] to help developers to overcome their fear of code failure. Another paper [S57] argued giving freedom to developers (e.g., full access to the company's code) enabled them to feel empowered to release new code within days of being hired.

Improve team qualification and expertise: Our review has identified the practices that aim at improving team qualification and expertise to bridge the skills gap to successfully implement continuous practices. We found several studies [S5, S6, S45, S48, S57] that provided formal training and coaching (for example through events) arranged by organizations. For instance, OANDA, a company studied in [S57], assigned new developers to the release engineering team for several months in order to get trained and familiar with CD practice. Claps et al. [S6] reported a software provider that leveraged CI developers' experience for the transition from CI to CD by integrating automated continuous deployment of software into the existing CI workflow of developers to ensure there is no or a low learning curve.

C) Clarifying Work Structures

Our analysis identified the practices that emphasize the importance of clarification of the work structures in successfully adopting and implementing continuous practices.

Define new roles and teams: A noticeable practice is defining new roles and responsibilities in software development lifecycle when a project adopts continuous practices [S1, S9, S29, S30, S45, S48, S49, S51]. Krusche and Alperowitz [S1] defined hierarchical roles such as release manager and release coordinator to introduce continuous delivery to multi-customer projects. Another work [S29] indicated that using a dedicated build sheriff role proved successful in practicing CI. The build sheriff engineer not only watches the build machine continuously but also aids developers by identifying and resolving the backouts that previously had to be addressed by developers. Another case [S45] reported the rotational policy implemented to enable team members to take different responsibilities to get a higher understanding of the status of the CI process.

Another study [S57] also reported similar practice as developers were encouraged to rotate between different teams. Hsieh and Chen [S30] advocated having a *single responsible person* on the team to constantly authorize and watch CI system. This helps to prevent ignoring broken builds by developers, particularly those happen during overnight. It was also reported that establishing a temporary or dedicated team to facilitate transitions towards continuous practices was helpful. The experience reported in [S37] highlighted that establishing a virtual Scrum team with expertise in infrastructures and operations was helpful to mitigate potential risks in a software release. Another study [S5] observed the usage of the pilot team who trained other team members and provided guidelines about CI goals to them through workshops and meetings to stimulate CI concepts. Two studies reported the establishment of a dedicated team for design and maintenance of infrastructure and deployment pipeline. This helps organizations in CD transformation [S57] and reduces the release cycle time [S58].

Adopt new rules and policies: Several studies have reported the need of new rules, regulations, policies, and strategies for enabling CI/CD [S26, S39, S45, S46, S48, S50, S58]. For example, one company [S39] enforced developers to solve the errors occurred during their commits in less than 30 minutes or revert the check-in. A paper [S46] reported a set of rules for improving deployability such as: creating tests cases at the system-level should take one day on average. In another paper [S26], the authors argued that having deployable software all the time has been reached by the following rule "*whenever a test complained, the integration of a change set failed, and the software engineer is obliged to update the test code or production code*".

3.4.5.2 Practices for implementing CI

This category presents three types of practices namely *improving testing activity*, *branching strategies* and *decomposing development into smaller units*, to enable and facilitate practicing CI.

A) Improve Testing Activity

Whilst Sections 3.4.2.1 and 3.4.2.3 summarized a set of approaches and tools proposed in the literature for improving test phase during CI, this section discusses three practices for this purpose. Karvonen et al. [S12] indicated that adopting test-driven development (TDD) and daily build practices are essential for CI practice. Neely and Stolt [S17] reported that one of the appropriate practices for removing manual tasks of QA was "***test planning***". This practice stimulates close collaboration between QA and developers to document a comprehensive list of automated tests. They argued that this practice liberates QAs from manually testing the majority of the software applications for regression bugs [S17]. The authors in [S39] suggested another practice called "***cross-team testing***", which means integration test of module *A* should be performed by programmers or testers who have not been involved in the implementation of module *A*. It has been argued that this practice helped detect more defects and build an objective appreciation of the modules. Rogers [S50] argued that the problem of slow unit tests in CI system can be alleviated by separating them from functional and acceptance tests.

B) Branching Strategies

Branching is a well-known CI practice. The practices such as repository use [S30, S44] and short-lived feature branching [S43] were presented as software development practices that support CI. Short-lived branching also supports the adoption of CDE practice as one study [S43] reported that an organization changed the long-lived feature branches to short-lived and small ones for exposing new features faster to the clients to receive feedback faster. Two studies [S29, S48] reported the practice of having developers to commit changes to a local repository and later on those changes would be committed to a central repository. However, in one case [S29], the code that passed all build and automated tests would be committed to the central repository by build sheriffs (i.e., introduced in Section 3.4.5.1.C). In this way, a release process will be more stable. It was also reported that having many branches hampers practicing CI. Feitelson et al. [S48] observed that working on a single stable branch of the code reduces time and effort on merging long-lived branches into trunks.

C) Decompose Development into Smaller Units

A set of the reviewed papers [S5, S10, S30, S36, S45, S47, S48, S49, S50, S51, S57] emphasized that software development process be decomposed into smaller units to successfully practice CI, but none of them provided concrete practice for this purpose. The main goal of this type of practice is to keep build and test time as much small as possible and receive faster feedback. Three papers [S10, S48, S49] argued that large features or changes should be decomposed into smaller and safer ones in order to shorten the build process so that the tests can be run faster and more frequently. For cross-platform applications, the complexity of dependency between components increases dramatically and it can be an obstacle to applying CI to them. Hsieh and Chen proposed a set of patterns namely *Interface Module*, *Platform Independent Module*, and *Native Module* to control dependency between modules of cross-platform applications [S30]. They suggested that the platform-independent code should be placed into *Platform Independent Module* and these modules should be built in the local build environment. Through this pattern not only the build time reduces, but also the build scripts remain simple. Another paper [S5] proposed dead code practice, which can reduce dependency between components before integration through activating and testing a code or component only if all dependencies among them are in place. Decomposing development process into independent tasks enables organizations to have smaller and more independent teams (e.g., cross-functional teams), which was argued as an enabler for fully practicing CI [S50] and CDE [S51, S57].

3.4.5.3 Practices for implementing CDE

A) Flexible and Modular Architecture

As discussed in Section 3.4.4.3.A, the technical dependency between codes or components can act as an obstacle to adopt CDE and CD. The reviewed studies reported that delivering software in days instead of months requires architectures that support CDE adoption [S7, S12, S28, S30, S45, S51, S57]. The software architecture should be designed in a way that software features can be developed and deployed independently. Loosely coupled architecture minimizes the impact of changes as well. For example, Laukkanen et al. [S45] observed that the studied organization had to re-architect their product (e.g., removing components caused trouble) to better adopt CI and CDE. It is also asserted that teams that are not architecturally dependent on (many) other, they would be more successful in implementing CDE and CD [S57]. The work reported in [S7] has conducted an empirical study on three projects that had adopted CI and CDE. The study concluded that most of the decisions (e.g., removing web services and collapsing the middle tier) made to achieve the desired state of deployment (i.e., deployability quality attribute) were architectural ones. The collected deployability goals and tactics from three projects have been used as building blocks for the deployability tactics tree. Two studies [S5, S30] recommend that the component interfaces be clearly defined for making continuous delivery- or deployment-ready architectures.

B) Engage all people in the deployment process

A set of papers [S6, S9, S43, S44, S48, S57, S58] argued that achieving the real benefits of continuous delivery and deployment practices requires developers and testers being more responsible for their codes in the production environment. With this new responsibility, they are involved in and aware of all the steps (e.g., deploy into production), and are forced to fix problems that appear after deployment [S44]. As an example of involving developers in the release process, Facebook adopted a policy, in which all engineers team who committed code should be on call during the release period [S48].

Table 3.10 A classification of practices and lessons learned for implementing CI, CDE, and CD

		Practices	Key Points and Included Papers	#
Common Practices for Implementing CI, CDE, CD	Team Awareness and Communication	**PR1**. Improve Team Awareness and Communication	▪ Listing the changed features in changelog entries [S43] ▪ Labelling the latest version and new features [S6, S47] ▪ Informing team members about branches that are completely outdated [S49] ▪ Improved knowledge sharing between technical and management staffs on different levels [S6, S31, S37, S44]	7
	Investment	**PR2**. Planning and documentation	▪ A planned path for adopting continuous practices [S28, S31, S36, S57, S58] ▪ Document builds, tests and other activities in integration processes [S6, S11, S17, S37, S43] ▪ Integration Flow Model [S11]	10
		PR3. Promote team mindset	▪ Organizing events about continuous practices to spread mindset and train team members [S5, S6, S45, S48] ▪ Giving much freedom to developers [S57] ▪ Empowering culture [S6, S57]	5
		PR4. Improve team qualification and expertise	▪ Formal training and coaching team members [S5, S6, S45, S48, S57]	5
	Clarifying Work Structures	**PR5**. Define new roles and teams	[S1, S9, S29, S30, S37, S45, S48, S49, S51, S57, S58] ▪ Establishing a dedicated team to develop and maintain the deployment pipeline [S57, S58] ▪ Sheriff engineer [S29] ▪ Piloting team [S5] ▪ Virtual Scrum team [S37]	11
		PR6. Adopt new rules and policies	[S26, S39, S45, S46, S48, S50, S58] ▪ All developers should be on call when releasing software [S58].	7
Practices for Implementing CI	Improve Testing Activity	**PR7**. Improve Testing Activity	▪ Practicing test-driven development [S12, S50] ▪ Test Planning practice [S17] ▪ Cross-team testing practice [S39] ▪ Designing decoupled tests by separating unit tests from functional and acceptance tests [S50]	4
	Branching Strategies	**PR8**. Branching Strategies	▪ Using integration or local repository [S29, S48] ▪ Short-lived feature branching [S43] ▪ The practice of repository use [S30, S44] ▪ Not too many branches [S48]	5

	Decompose Development into Smaller Units	**PR9.** Decompose Development into Smaller Units	[S5, S10, S30, S36, S45, S47, S48, S49, S50, S51, S57] ▪ Dead code practice [S5] ▪ Breaking down large features and changes into smaller and safer ones [S10, S48, S49] ▪ Small and independent teams [S50, S51, S57]	11
Practices for Implementing CDE	Flexible and Modular Architecture	**PR10.** Flexible and Modular Architecture	[S5, S7, S12, S28, S30, S45, S51, S57] ▪ Deployability concern in mind when designing software systems [S7] ▪ Defining component interface clearly [S5, S30]	8
	Engage all people in Deployment	**PR11**. Engage all people in deployment process	▪ Developers and testers should take more responsibility for their code [S6, S9, S43, S44, S48, S57, S58] ▪ On call developers [S48]	7
Practices for Implementing CD	Partial Release	**PR12**. Partial Release	▪ Zero release (Empty release) [S1] ▪ Hiding and disabling new or problematic functionalities to users [S6, S17, S44, S48] ▪ Deploying software to a small set of users [S17, S44, S57] ▪ Rolling back quickly to stable state [S48] ▪ Independent releases [S58]	7
	Customer Involvement	**PR13**. Customer Involvement	[S10, S12, S28, S36, S43, S44, S49, S61, S63] ▪ Lead customer [S10, S12] ▪ Pilot customer [S43] ▪ Involving customers in testing phase [S61, S63] ▪ Triage meeting [S36]	9

3.4.5.4 Practices for implementing CD

A) Partial Release

Releasing software to customers potentially may be risky for software providers as their customers may receive buggy software. This issue can intensify when deploying software on a continuous basis (i.e., practicing CD). It is critical for software organizations to adopt practices in order to reduce potential risks and issues in release time. We identified three types of practices for this purpose: (i) deploying software to a small set of users [S44, S17, S57]; (ii) hiding and disabling new or problematic functionalities to users [S6, S17, S44, S48]; (iii) rolling back quickly to a stable state [S48]. Three papers [S17, S44, S48] pointed out *dark* and *canary* deployment methods that can significantly help transit to continuous deployment. In canary deployment method, the new versions of software are incrementally deployed to a production environment with only a small set of users affected [104]. Deploying software by this method enables the team to understand how new code (i.e., the canary) works compared to the old code (i.e., the baseline). In [S57], it was found that both Facebook and OANDA released software products to a small subset of users rather than releasing them to all customers. For example, Facebook first releases the software products to its own employees to get feedback to improve the test coverage. Another incremental release method, dark deployment, hides the functional aspects of new versions to end-users [105]. This method tries to detect potential problems, which may be caused by new versions of software before end-users would be affected. In order to deal with the large features (i.e., dark features) in OnDemand software product that may not be developed and deployed in a small cycle, one organization [S6] employed the practice of small batches. Through this practice, the development process of dark features was hidden from customers. However, when the entire feature is finally developed, the switch of dark feature will be turned on and then customer is able to interact with and use them. Another study [S58] reported the implementation of microservices that were independently released while maintaining backward compatibility with each release as a tactic of addressing delays in the deployment pipeline. In order to introduce CD practice to novice developers, Krusche and Alperowitz [S1] suggested "***empty release***" practice, in which besides development teams get in touch with continuous workflows and infrastructures from day 0, the continuous pipeline is initially run with a simple application (e.g., "hello world").

B) Customer Involvement

Several papers [S10, S12, S28, S36, S43, S44, S49, S61, S63] aimed at exploring the role of customers or end-users as an enabler in the transition towards continuous deployment. A couple of papers [S10, S12] defined the concept of "***lead customer***", at which customers not only are incorporated in software development process, but also are eager to explore the concept of continuous deployment. The work reported in [S43] used the term "***pilot customer***" and argued that it would be better to apply CDE or CD to those companies that are willing to continuously receive updates. It has been noted that it is needed to renew existing engagement model with customers to be compatible with the spirit of CD. Agarwal [S36] described a process model based on Type C SCRUM, called Continuous SCRUM, and leveraged a number of best practices to augment this process model and achieve sustainable weekly release. One of the noticeable practices was "***triage meeting***", in which product-owner runs the meeting and she/he determines the triage committee. A product-owner review has been introduced into the sprint to enable and approve changes to product requirements as well as the product-owner was enabled to prioritize the back-log of product requirements. We found a set of papers [S61, S63] arguing the involvement of customer in testing was an effective practice for adopting CDE and CD practices. A study [S61] revealed that involving customers in the testing phase is a helpful practice for those companies that do not have enough resources for practicing CD. The study indicated that customers can be greatly successful in finding lower impact functional defects.

3.5 Discussion

In this section, we discuss the findings and reflect upon the potential areas for further research.

3.5.1 A Mapping of Challenges to Practices

Figure 3.4 presents a mapping of the identified challenges in Section 3.4.4 onto the practices reported in Section 3.4.5. This mapping is intended to provide a reader (i.e., researcher or practitioner) to quickly determine which challenges are related to which practices. For example, a *flexible and modular architecture* is expected to decrease *dependencies in design and code*. Figure 3.4 also indicates that that there might be dependencies among the challenges (i.e., *exacerbation*) or practices (i.e., *support*). A practice may support or positively affect another practice, for example, by making the implementation of that practice easier. For example, we found that *distributed organization* can exacerbate the challenge of and need for *coordination and collaboration* in adopting continuous practices; however, adopting and implementing *partial release* can be greatly supported by *engaging all people (in particular customer) in deployment process*.

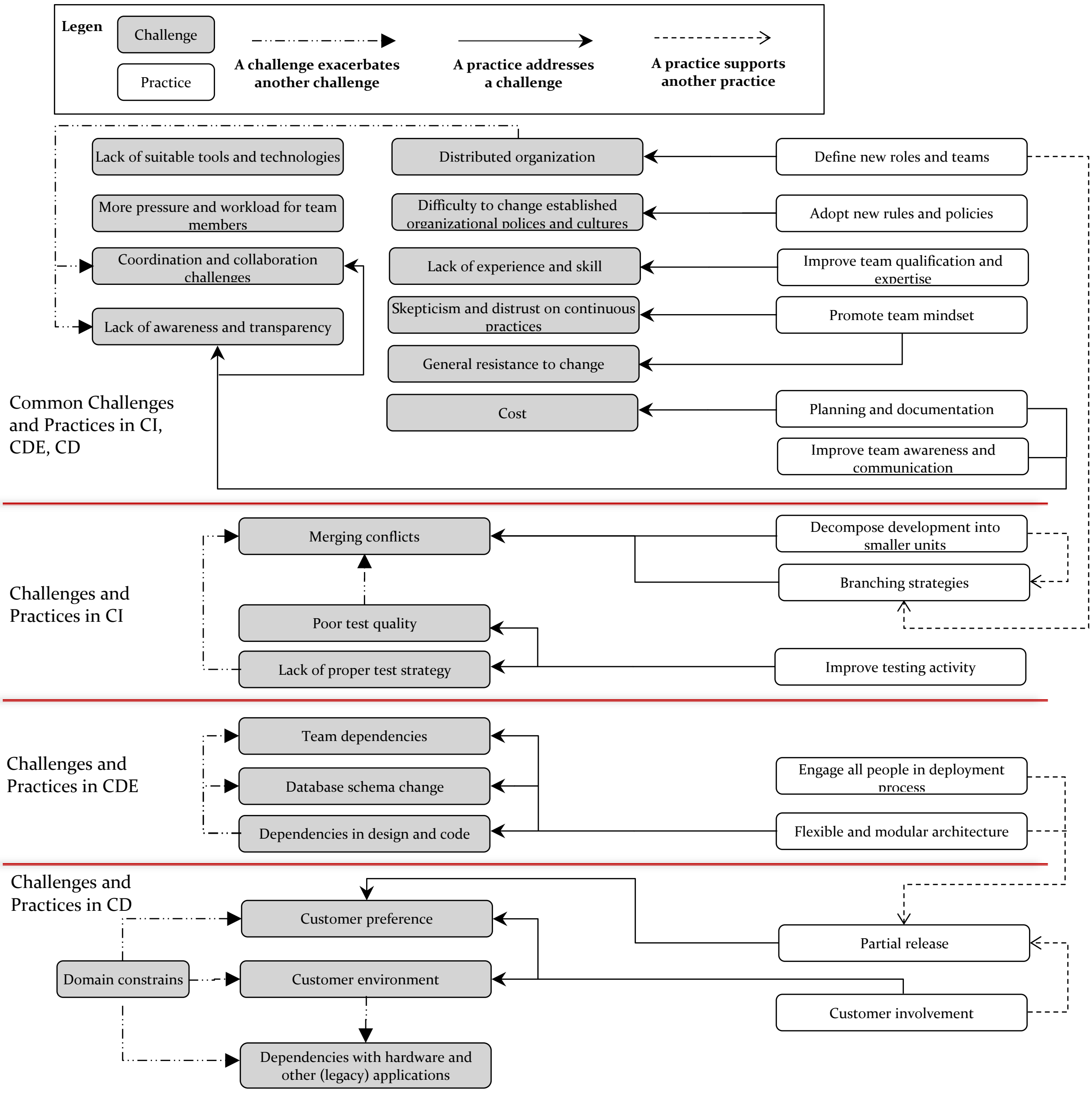

Legen
Challenge
Practice
A challenge exacerbates another challenge
A practice addresses a challenge
A practice supports another practice
Lack of suitable tools and technologies
More pressure and workload for team members
Coordination and collaboration challenges
Lack of awareness and transparency
Distributed organization
Difficulty to change established organizational polices and cultures
Lack of experience and skill
Skepticism and distrust on continuous practices
General resistance to change
Cost
Define new roles and teams
Adopt new rules and policies
Improve team qualification and expertise
Promote team mindset
Planning and documentation
Improve team awareness and communication
Common Challenges and Practices in CI, CDE, CD
Merging conflicts
Poor test quality
Lack of proper test strategy
Decompose development into smaller units
Branching strategies
Improve testing activity
Challenges and Practices in CI
Team dependencies
Database schema change
Dependencies in design and code
Engage all people in deployment process
Flexible and modular architecture
Challenges and Practices in CDE
Challenges and Practices in CD
Customer preference
Domain constrains
Customer environment
Dependencies with hardware and other (legacy) applications
Partial release
Customer involvement

Figure 3.4 An overview of challenges and practices of adopting CI, CDE, CD, and their relationship

3.5.2 Critical factors for continuous practices success

Based on our analysis in Sections 3.4.4 and 3.4.5, we have identified 20 challenges and 13 practices for CI, CDE, and CD. We have also found 30 approaches and associated tools that have been proposed by the reviewed studies to address particular challenges in each continuous practice. It is important to point out that there was no one-to-one relationship between the identified challenges and the proposed practices, approaches and associated tools as there were some challenges for which we were unable to identify any practice or approaches to address them and vice versa. We decided to define a set of critical factors that should be carefully considered to make continuous practices successful. To identify what factors (i.e., both in software development and customer organizations) are important to successfully adopt and implement continuous practices, we again analyzed the results reported in Sections 3.4.2, 3.4.4, and 3.4.5. A factor is accumulated challenges, approaches, and practices pertaining to a fact. For example, we found a number of challenges (Section 3.4.4.2.A), approaches and associated tools (Sections 3.4.2.1 and 3.4.2.3), and practices (Section 3.4.5.2.A) for testing activity in moving towards continuous practices. Therefore, we considered "testing" as a factor, which should be carefully considered when adopting continuous practices. If a factor is cited in at least 20% of the reviewed studies, then we regard that factor as a critical factor for making continuous practices successful.

Table 3.11 shows the list of 7 critical factors, which may impact the success of continuous practices. "Testing" (27 papers, 39.1%) is the most frequently mentioned factor for continuous practices success, followed by "team awareness and transparency" (24 papers, 34.7%), "good design principles" (21 papers, 30.4%) and "customer" (17 papers, 24.6%). Our results indicate that "testing" plays an important role in successfully establishing continuous practices in a given organization. Our research reveals that long-running tests, manual tests, and high frequency of test cases failure have failed most of the case organizations in the reviewed studies to realise and achieve the anticipated benefits of continuous practices. Whilst we have reviewed several papers that revealed a lack of test automation was a roadblock to move toward continuous practices, there were only a few papers (i.e., 7 papers), which had developed and proposed approaches, tools and practices for automating tests for this purpose.

Continuous practices promise to significantly reduce integration and deployment problems. It should be designed in a way that the status of a project, number of errors, who broke the build, and the time when features are finished are visible and transparent to all team members. We have found "team awareness and transparency" as the second-most critical factor for adopting continuous practices. Improved team awareness and transparency across the entire software development enables team members to timely find potential conflicts before delivering software to customers and also improves collaboration among all teams [106].

Our review has identified 17 papers that report challenges, practices, and lessons learned regarding customers, which enabled us to consider "customer" as a critical factor for successful implementation of continuous practices. It is worth mentioning that this factor mostly impacts on CD. We found that not always customer organizations are happy with the continuous release. That is why we need to investigate the level of customer satisfaction when moving to CD practice: unavailability of customer environments, extra computing resources required from customers, incompatibility of new release with existing components and systems, and increased chance of receiving buggy software all together can demotivate customers about advantages of continuous deployment. Our results also indicate that "highly skilled and motivated team" (15 out of 69, 21.7%) is a critical factor to drive software organizations towards continuous practices. We argue that releasing software continuously and automatically can be achieved with the solid foundation of technical and soft skills, shared responsibilities among team members, and having motivated teams to continuously learn new tools and technologies.

Whilst this SLR reveals that continuous practices have been applied successfully to both maintenance and greenfield projects, we argue that "application domain" can play a significant role in the transition towards continuous practices, in particular, continuous deployment. As discussed earlier, continuous delivery can be applied to all types of applications and organizations. However, practicing CD in some application domains (e.g., embedded systems domain) is associated with unique challenges, in which they make almost impossible to truly practice CD or affect the frequency of releases to customer environments. We emphasize that application domains and limitations of customers should be carefully studied before adopting continuous deployment. Our SLR reveals that one of the leading causes of failure in fully implementing continuous practices is missing or poor infrastructures. By "appropriate infrastructure", we mean all software development tools, infrastructures, networks, technologies and physical resources (e.g., build server and test automation servers) employed by an organization to do continuous practices well. This is mainly because implementing each continuous practice, in particular, continuous delivery and deployment in a given organization requires extra computing resources and also tools and technologies to automate end-to-end software development (e.g., testing) and release process as much as possible. This consequently would affect the organizational budget. We assert that one of the core components of an appropriate infrastructure, which considerably enables automation support and impacts the success of continuous practices, is the deployment pipeline. We will concretely discuss the engineering process of the deployment pipeline in Section 3.5.5.

Table 3.11 List of critical factors for continuous practices success

ID	Factor	#	%	Studies
F1	Testing (effort and time)	**27**	**39.1**	S3, S4, S5, S6, S12, S17, S19, S23, S25, S32, S34, S36, S38, S39, S40, S41, S43, S45, S50, S52, S53, S54, S55, S56, S62, S64, S67
F2	Team awareness and transparency	**24**	**34.7**	S1, S2, S4, S6, S10, S13, S22, S24, S31, S33, S37, S38, S41, S43, S44, S45, S47, S49, S50, S52, S56, S62, S64, S67
F3	Good design principles	**21**	**30.4**	S4, S5, S6, S7, S12, S10, S28, S30, S31, S36, S41, S45, S47, S48, S49, S50, S51, S57, S58, S60, S62
F4	Customer	**17**	**24.6**	S4, S6, S10, S12, S28, S29, S36, S43, S44, S49, S55, S56, S57, S60, S61, S62, S63
F5	Highly skilled and motivated team	**15**	**21.7**	S4, S5, S6, S9, S12, S43, S44, S45, S48, S49, S57, S56, S57, S58, S62
F6	Application domain	**14**	**20.2**	S4, S5, S6, S9, S10, S24, S31, S41, S44, S48, S56, S57, S60, S65
F7	Appropriate infrastructure	**14**	**20.2**	S5, S6, S8, S10, S27, S43, S47, S49, S56, S57, S59, S60, S66, S68

3.5.3 Contextual factor

The importance of contextual attributes and what should be reported as contextual attributes have been discussed in the software engineering literature [107-109]. It has been argued that software development approaches, tools, challenges, lessons learned and best practices need to be explored and understood along with their respective contexts [108, 110]. Particularly, we tried to understand in which methodological and organizational contextual settings (i.e., research type, project type, application domain, organization size, and domain) the proposed approaches, tools, best practices and challenges have been reported. According to the results reported in Section 3.4.1.2, the reviewed studies were evaluation research (25 papers, 36.2%), followed by validation research (24 papers, 34.7%) and experience report (15 papers, 21.7%). Since all of the experience papers were based on practitioners' experiences, the combination of both evaluation and experience papers means that 57.9% of the reviewed papers came from industry setting. The high percentage of the papers with industrial level evidence improves the practical applicability of the reported results and encourages practitioners to adopt and employ the proposed approaches, tools, practices and consider the challenges when adopting each continuous practice.

As reported in Section 3.4.1.4, a considerable number of the reviewed papers did not provide the information on application domain and type, resulting in these papers being categorized as "unclear". There was a general lack of information about the organizational contexts (i.e., size and domain) in the reviewed papers. We were forced to drop them for data analysis and interpretation. We strongly suggest that more attention should be paid to reporting the contextual information about the reported studies. The contextual information is likely to improve the quality and credibility of the reported approaches, tools and practices in continuous integration, delivery, and deployment. Such information can also help a reader to better understand the reported research.

3.5.4 Architecting for deployability

The results of this SLR indicate that sound architecture design (i.e., "good design principles" factor) has a significant influence on the success of practicing CI, CDE, and CD. Several of the reviewed papers have discussed modular architecture, loosely coupled components, and clearly defined interfaces as contributing factors for adopting and implementing continuous practices, in particular, CDE and CD. Based on Section 3.4.4.4.A, the importance of this issue increases sharply in heterogeneous environments as they can hinder continuous software deployment. We argue that one of the most pressing challenges of adopting and implementing continuous practices is how software applications should be (re-) architected to develop, integrate, test and deploy independently in multiple environments. Therefore, the architecting phase should be considered as one of the most important phases for appropriately adopting and implementing continuous practices [38]. Deployability as an emerging quality attribute has a high priority for continuous delivery and deployment [9, 37, 111]. By deployability, we mean *"how reliably and easily an application/component/service can be deployed to (heterogeneous) production environment"* [111].

Architecting with testability and deployability in mind during the design time has been featured in many white papers and practitioners' blogs [9, 37] as a noticeable practice for CDE and CD, but we could find only one paper [S7] that has explicitly considered the deployability scenarios for upfront design decisions and concluded that most of the decisions made for deployment-related issues were architectural one. We assert that there is an important need of research to gain a deep understanding of how continuous delivery or deployment adoption can influence the architecting process and their outcomes in an organization. We argue that this research area (i.e., architecting for deployability) should be more investigated in the future. This motivates the following questions: How can we evaluate and measure the deployability of a designed architecture at the early stage of development time? What are quality attributes in support of or in conflict with deployability? Which architectural patterns, tactics, and styles are more-friendly for deployability?

3.5.5 Engineering deployment pipeline

In Section 3.4.3, we have discussed that deployment pipeline is a key enabler for enterprises to successfully adopt continuous practices. Our review has revealed that despite a significant number of the reviewed papers conducted in industrial settings and reported by practitioners, many papers lacked sufficient details about how enterprises design and implement deployment pipelines and what challenges they might experience. In fact, only 36.2% of the included studies presented the tools, which have been employed to implement deployment pipelines. This investigation was interesting because there is no standard or single pipeline [78] and modelling and implementing a deployment pipeline in a given enterprise may be influenced by a number of factors such as team skills, experience and structure, organization's structure and budget, customer environments, and project domain [7]. Therefore, software development organizations need to allocate time and resources to appropriately select and integrate a wide variety of open source and commercial tools to form a deployment pipeline tailored to them. The evidence of this growing need is the emergent of consulting companies such as Sourced Group[18] and Xebia[19] that are assisting enterprises in designing and implementing deployment pipeline.

In the meanwhile, with the increasing size and complexity of software-intensive systems, the number of builds and test cases increase dramatically. Whilst infrastructures with high-performance computing resources and selecting appropriate tools are mandatory for implementing continuous practices and deployment pipeline, this is not sufficient to deal with such tremendous growth rate. Therefore, it is needed to develop innovative approaches and tools, which not only enable team members to receive build and test results correctly and timely but also they should be aligned and integrated with the deployment pipeline. In Section 3.4.2, thirty approaches and associated tools have been reported to support and facilitate continuous practices. Most of them (24 out of 30) only target CI practice; 18 out of 30 are stand-alone tools that have not been integrated and evaluated in a deployment pipeline. Another increasing concern in the deployment pipeline is how to secure a deployment pipeline [112]. According to [112], the main concern raised during RELENG[20] workshop in 2014 was "what happens if someone subverts the deployment pipeline". All stages and tools involved in the deployment pipeline as well as integrating application to other infrastructures can potentially be compromised by attackers. Two papers [S27, S66] have investigated the security issue in deployment pipelines. We conclude that there is a paucity of research aimed at systematically studying engineering process of deployment pipelines. We assert software engineering researchers and practitioners need to pay more attention to systematically architect deployment pipelines and rigorously selecting appropriate tools for the pipelines.

3.6 Threats to validity

Whilst we strictly followed the guidelines provided by [45], we had similar validity threats like other SLRs in software engineering. The findings of this SLR may have been affected by the following threats:

Search strategy: One of the threats that may occur in any SLR is the possibility of missing or excluding the relevant papers. To mitigate this threat, as discussed in Section 3.3.2.3, we used six popular digital libraries to retrieve the relevant papers. We argue that using Scopus as the largest indexing system which provides the most comprehensive search engine among other digital

[18] http://www.sourcedgroup.com/
[19] https://xebia.com/
[20] http://releng.polymtl.ca/RELENG2014/html/

libraries [110], enabled us to increase the coverage of the relevant studies. Additionally, we employed three strategies to mitigate any potential threat in the search strategy: i) search string was improved iteratively based on the pilot search and were tested carefully before executing for searching the relevant papers for this review; 2) we consulted the search strings used in the existing SLRs [83, 84] for building our search string; 3) a snowballing technique (i.e., manual search on references of the selected papers) was employed in the second round of the papers searching process (See Figure 3.1) to identify as many related papers as possible.

Study selection: This step can be influenced by researchers' subjective judgement about whether or not a paper meets the selection criteria for inclusion or exclusion. The potential biases in the study selection have been addressed by strictly following the pre-defined review protocol, recording the inclusion and exclusion reasons for on-going internal discussions about the papers that raised doubts about their inclusion or exclusion decisions. At the first step, the inclusion and exclusion criteria have been validated on a small subset of primary studies. Any disagreements during study selection were resolved through the internal discussions. Furthermore, a cross-check using a random number of the selected papers was conducted.

Data extraction: Researchers' bias in data extraction can be a basic threat in any SLR, which may negatively affect the results of SLRs. We implemented the following steps to address this threat. First, we created a data extraction form (See Table 3.4) to consistently extract and analyze the data for answering the research questions of this SLR. Second, since a large part of the data extraction step was conducted by one person (i.e., the present author); in the case of any doubt, continuous discussions were organized with other researchers for correcting any disparities in the extracted data. Third, a subset of the extracted data was verified by two other researchers.

Data synthesis: As we argued in Section 3.3.5.2, we applied quantitative and qualitative methods to analyze the extracted data. It should be noted that sometimes there were some difficulties in interpreting the extracted data due to lack of sufficient information about the data items. We had to subjectively interpret and analyze the data items, which might have had an effect on the data extraction outcomes. To reduce the researchers' bias in interpretation of the results, besides reading the given study, where possible we also referred the approach's and tool's website and any training movie (e.g., RQ3.1 and RQ3.2) to get more reliable information. It should be noted that for other data items, we did not have any interpretation unless the data items have been explicitly provided by the study (e.g., application domain).

3.7 Conclusions and Implications

This chapter has presented a Systematic Literature Review (SLR) of approaches, tools, challenges, and practices identified in empirical studies on continuous practices in order to provide an evidential body of knowledge about the state of the art of continuous practices and the potential areas of research. We selected 69 papers from 2004 to 1st June 2016 for data extraction, analysis, and synthesis based on pre-defined inclusion and exclusion criteria. A rigorous analysis and systematic synthesis of the data extracted from the 69 papers have enabled us to conclude:

(1) The research on continuous practices, in particular continuous delivery and deployment are gaining increasing interest and attention from software engineering researchers and practitioners according to the steady upward trend in the number of papers on continuous practices in the last decade (See Figure 3.2). More than half of the reviewed papers (39 papers, 56.5%) have been published in the last three years.

(2) With respect to the research type, most of the selected papers were evaluation (25 out of 69, 36.2%) and validation (24 out of 69, 34.7%) research papers. While 21.7% of the selected papers

were “experience papers”, a small number of papers were solution proposal (7.2%). A large majority of the papers were conducted in industrial (i.e., 64 out of 69, 92.7%) rather than academic (i.e., 5 papers) settings. With respect to the data analysis approach, the same number of the selected papers used quantitative and qualitative research approaches (i.e., 37.6% for each), while this statistic was 20.2% for mixed approaches.

(3) The approaches, tools, challenges, and practices reported for adopting and implementing continuous practices have been applied to a wide range of application domains, and among which “software/web development framework” and “utility software” have received the most attention. This SLR also revealed that continuous practices can be successfully applied to both greenfield and maintenance projects.

(4) Thirty approaches and associated tools have been identified by this SLR, which facilitate the implementation of continuous practices in the following ways (i.e., not mutually exclusive): *reducing build and test time in CI* (9 approaches), *increasing visibility and awareness on build and test results in CI* (10 approaches), *supporting (semi-) automated continuous testing* (7 approaches), *detecting violations, flaws and faults in CI* (11 approaches), *addressing security, scalability issues in deployment pipeline* (3 approaches), and *improve dependability and reliability of deployment process* (3 approaches).

(5) We observed that only 36.2% of the selected papers reported what and how tools and technologies were selected and integrated to implement deployment pipeline (i.e., modern release pipeline). *Subversion* and *Git/GitHub* as version control systems and *Jenkins* as integration server were the most popular tools used in deployment pipelines.

(6) The identified approaches (See Section 3.4.2), challenges (See Section 3.4.4) and practices (See Section 3.4.5) of CI, CDE, and CD have enabled us to find seven critical factors that impact the success of continuous practices, in an order of importance: “*testing (effort and time)*”, “*team awareness and transparency*”, “*good design principles*”, “*customer*”, “*highly skilled and motivated team*”, “*application domain*”, and “*appropriate infrastructure*”.

(7) Implications for researchers: (i) this SLR has revealed the scarcity of reporting contextual information (e.g., organization size and domain) in the selected papers. To improve the quality and credibility of the results, researchers ought to report detailed contextual information. (ii) In this review, we found only two papers that investigated the security issue in deployment pipelines. Given the increased importance of security in deployment pipelines, there is a need for further research to explore how deployment pipelines should be designed and implemented to mitigate security issues. (iii) Out of 30 approaches and associated tools reported in this SLR, only 12 approaches and tools were integrated and evaluated in the deployment pipeline. We encourage researchers to evaluate their proposed approaches and tools with real deployment pipelines. (v) As discussed in Section 3.5.4, architecture design and deployability quality attribute are very important factors in successfully adopting and implementing continuous practices, however, there is a lack of guidance of architecting for deployability. We suggest that researchers in cooperation with practitioners come up with frameworks, processes, and tools to support deployability quality attribute at design time.

(8) Implications for practitioners: (i) a very high percentage of the reviewed papers provide industrial level evidence (i.e., evaluation and practitioners’ experience papers as presented in Section 3.4.1.2). This improves the practical applicability of the reported results. Such findings are expected to encourage software engineering practitioners to adopt and employ appropriate approaches, tools, practices and consider the reported challenges in their daily work based on the suitability for different contexts. (ii) The identified approaches, tools, challenges, and

practices have been classified in a way that practitioners are enabled to understand what challenges are for adopting each continuous practice, what approaches, and practices exist for supporting and facilitating each continuous practice. We found a number of challenges and practices that were common in the transition towards all CI, CDE, and CD. (iii) The identified critical factors can make practitioners aware of the factors that may affect the success of continuous practices in their organizations. For example, whilst it is important for practitioners to know that a lack of team awareness and transparency may fail them to realize and achieve the real anticipated benefits of continuous practices, this SLR has identified several approaches, associated tools, and practical solutions to improve and sustain team awareness and transparency in continuous practices.

Chapter 4

Moving from Continuous Delivery to Continuous Deployment

Related publication:

This chapter is based on ESEM paper, "Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges" [3].

In Chapter 3, we observed many researchers have started investing significant efforts in studying different aspects of Continuous DElivery (CDE) and Continuous Deployment (CD). However, many studies refer to CDE (i.e., where an application is potentially capable of being deployed) and CD (i.e., where an application is automatically deployed to production on every update) as synonyms and do not distinguish them from each other. Despite CDE being successfully adopted by a large number of organizations, it is not empirically known why organizations still are unable or demotivated to have *automatic* and *continuous* deployment (i.e., CD practice). This chapter aims at empirically investigating and classifying the factors that may impact on adopting and implementing CD practice. Through a mixed-methods empirical study consisting of interviewing 21 software practitioners, followed by a survey with 98 respondents, we found 11 confounding factors that limit or demotivate software organizations to push changes automatically and continuously to production. Our findings highlight several areas for future research and provide suggestions for practitioners to streamline the deployment process.

4.1 Introduction

> "*We do continuous delivery even to on-premise environments. Using continuous delivery, we would not be pushing out every day. We might only push out new release to the client site every three months. We are still using continuous delivery to keep software deployable. It is quite important to distinguish continuous delivery and continuous deployment*" ***P18-Consultant***.
>
> "*You can apply continuous delivery and not implement continuous deployment yet (e.g., because the customer has security constraints to deploy remotely)*" ***R97-Team Lead***.

Whilst several studies have investigated the challenges and issues for adopting and implementing Continuous DElivery (CDE) [14, 24, 113] and Continuous Deployment (CD) practices [11, 23, 114, 115], they usually use CDE and CD as synonyms [33, 116]. However, there is the other side of the coin. A recently published literature review on CDE [33] reveals that existing literature uses the term CD while they actually refer to CDE because they do not include or provide *fully automatic* deployment to production. Furthermore, our SLR in the previous chapter [1] and the review done by Laukkanen et al. [33] could not find highly relevant scientific literature on CD implementation. Recently industrial communities [117-119] have underlined the challenges, experiences, and lessons learned in moving from CDE (i.e., where an application is *potentially* capable of being deployed) to CD (i.e., where an application is *automatically* deployed to production on *every update*). In addition, two studies [27, 120] reveal that delivery (i.e., CDE) and deployment (i.e., CD) capabilities of software organizations may be different as, for example, there might be a tension between software quality and deployment frequency. In [27], a new line of research is explicitly sought to explore the reasons for this difference.

To the best of our knowledge, none of the previously published literature reports this issue and specifically distinguishes between the challenges of adopting CDE and CD. This chapter aims at empirically investigating and classifying confounding factors that particularly impact on adopting and implementing CD practice: despite software potentially is production-ready (i.e., CDE), there are still factors that limit or demotivate organizations to continuously and automatically ship code changes from development into production without human intervention (i.e., CD). To help to close this literature gap, we investigate the following research question:

RQ4.1 What factors do limit or demotivate software organizations to move from continuous delivery to continuous deployment?

To answer this research question, we leveraged the data collected through the mixed-methods empirical study discussed in Chapter 2. Our analysis reveals 11 factors that are confounders to truly adopt CD practice, including: "lack of fully automated (user) acceptance test", "manual quality check", "deployment as business decision", "insufficient level of automated test coverage", "highly bureaucratic deployment process", "lack of efficient rollback mechanism", "dependency at application level", "demotivated customer", "customer environment", "domain constraints", and "manual interpretation of test results".

Chapter organization: Section 4.2 summarizes related work. We describe the research method in Section 4.3. Section 4.4 reports our findings. Section 4.5 reflects a discussion on findings. Finally, the chapter is concluded in Section 4.6.

4.2 Related Work

This section places our work in this chapter in the context of other related studies. Lwakatare et al. [115] present high-level challenges for the adoption of DevOps in embedded systems domain. The identified challenges are the huge dependency between hardware and software versions, lack of access to customer environments, and lack of appropriate technologies to automatically and continuously deploy software to customer environments. Whilst [14, 113] present the obstacles and challenges of adopting CDE practice, adopting CD practice has been evaluated by [11, 23, 114]. Claps et al. [23] report the challenges that a single case software company faced in transition to CD. The identified challenges are classified into technical and social ones including team experience, continuous integration, partner plugins, and changing database schemas. The study also reveals what solutions (e.g., adopting social rules and investment in infrastructures) the case company employed to address those challenges.

Savor et al. [22] investigate the effect of adopting CD practice on team productivity (i.e., number of added or modified code lines pushed to production per developer) and product quality (i.e., number of failures in production) at Facebook and OANDA. The study discusses a number of challenges including management support and extra effort for understanding updates that an organization may face in the path of CD adoption. The challenges identified for adopting CD [14, 113] are almost similar to those that are found for CDE [11, 23, 114]. For example, most of the studies indicate that manual testing, unsuitable architecture, and resistance to change are roadblocks to practicing CDE and CD. In addition to the challenges reported in [11, 14, 23, 114], Laukkanen et al. [24] show that the stage-gate process negatively impacts on CDE success. This is mainly because the attributes (i.e., tight schedule) of the stage-gate process significantly limit the time needed for CDE adoption. The study argues that it is almost impossible to adopt CDE in a stage-gate managed organization without changing the process.

Our findings in this chapter have significant differences to the existing studies [11, 14, 23, 24, 113-115]: (1) our findings come from interviewing 21 experts in CDE and CD practices and a survey of 98 software professionals from a wide range of organizations in terms of size, domain, and the way of working rather than one practitioner's own observations [14, 113] or a single case company [23, 24] and a particular domain [115]. The study [115] focuses only on adopting DevOps in embedded systems and the studies [14, 113] only identify the challenges and issues of adopting CDE and CD based on the experience of authors. (2) In this chapter, we discuss the current state of automation support to adopt CD, which has not been reported in the previous work. (3) Most of the previous studies *did* consider CDE and CD as one practice and did not distinguish the challenges associated with adopting CDE and CD. (4) To the best of our knowledge, our study reports the first (large scale) piece of work, which distinguishes CDE from CD and empirically investigates the confounding factors that limit or demotivate organizations from moving towards CD from CDE.

4.3 Research Method

This chapter uses the mixed-methods research approach, which was described in Chapter 2. The results of this chapter are based on 21 in-depth, semi-structured interviews and surveying 98 software professionals.

4.3.1 Interviews

The relevant parts of the interviews for this study are described as follows: the first part briefed the high-level objectives of the study to the interviewees. In the second part, we precisely defined CDE and CD terms for the interviewees and explained what differences exist between them. Next, questions related to the participant's background were asked (e.g., *what is your role and responsibilities in the project team?*). Fourth, the participants explained challenges, their personal experiences and concerns around moving from CDE to CD, and why they were still unable or demotivated to have a fully *automatic* deployment to production. In the last part, questions related to deployment pipeline and automation were asked (e.g., *is your deployment pipeline fully automated or not? why?*). We finished the interviews by asking the interviewees "*Is there any comment or issue you want us to know?*"

4.3.2 Survey

We precisely defined CDE and CD practices at the beginning of the survey instrument. The relevant survey questions used for this study were composed of 4 demographic, 3 five-point Likert-scale, 2 single-choice, 2 multiple-choice and 4 open-ended questions.

4.4 Findings

In this section, we first present our findings regarding differences in practicing CDE and CD in the participants' organizations. Next, as automation is a key component of CD, we describe the current state of automation support in this regard. Then, we report confounding factors in moving from CDE to CD.

4.4.1 Practicing CDE vs. CD

We aimed at understanding the differences between practicing CDE and CD and also assessing the maturity of CDE and CD practices in our participants' organizations. To this end [9, 11, 27], both the interviewees and the survey participants were asked two questions: (1) *on average, how often your applications are in releasable state or production-ready?* This question, to a large extent, indicates how an organization adopts and implements CDE practice. To measure CD adoption in an organization, we asked (2) *on average, how often do you deploy your applications to production?* Figure 4.1 shows that almost 53.7% (64 out of 119 (21+98)) of the participants indicated that on average the applications in their respective or client organizations were in deployable-state *multiple times a day* or *once a day*, indicating they were successful to truly implement CDE. However, CD was less adopted in the participants' organizations as in total 43 out of 119 (36.1%) participants indicated that the application changes were automatically pushed *multiple times a day* or *once a day* to production. This finding can have twofold implications: First, it shows that compared to the organizations studied in [11, 27], our participants' organizations were more successful in implementing CDE and CD practices. That means our findings came from reliable sources. Second, it reveals that the practitioners believe that CDE and CD *are* quite different practices: despite CDE being successfully adopted by the participants' organizations, there might be factors that limit or demotivate them to have the *automatic* and *continuous* deployment to production (i.e., CD practice). Section 4.4.3 reports these confounding factors.

Key Finding

Finding 1. *From a practitioner's perspective, continuous delivery and continuous deployment are indeed distinguishable practices in the industry.*

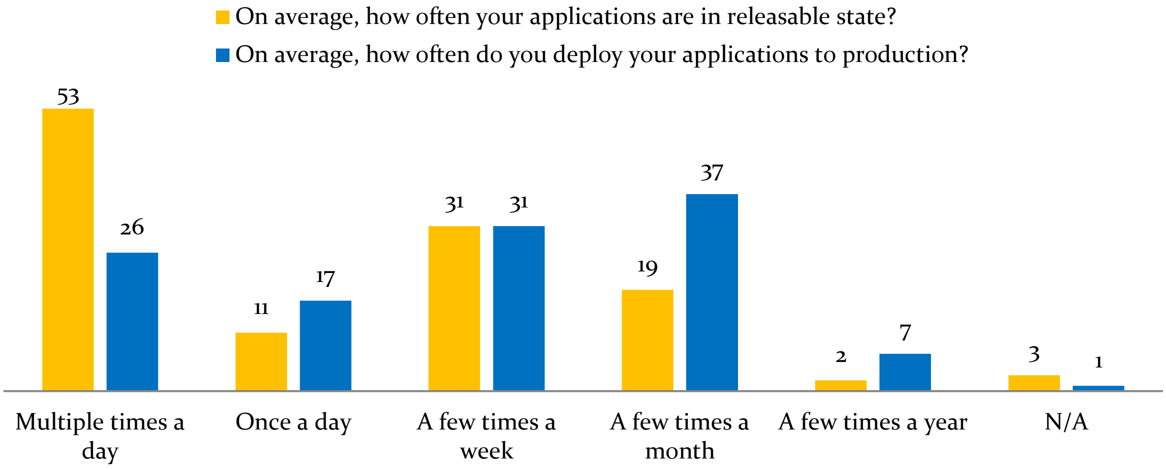


Figure 4.1 How continuous delivery and deployment are implemented – aggregated results of interviews and survey

4.4.2 Current State of Automation Support in Continuous Deployment Pipeline

Continuous Deployment Pipeline (CDP) (aka. continuous delivery pipeline) plays a significant role in helping organizations to achieve *continuous* and *automatic* deployment [98]. This means that the success of practicing CD in an organization heavily relies on the degree of automation support in the CDP [78]. A fully automated CDP enables organizations to automatically build, test, configure and deploy new features to production. Therefore, we were interested in understanding how automation is supported in CDPs in the practitioners' organizations. The survey respondents were asked to score their CDPs in terms of automation on a 1-5 scale (i.e., from *1-completely manual* to *5-completely automated*). The data from Figure 4.2 shows that over 70% (69 out of 98) of the surveyed participants scored their CDPs 3 or 4, which can be considered as semi-automated CDPs. Surprisingly, only 19 out of 98 of the respondents indicated that they had fully automated CDP to transfer the changes from the repository to production.

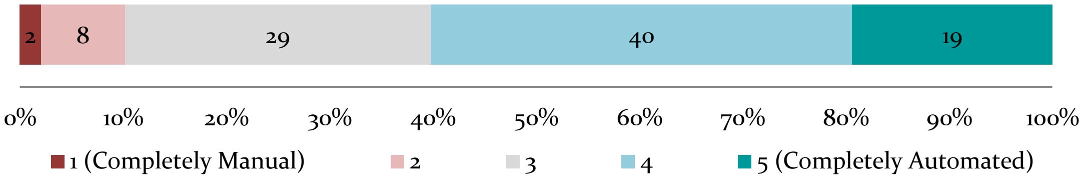


Figure 4.2 Statement 1: How you would grade your CDP in terms of automation?

Through a five-point Likert-scale question, we asked the survey participants to indicate how strongly they agree or disagree with this statement (S2): "*we have the right tools to set up fully automated continuous deployment pipeline*". Figure 4.3 shows that whilst 25.5% of the respondents *strongly* agreed, 43.4% agreed that there are the right tools for this purpose.

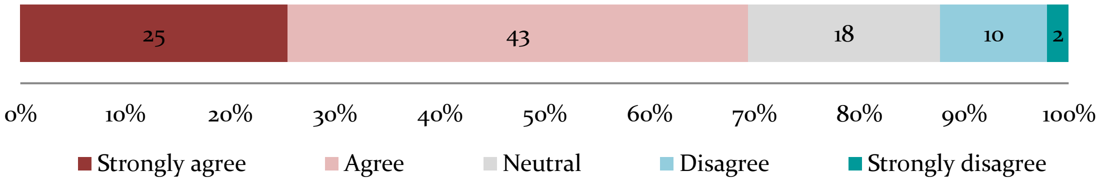


Figure 4.3 Statement 2: We have the right tools to set up fully automated CDP

Key Finding

Finding 2. *Whilst only 19.3% of the surveyed participants believe that their deployment pipeline is completely automated, the last stages of the pipeline including "acceptance testing", "production deployment", and "configuration and provisioning" stages are likely to be* ***semi-automated*** *or* ***manual****.*

Furthermore, we intended to explore what stages of a CDP may more or less support automation. Typically, a CDP is composed of explicit stages (e.g., build) to push code from the source code repository to production [7, 78]. There is no standard or single pipeline as organizations may design and implement their own CDPs with different stages and diverse tools [78]. Through a multiple-choice question, we first asked the surveyed participants which of the following stages constitutes their CDPs: "version control", "build", "continuous integration", "artifact repository management", "unit/integration testing", "acceptance testing", "production deployment", "configuration and provisioning", "log management and monitoring", and "containerization". We also included "Other" field to collect any other stages in a CDP. As Figure 4.4 shows, "version control", "build", "unit/integration testing", "continuous integration" and "production deployment" were the most common stages of CDPs. However, "containerization" was the least commonly mentioned stage in CDPs as only 37 survey participants referred to the "containerization" stage. We also found that not all stages are compulsory in a CDP as only 19 survey participants indicated that the CDP in their respective organization includes all of the abovementioned stages. Figure 4.4 demonstrates that only 5 respondents completed the "Other" field. One respondent pointed out that each application has its own CDP; therefore, there is high variability from application to application. Two others referred to "configuration and provisioning" stage in different ways (e.g., "*Cloud Management and Self-Service*" ***R80*** and "*Automated Provisioning of Environments*" ***R90***).

Through two open-ended questions, the survey respondents were asked which of the above-mentioned stage(s) have the most and the least automation support respectively. The responses to these questions indicate that "acceptance testing", "production deployment", and "configuration and provisioning" were the stages that had the least automation support respectively. In contrast, "build", "continuous integration" and "unit/integration testing" stages got the most automation support. Our analysis has revealed that the organizations with completely automated CDPs were much more successful to achieve *frequent* and *automatic* deployment. Among the respondents who scaled their CDPs as *semi-automated* or *manual*, only 30.3% indicated that (application) changes are *automatically* pushed to production *multiple times a day* or *once a day*. The responses to these two open-ended questions were fed into our qualitative analysis process, where applicable, to explore why some CDP's stages had less automation support and how lack of fully automated CDP limited the participants' organizations to truly adopt CD (See Section 4.4.3).

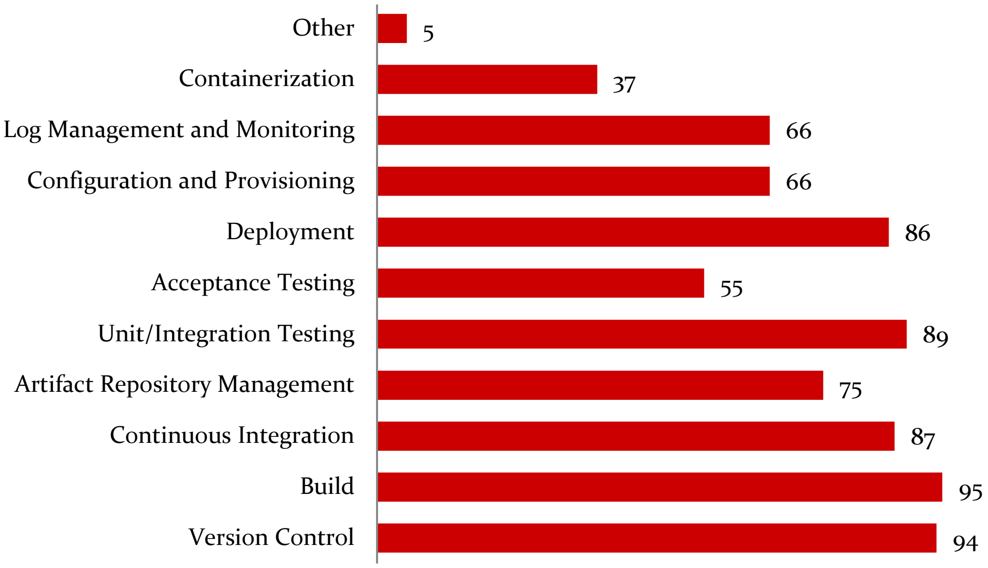


Figure 4.4 Stages of Continuous Deployment Pipeline

4.4.3 Moving from CDE to CD

This section reports the confounding factors in moving from CDE to CD, which are extracted from the interviews and the survey open-ended questions. We also assess and quantify these factors by indicating the number and percentage of the survey respondents who experienced these factors (See Table 4.1).

Table 4.1 Summary of confounding factors in moving from CDE to CD

Confounding Factors	**#**	**%**
F1. Lack of fully automated user acceptance test	**43**	**43.9**
F2. Manual quality check	**42**	**42.9**
F3. Deployment as business decision	**41**	**41.8**
F4. Insufficient level of automated test coverage	**34**	**34.7**
F5. Highly bureaucratic deployment process	**31**	**31.6**
F6. Lack of efficient rollback mechanism	**24**	**24.5**
F7. Dependency at application level	**23**	**23.5**
F8. Demotivated customer	**19**	**19.4**
F9. Customer environment	**16**	**16.3**
F10. Domain constraints	**15**	**15.3**
F11. Manual interpretation of test results	**11**	**11.2**

1) Lack of fully automated (user) acceptance test

We found that one of the major changes that usually would happen during the transition to CD is to identify reworks and eliminate their root causes effectively. An often-heard reason for reworks in the development process was manual testing. Several interviewees stated that an extensive effort and time in transition to both CDE and CD practices have been spent on automating existing manual tests (e.g., "*From a technical perspective, you have to reduce time and move fast, you have to care about testing. The problem is that you can't automate everything because it sounds time consuming*" ***P13***). We were interested in gathering viewpoints of the practitioners in this regard on a quantifiable scale. Hence, we asked the survey participants (n=93) to rank how important was the challenge of "*lack of full test automation*" in CD adoption, so because of which they faced serious challenges in *automatic* and *continuous* deployment. Figure 4.5 indicates that 78.7% of the respondents rated the severity of this challenge as *very important* or *important*.

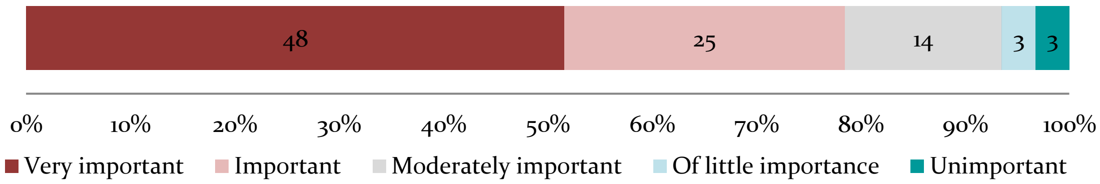


Figure 4.5 Statement 3: How important is "*lack of full test automation*" in adopting CD and put you in trouble to have automatic deployment

The interviewees disclosed that they considerably succeeded in automating unit and integration tests, but automating the tests occurring at the end of development process such as (user) acceptance test and performance test has remained a challenge and requires heavy workloads and time. As one consultant observed:

> *"They [organization] really often have challenges to get it [test automation] done, for acceptance tests most of the time it is not easy to fully automate"* ***P9***.

Our survey's results were aligned with our interviews' findings as "lack of full user acceptance test automation" has presented the most confounding factor (43 out of 98, 43.9%) for CD success. The survey participants shared the following reasons why (user) acceptance tests were or could not be fully automated.

a) Too much effort with low gain

More survey participants than expected mentioned concerns about the potential benefits of automating acceptance test at large scale compared to its associated complexities and costs (e.g., "*I'm guessing it [automating user acceptance test] is seen as a high effort for low gain*" ***R61***). That is why in some participants' organizations there is not enough demand for this purpose. Furthermore, some believed that acceptance testing should involve human intervention (e.g., for assessment of results) as it would bring more value and increase confidence in code quality. As explained by **R14**

> "*Acceptance testing [is manual] due to high frequent UI changes, [in which] it is more efficient for manual testing to validate user experience against requirements*".

In this regard, the survey participants also indicated that they experienced a significant burden of learning and changing the mindset of customers to support fully automated acceptance testing. As stated by **R40**

> "*Member acceptance test [is manual]; we cannot dictate customer workflow*".

b) Tools limitations

A few of the survey participants reported that current tools and technologies are immature to fully automate acceptance tests (e.g., "*Acceptance testing [is manual], because we have lacked the tools and technology to suitably automate this*" ***R2*** or "*The tools we use [for automating acceptance tests] are too crude*" ***R91***).

c) Lack of automation skills

Lack of automation skills was another reason for having manual acceptance testing. **R2** explained the reason to adopt manual acceptance test as "*The staff involved in acceptance test phase are often domain experts with little or no automation knowledge or development skills*".

2) Manual quality check

The analysis of the qualitative data also indicates that although automation is critical in CD practice, manual tasks are sometimes unavoidable. For example, the organization that P15 helped to adopt DevOps was always able to deploy working versions to lower environments (e.g., staging environment) in an hour, yet the step for getting a working version from a low-level environment into production involved additional verification and approval. That is why it was not truly CD, but it was more CDE. The most common manual task mentioned by the interviewees was code review. The interviewees' organizations performed several manual code reviews before deploying software to production. This is partially because some organizations may not have highly skilled developers for truly practicing CD. As explained by a participant:

> "*That is second code check because first code reviews done within team and there is a final check by release manager who is one of the most knowledgeable developers in the organization*" ***P10***.

Ideally, Quality Assurance (QA) tasks should be automated and integrated into CDP [121]. Several interviews' participants expressed that using automated testing could have enabled them to effectively deal with QA team and their tasks as they are not a severe bottleneck for practicing CD anymore. However, for some of them, it does not mean removing all manual QA tasks from CDP. One of the interviewees reported on the reason that changes are not immediately pushed to production in the following words:

> "*We fully automatically deploy on the test environments. But currently, there is still a second button to deploy it out to the customer side. It is just because of sort of caution around, basically it would not be hard for us to automate it but there is just too much concern around quality level and having another round of sign off. That is probably our biggest trouble how to get the initial quality to a level that they can be deployed simply with developers independently*" ***P7***.

Many interviewees' organizations still need to perform many manual quality checks before each release. Team members still need to release software (changes) to the QA team to get certified.

According to Table 4.1, the second most mentioned pain by the survey participants (i.e., 42.8%) for *continuous* and *automatic* deployment was conducting the manual check and confirming the changes before each release. Some of the survey participants also revealed that their organizations are not willing to automate all QA tasks in the CDP (e.g., due to lack of trust in existing tools) and believe that manual quality checking brings much more value to them. As one respondent stated:

> "*Deployments to production need a human trigger. We feel [it is] safer to look closely at the metrics during the deployment process*" ***R89***.

3) Deployment as a business decision

The interviewees indicated that in their organizations the development teams were not able to immediately deploy every change to the production environment, despite passing all the tests and quality checks. This is mainly because deploying to production was considered as a business decision, which had to be made by management or financial sectors. In other words, development team members have little control over production deployment. Furthermore, different organizations may adopt different policies and timeslots for release, which can bring the most value to their customers [13, 122]. As can be seen from the following quote, despite developers could deploy changes into lower environments at any point of time, production deployment occurred every three weeks through a formal process.

> "*At any point of time, [if] we wanted we could deploy into lower environments, but the major release was done every three weeks because the release process still was quite important for management to have sign off from the testers …*" ***P15***.

The survey respondents were asked to determine whether this factor (i.e., deployment as a business decision) impacted their capability to have *automatic* and *continuous* production deployment. According to the survey responses (See Table 4.1), this factor was ranked as the third most common confounding factor in this regard, in which out of 98 survey participants, 41 (41.8%) indicated that production deployment is considered as a business decision and is out of developers' control.

4) Insufficient level of automated test coverage

Lack of sufficient automated test coverage was also deemed as a bottleneck to transition from CDE to CD (e.g., "*Of really large problem [in this journey] was maintaining our automated test coverage*" ***P17***). It is important to note that insufficient level of automated test coverage reduced the confidence of some interviewees' organizations in the readiness of their applications for actual deployment. Our survey results also confirm this concern as a large number of the survey respondents (34 out of 98, 34.7%) indicated this as a CD challenge.

5) Highly bureaucratic deployment process

We found that the deployment process in some organizations is still highly bureaucratic. Our findings characterize a highly bureaucratic process as the one having a large number of formal tasks (e.g., getting approvals from various people) to be performed manually before each release. For example, one interviewee and one survey participant highlighted this in the following quotes:

> "*Solution deployment to customer side involves taking agreement and acceptance from their backend teams. So we were given some slots based on their delivery cycle or their priorities and we had to obey those slots*" ***P5***.

> "*Telecom operators have deployment processes that have formal bureaucratic checklists prior to deployment, due to multiple integrations in their network. Solutions come from various vendors and having CD to make eco-systems working across the vendors involve manual checks*" ***R67***.

Table 4.1 reveals that 31 of the survey respondents (31.6%) chose that deployment process in their respective organizations or client organizations was highly bureaucratic.

6) Lack of efficient rollback mechanism

In comparison with CDE practice, CD requires better monitoring solutions and fully automated rollback mechanisms [12]. Whilst **P17** stated that integrating automated rollback in CDP increased their confidence to have *automatic* deployment, our analysis of qualitative data reveals that lack of having efficient rollback mechanism, forced a couple of the interviewees' organizations to decrease the pace of pushing changes to production. Lack of efficient and automated rollback mechanism to quickly recover issues in deployment process may put software organizations at risk of delivering buggy code to their customers. 24.5% of the participants confirmed that this confounding factor was the reason for having the manual deployment. One survey participant, an architect, stated:

> "*Deployment [is manual], because it doesn't support rollback neither has power to provision of a machine/instance. The rollout could be better by testing the released version in production, load tests, micro benchmark, etc.*" ***R43***.

7) Dependency at the application level

Our study has found that albeit an application might be at deployable state, dependencies between that application and other systems may have inhibited some of the participants' organizations to transition from CDE to CD. It means organizations need to ensure that there is no integration problem when deploying an application to production. Deploying software changes on a continuous basis necessitates continuously deploying all dependencies (e.g., dependent applications). One interviewee described this situation vividly:

> "*The difficulties become visible when you automate deployment for a complex stack. Then you have sometimes challenges to get everything working within one task. To all dependencies of your deployment, if [you] have legacy applications, then you need to deploy all these things together and everything should work after deployment and you always face things more difficult. So, there are always some tasks to automate*" ***P9***.

As shown in Table 4.1, the survey results moderately confirmed our interviews' findings as 23.5% of participants agreed that dependency at application level was a confounding factor. To give an example, one survey participant discussed the reason for manually deploying their application in these words:

> "*Deployment is a manual process because once an artifact is created, [in order] to allow coordination with other services; dependencies must be known in advance and aren't written down*" ***R41***.

8) Demotivated customer

Based on our analysis and the interviews' discussions, we perceived that the time and pace of deployment to production greatly depends on customers' cultures, policies, and goals. We found that not all customers are mature enough to accept a continuous release. The interviewees pointed out that whilst they were able to give updates as frequently as possible to their client organizations, the established cultures and policies in the client organizations did not support fully transition to CD practice. Therefore, they had to follow pre-defined timeslots (i.e., calendar-based release) for releasing software.

Our survey participants confirmed this finding as 19 of them indicated that their customers were not happy or had no need to receive the *continuous* and *automatic* release. One architect and one consultant from the survey study explained respectively:

> "*Upgrading to a new release is expensive and strategic for customers, they don't want to run the risks of a continuous daily or weekly delivery, they want to upgrade once a year at most*" ***R78***.
>
> "*Moving into production [is manual] because that is not done too often; and it is a handover to Ops*" ***R13***.

Our study shows that compared to CDE, customers need to be actively involved in *continuous* and *automatic* deployment for truly practicing CD (e.g., "*I think our domain and customer are not yet in the position to have continuous deployment*" ***R23***).

9) Customer environment

Our findings have revealed that the participants often found themselves struggling with customer environment as a severe roadblock to moving from CDE to CD. It was often stated by the interviewees that lack of carefully studying and exploring customer environments before moving to CD led to challenges in the *continuous* and *automatic* release. One CD consultant observed:

> "*I saw a customer actually who did not take regulation compliance into consideration and invested a huge amount of time and money on doing microservices and fancy tools. In the end, they couldn't deliver more often than once a month, because of regulations. Many things should be taken into considerations for the success of this [CD] journey*" ***P14***.

Table 4.1 displays that 16 out of 98 survey respondents indicated that the challenges (e.g., manual configuration of complex software) associated to customer environment negatively affected their capability to *automatically* and *continuously* release software (changes). During our interviews and survey studies, we heard the following challenges associated with production environments as confounders to adopt CD.

a) Manual configuration of complex software

Anecdotally, several of the interviews' and survey participants have said that manual configuration of complex software, particularly when there is a tight coupling between software and hardware, and regulatory environments represented a significant obstacle to CD success. Here are just a few of the examples indicating the participants struggled with manual configuration:

> "*On the customer/onshore side, requirements developed, and additional expertise was brought in to manage and support the manual deployment due to firewall and legacy processes preventing CD on the client side*" ***P3***.

> "*This [production deployment] still involves a manual step to move the release artifact from one environment to another, due to network separation*" ***R71***.

> "*Configuration and provisioning vary from site to site at the customer location. Hence this involves manual configuration from team*" ***R67***.

Besides the complex and error-prone process of configuration and provisioning in some production environments, we also noticed several other reasons for having manual configuration and provisioning: (i) lack of mature tools (e.g., "*We do not have a robust toolset for automating configuration*" ***R49*** and "*Configuration is [manual] in Puppet but requires restarting of applications to bootstrap and load new configurations adding a manual step into the process. We use load balancing, so Puppet can't restart at will*" ***R41***); and (ii) not much value in automating configuration and provisioning. A team lead commented

> "*Provisioning [is manual], because we update instance images only a few times per week*" ***R52.***

b) Hard to simulate/access real production

Our analysis highlights that a lack of control on, access, and simulation of the production environment (e.g., on-premise environment) make it difficult to deploy potentially releasable changes on a continuous basis. When there is no direct and regular access to customer environments, a software development team needs more communication with the operations team at customer side to get confirmation and agreement for each release. The following quote depicts this issue:

> "*We had a project and we had concrete control through the infrastructure and choice of technologies. That kind of environment is pretty simple to kind of get deployment pipeline that you want. But when we are working with a customer that does not use that model of working, that's needed to have a lot of communication, a lot of mentorships, e.g., why we need to do these things*" ***P12***.

The interviewees frequently shared that it is not easy (if possible) to simulate production-like environments with realistic data. Therefore, lack of access to and control on production environment make it much harder to fully automate the deployment process. One interviewee described this situation perfectly:

> "*There is always challenge on how to keep testing environment in synchronization with production environment. Because you can't have the same testing environment like production environment, for example network is different*" ***P14***.

Due to the above-mentioned issues, there was some debate about the real benefits of staging environment in the context of CD. According to the interviewees, this is mainly because staging environments do not show how really software works as a few of them explicitly stated that staging environment can be disruptive to successfully adopt CD [117]. One interviewee told us:

> "*We used to have a staging environment because the reason that we needed to have a place where we integrate all the changes in one version, we test and then we deploy. But I think when we are going to more rapid deployment, all the time continuously deploy. Then we started to feel this [staging] doesn't fit this pattern because the problem is that you don't have the whole data. So, you have the risk because staging and production are not equal. So, it is like cheating you; the real life is different*" ***P16***.

10) Domain constraints

Many challenges in the applicability of CD arise due to domain constraints. This factor could change the frequency of releasing software (changes) to production [11]. Whilst CD practice is more easily applied to web-based applications, it may not be easily applied to other domains such as embedded systems and financial systems [18]. One possible reason for this is that such domains are more conservative to *automatic* deployment to customers and require more (manual) verifications before each release [81, 115]. Table 4.1 indicates that among 98 survey participants, 15 considered domain constraints as a confounding factor. Whilst the survey participant **R40** told us that they are a financial exchange and are not able to deploy during business hours, another survey participant described the domain constraints in the following words:

> "*I deal with Big Data style petabyte state. Large-scale state migration during deployment remains a challenge at this scale due to the cost of backup and impact of lost state*" ***R46***.

11) Manual interpretation of test results

Long-running tests and test results' interpretation were seen as other confounding factors. Long-running tests not only increased the cycle time (i.e., the time required to get the code from code repository into production) in a CDP but also have hindered developers from getting real-time feedback. One interviewee revealed that a large portion of cycle time had been spent on running regression tests and interpreting the tests results. Another issue mentioned by a number of the interviewees was the fact that there was very little automation support for regression tests. So it involves manual efforts and takes huge cycle time. Hence, it depends on the extent the regression tests can be automated, organizations can significantly reduce the overall cycle time in the CDP. Furthermore, with the increasing number of test cases, the interpretation of test results becomes quite time-consuming and labor-intensive process. One interviewee reflected:

> "*We have some challenges to fully automate deployment process; one of the challenges is the interpretation of test results. There is manual and intervention, between build the test and actual deployment, to interpret the results*" ***P6***.

Only 11 participants in our survey (See Table 4.1) confirmed that "manual interpretation of test results" was a confounder in the transition from CDE to CD as one of them stated "*Acceptance testing still requires manual assessment of results*" ***R24***.

Key Findings

Finding 3. *Overall, making the decision to move from continuous delivery to continuous deployment is influenced by both* ***technical*** *and* ***socio-technical*** *factors.*

Finding *4. The three most cited stumbling blocks to automatic and continuous deployment are (1) lack of fully automated user acceptance test (2) quality concerns (3) considering production deployment as a business decision.*

4.5 Discussion

This section first discusses some of the main findings from our study. Second, we suggest implications for practitioners and researchers based on the themes that emerged from five-top reported factors and analysis of the responses to an open-ended question: "*Given the increasing importance of automation in CD, in your understanding what are the top four things that you look for/need/would like to see in automation*".

4.5.1 Summary of main findings

Our study indicates that there ***is*** a well-understood difference between practicing Continuous DElivery (CDE) and Continuous Deployment (CD). Specifically, there are factors because of which organizations may be unable or demotivated to move from CDE to CD (i.e., having *automatic* and *continuous* deployment). These factors are "lack of fully automated (user) acceptance test", "manual quality check", "deployment as business decision", "insufficient level of automated test coverage", "highly bureaucratic deployment process", "lack of efficient rollback mechanism", "dependency at application level", "demotivated customer", "customer environment", "domain constraints", and "manual interpretation of test results". Moreover, we found that most of the participants' organizations still have semi-automated CDPs, in which "acceptance testing", "production deployment", and "configuration and provisioning" stages have least automation support.

4.5.2 Implications for research and practice

Better automated testing: Both the interview and survey data show a strong need for better support for automated testing, specifically (user) acceptance testing. Several of the participants mentioned that the current automated testing tools need significant improvements in order to harden them for different environments. The participants thought that (user) acceptance testing on the relative scale have to be run and assessed by a human as it brings more value and safety. The participants also mentioned the need to test all types of applications (for example mobile testing as it is fragile and expensive to automate), techniques and tools that enable parallelization of automated testing and infrastructure automation testing.

Integrating automated quality checks: Software quality was one of the major concerns in the interviewees' and the survey participants' organizations before each release. It was also among the top priorities for business leaders. Security and performance were the most frequently reported quality concerns. According to the participants, attention to security needs to increase and performance testing should be conducted at production scale to truly implement CD. However, an open question is how to efficiently automate quality checks inclusive of performance and security and incorporate them into CDP. For instance, there is a strong need for integrating performance baselines into CDP.

Management support: Participants perceive "managers" are hesitant to allow developers immediately push out every change to production because only business leaders of their organizations are responsible to make the decision about *when* and *what* to be deployed to production. Our analysis shows that compared to CDE, successfully adopting CD needs better management support. This is mainly because deployment on a continuous basis without human intervention may increase complexity as organizations need to deal with more components, more people, more roles, and more concerns. Hence, this can be much more a business or political problem rather than just an engineering problem. Management at both customer and vendor organizations is expected to have a clear understanding of business drivers of continuous deployment and get all the stakeholders on board. Whilst the main business leaders' concern is around the quality level, our results suggest that integrating automated quality checks and security test in both development and operations processes can alleviate this concern and to a large extent make continuous deployment compelling to business leaders. To achieve CD, organizations must break down barriers at production. This is mainly achievable by allowing developers to be part of deployment decision-making and placing more trust in them. By this, the manual approval process described in Section 4.4.3.5 will be significantly reduced.

Easier tools integration: As we discussed in Section 4.4.2, CDP is a tool-chain, which a number of open source and/or commercial tools should be integrated for this purpose. This is mainly because deploying with *one and only tool* would make *automatic* deployment process more complicated. However, a commonly mentioned issue was compose-ability of tools. Software organizations need to spend too much engineering effort to architect each of distinct tools to interface and integrate with other tools to make them work seamlessly. We observed that due to the availability of a gamut of tools and lack of standardization between them, there is too much of chaos in the way each organization adopts their continuous delivery or deployment journey. It is highly recommended that tool vendors consider applying standards that enable an organization to easily stitch tools together. Such standards would drastically minimize the difficulty and effort required for tools integration.

Digestible visualization and monitoring: Although there are lots of monitoring tools available, the survey participants often expect tools and techniques, which enable them to have full monitoring coverage. In the meanwhile, having a visual representation of end-to-end build, test and deployment would make a huge difference in the capability of organizations to release faster and often. Unfortunately, current tools are not great for this; presumably, because they do not do a good job (e.g., lack of domain specific monitoring tools) or are exceedingly complex for this purpose. Furthermore, scaling CD practice in large organizations with multiple teams and applications can worsen this problem as a wide range of stakeholders need to be able to understand what is happening, what has happened, and why in a real-time manner. Hence, there is a need to develop tools that provide real-time, digestible and customizable monitoring and alerting for the different types of stakeholders [123, 124].

Other needs: There are also serious needs for (1) better tools to simplify configuration and provisioning of environments and support automatic setup of distributed environments; (2) tools to manage, validate and automate schema upgrades and database migrations in CDP; and (3) better post-deployment checks (e.g., automated smoke and reliability testing after deployment).

4.6 Conclusion

This chapter has reported an empirical investigation into the reasons (e.g., manual user acceptance testing) because of which organizations may be unable or demotivated to *automatically* push out *every change* to production in order to have many production deployments every day. Our findings came from a mixed-methods study consisting of data collection and analysis from 21 semi-structured interviews and an online survey completed by 98 software practitioners. This research reveals the current state of automation support to truly implement continuous deployment. Interestingly, the majority of the participants' organizations did not have fully automated deployment pipelines, with mostly semi-automated or manual "acceptance testing", "production deployment", and "configuration and provisioning" stages. We have also identified several future research directions (e.g., better tooling support) along with a set of recommendations (e.g., management support) that can help streamline *continuous* and *automatic* deployment.

Chapter 5

Continuous Delivery and Deployment: Organizational Impact

Related publication:

This chapter is based on EASE paper, "Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities" [4].

While some efforts have been made to study different aspects of Continuous Delivery and Deployment (CD) practices, a little empirical work has been reported on the impact of CD on *team structures*, *collaboration,* and *team members' responsibilities*. To this end, we leveraged the data collected from 21 in-depth, semi-structured interviews in 19 organizations and a survey with 93 software practitioners (i.e., as described in Chapter 2) to empirically investigate how Development (Dev) and Operations (Ops) teams are organized in the software industry for adopting CD practices. We report that there are four patterns of organizing Dev and Ops teams for this purpose. The research presented in this chapter also provides insights into how software organizations actually improve collaboration among teams and team members for practicing CD. Furthermore, we highlight new responsibilities and skills (e.g., monitoring and logging skills), which are needed in this regard.

5.1 Introduction

The highly complex and challenging nature of DevOps practices, particularly Continuous Delivery and Deployment practices (i.e., they are referred to CD practices in this chapter), make it inevitable that organizations improve their skills, form the right teams, and investigate organizational processes, practices, and tool support to gain anticipated benefits from DevOps practices [23]. With the increasing popularity of CD practices, the research community has been conducting extensive research efforts to understand how organizations initiate and implement these practices. For example, a few papers have investigated the challenges that organizations may face in adopting CD practices [11, 23, 113]. The other area of interest in CD is to provide and integrate appropriate technologies and tools to support automated configuration and deployment processes [26]. On the other hand, it is asserted that achieving CD may require a new way of working and changes in team structures and responsibilities [23, 125, 126]. Furthermore, CD practices demand tighter and stronger collaboration and integration among teams and team members [127]. However, there is no systematic research about how organizations actually form and arrange Development (Dev) and Operations (Ops) teams and also how they increase collaboration among teams and team members to optimally embrace CD practices. We assert that such questions should be explored and answered through empirical studies involving practitioners from diverse organizations rather than through one case company or one practitioner's perspective [125, 126]. To address this gap, we report an empirical investigation to address the following research questions:

RQ5.1 How are Dev and Ops teams organized to initiate and adopt continuous delivery and deployment?

RQ5.2 How is collaboration among teams and team members improved for adopting continuous delivery and deployment?

RQ5.3 How does adoption of continuous delivery and deployment impact on team members' responsibility?

We used part of the mixed-methods study presented in Chapter 2 to answer these research questions. The main findings of this chapter are:

(i) There are four common types of patterns for organizing Dev and Ops teams to effectively initiate and adopt CD practices: (1) *separate Dev and Ops teams with higher collaboration*; (2) *separate Dev and Ops teams with facilitator(s) in the middle*; (3) *small Ops team with more responsibilities for Dev team*; (4) *no visible Ops teams*.

(ii) The participants shared that *co-locating teams*, *rapid feedback, joint work and shared responsibility, using collaboration tools more often, increased awareness and transparency,* and *empowering and engaging operations personnel* enabled them to increase the collaboration among teams and team members in the path of adopting CD.

(iii) Team members have three key high-level changes in their responsibilities: *expanding skill-set*, *adopting new solutions aligned with CD*, and *prioritizing tasks*.

Chapter organization: Section 5.2 summarizes related work. Section 5.3 describes the research methodology. We report our findings in Section 5.4. Finally, Section 5.5 closes the chapter with discussion and conclusions.

5.2 Related Work

There are a number of empirical studies that have investigated the challenges and practices of adopting DevOps and CD [2, 11, 23, 115]. Among the reported challenges (e.g., monolithic architectures [2]) and practices (e.g., management support [22]), the studies have also briefly discussed the skills required for practicing CD, and how collaboration and coordination among teams and their members can be consolidated for this purpose.

Savor et al. [22] report that the developers needed to gain new skills as a result of implementing continuous deployment at Facebook and OANDA. The studied companies assigned new developers to the release engineering team for several months. Claps et al. [23] identified 20 technical and social challenges (e.g., team experience and team coordination) that a single case company faced in the transition towards continuous deployment. In order to move from continuous integration (CI) to continuous deployment, the case company studied in [23] leveraged CI developers' experience by integrating automated continuous deployment of software into the existing CI workflow of developers. This approach helped them to reduce the learning curve for developers. Other studies discuss that when a project adopts CD practices, it would be helpful to define new roles and teams in software development lifecycle to smooth this path. Krusche and Alperowitz [128] define hierarchical roles such as release manager and release coordinator to adopt and implement continuous delivery in multi-customer projects. It is argued that these roles improve coordination among team members. Other studies argue that establishing a dedicated team for the design and maintenance of infrastructure and deployment pipeline helps organizations to smoothly transform to CD and reduce release cycle time [22, 129].

Wettinger et al. [127] present the idea of solution repositories to provide efficient collaboration between developers and other team members (i.e., operations stakeholders). Each team in software development lifecycle may use their own solutions and repositories to build and maintain knowledge and documents around corresponding solutions. This approach could significantly hinder knowledge sharing and collaborative work in a team. The collaborative solution repositories automatically collect and store different solutions and their metadata from diverse environments (e.g., test environments) and sources (e.g., Chef). Then this data is utilized to establish consolidated knowledge base instances for supporting collaboratively work.

Nybom et al. [130] conducted a case study with 14 practitioners in an organization to investigate the potential impact of mixing responsibilities between developers and operations staff. The study reveals that mixing responsibilities, among other impacts, remarkably increases collaboration and trust, and fosters team members' workflow. However, this approach is associated with a number of negative implications. For example, given more responsibilities to developers and constantly learning about operations tasks might demotivate some developers to have collaboration with operations personnel.

França et al. [131] conducted a multivocal literature review to characterize DevOps principles and practices. The study highlights that developers and operations personnel need to gain both social (e.g., communication) and technical (e.g., math skills for performance analysis) skills to truly perform DevOps. This also enables team members to effectively collaborate on fixing bugs. The study also found a number of practices to improve collaboration among team members including role rotation, face-to-face communication, and open information.

It should be noted that none of the above-mentioned studies has systematically and empirically explored the actual impact of CD practices on the structure of Dev and Ops teams, team members' responsibilities, and collaboration among them. Whilst analyzing our data for RQ5.1, we came across a few blogs [125, 126], which suggest different team structures (e.g., Ops as Infrastructure-as-a-Service) for DevOps success. That increased our confidence in the importance of exploring how Dev and Ops teams are organized in practice for adopting CD. We assert that our findings provide an evidence-based and detailed view of different team setups when adopting CD, as they are not restricted to a single case company or observations of one practitioner.

5.3 Research Method

As described in Chapter 2, a mixed-methods empirical study was adopted to answer the research questions introduced in Section 5.1. For this chapter, we have collected qualitative data through 21 in-depth, semi-structured interviews and a survey of 93 practitioners to further gain the evidence and understanding of our findings from the interview study.

5.3.1 Interviews

We initiated the analysis by breaking down the transcripts into three high-level segments according to our research questions: the impact of practicing CD on team structures (RQ5.1), collaboration (RQ5.2) and team members' responsibilities (RQ5.3). Then we rigorously reviewed the transcripts and extracted and coded data related to each of the research questions.

5.3.2 Survey

Apart from demographic questions (e.g., role and experiences), we particularly asked about how practicing CD has influenced team structure, responsibilities, and collaboration of team members. The relevant survey questions used for this study contained 11 questions including demographic (4 questions), five-point Likert-scale (3 questions), single-choice (2 questions) and open-ended (2 questions).

5.4 Findings

We present our findings of different team structures in software organizations to adopt CD, followed by strategies and practices adopted to effectively improve collaboration. Finally, we describe how CD adoption may change the responsibilities of team members.

5.4.1 Team Structures for Adopting CD Practices (RQ5.1)

This section reports how Development (Dev) and Operations (Ops) teams are organized to implement CD practices (See Figure 5.1[21]). First, we present the main patterns of team structures for this purpose, which are extracted from the interviews. Second, we assess and quantify these patterns by indicating the number of the survey respondents reported these patterns.

Separate Dev and Ops teams with higher collaboration: Our analysis has disclosed that for a couple of the interviewees' organizations, in particular hierarchical ones, adopting CD does not necessarily mean huge changes in team structure or complete breakdown of silos (i.e., divisions of labor) between teams. They tried to leverage their existing Dev and Ops teams by providing the needed infrastructures and emphasizing the culture of empowerment in order to make a higher and tighter collaboration between Dev and Ops teams (See Figure 5.1.A). Through this strategy, they were able to achieve DevOps and CD goals as much closeness as they could. The amount of collaboration between teams and team members, in particular application developers and operations team, increased after adopting CD. It was explained by one of the interviews' participants in the following words:

> "*They [organization] did very successful continuous delivery even though they have separate teams for development and operations. So, I mean you are on a spot that there should be close Ops cooperation, but it is not necessarily [to have] the full DevOps in the sense of making the teams [that] do operations and development tasks by themselves. I think you do need to have close collaboration, but you do not need to have teams to do both Ops and Dev [tasks]*" ***P11***.

We found that placing operations team next to developers (e.g., in the same office) and encouraging them to have more collaboration and face-to-face communication with other team members are simple strategies adopted by the interviewees' organizations to bridge the collaboration gap between Dev and Ops teams. One interviewee (i.e., a program manager) pointed out this in these words:

> "*There are two sub-groups, who reported me. There is some division of labor who focus on development and there is another subgroup of 3 people who are focusing on the operations and deployment. But they sit next to each other and they work very closely together*" ***P7***.

[21] Note that the icons that are used in this figure are taken from **freepik.com** and **thenounproject.com**

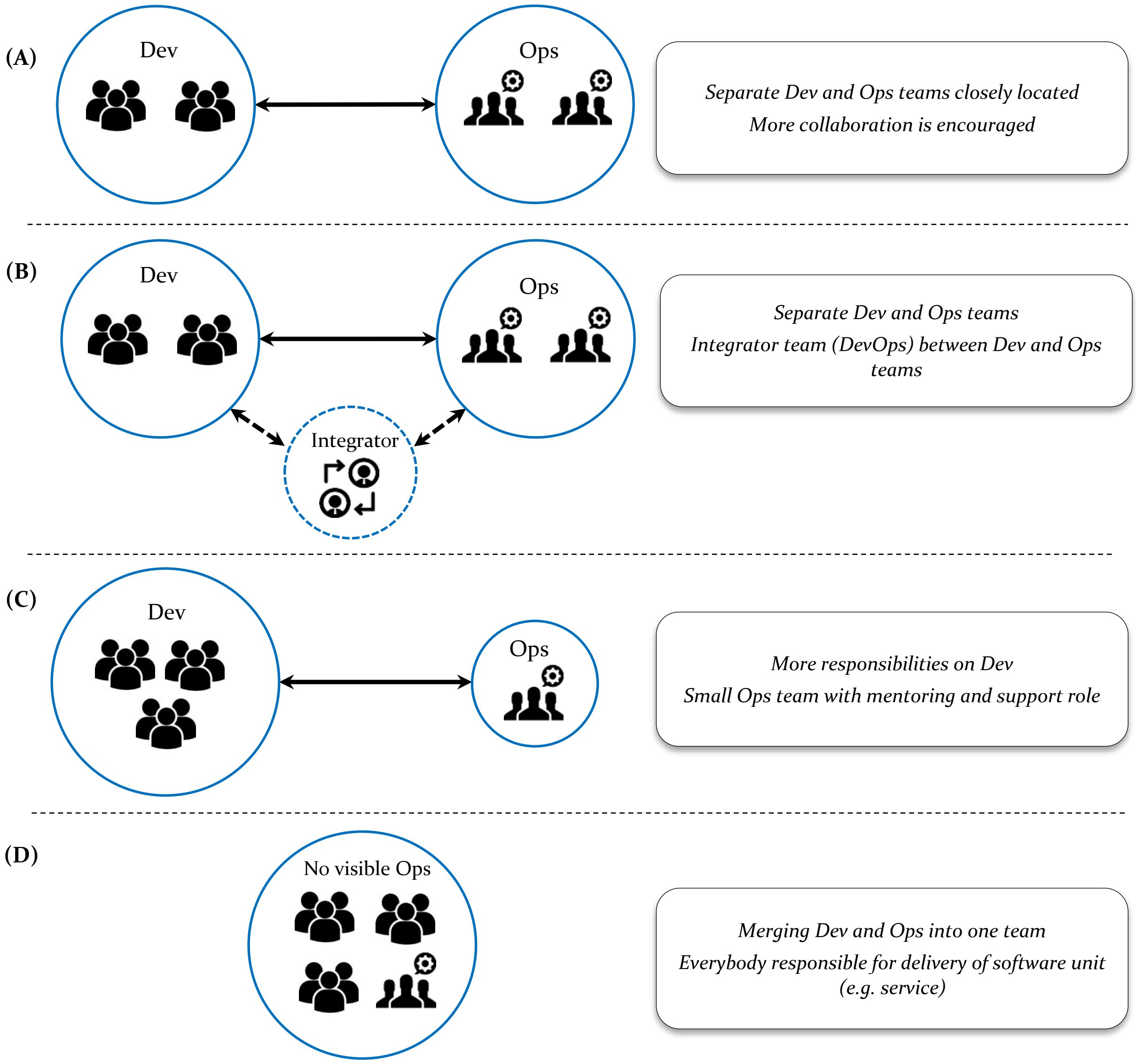

(A)
Dev
Ops
Separate Dev and Ops teams closely located
More collaboration is encouraged
(B)
Dev
Ops
Integrator
Separate Dev and Ops teams
Integrator team (DevOps) between Dev and Ops teams
(C)
Dev
Ops
More responsibilities on Dev
Small Ops team with mentoring and support role
(D)
No visible Ops
Merging Dev and Ops into one team
Everybody responsible for delivery of software unit (e.g. service)

Figure 5.1 Team Structure for effectively initiating CD practices

We asked the survey participants to determine whether this pattern describes the structure of Dev and Ops teams in their respective or client organizations. As shown in Figure 5.2, of 93 survey responses to this question, 33 (35.4%) of the respondents indicated that they still have separate Dev and Ops teams; however, it was reported that collaboration and coordination among even the separate teams had significantly improved. Interestingly, this pattern was mainly adopted by large organizations, followed by medium-sized organizations as 48.6% of the large organizations had structured their Dev and Ops teams in this way. While only 5 out of 26 small organizations chose this pattern.

Separate Dev and Ops teams with facilitator(s) in the middle: As part of the strategy to improve communication and collaboration between developers and operators, some interviewees' organizations would go a step further by defining and establishing a team, for example, so-called DevOps team, to facilitate communication and collaboration between Dev and Ops teams (See Figure 5.1.B). This team acts as an integrator between these teams to consolidate work together and knowledge sharing. The participant **P4** highlighted the role of this team as the follows:

> "*We had DevOps engineer [who has] job to integrate between development and operations. He [is] primarily responsible for integrating between Dev and Ops to make sure the all changes are applicable to operations"* ***P4***.

17.2% of the survey respondents stated that they are using a facilitator team as an enabler for communication and collaboration. Only one small size organization used this pattern, remainders were large (9) and medium-sized (6) organizations.

Small Ops team with more responsibilities for Dev team: DevOps often recommends that developers take more accountability about their code in production environments [7]. Some interviewees' organizations have gradually and smoothly shifted operational responsibilities from infrastructure and operations teams to Dev team. By applying this change, Ops team is more responsible for mentoring, coaching and helping developers to write operational aspects of the code, for example writing provisioning code. This strategy enabled the interviewees' organizations to make operations process easier and helped developers to commit codes that made less trouble. This is mainly because Ops team influenced the way the applications were configured to make them easier to deploy. Furthermore, Ops team may still exist to handle initial incidents in production environments. Hence, the development team is not available like 24/7 to address incidents in production and initial incidents handling will be out of developers' accountability. As one interviewee commented:

> "*They (operations team) often would pass the problems to the development team, if they cannot solve the problem itself and then the development team will get involved in operational things, incidents, that kind of things*" ***P18***.

Organizations within this category still had a distinct operations team, albeit a small one with limited responsibilities (See Figure 5.1.C). According to the interviews' participants, operations team is still needed to support deployed system in the production. They are mainly in charge of running the system, monitoring it, and fixing the performance issues. As **P12** stated that:

> "*The organization that I am talking about is a very hierarchical organization and we are not able to inroad and change the organizational hierarchical. I really like these things [operations tasks] run [by] product team where you have Ops people embedded in the product team and then whole team working together. We have not got that state but we could get through months and months like talking to each other and having bear with each other*" ***P12***.

We could not conclude that the Ops team in this category is a part of Dev team as there are always a bunch of tasks that are not really related to or out of the expertise of a development team. 27 out of 93 survey respondents indicated that there is a very small Ops team (e.g., 2-3 people) in their organizations to do specific tasks and most of the responsibilities of Ops team have been shifted to the Dev teams. The distribution of this pattern was almost similar among large (8), medium-sized (9), and small organizations (10). There is always a need to be someone on duty, particularly in critical systems such as financial systems, which has to be available 24/7. An IT architect, who worked in a company specialized in DevOps and CD and helped other organizations to adopt DevOps practices, pointed out that:

> *"For me, you may want to know, I have not seen in many organizations that DevOps team, the ideal situation, is really happened as a practice at the moment. So, what I mean this is a full responsibility; they are a really multi-disciplinary team and they can do all the technologies themselves and that requires highly skilled people to learn real DevOps team"* ***P9***.

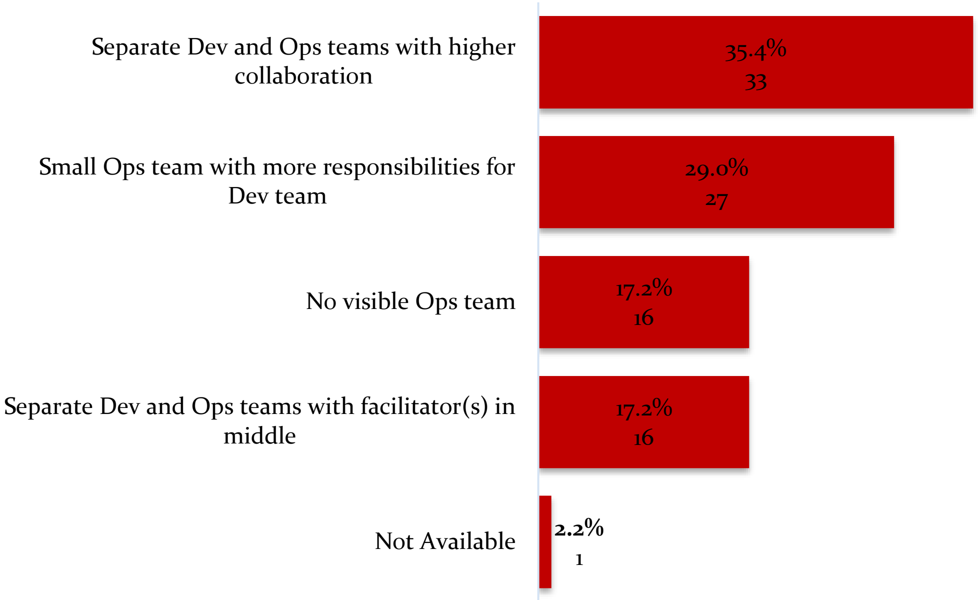


Figure 5.2 Survey results on patterns of organizing Dev and Ops teams for initiating and adopting CD (n=93)

No visible Ops team: Our analysis has revealed that in a few organizations, the Ops team has been an integrated part of Dev team (See Figure 5.1.D). There is no specific and visible Ops team; all team members have a shared responsibility and purpose to cover the entire spectrum of the software application, from requirement gathering, to continuously deploying, monitoring, and optimizing application in production environments.

> *"Well we had operational personnel at the team; they were there at every moment of the project as we [were] making decision. So they were integrated part of the team; there is no communication overhead for operation teams because we have no operations in a separate operations team"* ***P13***.

The results from the survey show that 17.2% of the respondents, especially in small organizations, stated that they do not have a visible and distinct Ops team. Those organizations have structured team members in the cross-functional team for each software unit (e.g., service and component); therefore, each team includes developers, business analyst, quality assurance (QA) people, and operations people. It is also asserted that creating a cross-functional team (e.g., operations team is completely embedded in development team) necessitates highly skilled people and this pattern has usually been found in Start-up or highly innovative web companies [125, 126].

> *"Initially we had separated operations team. There was a huge concern from the business because people came from IT background, you had to have developers who were far away from the production. It's risky stuff and we had to change this mindset and in about three years we moved to the cross-functional team where operations were part of the team"* ***P14***.

A small number of the interviewees emphasized that if organizations want to efficiently adopt and implement DevOps practices, in particular CD practices, they cannot really have operations silo (i.e., separate Ops team), even small one. Having operations silo may lead to a lot of frictions in the deployment process and fail organizations to achieve the real anticipated benefits of CD practices.

Five respondents chose the "Other" field, but they did not provide a new pattern for organizing Dev and Ops teams. They mainly used this field to describe their team structures in other ways. Thus, we assigned them to existing categories. For example, **R71** stated that

> "*We have 2 Web Ops teams serving > 30 Dev teams. The Web Ops provide infrastructure, tooling, and deployment, but the Dev teams are monitoring and managing their own services in production*".

5.4.1.1 Team Structures for Designing Pipelines

It is asserted that the success of adopting DevOps practices (e.g., in particular CD) in organizations would heavily depend on the choice of appropriate tools, technologies, infrastructures, and level of automation to implement Continuous Deployment Pipeline (CDP) or also known as continuous delivery pipeline [12, 78]. It is worth noting that organizations used different terminologies to refer to CDP. We observed that the participants' organizations adopted the following models to introduce CDP:

Organization-driven model: Software organization may found a team to build and maintain platforms, infrastructures and toolchain (e.g., Jenkins and Chef) to set up a (semi-) automated CDP [132]. Then all project teams in the organization are able to use this CDP to build, test, package, and run their applications [7]. Having a common CDP enables organizations to improve consistency, governability and team productivity [133]. Among the 19 interviewees' organizations, we observed three different patterns for organizing that team. For all cases, first an organization builds CDP by applying one of the following patterns, and then multiple projects simultaneously are fed into and ran on established CDP.

Centralized Team: According to our participants, adopting and scaling CD practices at a large organization with multiple teams and applications necessitates a CDP that supports traceability, scalability, and flexibility [124]. The CDP must be able to perform no matter how large or many applications it processes, or how large their test suites are. It must also be flexible in a way that organizations can extend and tune it or parts of it without major disruption or major effort. Furthermore, the CDP should support traceability, which enables a wide range of stakeholders to understand what is happening, what has happened, and why. For some interviewees' organizations, that is achievable by establishing a dedicated and centered team to design, develop and continuously improve CDP in the long term. One of the interviewees told us:

> *"We had Squad that was responsible for basically taking care of the platform. … So, my colleagues, Squad was responsible for DevOps platform layer"* ***P6***.

This was the most commonly chosen pattern by the survey respondents, with 39.7% (37) choosing that a central team in their organizations designed a CDP that would work best for all teams and applications. We found that this model of forming CDP team mainly appeared and practiced in large (20) and medium-sized (12) organizations.

Temporary Team: In contrast to the previous pattern, CDP in this pattern is built by a temporarily established team in an organization and then the members of that temporary team join other teams because there is no need for them anymore. As one interviewee explained that:

> *"Once we have set up continuous integration, they would call a pipeline, once the pipeline there, and if there is no problem, we will go back to the pool and we don't stay all days."* ***P8***.

Figure 5.3 shows that there were 23 (24.7%) survey participants that indicated this pattern. We observed a fairly uniform distribution of this pattern across small and medium-sized organizations.

External Team: An external consulting organization helps both software provider and customer organizations by creating a customized CDP and then team members in the organization are trained to use and maintain that pipeline. Our results show that a few numbers of the interviewees' and survey participants' organizations sought external organizations for this purpose.

Team-driven model: In this model, each team in an organization builds and develops their own pipeline to adhere to the needs of the team and project. This model was mainly found by the survey results. When we asked the participants about the formation of a CDP team, the "Other" field was also considered to gather more patterns. 22 survey respondents indicated that their organizations followed this model (i.e., individual team in Figure 5.3). As stated by **R89**, "*Each team has organized their continuous delivery pipeline*", and **R47**, "*Various pipelines are built by engineers and used by themselves*". We also found that in some organizations a central team provides consultancy to all project teams to help project teams to build and manage their own CDP ("*Each team builds its own pipeline with help from a central team*" ***R9***).

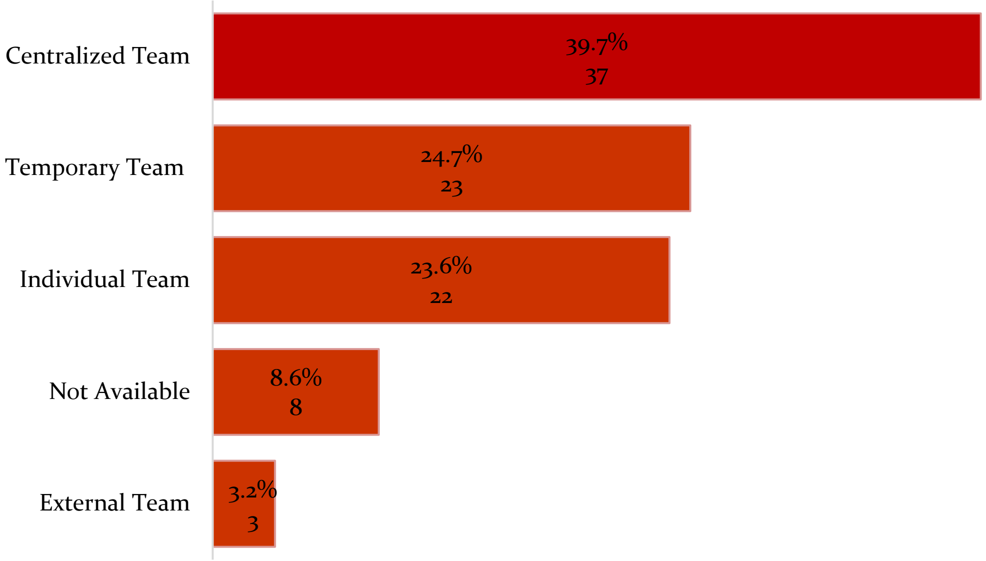


Figure 5.3 Survey results on CDP team patterns (n=93)

5.4.2 Collaboration (RQ5.2)

Based on the analysis of the interviews' data, we found that besides changing the team structures, organizations are increasingly improving collaboration among teams and team members to effectively initiate and adopt CD practices. We asked the survey respondents to rate how they strongly agree or disagree that the collaboration between teams (e.g., developers, quality assurance team, testers, and operations personnel) has increased in their respective organizations since the adoption of CD practices (See statements S1 in Figure 5.5). The results indicate that 73.1% of the respondents strongly agreed or agreed with this statement; only 6.4% of the respondents indicated disagreement with the statement S1, and none of them disagreed strongly.

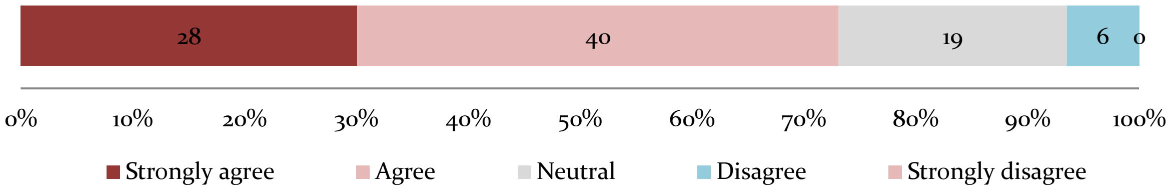


Figure 5.5 Statement S1: The collaboration between team members has increased in my organization since the adoption of CD

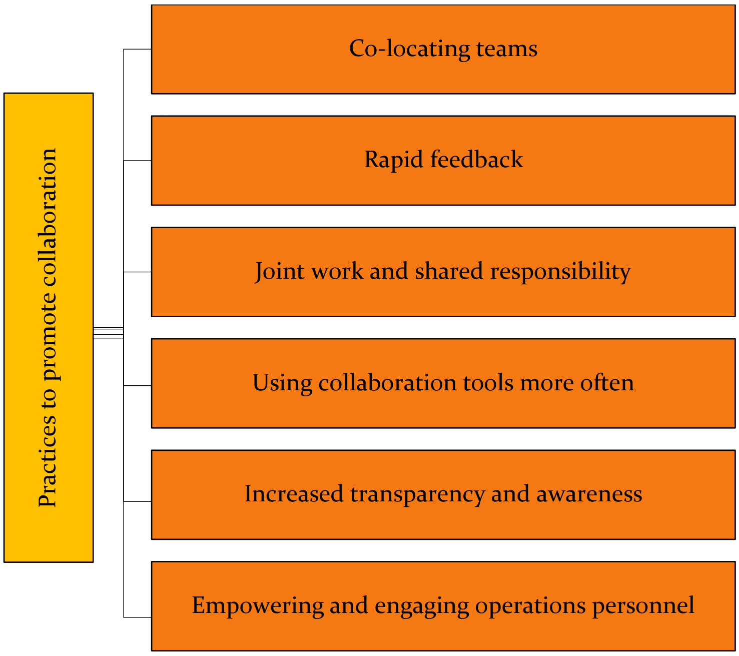

Practices to promote collaboration
Co-locating teams
Rapid feedback
Joint work and shared responsibility
Using collaboration tools more often
Increased transparency and awareness
Empowering and engaging operations personnel

Figure 5.4 The practices to promote collaboration

Through a mandatory open-ended question, we investigated how organizations could foster collaboration among teams. The respondents were expected to specify strategies employed by their organizations for this purpose. The analysis of the provided answers revealed the following practices (See Figure 5.4):

Co-locating teams: The most common strategy to improve collaboration is co-locating teams and discuss, for example, operational issues more often before an application is released to production or customers (“*Dev/Ops/InfoSec team co-location*” ***R17***). The respondents revealed that adopting CD not only needs tighter collaboration between Dev and Ops teams, but also other teams need to be physically close to each other to enable face-to-face communication, faster and easier interaction and knowledge sharing (e.g., “*Placed hardware [team] along with software [team]*” ***R57*** or “*Analysis [team] next to developers*” ***R60***).

Rapid feedback: A few numbers of the participants emphasized that having shorter feedback loop at each stage in CDP enables teams and team members to partner in producing high-quality software. As described by **R19** “*the rapid feedback loop has allowed developers and testers to partner in producing high-quality software*”. This also allows them to significantly reduce the time between problem identification and problem-solving (e.g., “*Shorter loop from feedback to fixes bugs*” ***R38***).

Joint work and shared responsibility: Our results reveal that the speed and frequency demanded by DevOps and CD practices drive the need for a more holistic view, in which team members from each side of the fence are needed to jointly work together and adopt shared responsibility as much as possible. As this quote from **R28** shows: "*Developers are working with operations to make sure their concerns are addressed as part of CD pipeline (i.e., monitoring and health checks)"*. The respondent **R33** also confirmed this in the following quote*:*

> *"Speaking for a large organization containing dozens of individual cases, it is my experience, however, engineers from each side of the fence need to sit down and discuss what the pipeline actually looks like, what it should look like, and what their respective roles in that pipeline are/should be".*

Empowering the culture of shared responsibility is crucial to achieve CD, as shown in this example given by **R88**

> "*The team as a whole is responsible for the quality of the application, everybody does testing, and everybody solves operations problems*".

It is argued by the participants that this is a success of a team as a whole, any failure impacts each tier exponentially in terms of cost; hence, the aim is to minimize the failures, particularly before deployment process.

Several survey respondents explained that the overhead of collaboration was sustainably reduced in their organizations by incorporating testing as an integral part of the development team instead of testers just being as assessors. For example, **R79** pointed out that

> "*Many testers were trained to be developers and have become valuable members of staff, better developers than those with years of experience*".

As indicated by **R68**, a QA team should be paired with the development team to successfully adopt CD:

> "*CD asks for maximum collaboration between different disciplines of the system, a QA has to pair a lot with developers to understand the delivery to give the signoff*".

Collaboration and communications among team members can considerably increase by establishing cross-functional teams as explained by **R85**

> "*Cross-functional teams became the norm and communication and collaboration increased ten-fold, as teams became more self-organizing*".

Using collaboration tools more often: Several participants indicated that the use of communication and collaboration tools to drive collaborative works between teams has increased since CD adoption (e.g., "*Communication over Slack increased*" ***R84***). Several participants indicated that using common tools and processes across teams in an organization decreases the overhead of collaboration and communication. This enables teams to have cross-collaboration to refine work prior to releasing applications to customers or production environments. A program manager described this vividly:

> "*Information is being shared in many ways across them [Dev and Ops] and sharing the same Wiki for example in terms of they both get notified when changes are made to documents on the Wiki, using the same JIRA system*" ***P7***.

Increased transparency and awareness: During the interviews, we found that the lack of suitable awareness on the status of the project (e.g., build status, release status) among team members can be a bottleneck for collaborative work and significantly hinders the CD success. To better understand this challenge, we asked the survey respondents to rate the severity of this challenge through a five-point Likert-scale question.

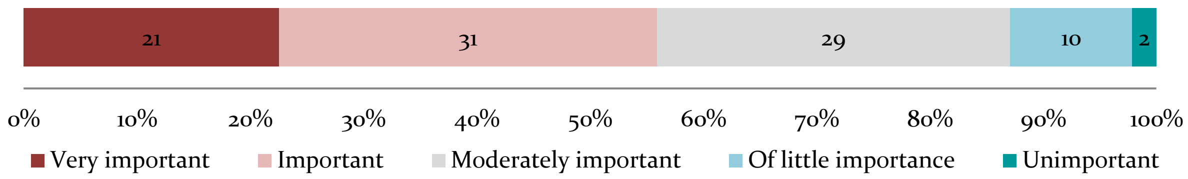


Figure 5.6 Statement S2: How important is "*lack of suitable awareness on the status of the project among team members*" in adopting CD

As shown in Figure 5.6, 55.9% of the survey participants voted the statement S2 as *very important* or *important*. While only 12 out of 93 participants considered this challenge as *unimportant* or *of little importance*. Besides visibility of build results and test suites execution results, our survey participants emphasized that operations tasks and stuff should be visible and traceable to everyone in the team. For example, **R67** explained that:

> "*Operations teams have now multi-channel feedback to Dev team (Email, Call, Monitoring Dashboard, Alarms, and Reports)*".

R43 elaborated further this in the following words: "*The operations team has a daily meeting and a Kanban that other teams can go and interact with*".

Empowering and engaging operations personnel: Our survey data shows that the overhead of collaboration and communication between development and operations teams reduced by shifting some of the operations' responsibilities to development team (e.g., "*Development and QA teams are together in a cross-functional team. We also both perform an Op's function, through monitoring and analyzing production behavior*" ***R71***). This situation gives more freedom and time to Ops team to directly and freely collaborate with other team members as stated by **R56:**

> "*Most of the operations works have reduced and they are able to help Dev and QA as they are having [more] time to help*".

Giving more power to Ops personnel and engaging them in software development life cycle right from the beginning was referred by a couple of the survey participants as enablers for collaboration. For example, participant **R14** pointed out:

> "*Collaboration between these groups has been high as it's always my intention to involve these groups early in the project lifecycle as possible to ensure the correct parties have their say early in the solution*".

A number of the participants mentioned that the interaction between Ops stakeholders and other team members previously used to happen only during production deployment. It has been observed that Ops team became more interactive before each deployment after gaining a voice in development and deployment decisions and ability to influence on design and formation of CDP pipeline. One of the interviewees described this perfectly:

> "*I think what we tried to do is to let operations team not only be responsible for operations tasks, and they may also be injecting requirements into build cycle within the project. So they need to be empowered to have an equal voice on the team in order to represent their needs and the team is empowered and required to support those needs. This is unlike to work traditional model in the past, where the operations team was a separate team. In our model, the operations team was tightly integrated into the development, decisions and planning on the daily basis*" ***P6***.

5.4.3 Responsibilities (RQ5.3)

We observed that adopting CD practices changes the responsibilities of some team members. Rather sometimes there are more responsibilities that require the acquisition of new skills to align themselves with the spirit of CD practices. For example, **P9** highlighted this change as the follows:

> "*So, the real responsibility of deployment moved from these different departments to the development team. So, development teams become more and more a DevOps team. That's what we tried to do it step by step. So, for example, we also tried to integrate database persons in the team; all the database changes are now performed by Dev team*" ***P9***.

This means by adopting CD every function of an organization might be touched, not just development. We were interested in understanding the changes brought about by the adoption of CD in the daily work routine of team members. According to data from statement S3 in Figure 5.7, 56.9% of the survey participants indicated that their responsibilities have changed *somewhat*, *much*, or *very much*. However, of the 23 (24.7%) participants that responded to this statement as *not at all*, more than 60% introduced themselves as consultant or mentioned that their responsibilities have not changed because when they joined their current organizations, CD had already been implemented (e.g., "*No [change], when I joined CD was already adopted*" ***R89***).

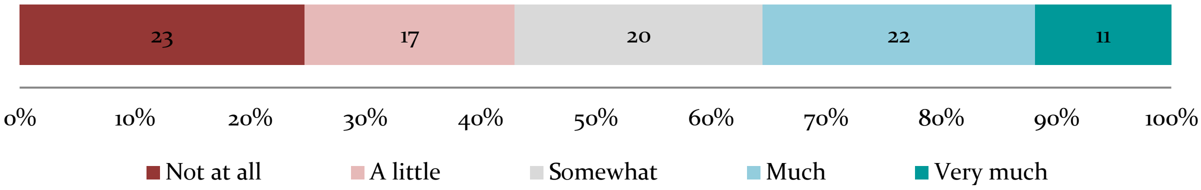


Figure 5.7 Statement S3: My responsibility has changed after our organization adopted CD practices

Through a follow-up question, we asked them to explain how their responsibilities have changed (e.g., what new skills they require for practicing CD) (See Figure 5.8).

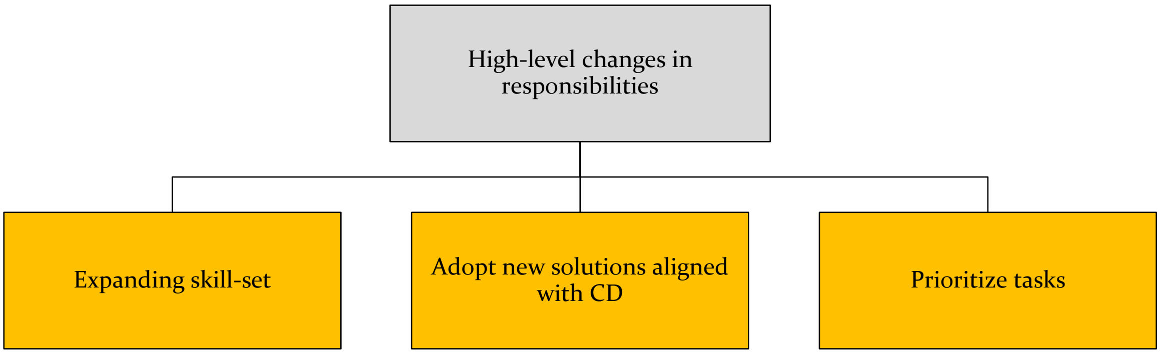

High-level changes in responsibilities
Expanding skill-set
Adopt new solutions aligned with CD
Prioritize tasks

Figure 5.8 Three high-level changes in team members responsibilities for practicing CD

Expand skill-set: Interestingly most of the respondents indicated that they have to constantly learn best practices and new tools for reliable release (e.g., "*[working in CD context] requires familiarity with cloud deployment tools*" ***R24*** or "*focus on tools of CI and CD*" ***R67***). In our survey, we perceived that the development team needs to significantly develop their operational skills as well. As the participant **R76** stated that

> *“Coming from a development side, I had to develop some "ops" skills. When your commit goes automatically to production, you have to care about security, on-call, and performance of your application”*.

One of the operational skills that mostly mentioned by the respondents was monitoring and logging skills. Working in CD context necessitates developing monitoring skills and spending more time on monitoring to triage and quickly respond to production incidents. As stated by two surveyed participants:

> *“Ensuring the product stays deployment-ready all the time. Each check-in and change gets monitored”* ***R20.***

> *“I have to be more watchful on the deliverables, more stress is on test automation”* ***R23.***

Scripting and automation skills were another skills that were referred by several survey participants (e.g., “*Scripting, deploying, automate everything instead of programming only*” ***R58***). We found that CD seeks new bureaucracies to access and manage production environments (e.g., “*Infrastructure and Platform now treated as code [in CD context] and environments defined at last minute*” ***R45***). This helped them to reduce security problems, avoid downtime in the production environment and better follow ITIL (i.e., Information Technology Infrastructure Library) in the transition towards CD practices. In addition, for some of the respondents adopting CD means to understand the whole stack of the application: database, backend, front-end, OS, and build. One of them stated this in these words: “*Skillset required has expanded to more of complete DevOps workflow*” ***R65***. This helped them to further and better be involved in bug fixing (e.g., “*More in-depth knowledge of the entire stack - to debug when something fails*” ***R38***).

Adopt new solutions aligned with CD: The findings from the interviews suggest that CD expands and changes the role of architect (e.g., “*You know in terms of overall architecture [for CD], it is not just to know about architecting actual product, it is about architecting the whole picture*” ***P7***) [134]. Our survey results were aligned with this finding; the role and responsibilities of software architects have significantly changed in CD context as only 6 out of 39 architects shared that their responsibility did not change at all. Architects are expected to define and design modern architectures that work with CD process of their organizations (i.e., CD-driven architectures). As explained by **R68**

> *“As an architect, I had to rethink on how we design the systems for continuous delivery”*.

Another participant validated this change through this quote:

> *“I have taken on completely new roles; leading architecture work to define internal services to enable these practices [CD practices]”* ***R33***.

Understanding and applying microservices architectural style and designing for different deployment models (e.g., Blue-Green deployment) were two main skills and changes reported by the architects to better support CD.

Prioritize tasks: CD greatly helps some team members to concentrate on more valuable tasks (e.g., “*[CD] allows me to focus more on solving business problems instead of release coordination and ceremony*” ***R6***). We also found that building high-quality applications to be deployed frequently and reliably may force team members to spend more time for standardizing their solutions and also improving confidence in the code. This is mainly achieved by performing excessive (automated) testing or shifting part of testing responsibilities to Dev team. One participant stated that

> *"My responsibilities have shifted from always being able to reproduce every version of code to a model where you always move forward. So I have to think about ways to give trust and confidence in code"* ***R11***.

Another example came from participant **R32**, a software engineer, who explained that online functional testing and deployment model checking were two of his new responsibilities towards CD. Furthermore, the focus has shifted more toward automating tasks as much as possible, for example, more concentrating on test automation, creating automated test-cases, and less on tracking down build failures in order to better allocate resources. As explicitly explained by participant **R48**

> *"My responsibilities are not to do "operations" anymore but to think how we are organized and find solutions to provide automation, security and quality for repeatable, and trustable deployments"*.

5.5 Discussion and Conclusions

DevOps paradigm stresses higher coordination and collaboration between members involved in software delivery to release high-quality software faster and more reliably [28]. It is asserted that collaboration is one of the key dimensions of DevOps [135, 136] for supporting the changes in organization' structure and culture as a result of adopting DevOps. In many organizations, Development (Dev) and Operations (Ops) teams form silos that are possibly located in separate department [136]. Whilst this structure was motivated by traditional methodologies (e.g., waterfall), it is not suitable for recent software development practices that simultaneously deal with agility, and maintaining software in different environments [135]. Iden et al. [137] highlight that effective cooperation between Dev and Ops teams has a great impact on the quality of the final product. Any shortcomings in the interaction of these members usually manifest in problems such as excluding IT operations from requirement specifications, poor communication and information flow, and lack of knowledge transfer [137].

Lwakatare et al. [138] indicate that collaboration can be enforced through practices such as broadening skill-set, information sharing and shifting responsibilities among these members. Nevertheless, implementing these practices demand changes in team structures, required responsibilities, the work culture of organization and mindset of team members [136, 138]. Some researchers have identified best practices to implement these changes (e.g., team structure). For example, Humble and Molskey [135] suggest re-architecting software product in form of strategic services and assigning each service to a small cross-functional team who takes the full ownership of it during the whole development lifecycle. Our research has been motivated by the need of empirically studying and understanding organizational practices promoting collaboration principle of DevOps. Our empirical study's findings have identified four key patterns for structuring Dev and Ops teams to effectively initiate CD practices.

The popularity and applicability of these patterns vary given the organizational context (e.g., hierarchies and size). There is a higher tendency among large organizations to initiate CD practices while maintaining separate Dev and Ops units. These organizations have promoted collaboration among their Dev and Ops teams through different means (e.g., collocating members and employing facilitator team); yet they have not drastically changed the organizational structure for breaking the silos. We have observed that small organizations have more flexibility and tendency to employ different patterns aimed at merging these units in form of unified multidisciplinary team(s). We have pointed out that this difference may be rooted in the challenges that organizations face in adopting CD ideally with unified multidisciplinary teams. We can enumerate some of these challenges: availability of highly skilled members to form the multidisciplinary team, the possibility of re-structuring units/ departments, (re) architect software product to independent units (e.g., services) for assigning to multidisciplinary teams and having a highly cooperative organizational culture for forming and running unified teams. We speculate that larger organizations may face more restrictions to change established practices and address these challenges. Future research can extend our findings and investigate the role of different contextual factors (e.g., global distribution of sites, business domain, and type of products) in adopting different team structures when moving to CD practices.

Our study has revealed several organizational practices improving collaboration among teams and team members to effectively implement CD. Our findings in this regard are aligned with the previous research [28, 135, 136] while providing significantly additional insights. It is evident that sharing the responsibilities of software delivery with *all* members [135] could promote coordination and collaboration in a team. We observed the practice of rotating roles [136] between developers and operations staff, i.e., involving developers in testing and QAs in development tasks. Our findings have highlighted the significant role of visibility and awareness of a project status for improving collaboration in a team and successfully adopting CD. Humble and Farley [28] recommend using big, ubiquitous dashboards in each team room to visualize the status of builds and sharing feedback with everybody. Our participants also indicated raising awareness in teams by involving operations staff in daily meetings [136] and interacting with Kanban board. We have discussed that collaboration among team members not only can be improved through processes but also by provisioning appropriate tool support. Some studies (e.g., [135]) have demonstrated that while organizations extensively utilize toolchains for building deployment pipeline, there is less focus on technologies facilitating communication and knowledge sharing in teams. Future research should explore the possibilities for promoting communication and collaboration through tools.

Implementing CD practices demands skill-set and knowledge that are either brand new (e.g., tools for automating CD process), or lie at the intersection of development and operations responsibilities. Similar to [136], our study reveals that adopting CD broadens the scope of responsibilities and skill-set. It is evident that these changes are particularly significant for developers who sometimes take larger shares in operations activities [136]. Whilst developers are expected to take an active part in deployment for successful adoption of CD, it should not demolish operations' functions. Broadening responsibilities of developers to a larger extent may negatively impact their productivity in core tasks. Shifting extensive amount of operations' responsibilities to developers could cause fear of losing jobs for Ops team and may negatively affect the success of the transition to CD. Organizations should extensively promote knowledge sharing among team members to complement areas of skill-set and collaboratively work towards a shared goal.

Chapter 6

Architectural Impact of Continuous Delivery and Deployment: Practitioners' Perspectives

Related publications:

This chapter is based on ESEM paper, "The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives" [2] and EMSE paper (minor revision), "An Empirical Study of Architecting for Continuous Delivery and Deployment [5].

In Chapter 3, we concluded that one of the fundamental limitations to rapid and safe deployment is rooted in the architecture of a system. However, only little is known about the role of software architecture in Continuous Delivery and Deployment (CD) and how an application should be (re-) architected to enable and support CD. This chapter aims to fill this gap through a systematic and rigorous analysis of the main part of the mixed-methods empirical study described in Chapter 2. We present a conceptual framework to support the process of (re-) architecting for CD. We provide evidence-based insights about practicing CD within monolithic systems and characterize the principle of "*small and independent deployment units*" as an alternative to the monoliths. Our framework supplements the architecting process in a CD context through introducing the quality attributes (e.g., resilience) that require more attention and demonstrating the strategies (e.g., prioritizing operations concerns) to design operations-friendly architectures. We discuss the key insights (e.g., monoliths and CD are not intrinsically oxymoronic) gained from our study and draw implications for research and practice.

6.1 Introduction

As we described in the previous chapters, given the increasing popularity of Continuous Delivery and Deployment practices (i.e., they are referred to CD practices in this chapter) [9], several efforts have been allocated to study and understand how organizations effectively adopt and implement CD practices. We discussed the importance of choosing appropriate tools and technologies to set up modern deployment pipelines to improve the software delivery cycle in Chapter 3 [1, 14, 27]. Along with other research [11], Chapter 5 argues that the highly complex and challenging nature of CD practices require changes in a team's structures, mindset, and skill set to gain the maximum benefits from these practices. As described in Chapter 3, the other area of interest in CD research is to support (semi-) automated continuous testing and increasing awareness and transparency in the CD pipeline [1].

On the other hand, it has recently been claimed that the fundamental limitations to adopting CD practices are deeply ingrained in the architecture of a system and these practices may bring substantial architectural implications [1, 7, 9, 33]. Whilst the industrial community through white papers and practitioners' blogs has investigated the role of software architecture in CD adoption [9, 37, 39], there is little empirical effort to study how software architecture is being impacted by or is impacting CD practices [38, 40, 139]. This is evident in the recently published systematic reviews on CD [1, 33, 34], in which a new line of research has been called to explore how an application should be (re-) architected for CD. Furthermore, to succeed in the DevOps/CD movement, which emphasizes on treating operations teams and operational aspects as first-class entities in the software development process, modern architectures should deal with both design and runtime considerations (e.g., predictive monitoring) [7, 140-143]. For example, software architecture in a CD context should ensure the desired level of quality attributes (e.g., deployability), and reduce the feedback cycle time from operations to development [75]. We assert that an appropriate software architecture (SA) is required to maximize the potential benefits of CD practices. To characterize CD-driven architectures, this chapter analyzes the main part of the mixed-methods empirical study described in Chapter 2. This chapter is guided by the following research question:

RQ6.1 How should an application be (re-) architected to enable and support continuous delivery and deployment?

Our results suggest monoliths and CD are not intrinsically oxymoronic. However, adopting CD in this class of systems is more difficult due to the hurdles that they present in terms of *team autonomy*, *fast and quick feedback*, *enabling automation* (e.g., test automation) and *scalable deployment*. To that end, the principle of "*small and independent deployment units*" is used as an alternative to the monoliths by some participants' organizations. We find that *autonomy* in terms of *deployability*, *modifiability*, *testability*, *scalability*, and *isolation of business domain* are the main characteristics of this principle. With that information, we intend to support organizations to move from a monolith to "*small and independent deployment units*". We also discuss quality attributes including deployability, modifiability, testability, loggability, monitorability, and resilience, which require more attention when designing an application in a CD context. We provide concrete examples of these quality attributes in action and discuss their role for CD success. Finally, we demonstrate three strategies (e.g., prioritizing operations concerns) suggested by our participants to design operations-friendly architectures. This chapter makes the following contributions:

(i) One of the largest empirical study, with 100+ experts, investigating the state of the practice of architecture side of CD;

(ii) A better understanding of the characteristics of CD-driven architectures;

(iii) A catalogue of 23 empirically-justified findings that can be used to help organizations and practitioners to adopt CD and to guide them in creating CD-driven architectures;

(iv) A conceptual framework to support the process of (re-) architecting for CD;

(v) Concrete and actionable recommendations for both practice and research.

Chapter organization: Section 6.2 summarizes prior related work. Section 6.3 presents our research method. We present the current state of CD practices in the participants' organization in Section 6.4. The quantitative and qualitative results are described in Section 6.5. Our discussion and reflection on the findings are presented in Section 6.6.

6.2 Related Work

The related work is divided into the general literature on CD practices and specific literature on the role of SA in CD adoption.

6.2.1 General Literature on CD

Primary Studies on CD: Transition to CD practices is a nontrivial process and necessitates technical, cultural, process and tooling changes in organizations to support the highly complex and challenging nature of these practices. Adams and McIntosh [98] characterize a modern release engineering pipeline by defining six major phases (e.g., infrastructure-as-code), which can help to ease the adoption of modern release practices (e.g., CD practices). The study [98] argues that whilst the industry has started widely implementing these practices, the empirical evaluation of these practices is in the nascent phase. A number of studies have examined the challenges, pitfalls, and changes that organizations may have experienced in CD adoption and/or adopted practices for this purpose. Claps et al. [23] studied the technical and social challenges of adopting continuous deployment in a single case software company. Based on the analysis of the data gathered from 20 interviews with software practitioners in the case company, they identified 20 challenges such as team experience, team coordination, continuous integration, infrastructure, and partner plugins. They reported on the strategies (e.g., rigorous testing for databases) adopted by the case company to alleviate the challenges. The findings of this study reveal that adopting continuous deployment necessitates changing team responsibilities, greater team coordination and involves several risks.

Savor et al. [22] present an experience report of implementing continuous deployment at Facebook and OANDA. They reveal that despite the tremendous increase in the number of team members and the complexity of code size over six years, continuous deployment did not negatively affect the team productivity (i.e., lines of added or modified code deployed to production per developer) and software quality (i.e., number of production failures). Furthermore, the study distills some issues such as management support and extra effort for understanding updates that an organization may encounter during the journey towards continuous deployment adoption. Two other studies [11, 14] report the benefits that CD practices provide for software development organizations including reduced deployment risks, lower development and deployment costs, and faster user feedback. A number of obstacles and challenges (e.g., resistance to change, customer preference, and domain constraints) to CD are also reported in [11, 14], which are in line with the challenges reported by Claps [23]. Apart from the challenges reported in [11, 14, 23], the stage-gate process is also a roadblock to continuous delivery success [24]. A stage-gate process has several quality gates to ensure the quality of new releases before entering the next stage. Laukkanen et al. [24] indicate that tight schedules, process overheads and multiple branches, which are associated with a stage-gate process, make it almost impossible to adopt the continuous delivery practice in a stage-gate managed organization.

Literature Reviews on CD: Recently, there have been several reviews published on CD practices [1, 33, 34] and rapid release [83]. Mäntylä et al. [83] conducted an SLR aimed at identifying the benefits (e.g., customer satisfaction), enablers (e.g., tools for automatic deployment) and challenges (e.g., time pressure) of rapid release (including CI and CD). Based on 24 primary studies, the review concludes that the rapid release is a popular practice in the industry; however, there is a need for empirical studies to prove the claimed advantages and disadvantages of rapid release. Three recently published reviews [1, 33, 34] on CD practices mostly focused on the issues that hinder adopting CD, along with solutions to address those issues. Some of the major stumbling blocks to CD are rooted in the system design, production environment, and testing practices. Furthermore, it has been revealed that the solutions to testing and system design problems in CD are rare. In Chapter 3 [1], we also defined the critical factors that a given organization needs to consider carefully along the CD adoption path, including testing (effort and time), team awareness and transparency, good design principles, customer satisfaction, highly skilled and motivated teams, application domains, and appropriate infrastructure(s). Rodríguez et al. [34] identify 10 factors such as fast and frequent release, continuous testing and quality assurance, and the configuration of deployment environments, which together characterize CD practices. The configuration management process in CD refers to storing, tracking, querying, and modifying all artifacts relevant to a project (e.g., application) and the relationships between them in a fully automated fashion [28].

6.2.2 Architecting for CD Practices

The software architecting process aims at designing, documenting, evaluating, and evolving software architecture [144]. Recently, Software Architecture (SA) research has been experiencing a paradigm shift from describing SA with quality attributes and the constraints of context (i.e., the *context and requirement* aspect) and structuring it as components, connectors and views (i.e., the *structure* aspect) to focusing on how stakeholders (e.g., the architect) make architectural decisions and reason about chosen structures and decisions (i.e., the *decision* aspect) [142, 145]. Whilst current research and practice mostly consider architecting as a decision-making process [146], it is been recently argued that SA in the new digitalization movements (e.g., DevOps) should cover the *realization* aspect as well [142]. The *realization* aspect of SA design is expected to deal with operational considerations such as automated deployment, monitoring, and operational concerns. However, the SA research community has provided few guidelines and systematic solutions for this aspect of software architecture [143]. Accordingly, our findings in this chapter are placed in all four aspects of software architecture in the context of CD.

Nonetheless, whilst works indicate that an unsuitable architecture would be a major barrier to CD transition [1, 24, 33], there has been little empirical research on the role of SA as a contributing factor when adopting CD practices. Some initial efforts on this topic have been reported in [38, 40, 75, 147, 148]. Mårtensson et al. [148] have conducted a case study to explore the behavior of developers in CI practice in two case organizations. They identify 12 enabling factors (e.g., "work breakdown" and "test before commit") impacting the capability of the developers to deliver changes to the mainline. The study [148] argues that some of these factors (e.g., work breakdown) can be limited by the architecture of a system. Chen [38] reports on the experience of architecting 25 software applications for continuous delivery. The study indicates that continuous delivery creates new challenges for architecting software applications. According to Chen, continuous delivery heavily influences a set of Architecturally Significant Requirements (ASRs), such as deployability, security, modifiability, and monitorability. It is asserted that the maximum benefits of continuous delivery are achieved by effectively satisfying the aforementioned quality attributes. Bellomo et al. [75] studied three projects involving continuous integration and delivery to understand deployability goals of those projects. The study [75] reveals that most of the decisions made to achieve the desired state of deployment (i.e., the deployability quality attribute) were related to the architecture of the systems in those projects. Based on the study's findings, the authors formed a deployability tactics tree.

Schermann et al. [40] conducted an empirical study to identify the current practices and principles in the software industry to enable CD practices. They observed that A/B test and dark launches as practices of CD are not often applied in industry and the feature toggles technique may bring unwanted complexity. Recently, a few of academic and white papers have discussed microservices architecture as a first and promising architectural style for CD practices [111, 147, 149]. Microservices architecture aims to design software applications as a set of independently deployable services [111, 150]. Balalaie et al. [147] report an experience of migrating a monolith to a microservices architecture. Apart from changing team structures (e.g., forming small cross-functional teams), they applied a number of migration patterns (e.g., change code dependency to service call) to decompose the monolithic system into microservices [147]. Furthermore, the migration process included introducing new supporting components and using containerization to support CD practices.

6.3 Research Method

As described in Chapter 2, a mixed-methods empirical study was adopted to answer the research question. The findings of this chapter come from 21 interviews and a survey of 91 practitioners.

6.3.1 Interviews

The interview study for this chapter involved 26 open-ended questions. After asking demographic questions (e.g., role and experience years), the interviewees were asked to share the challenges, pitfalls, and the changes that CD practices may have brought to the architecting process, and the architectural principles and practices that they used to address them. Then, we asked them to share how they took into consideration the operations teams and their concerns in their respective development processes. At the end of each interview, we asked the interviewees to share any other comments and potential issues regarding the questions. We initiated the analysis by creating three top-level nodes in NVivo: (1) the challenges and pitfalls that the interviewees faced at the architecture level in transition to CD; (2) the architectural principles, solutions, and practices they employed to better support CD; and (3) the strategies and practices that the interviewees' organizations employed to treat operations team and their concerns as first-class entities.

6.3.2 Survey

Like the interview guide, the survey targeted the questions about the participants' background, the impact of CD on the architecting process, and operational aspects. Apart from demographic questions, the survey questions used for this chapter included five-point Likert-scale (25 questions), multiple-choice (1 question), single-choice (2 questions) and open-ended (2 questions).

6.4 Practicing Continuous Delivery and Deployment

We asked both the interviewees and the survey participants to indicate how often, on overage, the applications in their respective or client organizations are in the deployable state (i.e., implementing continuous delivery) and how often they deploy the applications to production (i.e., implementing continuous deployment). These questions were designed, to a large extent, to determine the maturity of implementing continuous delivery and deployment practices (i.e., how an organization adopts and implements continuous delivery and deployment) [11].

It is clear from Figure 6.1 that 57.1% of the interviewees (12) and 54.9% of the survey participants (50) indicated that on average the applications in their respective or client organizations were in a releasable state *multiple times a day* or *once a day*. This number for continuous deployment practice was lower, as 7 interviewees (33.3%) and 34 survey respondents (37.3%) stated that they automatically deploy their applications *multiple times a day* or *once a day* to production. These results indicate that our findings came from reliable sources as the participants' organizations successfully implemented CD practices. Three interviewees (i.e., P11, P12, and P19) had no idea about how often the application changes were in the deployable state (i.e., shown as N/A in Figure 6.1). Interestingly, 6 participants indicating the changes were production-ready at least *a few times a month* had actual production deployment *a few times a year*. In Chapter 4, we argued that the reason for this may stem from factors such as domain constraints and quality concerns [3].

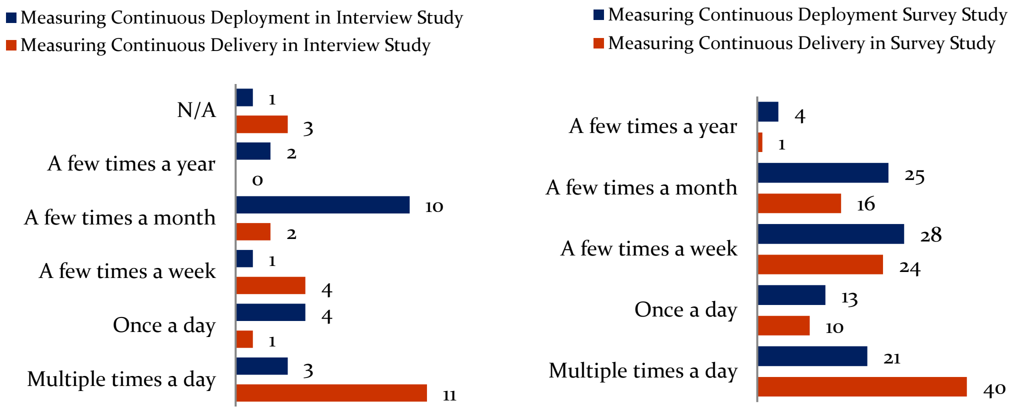


Figure 6.1 How continuous delivery and deployment are implemented in the interviewees' (left) and the surveyed participants' (right) organizations

6.5 Findings

In the following, we present a conceptual framework emerged from our findings to support (re-) architecting a system for CD (See Figure 6.2). We briefly introduce the framework here and provide details in the corresponding sections. The framework consists of five main parts: *Monoliths*, *Migration*, *Small and Independent Deployment Units*, *Quality Attributes*, and *Operational Aspects*. *Monoliths* part is shown on the top left side of Figure 6.2, which reflects the possibility of practicing CD within monoliths with potential challenges that may hinder CD adoption in this class of systems (See Section 6.5.1). Since monolithic architecture is predominant in software industries [31], there may be organizations that want to achieve CD with their monoliths. To this end, they need to augment/improve the architecture of their systems by applying the practices and strategies presented in *Quality Attributes* and *Operational Aspects* parts. *Quality Attributes* and *Operational Aspects* have the main goal of creating a CD-driven architecture apart from the chosen architecture style. It is worth mentioning that this is mainly because the practices and strategies in these two parts were often reported by our participants as prerequisites to, or supportive of CD-driven architectures. We show this fact in the framework by indicating *Operational Aspects* can be injected into the architecting process in the context of CD and *Quality Attributes* serves as input to both *Monoliths* and *Small and Independent Deployment Units*. *Operational Aspects* (left bottom side of Figure 6.2) provide the strategies to design operations-friendly architectures (See Section 6.5.4). At the bottom of Figure 6.2, we have the *Quality Attributes* that need to be carefully considered to design CD-driven architectures (See Section 6.5.3). We find that these quality attributes impact the architecture in the CD context in two dimensions: positive (+) and negative (-).

The challenging nature of the monoliths may compel organizations to move from monolithic systems with long development and deployment cycles to *Small and Independent Deployment Units* (top right side of Figure 6.2). The framework supports organizations in this journey by providing a list of reliable factors, shown as *Migration* in the middle of Figure 6.2. They can be used to characterize *Small and Independent Deployment Units* (See Section 6.5.2.1). The migration journey results in vertical layering or microservices (See Section 6.5.2.2).

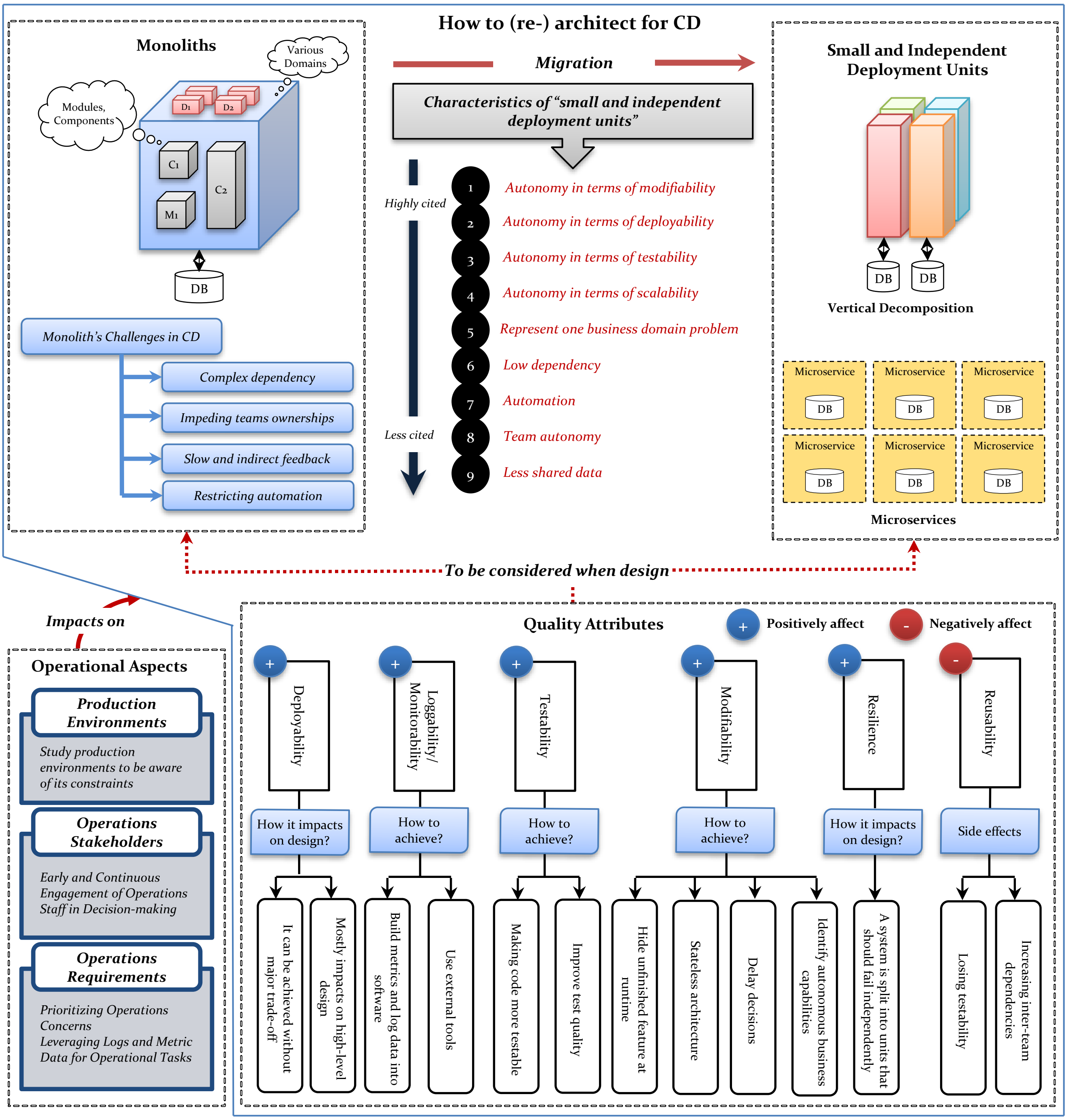

How to (re-) architect for CD
Monoliths
Various Domains
Modules, Components
D1
D2
C1
C2
M1
DB
Monolith's Challenges in CD
Complex dependency
Impeding teams ownerships
Slow and indirect feedback
Restricting automation
Migration
Characteristics of "small and independent deployment units"
Highly cited
Less cited
1 Autonomy in terms of modifiability
2 Autonomy in terms of deployability
3 Autonomy in terms of testability
4 Autonomy in terms of scalability
5 Represent one business domain problem
6 Low dependency
7 Automation
8 Team autonomy
9 Less shared data
Small and Independent Deployment Units
DB
DB
Vertical Decomposition
Microservice
DB
Microservices
To be considered when design
Impacts on
Operational Aspects
Production Environments
Study production environments to be aware of its constraints
Operations Stakeholders
Early and Continuous Engagement of Operations Staff in Decision-making
Operations Requirements
Prioritizing Operations Concerns
Leveraging Logs and Metric Data for Operational Tasks
Quality Attributes
+ Positively affect
- Negatively affect
+ Deployability
How it impacts on design?
It can be achieved without major trade-off
Mostly impacts on high-level design
+ Loggability/ Monitorability
How to achieve?
Build metrics and log data into software
Use external tools
+ Testability
How to achieve?
Making code more testable
Improve test quality
+ Modifiability
How to achieve?
Hide unfinished feature at runtime
Stateless architecture
Delay decisions
Identify autonomous business capabilities
+ Resilience
How it impacts on design?
A system is split into units that should fail independently
- Reusability
Side effects
Losing testability
Increasing inter-team dependencies

Figure 6.2 A conceptual framework of the findings showing how to (re-) architect for CD

6.5.1 Monoliths and CD

In its non-technical definition, a monolith is defined as "*a large block of stone*". Exploring literature and practitioners' blogs shows that there is no common understanding and interpretation of the term monolith as different types of monoliths may be created during software development lifecycle: for example, monolithic applications, monolithic builds and monolithic releases [151]. In our study, we define a monolith as a single executable artifact that contains various domains, layers, and many components, modules, and libraries, in which all the functionality is managed and packaged in one deployable unit (See Figure 6.2) [152-154]. Our interview study demonstrates that monolithic architectures are seen as a major problem in organizations during CD adoption journey. The following is a representative quotation about the negative impressions given of the monoliths:

> "*I do not know [if] they [the architects] are aware of that; they create kind of monolithic applications and the monolithic application contains large functional domains [that] would be hard to use in a large-scale organization if you want to have continuous delivery*" ***P10***.

Whilst the monolithic architecture, as a barrier to CD, was one of the most commonly occurring codes in the interviews (i.e., indicated by 15 interviewees, of which 6 of them were in the role of architect), we also found empirical the evidence in the interviews that CD can be implemented in the monoliths. Particularly, when there is a loosely coupled and modular architecture with clearly defined interfaces and one single team working on. As an example, we have:

> "*I have seen [examples of adopting] CD in monolithic applications, when they [organizations] don't split them and they go very fast because [there is] still a single team, they can be extremely fast, high-quality. The software is modular, and it is not split, and it works well for them*" ***P14.***

Furthermore, while conducting the survey, we received comments from the survey respondents that they had been successful in adopting CD within monolithic applications (e.g., *"A monolithic service can use CD"* ***R44*** and "*Microservices are not required nor is breaking up a monolith [for CD]*" ***R22).*** All this motivated us to add a new statement to the survey: "*It is possible to practice CD successfully in monolithic applications*". Figure 6.3 shows that 59.5% (25 out of 42) of the survey respondents answered this question as *agree* or *strongly agree*. Only 21.4% responded *strongly disagree* or *disagree* and others (i.e., 19%) took a *neutral* position in this regard (See statement S1 in Figure 6.3). It is interesting to note that among 25 respondents who believed in the possibility of implementing CD within monoliths, only 8 were in the role of architect. 6 architects (strongly) disagreed with the statement S1 and 2 adopted a neutral position. Team leads, consultants, and developers were the roles that were the most positive about monoliths. We assert that this finding is not necessarily in conflict with the interviews' findings as we did not conclude that it is *impossible* to practice CD within monoliths, but it seems that it would be much harder and more complicated to achieve CD within monolithic applications (e.g., "*Any component or service can adopt CD, a larger one [has] a slower cycle. Components without tests cannot adopt CD"* ***R85***).

In line with the work by Schermann et al. [40], our interview study shows that once the size of an application grows (e.g., by expanding its functionality) and the number of teams increases, the monoliths significantly impeded achieving scalable continuous deployment (e.g., "*The main challenge is the weight of your architecture, it could be the reason why you can't move to CD*" **P15** and "*We had to change the structure of our business as we had a large monolithic codebase and it was hard to work on when you have 50 or 100 people working on the same codebase*" **P14**). This is mainly because the monoliths may slow down the deployment process and a small change may necessitate rebuilding, retesting and redeploying the entire application [155, 156]. The next subsection lists the main categories of challenges about monoliths, which together hinder achieving CD.

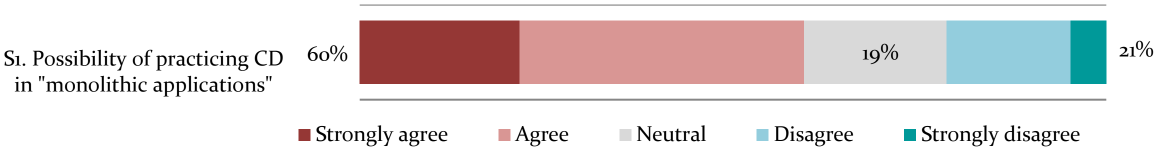


Figure 6.3 Survey responses to the statement on the possibility of practicing CD within the monoliths (n=42)

6.5.1.1 Why it is difficult to practice CD within the monoliths

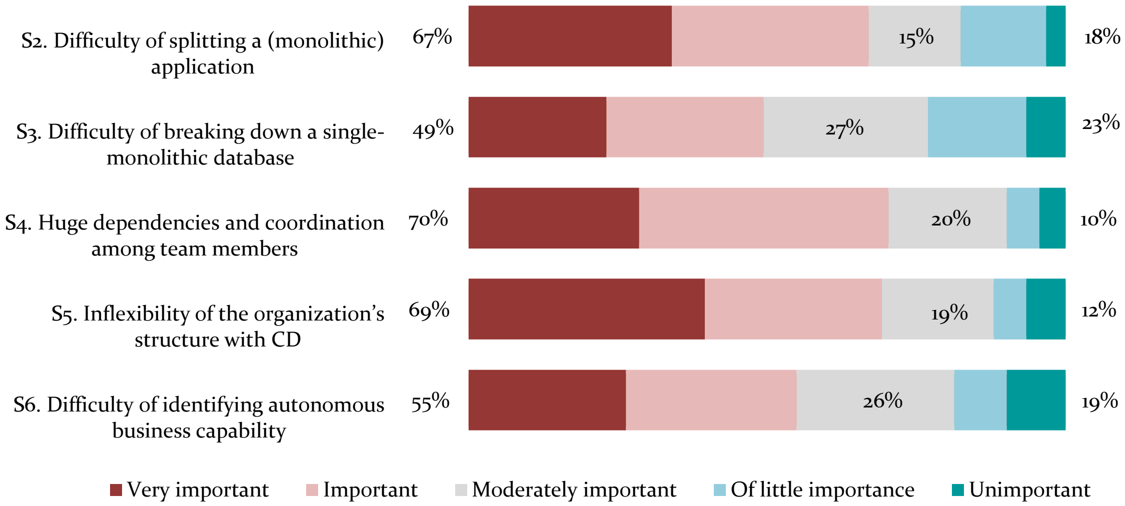


Figure 6.4 The survey respondents indicated the most important challenges in architecting for CD. The inflexibility of the organizational structure with CD is the most critical barrier for implementing CD.

C1. Dependency (Tightly Coupled): As the size and complexity of a monolith grows, the dependencies across the system will be so strong that it will become difficult to change different parts of the monolithic system independently. This impedes deploying software on a continuous basis as there is a need to thoroughly analyze and maintain all the dependencies in the deployment process [157]. One interviewee described this point vividly,

> "*When you say we have software component X and I would like to deploy that to the customer; do I need to deploy other software components as well; do I really need to or not? These are the design issues that we really face*" ***P9***.

External dependencies with other applications were another roadblock to frequent deployment (i.e., also confirmed by 24.2% of the survey respondents). Whilst an application might always be at releasable state, deploying a modified functionality within that application on a continuous basis may also necessitate the deployment of all dependencies (e.g., dependent applications). Therefore, software organizations need to refactor other legacy applications and ensure that there is no integration problem in the deployment process. As explained by an architect,

> "*You always get integration challenges [in the deployment process] because you're mixing new code, new systems with old systems; you still rely on old systems to do something*" ***P13***.

Our study identifies challenges regarding (monolithic) databases in the context of CD. The interviewees' organizations often faced difficulties deploying monolithic databases continuously as modifying any functionality in an application demands changing and incorporating database schema as well. Therefore, they gradually become an operational bottleneck and an undeployable unit. The interviewees expressed dissatisfaction with monolithic databases, for example,

> "*One of the traditional [approaches] is that for big applications, they [organization] used one database. Then you know for every piece of functionality you change, you also need to incorporate database changes, you need to test the database. These are a source of a single point of [failure] for these big organizations, so they cannot continuously deploy the database. So, they need to refactor [it] and do changes [in] the software architecture*" ***P9***.

Our combined findings indicate that software organizations often need to split their existing monoliths into small deployable parts that can be maintained and deployed independently for supporting CD. However, the interviewees highlighted the difficulties of breaking down a monolith (at application and database level) into smaller units. Through the survey, we asked the participants the extent to which they understand this issue as a challenge. As can be seen in Figure 6.4, 82.4% of the respondents considered the difficulty of splitting a (monolithic) application into independently deployable and autonomous components/services (i.e., statement S2) as *very important*, *important* or *moderately important* challenge. Furthermore, the difficulty of splitting a single-monolithic database (i.e., statement S3) was also widely recognized by the participants, as only 23% of the survey respondents considered it as *unimportant* or *of little importance*.

C2. Team: According to Conway's law, the architecture of a system mirrors the communication structure of the organization that designed it [36]. Our analysis demonstrates that introducing CD is challenging in the organizations where multiple teams work on a monolithic application. We observed that *cross-team dependencies* could cause frictions in CD pipeline. While one team could have full control over its development process, run quality-driven builds and release its output constantly, they can still be dependent on the performance of other teams. Therefore, they can easily break the changes that other teams are working on. As **P12** explained,

> "*We had multiple teams working on the same [monolithic] codebase. We started noticing that it was really affecting our ability to deploy software [continuously], only because many teams were trying to push many things at the same time*".

The interviewees also referred to tension between software and hardware teams as a challenge to truly practicing CD, as software teams sometimes rely on infrastructure readiness at production,

> "*Integration issues were part of the deployment [process] and [we had to] deal with their [hardware team's] backend systems. They had their [own] infrastructure, [which] was not ready; they had a serious quality cycle. So, we had to go through this [cycle], which caused additional preparation work. It was exhausting [for us] when we [as the development team] were going to the deployment cycle. So, it was a challenge to adopt DevOps*" ***P5***.

Furthermore, the interviewees' organizations that had large teams working on the same codebase experienced difficulties in *coordinating* among interdependent teams and *planning* before each release. As shown in statement S4 in Figure 6.4, 70.3% of the survey respondents ranked the challenge of *team interdependencies* and *coordination effort* when adopting CD as *very important* or *important*. CD may also require changes in organizational structures in order to align them with CD throughput [158]. When we asked the respondents how important the challenge of "*inflexibility of the organization's structure with the spirit of CD practice*" is, 69.2% of them ranked this as *very important* or *important* (See statement S5 in Figure 6.4)

C3. Feedback: Difficulties in getting fast, direct, and optimum feedback on code and test results in the monolithic applications was another category of challenges. In most of the statements that shared with us, *long build time*, *long-running tests*, and *size of changes* in the monoliths were main causes of slow and indirect feedback. As **P12** explained,

> "*[In our monolithic architecture] we started noticing that the number of tests started loading up, the feedback cycle was becoming very slow. We arrived at a state where [we had] very little ownership of [things] like test failures. You would have taken like around 3 or 4 hours to get any feedback. You could see a lot of friction on the deployment pipeline itself*".

C4. Automation: A key practice in CD is automation [28]. A few interviewees 'experience suggested that the heavy-weight nature of monoliths can often be a challenge (e.g., extra effort is needed) to fully automate tests and deployment across different environments. For example, **P15** explained how this situation would result in a longer and slower CD pipeline and extreme difficulties to move to automation:

> "*If your monolithic architecture is to be complex, it may be hard to move to continuous delivery because the more components you have, the harder it is to install them, to deploy them, the pipeline will be longer, and harder to automate that*".

Key Findings

Finding 1. *Monoliths (application and database) are great sources of pains for CD success, as they are hurdles for having* ***team autonomy****,* ***fast*** *and* ***quick feedback****, enabling* ***automation*** *(e.g., test automation) and* ***scalable deployment****.*

Finding 2. *Both architects and other roles are likely to believe that implementing CD in monoliths is difficult. Furthermore, among other roles (e.g., consultants), architects are more pessimistic about CD success within monoliths.*

Finding 3. *Breaking down monoliths into smaller pieces brings more value and flexibility to CD adoption; however, the participants' organizations experienced it as a challenging process.*

Finding 4. *Implementing CD is a challenge where multiple teams are working on one monolithic application as they gradually become* ***dependent*** *and need to spend too much time to* ***coordinate*** *and* ***plan*** *the delivery process.*

Finding 5. *The inflexibility of organizational structure with CD is the most critical barrier for implementing CD.*

6.5.2 Moving Beyond the Monoliths

In the previous section, we discussed how practicing CD in large monolithic systems with multiple teams is not a straightforward approach, as it might have negative impacts on team autonomy, direct and quick feedback, and automation. In this section, we first characterize a key architectural principle adopted by the participants to address the reported monolith-related challenges and then investigate how the key principle is implemented in industry.

6.5.2.1 Characterization of an Architectural Principle: *Small and Independent Deployment Units*

Our findings agree with the argument of Lewis and Fowler [150], that a key architectural principle employed by the participants' organizations for successfully practicing CD is to design software-intensive systems based upon "*small and independent deployment units*" (e.g., service, component, and database). By 'architectural principle', we mean the fundamental rules and approaches that serve as a guide to architects to govern and reason architecture designs [159]. **P18** described that applying this principle enabled them to bring the small units into production independently,

> "*We are moving to smaller services, where microservices to my mind are a less interesting aspect, but the services are much smaller than in the past and they will be deployed into their own server and they [are] managed [e.g., deployed] by one team*".

As we described in Section 6.5.1, the participants' organizations had challenges with splitting the monoliths into smaller chunks. That is why they usually perform this process incrementally. As an example, we have,

> "*We started with a big monolith. To be honest, initially we simply split it into a number of smaller chunks; maybe like seven or eight. That already gives us more value [in the deployment process]. In the future we might split things further" **P18**.*

Gradually breaking down applications should avoid the difficulties of service granularity. **P13** shared that it is easier to initiate the decomposing process with few large services and incrementally decompose them to smaller, fine-grained units:

> "*If you are trying to go from monolithic to something like service-based or microservices, don't try to break it down into little tiny pieces first of all right away. You'll get struggles at service level, structurally you have to look at how coupled your components are*".

Nevertheless, the above principle leaves us with yet another question: what does it actually mean? To characterize this principle, we asked the interviewees a number of questions to elaborate on their perception from a small and independent deployment unit, which serves as the foundation for CD success. We found that the practitioners typically consider four main criteria to characterize small and independent units, and accordingly design applications or break down large components or monolithic applications for practicing CD. These criteria include autonomy in terms of **deployability** (e.g., "*[The monolith is split into] those small components [that] can be deployed and reproduced more quickly so as to map smaller iterative deliverables*" ***P5***), **scalability** (e.g., "*The majority [of criteria in decomposing process] was [to have] components that are being very scalable, which means that whatever you design and whatever you develop must be scalable*" ***P5***), **modifiability** (e.g., "*[The monolith is] split into components, so that changes are likely to influence just one component*" ***P11***), and **testability** (e.g., "*It can be tested/ qualified by itself, and won't break product or require other services/components to be pushed" **P19***). Furthermore, some other criteria emerged from data that were only described by a few interviewees. To give an example, for one of the interviewees *small* meant the maximum amount of work required for coding and testing a single feature should not exceed a three-day effort for one person.

We asked the survey respondents to rate these four top criteria (See statements S7-S10 in Figure 6.5) with a 5-level Likert-scale. The results show that, in general, the respondents considered all the above-mentioned criteria for this purpose. As shown in Figure 6.5, for each of the statements fewer than 10 respondents *disagreed* or *strongly disagreed*. However, the statement S7 (i.e., "a component/service is small if it can be modified (changed) independently") received the most attention, and 78%, 76.9% and 68.1% of the survey respondents answered to statements S9, S8 and S10 *strongly agree* or *agree* respectively.

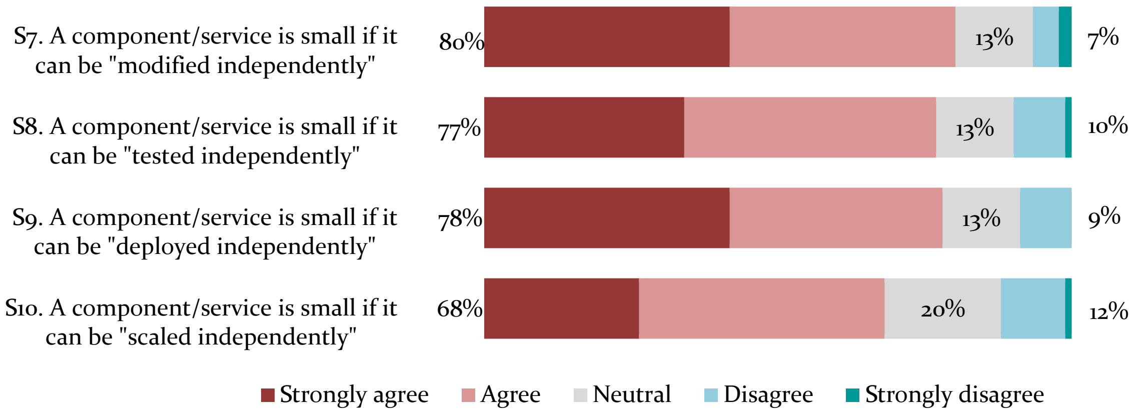


Figure 6.5 The main characteristics of the principle of "small and independent deployment units"; A few survey participants (less than 12%) disagree with the above-mentioned criteria.

Through a mandatory open-ended question, we also asked the surveyed practitioners to share any other criteria or factors (i.e., those not covered by statements S7-S10) that need to be considered to characterize small and independent units for CD and also to decompose the monoliths for this purpose. Some of the respondents tried to elaborate what they had previously chosen in the Likert-scale questions (e.g., "*It [the service] can be deployed/updated independently of any other service*" ***R84***, "*Every feature (roughly mapping onto an Epic) should be small enough to be deployed independently*" ***R31***). The analysis of the open-ended question revealed five other criteria or factors as listed below, in order of their popularity (See Figure 6.6):

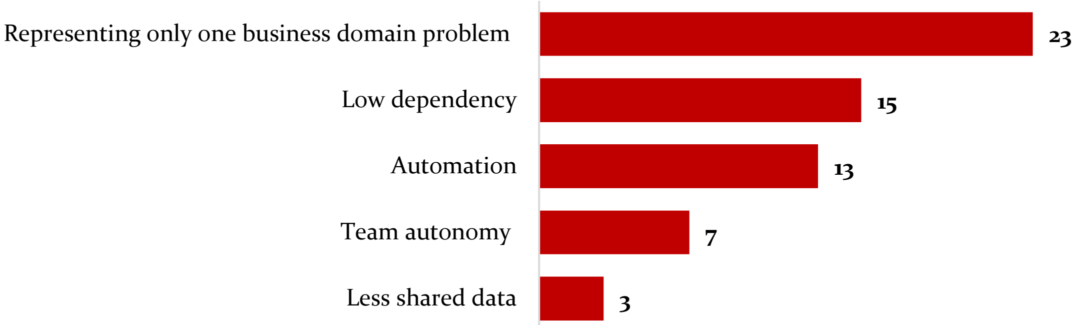


Figure 6.6 The additional criteria/factors shared by the survey participants to characterize "small and independent deployment units"

Representing *only* one business domain problem (23): Although a few interviewees indicated that monoliths need to be decomposed into smaller units, so they only cover and solve one business domain problem, this criterion was highly cited by the survey participants. For example, **R88** explained, "*A component [in a CD context] should implement a frontend and backend for one business concept*". We found that the definition of *one domain problem* varied significantly among the respondents. For some respondents, it means one function or one task (e.g., "*[A small deployment unit in CD] should encapsulate one unit task end-to-end*" ***R20***), however, others believed that it can serve a logical set of functionalities for a line of business capability, e.g., "*One [service] that solves one specific domain problem, not too small to be just a function, not bigger to* *try to solve different domains business*" ***R43*** and "*It [the service] should manage one aspect or collection of features [within] a bounded context*" ***R41***.

Low dependency (15): Some participants stated that the size of a component/service is not an important factor for them, but a component or service should have little dependency on other components/services, in particular during the deployment process: "*[It is] less about size (e.g., lines of code), [but I think] more about loose coupling and clear implementation for component API's, such as RESTful interfaces*" ***R46***. Respondent **R91** emphasized that a service in a CD context needs to be decoupled from other services in such a way that the upgrade and downgrade of the service would have no impact on SLAs and ongoing transactions. This factor can be augmented by implementing a well-defined interface for each service or component (e.g., "*[A component should be a] small code base, independent of all other services except via API calls*" ***R72***).

Automation (13): Another factor that emerged from the responses was that a small and independent unit can be tested and released in an entirely automated fashion within a time-boxed slot, in which it does not require to coordinate with external components/services: "*It must be designed with the intention of being easily built and automatically tested and deployed*" ***R53***. The respondents shared that CD essentially requires components or services with fast build times and fast testing loop in order to have quick and direct feedback, e.g., "*The service should be built and tested in less than a couple of minutes*" ***R76***.

Team autonomy (7): We perceived that having team-scale autonomy strongly affects the size of a component and decisions during decomposition. The respondents shared that adopting CD practices for a component/service is easier if one team can comprehend, build, test and deploy it (e.g., "*[A component should be] small enough for the owning team to comprehend it*" ***R9***). Another participant answered as "*A team should be able to own a backlog of multiple deployable [items]*" ***R30***.

Less shared data (3): A few participants also reported that having separate data storage per service or component (i.e., no common database) can help successfully implement CD. Respondent **R61** stated that a service should have "*no shared storage (with other services)*", while another respondent pointed out that "*my definition isn't about the size, it's [about] the boundaries and responsibilities, including what data it owns versus what data it uses*" ***R39***.

Key Findings

Finding 6. *The participants' organizations are increasingly considering* ***small and independent deployment units*** *as a key architectural principle to provide them with more flexibility in a CD path and achieve frequent and reliable deployments.*

Finding 7. *Autonomy in terms of* ***deployability***, ***modifiability***, ***testability***, ***scalability***, *and isolation of the* ***business domain*** *are the main factors to characterize the principle of "small and independent deployment units", and they significantly drive decomposing strategies to* ***safely*** *and* ***incrementally*** *break down a monolith into smaller independent parts.*

6.5.2.2 How is the key principle implemented in the industry?

To embrace the principle of "small and independent deployment units", the participants' organizations usually adopted two approaches: vertical layering and microservices (See Figure 6.2). Whilst these approaches might be used in many different scenarios to reach different goals (e.g., scalability), we examine them from the perspective of CD practices (i.e., promoting delivery speed). We note that vertical layering and microservices share many common characteristics. As shown in Figure 6.2, both representing independent deployment units, but the main difference between them is the level of granularity. Vertical layering is more coarse-grained than microservice [153, 160]. Verticals are autonomous systems that are larger than microservices. Verticals are mainly used to reduce operations' complexity, as there would be fewer number of deployment units. Another notable difference is that vertical layering allows sharing of some assets such as databases and infrastructures, as exemplified by **P17**,

> "*Every vertical slice would have talked to the same databases in the same way unless we explicitly decided that we didn't want one service talking to a specific backend. It was a hybrid model that worked very well for us*".

Vertical layering (decomposition) was a significant approach attempting to embrace the principle of "*small and independent deployment units*". Through this approach, a software application or a large component is decomposed into vertical layers (i.e., independent, autonomous systems) rather than horizontal layers; accordingly one team (ideally) would be responsible for one layer or component during the full development lifecycle (e.g., "*We are changing the paradigm of a more horizontal layered architecture to a more vertical one and isolating architectures, as it will help us to get this performance [deployability]" **P9***) [153]. Vertical layering also decreases interdependency among teams as a team minimally depends on what other teams are working on or are responsible for. Furthermore, this approach hampers the ripple effects resulting from any changes. Nevertheless, it was found that adopting this approach was associated with challenges in the formation of one integrated development team comprising all required skills (e.g., operations skills) as there is a tendency in software organizations to team up those human resources with similar skill sets. Here are just a few of the examples indicating the benefits of this approach for CD:

> *"We had separated layers in the architecture. If there are three teams responsible for one of these layers, all teams need to work together to bring a new piece of functional alive [in production]. To make it better [the delivery process], we swapped the organization and now each team is focused on one functional domain and works in its own layer. So, they are working in large isolation and we minimize the dependencies" **P10**.*

> "*We also changed the teams to align them to one or two (autonomous) subsystems. It was a big change because before that the teams worked on any part of the system. So that contributed to the monolith. It's not perfect but we definitely get some benefits from splitting down the domain lines and restricting development of that part to just one team as they can focus on it and have ownership around that*" ***P18***.

A microservices architecture style was adopted by some interviewees' organizations as a practical architectural style that suits CD [7, 161]. Our interviews revealed that some of the organizations have been able to successfully adopt the microservices style to smooth their CD adoption journey. Nevertheless, it was evident that implementing this architectural style was also associated with challenges that could negatively impact on the deployment capability of organizations. As mentioned by **P15**,

> "*So microservices solve some of the issues but also introduce some other issues, especially the orchestration and the configuration of that. Microservices can bring some overhead for operations team as well… if you split too much, you have too many microservices. The operational aspects [of] managing all those services become more complex because you have to make all the configuration options available, you have to orchestrate all those services*".

Due to these difficulties, some of the participants' organizations either avoided implementing microservices or did not experience promising results from this architectural style. One of the frequently raised issues regarding microservices was the operational overhead that they introduce (e.g., for monitoring and administrating services), which require highly skilled operations teams for implementation. That is why **P12**, a technical lead and architect, opted to break a monolith codebase into smaller components, each with its own repository. Whilst this decision gives more flexibility to the teams working on the codebase, as the teams were not mature enough to handle multiple runtime services, they decided to have one deployment unit rather than several runtime services. As stated by **P12**, "*We felt that the teams were not ready for that [implementing microservices] and then we compromised, deciding we are not going to make the application as microservices at least for now*". Each component, which in this case is a binary dependency, is added as a runtime dependency on the parent application. We recognize that the implementation of a microservices style of architecture enforces changes in the organizational structure. The organizations that successfully implemented the microservices style tended to have the structure of teams and their communication pattern aligned with this architectural style. Yet, being able to implement the microservices style at this level requires team maturity and organizational readiness [162]. **P14** observed that

> *"A company struggles with microservices architecture and they have implemented microservices, as they slow down [in software releases]. [It is] because they are not ready to change the organization to support this microservices architecture. The microservices' interactions don't reflect the structure that they have*".

We have previously, in Section 6.5.1, discussed how (monolithic) databases bring unique challenges to CD. Both two above-mentioned approaches highlight the importance of revising the core database design and decomposing it into *smaller individual databases*. There are repeated statements in the interviews which support the need to treat the database as a *continuously deployable units,* similar to other software components. *Incorporating the database in the CD pipeline* as a software component is expected to avoid unexpected issues that database updates may pose at production deployment. We found that *keeping everything related to databases (e.g., associated configurations) in a version control system* in a consistent manner plays a crucial role in improving the deployability of databases, because that helps trace the changes to the database schema. **P18** emphasized that this practice "*is really useful to monitor databases in different environments and [to] compare them to know databases have deviated [from] version control*". Using *tools to automate* the database schema changes and configurations, and also to automate detection of database changes in different environments was also indicated as helpful for continuously deploying databases. Furthermore, our findings show that a *schema-less database* (i.e., the schema exists in code, not in the database) is more compatible with the spirit of CD practices. As mentioned by participant **P17**,

> *"Scheme just exists in the code instead of living permanently in the database. Schema-less lets you change the schema with a code, which is exactly what you want with continuous deployment. This is a significant architecture change driven almost exclusively by continuous deployment requirements".*

Key Findings

Finding 8. *Vertical decomposition and microservices are two primary architectural styles to implement the principle of "small and independent deployment units" principle in industry.*

Finding 9. *Adopting vertical decomposition and microservices architecture styles to promote delivery speed comes at a cost as they necessitate considering organizational structures and highly skilled teams (e.g., operations skills). Ignoring this necessity may negatively impact the deployment capability of an organization.*

Finding 10. *The key practices to improve the deployability of the database in a CD context are (1) incorporating the database in a CD pipeline as a software component; (2) keeping database configurations in version control; (3) automating database schema changes and migration, and (4) using the schema-less database.*

6.5.3 Quality Attributes that Matter (Largely/Less) in CD

While we have investigated both practicing CD within the monoliths and breaking apart the monoliths into "*small and independently deployment units*", we are also interested in the quality attributes that are most likely to affect (negatively or positively) the CD success (See Figure 6.2). In the following sections, we try to answer this question regardless of the choice of architectural styles, as the following quality attributes deserve serious consideration to realize the anticipated benefits from CD.

6.5.3.1 Deployability

The findings from our interviews indicated that deployability has gained a high priority in CD as the interviewees frequently shared that deployability concerns are accredited during architectural design and drive many decisions to have independently deployable units [9, 38]. We were interested in understanding how deployability concerns impact different aspects of (architecture) design. Taking inspiration from Manotas et al. [163], we introduced the statements S11-S14 to ask the surveyed practitioners how often deployability concerns impact the ***design of*** individual classes, components/services, interactions among components/services and the entire application. We provided a definition of deployability for all the participants to ensure common understanding. We defined deployability as: "*deployability is a quality attribute (non-functional requirement) which means how reliably and easily an application/component/service can be deployed to a (heterogeneous) production environment*". Figure 6.7 indicates that the frequency of impacting deployability concerns on low-level designs is not significant as a sustainable number of the respondents (70 out of 91, 76.9%) indicated that in the projects adopting CD practices, deployability impacted class's design *sometimes*, *rarely* or *never*. In contrast, high-level designs were more influenced by deployability concerns (See statements S12-S14 in Figure 6.7). It is a commonly held belief among over 60% of the respondents that deployability impacted the design of components/services, interactions, and entire applications *often* or *always*. We observed that none of the respondents answered the statements S12 and S13 as *never* and only 3 respondents believed that the design of the entire application was *never* impacted by deployability concerns.

During the interviews, we found that deployability has a minimum conflict with other quality attributes (e.g., "*They [quality attributes] have sometimes conflicts; deployability has a minimum conflict with other quality attributes*" ***P4***). Another interviewee, **P14**, described the relationship between deployability and other quality attributes as follows: "*I would say that deployability doesn't really conflict with other aspects of continuous delivery, but it requires some practices*". To investigate this claim further, we asked the participants how frequently they could compromise other quality attributes to improve the deployability of an application. Figure 6.7 shows that the vast majority of the respondents are not willing to compromise other quality attributes to improve deployability. 68.1% of the respondents answered to the statement S15 with *rarely* or *often*, while 29.6% believed that they could *sometimes* sacrifice other quality attributes to achieve more deployability.

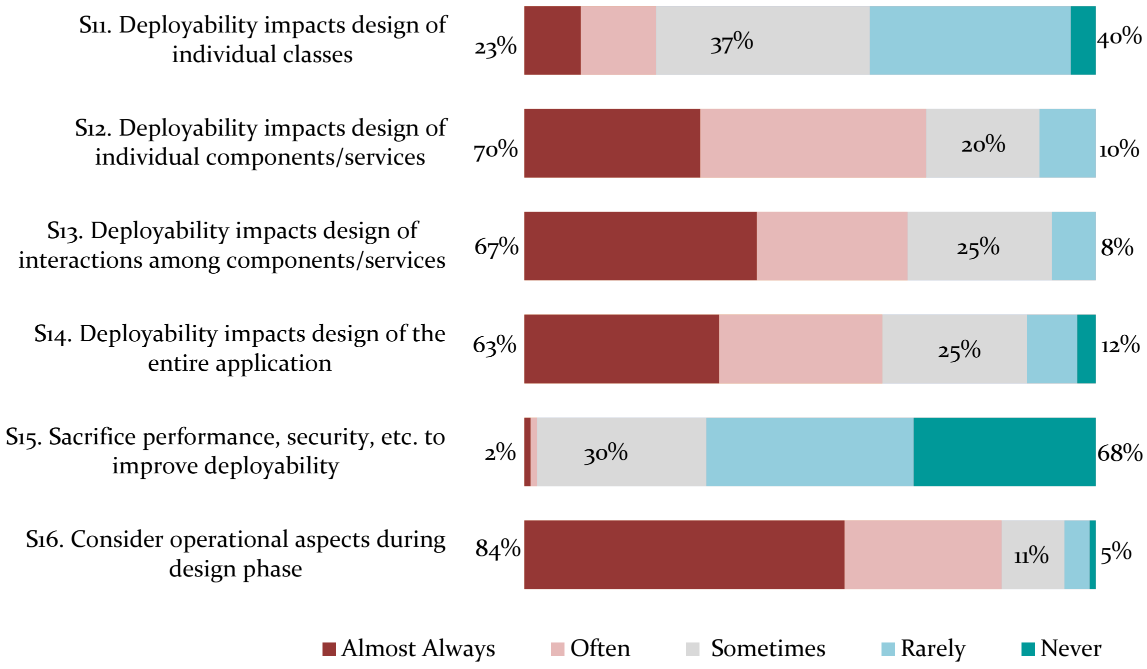


Figure 6.7 Survey responses to the statements on deployability and operational aspects

Key Findings

Finding 11. *Concerns about deployability (e.g., ease of deployment) impact how applications are designed. However, high-level designs, in particular* ***interactions among components/services****, are more influenced by deployability concerns than low-level designs (e.g., the design of individual classes).*

Finding 12*. Deployability can be supported in CD without major trade-off as it has a minimum conflict with other quality attributes and most of the quality attributes are in support of deployability.*

6.5.3.2 Modifiability

One of the top priorities for the participants when designing an application in a CD context was to support frequent and incremental changes. They reflected that they break down software into units that are small enough to be modified, replaced and run independently. The participants also tried to minimize the impact of changes for CD as described by one of the interviewees (i.e., an architect) as: "*For me, the autonomy is the most important quality attribute for continuous delivery. If you don't have dependency, you can isolate your changes and, as a team, you can independently do your lifecycle and of course, your lifecycle also means bringing it to production*" ***P19***. We learned that the participants employ four main techniques for this purpose.

T1. Identify Autonomous Business Capabilities: Our findings show there is a tendency among the participants' organizations to structure an architecture based on *business capabilities rather than functionalities* [150]. As revealed by the interview study, techniques such as *domain-driven* *design*, *bounded context*[22], and *event storming* were employed to determine the *business capabilities of software architecture* that can be independently developed, modified and deployed into production environment [164]. Domain Driven Design (DDD) aims to "design software based on connecting the implementation to an evolving model of the core business concepts" [165]. Bounded context is a concept in DDD to describe the conditions under which a specific model is defined [165, 166]. The models in each bounded context do not need to communicate with the models inside other contexts [111]. For example, one of the interviewees told us:

> *"We primarily use techniques such as domain-driven design and event storming to actually identify the autonomous business capability of software architecture"* ***P21.***

Event Storming is a collaborative design technique, in which all the key stakeholders (e.g., domain experts) assemble to identify and describe what operations (i.e., domain events) happen within a business domain [167]. These techniques were also deemed very useful by the interviewees to *decrease team interdependencies* as mentioned by **P14**:

> "*[In our CD journey] we needed to break down requirements into different, isolated, and autonomous business values and then each team had to be assigned to them. So, we could have reduced the dependency between the teams*".

Our survey results were aligned with the findings from the interviews that domain-driven design and bounded context patterns have become a mainstream in the CD context when designing CD-driven architectures. A majority of the survey respondents (54 out of 91, 59.3%) answered the statement S20 (See Figure 6.8) as *strongly agree* or *agree*, while only 7.6% strongly disagreed or disagreed. Interestingly, this statement (S20) received the highest number of "*neutral*" responses (30 out of 91) by the survey respondents. Only 9 out of 30 respondents who selected "*neutral*" to respond to this statement introduced themselves as an architect. This may suggest that this statement is a highly specialized statement to architects, with which other roles may be unfamiliar with. An initial step of applying these patterns is to find the *autonomous business capabilities* of software architecture. However, it is interesting to note that the majority of the interviewees and the survey respondents believed that it was difficult to find such autonomous business capabilities as 81.3% of the survey respondents rated the severity of this challenge as *very important*, *important* or *moderately important* (See statement S6 in Figure 6.4).

T2. Delay Decisions: Software architecture in CD should be extremely adaptable to unpredictable and incremental changes [149]. That is why we found that it is difficult to make many upfront (architectural) design decisions in CD context. Instead, the participants in our study only focused on an initial set of core decisions and other architectural decisions (e.g., decisions to choose a technology stack) are made as late as possible. Decisions are made when the time is right, for example, when requirements and facts are known [168]. This enables architects to keep architecture alternatives open to the last possible moment. Our findings show that deploying software changes frequently may necessitate making (architectural) design decisions on a daily basis. Furthermore, the participants also reported that the decisions were sometimes made unconsciously in CD context. For example, **P16** stated:

> "*I try to avoid making big decisions. We need to probably decide what kind of technologies [are] useful for running microservices; but I try to, even for those things, keep them as flexible as possible. We probably want to change it [in the future] and I also try to not look down and make too many strong decisions*".

[22] http://martinfowler.com/bliki/BoundedContext.html

To support this claim further, empirically, we asked the survey participants how strongly they agree or disagree with this statement: "*compared with less frequent releases, we avoid big upfront architectural decisions for CD practice to support evolutionary changes*". As shown in Figure 6.8, 65% of the respondents (54 out of 83) strongly agreed or agreed that architectural decisions in a CD context are made as late as possible (See statement S17 in Figure 6.8).

T3. Stateless Architecture: Having a *stateless application*, in which there is no need to maintain the state within the application, was another technique used to support incremental changes at operations level, e.g.,

> "*If a WAR file is completely stateless and if I want to deploy a new version, I can simply deploy a new version on the top of it and there is no state to keep for migrating to the new version*" ***P10***.

T4. Hide Unfinished Features at Run-time: Our study shows that working in CD mode needs everybody to push their changes to the master branch on a continuous basis. We found from the interview study (i.e., confirmed by **P12**, **P20**, and **P21**) that having long-lived feature branches is a stumbling block for CD, as they are associated with the challenges such as delayed feedback and increased merge complexity [169, 170]. The participants' organizations realized that instead of creating new branches, which developers may work on for a couple of months and then integrate them back to the master, they need to switch off the long-lived feature (i.e., the unfinished feature) and then release it to the users only when the feature is ready. A *feature toggle*[23] pattern was indicated by some interviewees to achieve this goal. This pattern helps a team to release new changes to end users safely as the features that the team are developing are still in production, but nobody can see them because they are toggled. This pattern can be combined with and made more elegant by the *branch by abstract*[24] technique, which aims at making large-scale changes to a system in production. This technique first creates an abstraction layer around an old component and them gradually reroutes all interactions to the already created abstraction layer. Once the new implementation of the old component finishes, all the interactions are rerouted to the new implementation. **P12** stated,

> "*We can also make feature toggle much more elegant by using branch by abstraction when you have implementation details separated out by interfaces and the child of implementation is on toggles*".

Other participants realized incremental changes by having side-by-side execution at the component level rather than at feature level. **P7** explained,

> "*We needed to improve the architecture to have side-by-side execution of components as it might be two versions of components, both installed at run-time. We could start to gradually cut the load over from the old version to the new version*".

[23] https://martinfowler.com/articles/feature-toggles.html

[24] http://polysingularity.com/branch-by-abstraction/

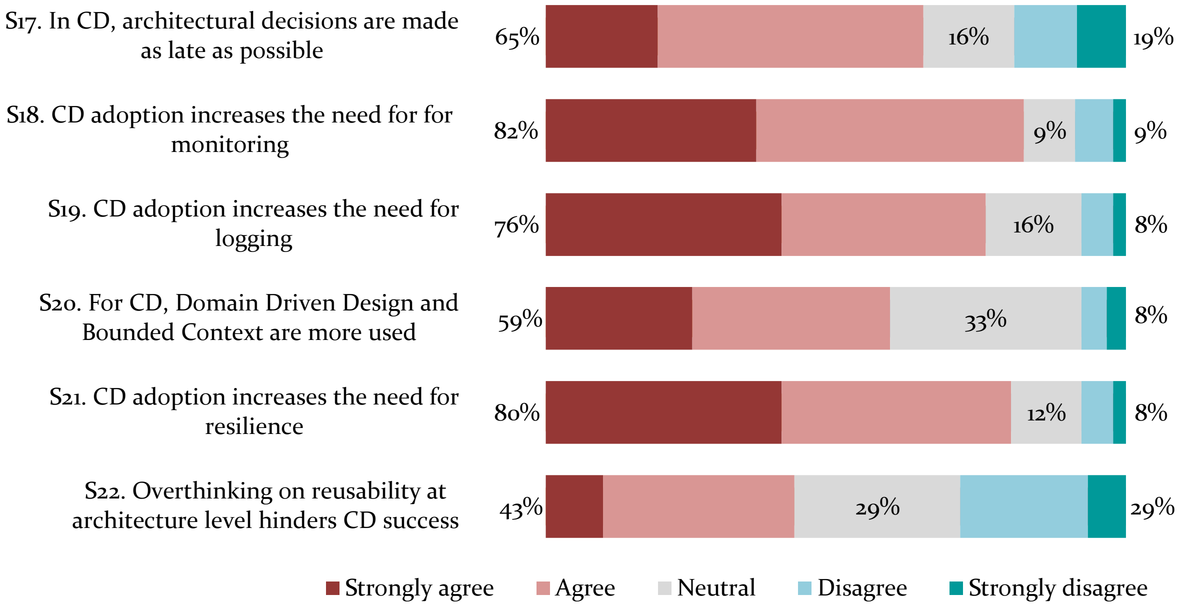


Figure 6.8 Survey responses to the statements on the quality attributes that need more attention in CD context. While the importance of monitorability, loggability, and modifiability has increased, overthinking about "reusability" (confirmed by 43% of the surveyed participants) at architecture level may overturn CD adoption.

Key Findings

Finding 13. *Domain-driven design and bounded context patterns are increasingly used by organizations to design loosely coupled architectures based on autonomous business capabilities*. *However, most (81.3%) of the participants stated that identifying the autonomous business capabilities of software architecture is difficult.*

Finding 14. *Compared with less frequent releases, CD places greater emphasis on* ***evolutionary changes****. This requires delaying (architectural) design decisions to the last possible moment.*

6.5.3.3 Loggability and Monitorability

Building and delivering an application on a continuous basis may potentially increase the number of errors, and the chance of unavailability of the application [7, 38]. This compelled the participants' organizations to have more investment in logging (i.e., the process of recording a time series of events of a system at runtime [7]) and monitoring (i.e., the process of checking the state of a system) in order to hypothesize and run experiments for examining different functionalities of a system, identify and resolve performance issues, and recover the application in case of problems. Therefore, the organizations practicing CD need to appropriately record, aggregate and analyze logs and metrics as an integral part of their CD environment. One interviewee highlighted this trend as

> *"We have everything based on the log. One aspect can be getting visibility in any part of the components, which is being pushed in the DevOps cycle. We already have it as part of our framework"* ***P5****.*

When we asked, "*since moving to CD practice, the need for monitoring (i.e., a centralized monitoring system) has increased*", the survey participants rated this statement (S18) *strongly agree* (36.2%), *agree* (46.1%), *neutral* (8.7%), *disagree* (6.5%), and *strongly disagree* (2.1%). As shown in Figure 6.8, the responses for statement S19 also indicate that 75.8% of the respondents strongly agreed or agreed that the need for centralized logging system has increased since their clients or respective organizations moved to CD. In the following, we provide more insights into how monitorability and loggability are improved:

T1. Use external tools: Our analysis shows that there are *limitations in CD tools* (e.g., Jenkins), as they are not able to provide sufficient log data and also enable aggregation and analysis of logs from different sources. The participants' organizations extensively used external tools to address these limitations. As an example, we have,

> *"So, basically those logs will be collected by other (external) systems. Then, we (the infrastructure team) can easily download and aggregate those logs on our own big data platform"* ***P2***.

T2. Build metrics and logging data into software was another solution adopted by the participants' organizations. The participants stated that large-scale applications adopting CD need to expose different monitoring and logging data. For example, two interviewees mentioned:

> *"You build log capability such they can be turned on and off in the code in order to eliminate the performance problems"* ***P6.***
>
> *"If we are talking about the logging aspect, if any tier fails for any particular reasons, then appropriate reason code must exist"* ***P5***.

Regarding the two above-mentioned solutions, the interviews' participants raised two areas that need further consideration. Firstly, it is necessary to determine to *what extent the log data* should be produced (e.g., *"We have internal meeting discussions about how much effort we can put to capture [logs] per system; you know how much with the level of detail [is] appropriate"* ***P7***). Secondly, *readability of the logs* for all stakeholders should be taken into consideration. As the developers mostly build the logging mechanism in the code, there is a chance that logs become unfathomable for operations teams and may not be efficiently employed to perform diagnostic activities, e.g.,

> *"Support people (the operations team) feel that there is too much logging. So, they have to swim [in] too much information and find problems. The support people need to have cleaner and simpler logs to look at, where developers get all this information they need"* ***P13***.

We also observed that the importance of loggability and monitorability considerably increases when a monolithic application is split into smaller units (e.g., microservices). This is due to the fact that splitting a monolith into multiple independently deployable services introduces challenges in identifying problems (e.g., performance issues) in a system. Furthermore, it is needed to trace how services go through a system. It is essential to ensure that there is enough trace information for logs.

Key Findings

Finding 15. *Monitorability and loggability quality attributes really matter in CD. They help build architectures that are responsive to operational needs over time.*

Findings 16. *While software organizations* ***extensively use monitoring tools*** *to address the shortcomings of CD tools, the applications in a CD context should* ***expose the different types of log and monitoring data with a standard format****.*

6.5.3.4 Testability

One of the main concerns stated by the participants in transition to CD was about *testing*. It was observed during the interviews that addressing testability concerns influences architectural design decisions for explicitly *building testability inside a system*. It is emphasized that architectures in a CD context should support testing to the extent that tests should be performed quickly and automatically. Another solution adopted by the participants to improve testability was using the right tools and technologies. Nevertheless, the participants worried about the potential limitations of tools, as tools alone are insufficient for addressing all the testing problems. For example, we were told that existing automated testing tools tend to be fragile in some types of applications (e.g., mobile applications) or environments (See Chapter 4 for more information). That is why the participants typically approached the testing challenges by combining architectural (i.e., testability inside the design) and technological (i.e., selecting the right tools) solutions. Apart from choosing appropriate testing tools, we found the following techniques support testability inside a system:

T1. Improving test quality: To avoid putting overhead on a CD pipeline, it was frequently suggested that there should be improvements to the quality of tests (data) rather than increasing the number of test cases. As one interviewee put it,

> "*We are looking at the quality to improve the DevOps model. One of the things that we can improve is the quality of the tests themselves; do we have the right test suites? Do we have the right coverage?*" ***P6***.

This includes improving quality by selecting the right number of test cases to be performed at the right time on quality test data (e.g., it was also affirmed by **P5**, **P12**, **P15**, and **P17**). The cycle time of a CD pipeline (i.e., the required time to push code from the repository to the production environment) can be increased by long-running tests. Furthermore, the long-running tests can slow down the feedback cycle in a CD pipeline. Respondent **R5** commented that running tests in parallel can decrease the cycle time of a CD pipeline. These issues happen when test cases are poorly written and designed. One of the interviewees described that how improving the quality of tests could help them to accelerate the feedback cycle in the CD pipeline. He (**P12**) mainly highlighted two practices: (i) designing smaller test units that result in immediate feedback, (ii) decreasing the number of functional tests. The applied practices resulted in receiving the same level of feedback from applications, in a timely manner.

T2. Making code more testable It was also found that testability should be addressed at code level. Whilst services and components need to be tested independently, testing them within a larger CD pipeline with all dependencies can be a complicated and costly process. For instance, it may be costly to provide all the resources (e.g., databases) for testing in staging environments. A typical solution to this problem was dependency injection, which would bring more flexibility to the testing process. One participant opined:

> "*From the architectural standpoint, we very often build projects that use dependency injection a mechanism in, like Spring so that we can inject a set of dependencies. For example, you inject an in-memory fake database rather than production database, you inject a fake set of resources rather than relying on [real] resources, which makes the code much more easily testable*" ***P13***.

Another example comes from participant **P6**, who built performance tests into the code. Rather than just running performance tests as black box testing from the outside of the code, performance testing was actually tightly integrated into the code. Put it another way, the code itself would be measuring start and stop times between operations and between events occurring within the code. All these times (i.e., the start and stop times for different operations in the system) are recorded to compare them with a baseline to decide whether there is a problem or not. According to **P6**, this practice significantly improved the overall performance of the system.

Key Findings

Finding 17. *CD practices require* ***high quality*** *tests that* ***run easily****. Therefore, it is necessary to make informed (architectural) design decisions to improve the testability of a system by (1) improving test quality (e.g., more suitable, simpler test methods and decreasing test cycle times); (2) making code more testable; (3) using smaller components.*

6.5.3.5 Resilience

Increasing the frequency of deployment to production may increase the number and severity of failures. So, the failures are inevitable in a CD context. Through the interview study, we found that for some participants designing for failure guides their architecture designs. **P11** characterized the software architecture in the CD context as follows: "*an architecture that splits the system also by units that should fail independently" and* ***P14*** *told us, "resilience is also a matter of integration aspect: how to integrate with the third party [units]; what if a third party starts to fail*". We also observed that many discussions among the participants were centered on resilience versus reliability as the main concern was *not* to prevent failures but to think about how fast the failures can be identified and isolated, and how to automatically rollback to an early stable version. An example of this trend is vividly typified in the following anecdote:

> "*I think when we are moving to this paradigm [CD], rather than ensuring your software is gold before you deploy it to production, for example, by three month testing cycles to validate every bit of it, we are going to a mode [where the] mean time to recovery [is important]. [For example] if I face an issue in production, how soon I can recover from that failure*" ***P10***.

Whilst we could not conclude that the resilience quality attribute needs to be prioritized over reliability within CD context [171], the responses to the statement S21 indicate that the survey participants are significantly concerned with resilience when designing and implementing applications. This quality attribute appeals to a large portion of respondents: 80.2% strongly agreed or agreed that the CD practices increase the need for resilience (See statement S21 in Figure 6.8).

Key Finding

Finding 18. *Design for failure is considered as the foundation of the architecture in a CD context, in which a system is split into units that should* ***fail independently****. Instead of preventing failures (reliability), it is more important to learn how to deal with failures (resilience).*

6.5.3.6 Reusability

Our analysis reveals the architecture designs that emphasize reusability could make practicing CD more challenging. The drawbacks of overthinking on reusability at architecture level (e.g., using shared components, or packaged software) in the context of CD are twofold: **(1)** *inter-dependencies between software development teams* increase in the sense that they rely on shared software units. It means that changing the shared units requires seeking inputs and reaching agreements among all the relevant stakeholders, which demands significant time and effort (e.g., "*If you really do not control explicitly what you really expose and what you really reuse, you will end up with a lot of mess in the code*." ***P14***). This approach hinders the *autonomy of software teams* in building and deploying software components and applications. **(2)** The other concern is about being able to *test the configuration of shared software units* (e.g., well-known packaged software). Our interviews revealed that it is vital for the application developers working in the context of CD to write test cases, through which they can validate the configuration of software packages used in the application. Hence, reusing packaged software may hinder their ability to perform potential unit tests and push the packaged software into CD pipeline on a continuous basis. This leads to *losing the testability of those configurations*, which is significant. As **P14** explained:

> *"We had a plenty of products and there are a lot of [common] things between them. We reused [things] like utilities and even some part of the domain. We spent a vast amount of time maintaining agreements between teams; [for example] how to evolve them. We decided to fork them; just go your own way, whatever you want and just leave us alone, because reuse has side effects. So, we have to understand duplication is not evil"* ***P14***.

As shown in Figure 6.8, about 42.8% of the survey respondents (39 out of 91) strongly agreed or agreed that overthinking about *reusability* at architecture level (e.g., reusing packaged software) hinders CD success, while 28.5% disagreed or strongly disagreed. This fairly agrees with the interviews' findings. We found that such attitudes were mostly dominant among software architects as 20 out of the 39 architects who participated in the survey have voted the statement S22 as *strongly agree* and *agree*, while only 7 architects rated it as *strongly disagree* or *disagree*. Other respondents (28.5 %) had no idea about whether or not having too much reusability is a major roadblock for CD.

Key Finding

Finding 19. *Overthinking about reusability at architecture level has two side effects: (1) increased inter-team dependencies and (2) losing testability. However, among other roles (e.g., developers), software architects strongly believe that a lack of explicit control on reuse at architecture level (e.g., shared components) would make practicing CD harder.*

6.5.4 Perspectives on Operational Aspects

The importance of operational aspects management (i.e., of the production environment, operations stakeholders and their requirements) for the architecting process in a CD context was highlighted during the interview study. We observed that the participants' organizations have been shifting from considering operational aspects as separate and sometimes uncontrolled entities to treating them as first-class entities in the architecting process, particularly after CD adoption [7]. The shift from the first to the second paradigm was also confirmed by the survey results, in which 83.5% of the respondents indicated that they consider operational aspects during the architecting process *almost always* or *often* (See statement S16 in Figure 6.7). Additionally, the respondents believed that operations requirements and aspects significantly impact on architecture design and design decisions (See Figure 6.9). 83.5% of the surveyed practitioners strongly agreed or agreed with the statement S24 in Figure 6.9 ("*Operational aspects and concerns impact on our architecture design decisions*"). Other studies have also found that involving relevant stakeholders in the architecting process leads to informed and balanced architectural decisions [172, 173]. This might help to mature the architecting process [161]. In the following subsections, we provide more insights into why operational aspects need to be perceived as important in a CD context.

6.5.4.1 Production Environment Challenges

The interview study revealed that production environments may pose challenges to architecture design. One of the top challenges is to make sure software changes are being seamlessly transferred and deployed into multiple, inconsistent production environments. A program manager (**P4**) described the challenge of the inconsistent environments thus,

> *"One challenge is that there are multiple environments. [We must] make sure that the system of development is compatible with all environments, which means we [need to] give the current infrastructure, hardware and technical details of all environments".*

Regulatory and controlled environments were also stumbling blocks to CD as these environments usually follow a formal deployment process. There is a need to adjust and adapt architectures in the regulated environments so that apart from those parts of a system that are not really able to adopt CD, the rest being deployed on a continuous basis. As **P14** mentioned,

> *"Architecture is also about compliance and security as well. Because let's say there is a kind of regulated environment and you want to have CD there; how [should we] to adjust the architecture? For example, you can rip out part of your software and make sure it is part of your architecture which can't really support CD but let the rest of the architecture and software be deployed in a CD fashion".*

Our results in Chapter 4 show that a lack of control of the production environment by the development team negatively impacts on deployment frequency (e.g., "*Inside the client network, the processes are less mature and more overhead exists to deploy code to controlled environments, in which requires manual processes to turnover code*" ***P3***).

Another category of the challenges is related to *the extent to which operations requirements* need to be collected and shared with software architects. Taking into account too many requirements of an operations environment could result in designing a system too specific for that environment. This introduces the risk that design decisions become dependent on the operational configurations and are therefore fragile to the changes that are made in that environment. On the other hand, ignoring the characteristics of an operations environment during the architectural design of an application is likely to result in losing the operability of the designed architecture. This challenge was explained by **P5**,

> *"They [the operations team] had purchased some infrastructure and some hardware and what they wanted from us [was] that our solution (architecture design) should be in compliance with their already purchased infrastructures. This all added the challenges [to the architecture design]".*

All the above challenges may lead to the expansion of the role of architects. Architecting for CD goes beyond designing only a software application: it requires incorporating consideration of the whole environment (e.g., database, automation, and test data). Architects are expected to look into a broader spectrum and propose solutions that keep consistency after the application is deployed and upgraded. This implies versioning the whole environment rather than versioning the source code alone (i.e., everything-as-code [174]). One interviewee described this trend in these words:

> “*It is not architecting things in terms of source code, but it is more about architecting the deployment as a larger entity, [for example], in finding ways to drive and maintain the consistency across those environments after we deploy and upgrade because they are on different servers. It is not [similar to] a cloud system in a multi-tenant environment*” ***P7***.

6.5.4.2 Operations Stakeholders and Requirements

In this section, we discuss what strategies have been applied in industry to treat the operations team and their concerns as the first-class entity in software development process, which also affects the architecting process (See Figure 6.7).

S1. Leveraging logs and metric data in an appropriate format for operational tasks: The most cited strategy, indicated by 62 (68.1%) survey respondents, refers to collecting and structuring logs, metrics (e.g., CPU and memory usage) and operational data in appropriate formats to enable the operations team to make faster and more informed operational decisions in post-deployment activities such as assessing the deployed application in production. For example, we have:

> “*One of the key aspects [of CD] would be around having sufficient logging in place, which you can identify when things go bad. You have to build a dashboard to tell you what the current load is and also aggregate that data on the period of time*” ***P12***.

S2. Early and continuous engagement of operations staff in decision-making: The participants also reported that early and continuous involvement of the operations team in the development process, particularly in the design process boosted the productivity of their respective organizations in delivering value on a continuous basis. It is mainly because deployability of the application is more thoroughly considered in the development and design phases; the deployment process would not be a big issue at the end. As shown in Table 6.1, this was the second most cited (55, 60.4%) strategy by the survey participants. Here are just a few of the example quotations supporting this strategy:

> “*Once the operations team was brought into planning meetings, we got things much smoother. Most of the operations requirements could be negotiated off and we would find cheaper wins that would be less expensive than the original plans of the software developers. So those operations requirements could be better implemented and landed in production*” ***P17***.

> “*When people do the design, they normally do not engage with operations teams. So, this is an area [that] we are trying to improve; when we finalize the design, we are getting the operations teams to sign-off the design*” ***P4***.

S3. Prioritizing operations concerns: It is argued that the operations team has a set of concerns (e.g., quickly detecting and predicting failures), which should be effectively addressed [7]. During the interviews, we learned that despite the adoption of CD, operations concerns and requirements still have a lower priority than other requirements. This is moderately confirmed by the survey study as 41.7% of the survey respondents (strongly) agreed with this finding (See statement S23 in Figure 6.9). It is interesting to note that there is a relatively large proportion of the respondents (29.6%) who are neutral about the statement S23 and only 28.5% strongly disagreed or disagreed. The interviewees revealed several reasons for deprioritizing operations requirements. For example, developers usually believe that the role of the operations team is only to serve the development team. It has been discussed that people who historically decide on priorities (e.g., the product owner) usually do not consider so much business value from operational aspects. According to the interviewees, ignoring and deprioritizing operations requirements have resulted in severe bottlenecks for deploying applications on a continuous basis. The interviewees' organizations started treating the operations personnel and their requirements as having the same priority as others. So, the operations requirements became an integrated part of the requirements engineering and architecture design phases. Table 6.1 shows that 41 (45.1%) of the surveyed practitioners adopted this stagey. **P18** referred to this strategy as follows:

> *"One thing [that] we are doing for this client, for this software is to emphasize the importance and address operational things much more. That is starting to happen, but it is quite a big change"*.

When we asked the participants about their experiences in thinking about operational aspects from day one of a project, we noted several benefits: (i) it puts operations staff on the team and reviews their concerns quite early; (ii) it influences how developers look at applications in production effectively; (iii) it forces architects to think much more about the problems that may happen in the deployment process (e.g., "*Only recently we have realized that operations concerns are really fundamental part of architecture. If you take that as part of the architecture and treat them as first-class citizens, it allows you to build an architecture that is much more flexible*" ***P13***); (iv) it provides much more flexibility and enables organizations to get applications deployed much faster and therefore realize CD.

We also added the "Other" field to gather more strategies in this regard. Interestingly, only 8.7% of the survey respondents filled in the "Other" field. The respondents mostly used the "Other" field to elaborate their previous choices (e.g., "*Ensuring we have scheduled placeholders to revisit changes in operational requirements, and that we have an operational requirements champion*" ***R16***, "*Operations [team] owns the final solution and must sign off on it*" ***R40***).

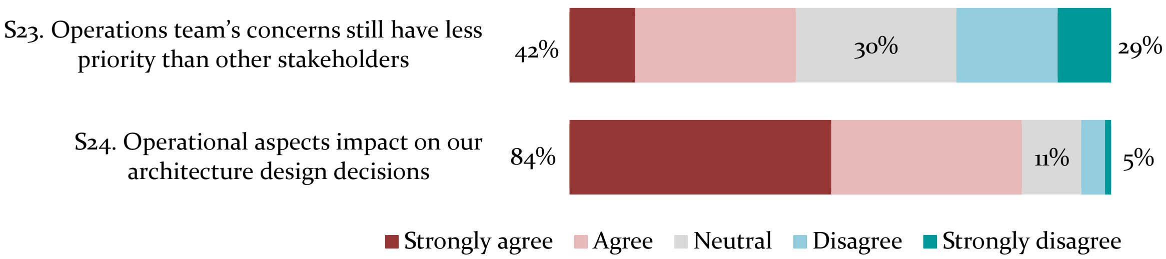


Figure 6.9 Survey responses to the statements on operational aspects: The respondents indicated that there is a trend to consider operations aspects at early stages of the development process (e.g., architecture design), which helps design CD-driven architectures.

Table 6.1 Strategies for increasing the amount of attention paid to the operations team and their concerns

Strategy	#	%
Leveraging logs and metric data for operational tasks	62	68.1
Early and continuous engagement of operations staff in decision-making process	55	60.4
Prioritizing operational concerns	41	45.1
Others	8	8.7

Key Findings

***Finding 20. Inconsistent**, **regulatory** and **controlled** production environments hinder the implementation of CD.*

***Finding 21.** The requirement for continuously and automatically deploying to unpredictable production environments can **expand the role of architects** as apart from software design, they need to collaborate substantially with operations personnel and deal with infrastructure architecture, test architecture, and automation.*

***Finding 22.** Designing highly **operations-friendly architectures** are achieved by (1) leveraging logs and metric data; (2) engaging operations personnel in decision-making; (3) prioritizing operational concerns; and (4) understanding production environments' constraints.*

***Finding 23.** The participants do not have a clear understanding of **the extent to which operations requirements** should be collected to simultaneously deal with the fragility of design decisions and improve the operability of software systems.*

6.6 Discussion and Conclusion

This chapter has empirically explored how an application should be (re-) architected for CD. We applied a mix-methods approach consisting of semi-structured interviews and an online survey for collecting qualitative and quantitative data, which have been systematically and rigorously analyzed using appropriate data analysis methods. The research presented in this chapter are expected to make significant contributions to the growing body of evidential knowledge about the state of the art of architecting for CD practices. Furthermore, the results of this research can provide improvements to the adoption of DevOps and CD. In this section, we first discuss the main findings of this chapter while comparing and contrasting them with the existing literature on this topic. Then, we present some implications for practitioners. It is worth noting that from the methodological perspective, our study includes varied demographics rather than the peculiarities of practitioners' own experiences [7, 28, 38, 111] or a single case company and particular domain [38, 147].

6.6.1 Main Findings

Monoliths and CD are not intrinsically oxymoronic. Our conclusion is in line with the results of [33, 40, 111] that monoliths present the most significant challenges to CD success. Yet, we have found some examples in both the interview and the survey parts of this study that it is *possible* to do CD with monoliths. This has also been indicated by personal experience reports by Prewer [175], Vishal [39], Schauenberg [176] and Savor et al. [22]. However, an overwhelming majority of the interviewees and the survey participants believe that it is much more difficult to adopt CD within large monolithic systems or large components. Compared with the works by Bass et al. [7], Schermann et al. [40], and Newman [111], our study has independently identified the reasons why practicing CD within the monoliths is difficult. We found that growing monoliths leads to increased complexity of internal and external dependencies, restricted automation (test and deployment), impeding teams' ownership and having slow and inconsistent feedbacks [33, 40, 156], which together can be roadblocks to frequent and automatic deployment. Finally, we have provided principles, practices, and strategies with concrete examples to achieve CD with monoliths. Specifically, those organizations that effectively embraced CD for their monoliths mostly employ the following practices to optimize their deployments: (i) developing highly customized tools and infrastructures for monitoring and logging [39]; (ii) reducing test run times by improving the test quality (e.g., parallelization of automated testing); (iii) describing, testing and deploying components/services *only* through interface specification with backwards capability; and (iv) deploying all entities including components and databases through a rigorous CD pipeline.

Characteristics the principle of "small and independent deployment units". All of the above-mentioned monolith-related challenges compelled many participants' organizations to move beyond the monoliths to facilitate the CD adoption journey. Similar to Newman [111] and Lewis and Fowler [150], we can conclude that the principle of "small and independent deployment units" is an alternative to monolithic systems for this purpose and serves as a foundation to design CD-driven architectures. As elaborated in Section 6.5.2.2, this principle has mostly been implemented in the industry using two concrete architectural styles: vertical layering (decomposing) and microservices. Our findings particularly emphasize that adopting these architectural styles may result in overhead costs to deal with complex deployment processes, which require having sophisticated logging and monitoring mechanisms and high operations skills [162]. For example, the microservices architecture style requires every team to build a CD pipeline for each component or service rather than having one CD pipeline for the whole application. Software organizations incrementally split their applications into "small and independent deployment units" that can be independently managed.

Our study includes other scopes compared with [111, 150] as we also focused on the characterization of "small and independent deployment units". We have proposed that "deployment units" need to represent *only* one business domain problem (i.e., it should not cross its bounded context) and each of the deployment units should be autonomous in terms of *deployability*, *modifiability*, *testability*, and *scalability*. These factors are among the highest rated statements by the participants (i.e., more than 68% of the participants (strongly) agreed with all these factors). That means organizations can use these factors as reliable criteria when moving towards CD. Other factors such as having team-scale autonomy per "deployment unit" are desirable but less cited by the participants.

First class quality attributes in the CD context. Our findings indicate that CD and its associated constraints (e.g., unpredictable environments) expect a certain kind of architecture to emerge. That means CD practices significantly change the priority of some quality attributes such as deployability, modifiability, testability, monitorability, loggability, and resilience. Whilst these quality attributes are important for all types of contexts and systems, they are considered critically important for a CD-driven architecture design.

Previous research also highlights the importance of these quality attributes [38, 75]. Bellomo et al. [75] only focus on deployability as a new quality attribute in three projects and introduce tactics to achieve deployability. In contrast to [75], we have investigated the impact of deployability on different aspects of (architecture) design (high-level design vs. low-level design) and also we have indicated that deployability has a minimum conflict with other quality attributes. Chen [38] also highlights what quality attributes (i.e., deployability, modifiability, security, monitorability, testability, and loggability) can be more important in the context of CD; however, he did not provide any details about how these quality attributes can impact on the architecture design in CD context and how they can be achieved. Our results confirm and extend the previous findings [38, 75] (i) by improving our understanding of how the aforementioned quality attributes affect architectural decisions; and (ii) by introducing techniques to achieve these quality attributes (e.g., keeping components stateless helps improve modifiability). Whilst we targeted the quality attributes of applications in the context of CD in this chapter, Bass et al. [7] focus on the quality concerns of a CD pipeline rather than applications. They indicate that the primary quality concerns that need to be built early in a CD pipeline include repeatability, performance, reliability, recoverability, interoperability, testability, modifiability, and security.

We have also observed that from an architect's perspective, "reusability" at the architecture level is the only quality attribute that should *not* be overthought, as it contributes to overturning CD adoption. It should be noted that we could not conclude that CD substantially influences other quality attributes (e.g., security as indicated by Chen [38]). On the other hand, our study highlights that CD practices seek software architectures that are easily composable, self-contained, and less monolithic with less shared data among components/services. Architectures in CD should be atomized for autonomy and independent evolution to make sure there is a minimum overlap between teams. Consequently, software architects should avoid upfront (architectural) design decisions and delay them to the last possible moment to effectively meet the aforementioned quality attributes.

Operations-friendly architectures One of the most intriguing findings of this study is that there is an increasing trend of paying significant attention to operational aspects (e.g., operations requirements) in the architecting process. Interestingly we have observed that the constraints of the production environment (e.g., lack of access to the production environment) may slow down the pace of the production deployment [3, 11, 115]. We can conclude that one of the reasons for the participants' organizations not continuously and automatically deploying into production environments stems from not carefully understanding and dealing with the production environments' constraints. We recommend that software development organizations invest time in understanding their production environments and their respective constraints that may not support CD. Accordingly, software architects need to collaborate substantially with operations teams during the requirements engineering and architecture design phases. This practice helps architects to design and implement operations-friendly architectures. Though we know from other studies (e.g., Bass et al. [7]) that the operations team and their requirements play significant roles in CD adoption, our study is the first large-scale piece of empirical work that has specifically explored the strategies (e.g., early and continuous engagement of the operations team in architecture design) that are practiced in industry to achieve this goal. We emphasize that considering the operational aspects of a system early in the software development process helps to scale the complexity of the deployment process at approximately the same pace as the code base. This trend has changed the role and responsibilities of the architect. From the research perspective, further investigation is necessary for understanding how software architects perform their work in a CD context to provide guidance for the required job skills and education programs [21, 142].

6.6.2 Practical Implications

It is argued that the findings of empirical studies should provide insights for practitioners. At the end of the online survey, the participants were asked if they wanted to receive the results of our study by providing their email addresses. Surprisingly, 81.3% (74 out of 91) of the respondents provided their email as they were interested in receiving the findings of this study. We believe this indicates that our findings are highly likely to draw the interest of software engineering practitioners. Based on the findings from this study, we draw implications for practice.

Deployable units do not only mean software components/services. The results of our study in this chapter suggest the concept of "*Continuously Deployable Units*". This indicates that in addition to software components/services, every entity (e.g., database and dependencies) that an application depends on, should be a unit of deployment in the CD pipeline. We have observed that database should be continuously provided with as much automation support as possible for successfully implementing CD practices. Whilst the participants in our study employed some practices (e.g., schema-less databases) and used appropriate tools for this purpose, it is clear that there is not much support for automatically delivering database changes on a continuous basis [25, 177].

Our study reveals that there are a number of organizations that could not succeed with CD due to a lack of database automation in their CD pipeline. The reason for this stems from the fact that validating data and managing schema migration in the CD pipeline remain immature (e.g., manual steps requiring DBA involvement). Hence, organizations prefer to do it offline. We believe that this can be a reason that our study participants rated the database-related challenges as the least important in their CD journey (See Section 6.5.1.1). On the research side, more focus is needed on mining and analyzing database-related failures in the CD pipeline to thoroughly understand the behavior of databases in a CD context. This helps to design effective and robust solutions (e.g., tools) to support organizations in automating the deployment of schema modifications and data conversions in CD pipelines without compromising the continuous deployment of application code changes [25].

Adjust the organizational structure to support CD. Our experience in this study was that the process of changing to a CD organization requires more than providing tool support and automation. CD impacts on organizational structures (e.g., team structures), roles, and responsibilities. From practitioners' perspectives, the inflexibility of organizational structures with the spirit of CD practices is the most critical challenge for implementing CD (See Section 6.5.1.1). A software organization can succeed in CD adoption once there is flexibility in optimizing organizational structures to be aligned with CD and certain skills and responsibilities need to be sought. Finding the best organizational structures that suit an organization depends on many factors, such as the flexibility of the current organizational structure, available skills, and management procedures [125, 126]. Answering this question requires extensive empirical studies to explore these organizational structures supporting CD practices in different situations. Chapter 4 [4] has provided preliminary findings about how team structures, roles, and responsibilities can change while adopting CD. We have identified four main patterns for organizing development and operations teams (e.g., no visible operations team) in industry. We believe that there is a need for further research to gain an in-depth understanding of suitable organizational structures for implementing CD.

Real-time, digestible and customizable monitoring is key. As elaborated in Section 6.5.3.3, we have seen the growing importance of monitorability and loggability in the transition to CD. In fact, there is a growing tendency to adopt log-driven, log-specific architectures that support the continuous collection of operational data, facilitate aggregation of logs and transform them into a searchable format [178]. More importantly, breaking down a monolith into "small and independent deployment units" can worsen the monitoring challenges because tracing a large number of independently deployable units and identifying problems inside a system can be even more challenging. Such issues can significantly impact the way that a system is composed of smaller pieces. Whilst a number of solutions (e.g., *extensive use of monitoring tools*, *creating monitoring and log data into applications*) have been introduced, there are still several challenges to be addressed in order to implement monitoring and logging in CD context. Based on our knowledge, two main areas that usually cause practical challenges in this regard are *the readability of logs for all stakeholders* and *the abundance of logs and monitoring data*.

We believe these challenges can be alleviated by (1) establishing a standard for logging and monitoring by which all developers and applications in an organization create consistent logging and monitoring data [179], and (2) applying analytical techniques (e.g., machine learning approaches) to summarize, prioritize or filter monitoring and logging data. Our findings re-emphasize the conclusion by Bass et al. [7] and Humble and Farley [28] that to achieve effective monitoring in the CD context, there is a need to build a centralized platform for providing real-time, digestible and customizable monitoring and alerting for the different types of stakeholders. This would enable stakeholders to understand what happens and why in real (or near) time.

Chapter 7

On the Role of Software Architecture in DevOps Success

Chapter 6 was focused on the perception of practitioners from different organizations around the world on how software architecture being impacted by or impacting two key DevOps practices, namely Continuous Delivery and Deployment (CD), which led to a conceptual framework to (re-) architect for CD. However, the practical applicability of architectural practices or tactics for enabling and supporting DevOps is tightly associated with organizational context (i.e., domain), and very little rigorous research conducted in this area. This chapter reports on an in-depth, industrial case study with two teams in a company working on Big Data technologies, which is analyzed with analysis approaches of Grounded Theory. We identify the architectural decisions and tactics the studied teams find essential to DevOps transition and describe the implications (costs and benefits) of those architectural decisions. Our key findings are DevOps works best with modular architectures and developers have difficulties with operations tasks that require an advanced level of expertise.

7.1 Introduction

Attracted by increasing the need of being able to improve business competitiveness and performance, many organizations have adopted DevOps to develop and deliver high-quality values to customers quickly and reliably [35]. The current research typically focuses on challenges and practices related to organizational culture, process, and tools. Several studies have focused on tools and techniques to integrate and improve security [180, 181], automation [26, 27, 182], and performance [30] in DevOps tool-chains (also known as deployment pipeline) to automatically and securely transfer code from a repository to production. Other studies report the challenges (e.g., limited visibility of customer environments) that organizations encountered in DevOps adoption and the practices (e.g., test automation) employed to address those challenges [115, 182, 183].

As reported in Chapter 4, another line of research investigates how organizations have restructured (e.g., re-organizing development and operations teams), trained developers, and sought new skills for adopting DevOps [4, 126]. However, it is increasingly becoming important for both researchers and practitioners to understand how an application should be (re-) architected to support DevOps [7, 35]. Besides our study in Chapter 6 [2, 5], a few studies can be found on software architecture and DevOps, yet they have mostly investigated the software architecture in Continuous Delivery and Deployment (CD) as two key practices of DevOps [38, 75]. Another type of studies in this regard has primarily leveraged microservices-based architectures to provide a flexible deployment process in DevOps context [129, 147].

Whilst DevOps, CD and microservices share common characteristics (e.g., automating repetitive tasks) and reinforce each other [184, 185], there might be organizations that follow and adopt only one of them to achieve their business goals, e.g., delivering better quality software in a shorter time and more reliable way than before [183]. For example, some of the essential characteristics of CD appear to be incompatible with the constraints imposed by organizational domain [3]. Furthermore, architecting an application to be compatible with DevOps is in its infancy. This deserves a fair, thorough, specific, and contextual investigation of different architectural tactics, practices, and challenges for DevOps as argued by Bass et. al [186], this helps to understand "which architectural practices are best for which kinds of systems in which kinds of organizations?". Studying a DevOps journey from a software architecture perspective is important in order to develop deep insights into the nature of the relation between software architecture and DevOps. To achieve this goal, we carried out an exploratory case study (i.e., described in detail in Chapter 2) with two teams in a company, which is referred to the case company in this chapter, to investigate the following research question:

RQ7.1 What key architectural decisions are made by the case company to adopt DevOps?

Description: We concluded in Chapter 6 that the role and responsibilities of architects in the context of DevOps expand as apart from software design they need to deal with infrastructure architecture, test architecture, and automation. The goal of this question is to learn more about the correlation of software architecture and DevOps through identifying those architectural decisions/tactics, along with their consequences that fall under the name DevOps in the case company.

Our case study in an industrial context presents in-depth results and confirms certain findings from Chapter 6, which can benefit other practitioners and researchers. Furthermore, our study identifies two concrete improvement areas for the case company: (i) leveraging operations specialists to manage shared infrastructures among teams and to perform the operations tasks that require a deep understanding of advanced expertise, and (ii) investing in testing could be a significant driver for releasing software more quickly.

Chapter organization: Section 7.2 outline our research method. It is followed by presenting the results of the qualitative study in Section 7.3. Section 7.4 discusses the lessons learned from our work and then we examine the related work in Section 7.5. Finally, we close the chapter in Section 7.6.

7.2 Research Design

As described in Section 2.3 in Chapter 2, we used a case study to deeply investigate the role of software architecture in DevOps transition in a software development context. We conducted 6 face-to-face, semi-structured interviews with the members of two teams, namely TeamA and TeamB in the case company. Besides the interview data, we had access to a number of documents (e.g., software architecture document, internal Wiki documentation) provided by the case company, which enabled the data triangulation process. It should be noted that when we refer to data from the interviews with TeamA and TeamB, we use **PAX** and **PBX** notations respectively. For instance, **PA1** refers the interviewee 1 in TeamA (See Table 2.3 in Chapter 2). The excerpts taken from the documents are marked as **D**.

7.3 Results

This section presents the results of analysing the interviews and documents. We identified 8 high-level (hard-won) architectural decisions of DevOps transformation. As mentioned in Section 7.1, apart from software design, architects in the context of DevOps need to deal with infrastructure architecture, test architecture, and automation. Therefore, the identified architectural decisions target application architecture, deployment pipeline architecture, and infrastructure architecture. For the sake of readability, the identified decisions are presented using a decision template including *concern*, *decision*, *implication*, and *technology option* [187, 188]. *Concern* describes the problem that is solved by a decision (See Tables 7.1 to 7.11). The potential positive (indicated by "+") or negative (indicated by "-") consequences of a decision are captured in *implication*. A plus/minus sign (+/-) shows that a decision might have both positive and negative impacts. Important implications are **bold**. For example, a decision might influence one or more system quality attributes [7, 189]. Finally, we show the technologies, tools, and tactics that help to implement or complement a decision with *technology option*.

D1. External configuration

We have found that the key architectural decisions in DevOps transition were about configuration. From the interviews and documents, it is clear that the concept of **external configuration** is at the centre of our findings. This can be illustrated by the following quote:

> "*The key architectural thing is [to] making [application] externally configurable as you can be aware of the test environment, production environment etc.*" **PB2**.

External configuration aims at making an application externally configurable, in which each application has embedded configuration (i.e., configurations are bundled within the application) for development and test environments and uses the configuration that lives in the production environment. Here *configuration* refers to storing, tracking, querying, and modifying all artefacts relevant to a project (e.g., application) and the relationships between them in a fully automated fashion [28]. The external configuration also implies **multiple-level of configuration** as each environment has a separate configuration. All makes deploying applications to different environments trivially easy as there is no complicated setup procedure. By this approach different instances of one artefact in different environments are considered as the same artefact; once one artefact is deployed to test environment, the same artefact gets deployed into the production environment. The solution architect from TeamA said:

> "*We externalize its [application's] configuration as you can provide different [instances] in different environments but as the same artefact*" **PA2.**

Apart from positively impacting on **deployability**, this decision leads to improving **configurability**. This is mainly because the embedded configurations inside applications can be easily overwritten at target environments and there is no need to reconfigure the whole infrastructure as it is only needed to touch the things required. Those configurations that might rapidly change will be read from Zookeeper[25], but large and static ones are read from HDFS[26] (Hadoop Distributed File System). **PB2** explained the benefits of this decision vividly:

> "*We also use Spring Boot. It is always looking for module name first, then for the file directly next to JAR and then for the embedded JAR. You can have multiple levels of configuration verities. So, inside our JAR, we have the same default and they always target the local favor environment. If you accidentally run a JAR file and you might do something crazy like delete data, so it lonely targets your local environment. Then the application YML one, like the external one, gets deployed to test environment*".

[25] https://zookeeper.apache.org/
[26] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

Table 7.1 Decision for "external configuration"

Concern	Different instances of an application/service should be treated as the same artefact in different environments.
Decision	External configuration
Implication	**Configurability** (+), Deployability (+), Co-existence (+)
Technology	Zookeeper, Hadoop HDFS, Spring Framework

D2. Smaller, more loosely coupled units

Our participants confirm that DevOps-driven architectures need to be loosely coupled to ensure team members working on the architecture would have a minimum dependency. By this, developers can get their works (develop, modify and test) done with high isolation [35]. This resulted in **smaller, more loosely coupled units** are considered as the foundation of the architectures in DevOps context for both the projects of TeamA and TeamB. For example, we have:

> "*I guess [we are] trying to be mindful to reduce coupling between different parts of the application as we can separate those things*" **PB1**.
>
> "*[In this new architecture] we want to be able to address each of these stories, without being tightly coupled to another*" **D**.

The interviewees reported that the main reason behind this decision was to improve **testability** and **deployability**. If the decoupled architecture is fulfilled, it appears that team members can independently and easily test (e.g., better test coverage) units and drastically decrease the cycle time of test and deployment processes. One interviewee, **PA1**, affirmed that they were successful in implementing this principle as everything for unit tests is independent and substitutable, which allows them to do mocking if they need it. In both projects, the interviewees explained that decoupled architecture was achieved by extensive use of **dependency injection**[27], **feature toggle**[28], and building units that are **backward and forward compatible**. A participant from TeamA put it like this:

> "*In terms of architecture, [we] build things that they are decoupled through interfaces, using things like dependency injection, using feature flags (feature on/off and if it is not ready for production; in the application features toggle that); these are [the topics related to] the architecture that we use to support deployability*". **PA2.**

This architectural decision enabled the teams to make sure everything is nicely separable, reconfigurable, and extensible. According to the participants, all self-contained applications and components are intentionally designed small to be tested in isolation. As an example, we have:

> "*Number one thing [in DevOps] is to have software to be well tested, which you need to separate concerns into separate components that you can test individual piece of functionality [without] influencing other components*" **PB1**.

PB2 discussed the benefit of small and loosely coupled units from a different perspective. According to him, breaking down a large component (called Enricher) into five smaller and independent units enabled them to increase test coverage of each to 90%. **PA3**, on the other hand, pointed out that this was also helpful for having more efficient deployment pipeline as large

[27] https://martinfowler.com/articles/injection.html

[28] https://martinfowler.com/articles/feature-toggles.html

components increase the deployment pipeline time and are hurdles for having quick and direct feedback from test results.

Table 7.2 Decision for "smaller, more loosely coupled units"

Concern	Application's architecture should allow the team to develop and deliver software more quickly.
Decision	Smaller, more loosely coupled units
Implication	**Testability** (+), **Deployability** (+), Modularity (+), Team collaboration (+), Time behaviour of pipeline (+), Modifiability (+/-)
Technology	Feature toggle, Dependency injection

D3. One monolithic deployment unit vs. multiple deployment units

Whilst TeamA and TeamB had many common architectural decisions (e.g., external configuration) to architect their systems in the context of DevOps, they differently deal with deployment process. This significantly impacted on their architectural decisions. TeamA had started with microservices architecture style, but once the team felt difficulties in the deployment process, they switched to monolith to have the minimum number of deployment units in operations. This significantly helped them to address the deployment's challenges encountered earlier with the microservices style. **PA2** and **PA3** explained it in the following words:

> "*We actually started with microservices. The reason was that we wanted to scale out some analysis components across machines. The requirements for the application changed, and this led to [move to] this monolith*" and "*We deploy it [the application] as one JAR file*".

The following reasons were mentioned by TeamA members to build a monolithic-aware architecture: (i) they found it is much easier to deploy one JAR file instead of deploying multiple deployment units. This mainly because the deployment units are always required to be locked at the end, as changes are made to all of them at the same time. (ii) Having multiple deployment units can increase overhead in operations time. When **PA1** was asked why their system (i.e., platform) uses one monolithic deployment, he replied: *"There was a lot of overhead in trying to make sure that this version is backward compatible this version of that one. So, we make the whole thing as one big monolithic application"*. (iii) It can be difficult to manage changes in the development side, where there are multiple, independent deployment units. In this scenario, TeamA found that it is easier to bundle all units together into one monolithic release. **PA1** added: "*We are doing monolith. I think that's been slowly changing idea; we were originally going to have several suites of [deployment units] that work together but I think to cut down on development effort we are just going to put all in one tool*".

Table 7.3 Decision for "one monolithic deployment unit"

Concern	The number of deployment units should be minimum in production.
Decision	One monolithic deployment unit in production.
Implication	**Deployment** (+), Supportability (+), Operability (+), Time behaviour of pipeline (-), Scalability (-), Testability (-)
Technology	-

Opposite scenario happened to TeamB's project as they invested a lot of time and effort in re-architecting their monolithic system to have multiple deployment units. As expressed by **PB2**, "*We cannot have a thing like TeamA [as] they have one monolithic JAR, and everything is just in* *there and just deploy to one post. We are very much more microservices; we like to be far away from that [monolithic deployment] as much as possible*". This new architecture, is called microarchitecture by TeamB, is a combination of microservices and monolith approaches. Among others (e.g., parts of TeamB's platform should scale and be tested independently of others), we observed that evolvability (modifiability) was the main business driver for this transition. A member from TeamB commented:

> "*We used to have one big monolithic application and changing anything required to redeployment the whole application, which interrupted everything and interrupted all processes. [It is mainly] because we couldn't change anything really without taking down others*" **PB1**.

TeamB uses bounded contexts and domain-driven design [166] as a way to split large domains, resulting in smaller, independent and autonomous deployment units (e.g., microservices and self-contained applications). TeamB's project includes more than forty libraries, microservices, and autonomous applications that automatically and independently are built and deployed. They are very small, and the intention was that keeping theme small as each of them should be single bounded and should do *only* one specific task. This enabled them to minimize external dependencies. **PB2** reported:

> "*We have seven Ingestion apps running. So, every single app has very specific things that it does, like Twitter Ingestor only does Tweets; the minimum stuff that they can do*".

Table 7.4 Decision for "multiple deployment units in production"

Concern	Parts of the system should be tested, deployed and scaled independently.
Decision	Multiple deployment units in production.
Implication	**Modifiability** (+), Testability (+), Scalability (+), Deployability (+), Modularity (+), Supportability (-)
Technology	Domain driven design, bounded context

D4. One massive repository vs. one repository per unit

Another difference we discovered between TeamA's and TeamB's projects was their decision about the repository. Driven by the team's intention to build, test and deploy the artefacts in isolation, the TeamB has decided to build and maintain one repository per each artefact. They were frustrated with keeping all artefacts into one repository because with a monolithic repository, changing one thing required running all tests and re-deploy all things. **PB3** referred to this decision as a key architectural decision which was deliberately made to simplify DevOps transition because it helped them to manage versioning of different libraries, modules, and self-contained applications in the deployment process. Interviewee **PB1** had a similar opinion to **PB3** and said:

> "*I guess number one thing is that reusable libraries or components should potentially be in own artefact. So, you can build, test and deploy that artefact in isolation. With the monolith [repository] you change one thing to test that, the monolith is going to run all tests for everything for being (re) deployed*".

We discovered that the concern about cycle time of deployment pipeline was another reason to move from one massive repository to multiple repositories in the TeamB's project. **PB3** described it as "*build cycle in our old architecture was problematic as it hit the development and iteration*". In the previous architecture, *only* build cycle took around 10 minutes. By re-architecting the deployment pipeline and having an individual repository for each artefact, the TeamB was enabled to rebuild a component that it has changed without rebuilding others. Therefore, the cycle time of the deployment pipeline turns around as currently all the build time, Ansiblizing the deployment process, and release to Nexus[29] take approximately 10 minutes.

Table 7.5 Decision for "one repository per unit/artefact"

Concern	Build, test and deployment of each artefact should be performed in isolation.
Decision	One repository per unit/artefact
Implication	**Time behaviour of pipeline** (+), Testability (+), Deployability (+), Modifiability (+), Team collaboration (+), Versioning artefacts (+)
Technology	-

Contrast to TeamB, TeamA uses one repository for all modules and libraries. This decision is heavily influenced by their monolithic architecture design. According to TeamA, their project used to have multiple repositories for some of components and libraries. This caused challenges for them in the deployment process such as it makes harder to synchronize units' changes and it presents the integration problem. As explained by **PA3**,

> "*So, if you make a change to one component, it is still built and passed the tests but when they would need to be together, it wouldn't work. So, we merged all the components in the same repository, which then alleviated all those issues*".

Table 7.6 Decision for "one massive repository for all units/artefacts"

Concern	Managing repository should be aligned with the architecture design.
Decision	One massive repository for all units/artefacts
Implication	**Operability** (+), Modifiability (+/-), Supportability (+)
Technology	-

D5. Application should capture and report status properly

Monitoring and logging were identified and discussed by all interviewees as a primary area, which needs more attention in DevOps context. In both projects, they build logs and metrics data into software applications as these data can be leveraged by monitoring tools. Consequently, applications in DevOps context need to capture and report data about their operations properly [178]. The system architect in TeamA (**PA3**) believed that this is the most important implication driven by DevOps in terms of architecture. Another interviewee described the relationship between software architecture and monitoring as follows:

> "*In order to trust your ecosystem works especially when you are changing things to be continuously deployed; if you don't have strong architecture, then the application' changes are less clear and is more difficult to monitor*" **PB1**.

Both TeamA and TeamB use two monitoring infrastructures, Consul and Ganglia, which are shared among a couple of projects in the case company. These systems are used for different purposes. Consul[30] is mostly used as a high service to check the state of an application. This monitoring system aggregates and orchestrates metrics for cluster state, in which both the teams can utilize them to identify components' changes among *critical*, *warring* or *good* states.

[29] https://www.sonatype.com/nexus-repository-sonatype
[30] https://www.consul.io/

According to an interviewee, this enables them to prioritize stability issues in clusters and from that, they are able to implement new fixes and deploy those fixes and capture the relevant information. As opposed to *good, critical,* and *warring* states, Ganglia[31] is used for aggregating metrics like disc usages, input and output network, interfaces, memory usage, CPU usage. A software engineer in TeamB described the function of Ganglia system as follows:

> "*It [Ganglia] is kind of aggregating the information. For example, this cluster node is constantly 100% CPU usage, this cluster almost is full disc usage. We can identify those stability issues as well and find out them early before becomes a real problem*" **PB3**.

It appears that whilst both the teams extensively improve their logging and monitoring capability using Consul and Ganglia to become DevOps, they are *not* interested in metrics data analytics. Apparently, the log and metrics data are mostly used for diagnostic purposes and they are not used for an introspective purpose- understanding how an application/service can be improved based on runtime data. We found that the low number of end users was the reason why TeamA and TeamB are not interested in metric analysis. A participant told us:

> "*We don't do a lot of analysis of the metrics. We do collect a lot of metrics, [for example] we have some centralized logging and centralized monitoring servers and pull all stuff in there, but it is mostly for diagnostic if the application fails and trying to figure out what went wrong*" **PA1**.

In both projects, the abundance of monitoring and logging data produced by the applications presents severe challenges, in which scaling up machines and capacities cannot solve the problem anymore. As a result, most participants felt that they have to re-think about their logging and monitoring approaches and how to reduce the size of logs. One participant complained about this pain point as follows:

> "*We've got Consul to let us know memory usage in that computer, we've got a server, that executed by multi-tenant solution, and each of them is running their own application that may spin up in parallel; that consumes memory. So, you get the capacity issues. So maybe the solution was to scale out machine initially, but now I think that is an issue again*" **PB1**.

Table 7.7 Decision for "application should capture and report status properly"

Concern	How observability of an application can increase in DevOps context.
Decision	Application should capture and report status properly.
Implication	**Monitorability** (+), Loggability (+), Supportability (+), Resources utilization (+), Analysability (+), Cost (-)
Technology	Consul, Ganglia

D6. Application should be standalone, self-contained

We found in the interviews frequent references to how the applications can get easily and reliably deployed to different environments. Besides applying "external configuration", both the teams have decided that the deployment of an application or service should be locked and independent to other applications and services that it depends on. To this end, they have adopted a self-contained and self-hosting approach [190], in which all dependencies need to be bundled to an application. An interviewee described the impact of this approach as:

[31] http://ganglia.sourceforge.net/

> "*Each of them [applications] like Ingestors are considered as a single standalone, self-contained application, the pipeline is a self-contained app. They don't really talk with any things else, they don't have any external dependencies like taking one of those down doesn't affect the other ones*" **PB2**.

To avoid interruptions of self-contained applications at runtime, they provide, where possible, the necessary infrastructures (e.g., load balancers) and multiple instances of applications or services behind them. Using Apache Kafka[32] also enables them to segregate service components and isolate them as they can be easily swapped in/with new ones at operations time:

> "*By using Kafka Buffer, you take a component out and then message its Buffers to either redeployed artefacts or new artefacts that are connected and consume messaging*" **PB3**.

Ansible[33] tool is used as the main automation deployment tool to make sure everything is deployable using infrastructure-as-code. Our interviewees reported two main methods to achieve this. First, it should be ensured that everything along with its dependencies that is going to be deployed should be captured in Ansible. Second, Ansible playbooks should be written well. From TeamB's architect perspective, writing good Ansible playbooks was the biggest challenge for them in the path of DevOps adoption and he defined a "good" Ansible playbook as: "*I can run, there is no crush in the system if [there are] no changes. If no changes need to be made, [it] skips those little tasks*" **PB2**.

In TeamB's project, everything is stateless as well. In addition, data fields are designed optional, which they can either exist or null. This was indeed helpful to the deployment process because there is no need to have the right database with the right data in it to restart the self-contained applications. The TeamB extensively uses Zookeeper[34] tool to make code stateless. **PB2** explained:

> *"A good thing about [the applications] is that there is a clean separation between the apps. So, I can go and update Ingestors; I can go and updates DTO and add more data or meta-data to it without affecting anything downstream like I just continue operating on the old version of schema*".

Table 7.8 Decision for "application should be standalone, self-contained"

Concern	Application in DevOps context should have the least dependency with others in production (i.e., operational efficiency).
Decision	Application should be standalone, self-contained.
Implication	**Deployability** (+), Replaceability (+), Recoverability (+), Availability (+), Team effort (-), Cost (-)
Technology	Zookeeper, Kafka, Ansible

D7. Three distinct environments are provided to support different levels of development and deployment processes

Providing sufficient physical resources (e.g., CPU and memories) to both the teams enabled them to establish three distinct environments including *development*, *integration* (or *test*), and *production* environments. In TeamA and TeamB, this was indeed a conscious decision to manage different levels of development, testing, and deployment in DevOps transformation [7].

[32] https://kafka.apache.org/
[33] https://www.ansible.com/
[34] https://zookeeper.apache.org/

Table 7.9 How often the teams deploy to different environments

	Deployment Frequency to		
	Development Environment	Integration Environment	Production Environment
TeamA's Project	Multiple-time a day	Once per week	Once per Sprint (two-week)
TeamB's Project	Multiple-time a day	Multiple-time per Sprint	At least once per Sprint

Table 7.9 shows the frequency of deployments to these environments is different. They strictly follow three upfront rules in the above-mentioned environments to ease DevOps adoption: (i) in the development environment ***only*** unit tests need to be performed, and integration environment should include ***all dependencies*** to properly run integration tests against all snapshot builds. One participant described the test environment like this:

> "*You can bring to own laptop all the infrastructure requirements of the cluster like Kafka Buffer, Postgre databases and Zookeeper instances. Then you locally test against that and then Jenkins automatically tests against the test environment*" **PB1**.

(ii) Code reviews should be performed before committing to the development branch. TeamA and TeamB use the Gitflow branching model[35], in which developers do their work on feature branches and before merging changes to the development branch they should raise a pull request to review the changes in Bitbucket[36] [57]. Then that pull is merged to the development branch. Except for large tasks which might not be merge-able in few days, multiple pull requests occur a day. The interviewee **PB2** reported that pull request approach "*would improve the merge [quality], and [make] development branch builds always ready for snapshot and deploying to the test environment*".

(iii) Critical bugs should be fixed ***only*** on release candidate branch, not on the development branch. This working style along with automation support was deemed helpful to faster repair bugs and reduce the risk of deployed changes to production. The interviewee **PB3** said: "*If there is an issue, we can quickly fix and redeploy quickly. So, the turnaround is very small because all are automated at the moment*".

Applying the above-mentioned principles leads to the development branch being potentially in the releasable state anytime, but it is not necessarily stable. An interviewee summarized it as:

> "*It [main branch] is a stable artefact. The development branch, which is all integrated features, ready to be merged into master anytime, [but] that might not be quite stable*" **PB3**.

Therefore, both the teams have always stable version software that can be released anytime. Although they deploy to production at the end of Sprint, actual production deployments are not tied to the Sprint and can be frequent. One interviewee said:

> "*We are working two weeks Sprint. We always try to our releasable is done in two weeks Sprint but often [we have] more releases, quite often. I think it is releasable in a couple of times a week*" **PB1**.

> "*Releases are not tied to the Sprint tempo - they can be more or less frequent*" **D**.

[35] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
[36] https://bitbucket.org/

Table 7.10. Decision for "Three distinct environments should be provided to support different levels of development, testing, and deployment."

Concern	How development and deployment processes can be easier for DevOps context.
Decision	Three distinct environments are provided to support different levels of development, testing, and deployment
Implication	**Supportability** (+), Testability (+), Deployability (+), Cost (-)
Technology	-

D8. Teams should be cross-functional, autonomous

Another key decision made by the studied organization to support DevOps transition is to set up small, cross-functional, self-organized teams. Both studied teams have **end-to-end ownership** of their own products, from design to deployment to supporting the code in production: "*The significant part of engineering team's time is to maintain the DevOps aspect of the app– I mean the complexity of operations as we have to maintain infrastructure as code and understand that, we have to spend time looking at metrics*" **PB1**. Furthermore, each team is a **full-stack team**, in which not only all skills exist in the team, but also everybody should know about testing and operations skills (e.g., writing Ansible playbooks). In both the teams, testing is considered as a rotating activity, not as a phase, which should be performed by all developers. It should be noted that some software engineers have been frustrated with doing operations tasks as this can be a source of distraction for doing real software development tasks (e.g., debugging). As an example, we have:

> "*Sometimes you [as a developer] spend a lot of time on debugging Ansible and Jenkins and infrastructures things, for example why Kafka Buffer is not coming from this host. It is kind of being dust by DevOps on deployment task*" **PB2**.

Both TeamA and TeamB are also **autonomous** by which they have the freedom to act independently and make their own architectural and technology decisions. For instance, as discussed earlier, they have chosen two different approaches for architecting their systems. **Collaboration culture** is well established in both the teams through co-design and shared responsibilities. Regarding the partnership in testing activity, the interviewee **PA1** added: "*We are rotating testing role so one person is responsible for testing everything in previous Sprint and then press proof button and say: to get deployed to the production*". Whilst each team has an architect inside it, other team members actively and substantially participated in the architecting process. According to an interviewee, the feedbacks of other team members constantly improve the architecture. Architects are mainly responsible for designing and evaluating the systems, however, in addition to be an architect they also do coding and testing. The architect of TeamA put the collaborative design in these words:

> "*Everybody is involved in architecture [design], we are not going to have only I as a solution architect, everyone in the team can do that. The architecture is done in daily work and everyone can have his [own] idea*" **PA2**.

Table 7.11 Decision for "Teams should be cross-functional, autonomous."

Concern	How teams should be reorganized to effectively adopt DevOps.
Decision	Teams should be cross-functional, autonomous.
Implication	**Team collaboration** (+), Team effort (-)
Technology	Confluence, HipChat, JIRA, Bitbucket

7.4 Lessons Learned

In the previous section, we have provided the key insights about the architectural decisions and tactics and their potential implications that fall under name DevOps in our case study. In the following, we discuss some of the key lessons learned in the context of related works and the implications of our results, which might be helpful for other organizations and practitioners trying to adopt DevOps.

Microservice-based architectures are not only DevOps-driven architectures: Whilst many software organizations [129, 147, 191] have adopted microservice-based architecture as a driver to succeed in DevOps, our results in this study have taught us that achieving a truly DevOps-driven architecture requires loosely coupled architectures and prioritizing deployability, testability, modifiability, monitorability, and loggability over other quality attributes. Our findings in this chapter and previous chapter are in line with recently published technical report by 2017 State of DevOps Report [35], which reveals that apart from architecting a system with microservices style or service-oriented architecture, loosely coupled architectures and teams are the biggest contributors to DevOps success (i.e., continuous delivery enabled by DevOps). Etsy is a notable example of this scenario as it has successfully implemented DevOps using a monolithic system [176]. Whilst both TeamA and TeamB realized that the fundamental limitation to rapid and safe delivery resides in the architectures of their systems, we *did* find significant differences in the architectural approaches employed by TeamA and TeamB to architect their systems: monolith vs. microarchitecture. Here monolith means a single build system, in which all the functionality is managed and deployed in one deployable unit [152, 153].

This does not necessarily mean that a monolith is a highly coupled architecture. TeamA's project uses one monolithic deployment unit, whilst TeamB's project is composed of several autonomous, self-contained applications and microservices, which are built, tested, and deployed independently from each other. We observed that TeamB is more likely to have loosely coupled architecture than that of TeamA. It is mainly because changing architecture in TeamA's project is not straightforward as the architect needs to get involved frequently when changes happen. Conversely, TeamB members could apply small-scale changes to their architecture without the need for communication and coordination with the software architect. However, in both systems applying large-scale changes requires involving architects. Another sign that shows the architecture of TeamB's project is more modular is that TeamB has higher deployment frequency to lower environments (e.g., test environment) compared to TeamA (See Table 7.9).

Our results in the previous chapter show that the architectures in DevOps context should support evolutionary changes. This implies that architectural decisions should be delayed until they are necessary (i.e., delaying decisions to the last possible moment) [168]. However, it does not mean that there is no longer a need to have a long-term vision of architecture. Put another way, core architectural decisions need to be made at the early stage of development in DevOps context. By ignoring this necessity, TeamA experienced a pain point in the architecting process as they ended up major refactoring of the whole stack of their system several times.

Operations specialists always required: Our study shows that establishing cross-functional, autonomous teams was one of the key decisions made in the case company to implement DevOps. We observed that operations expert is not embodied as a distinct role on both TeamA and TeamB or there is no a dedicated operations team in the case company. Operations skills are regarded as a skillset that blends with other skills such as software development and need to be performed by developers. We have observed that performing the operations tasks that require a deep expertise in operations (e.g., provisioning infrastructures) is burdensome for developers.

This lead to the significant part of developers' time is spent on operations tasks (e.g., writing Ansible playbook) rather than real software development tasks. Furthermore, the participants in this study found that this can give a good excuse for them to do not perfectly perform operations tasks. This can be exemplified by the Ansible playbooks written by the developers as part of their new responsibilities in DevOps transformation. Whilst teams' members found that the written Ansible playbooks are not good, we found that there is no any demand to improve them. In our view, this could originate from the fact that operations responsibilities are ambiguous for developers and some of the operations tasks are not clear who should do. Furthermore, in DevOps transformation development side is more emphasized than operations side (See Section 6.5.4 in Chapter 6 for more information). This could be due to the fact that most of the business values come from the development side (e.g., adding more features).

Whilst the interviewees strongly emphasized that deployment tasks should be performed by developers in the context of DevOps, but we emphasize that operations specialties need to be embedded in each team for complex operations tasks [192, 193]. This really would be beneficial for development people as they can more concentrate on real software engineering's tasks. This is in line with the findings from Chapter 3, which show that the majority of the surveyed organizations (76 out of 93) prefer having distinct operations team or operations specialists for specific operations tasks.

The Road Ahead (Remaining Challenges): One of the most emphasized DevOps practices is continuous delivery or continuous deployment [21]. Both teams successfully practice continuous integration as the developers in each team integrate code into a shared repository multiple times a day [79]. However, despite having an automated deployment to production, they are not able to or do not practice continuous delivery or deployment as actual production deployment happens once per two weeks. Table 7.9 also shows that deployment pace to integration or test environment is low. As we discussed in Chapter 4, whilst deploying continuously into production might be influenced by many socio-technical factors (e.g., in this case study: the circumstance of the projects), we have observed a strong influence of test automation on rapid deployment.

The problems that both TeamA and TeamB have about testing are *not having enough test coverage, lack of comprehensive end-to-end integration test automation,* and *performance, security and acceptance tests are not part of the deployment pipeline*. All these issues have reduced the confidence of the teams to have multiple times deployments a day. Both TeamA and TeamB have started automating end-to-end integration test and improving test coverage. However, given the size of the teams and end users, and the overhead (i.e., time and cost) of developing and maintaining automated performance and acceptance tests, both teams prefer to do these sorts of tests manually. For instance, as the overhead of maintaining Selenium-based User Interface (UI) testing increased (e.g., because UI changes a lot), the TeamA found that it is better to turn off UI testing in the deployment pipeline. That is why currently they do it manually on the release branch.

Many architectural decisions reported in Section 7.3 were made to improve the testability of an application in DevOps context, however, as argued in [1, 21] organizations should ensure to have good test coverage, write tests that less consume cycle time of the deployment pipeline, and automate tests (e.g., performance) that occur at the last stages of the deployment pipeline for implementing continuous delivery and deployment. This provides confidence to deploy to production continuously and automatically.

7.5 Related Work

This section presents existing research which has investigated the role of software architecture in the context of DevOps.

Over the last seven years, Puppet[37] has annually released non-peer reviewed reports to study the current state of DevOps in practice [9, 35]. In 2017 [35], the role of software architecture in DevOps was deeply examined to investigate how application architecture, and the structure of the teams that work on, impact the delivery capability of an organization. The main finding of this report reads, "loosely coupled architectures and teams are the strongest predictors of continuous delivery". Surprisingly, it is also revealed that many so-called service-oriented architectures (e.g., microservices) in practice may prevent testing and deploying services independently from each other. Subsequently, it can negatively influence teams to develop and deliver software. The relation between software architecture and Continuous Delivery or Deployment (CD) as a key practice of DevOps was investigated in [38, 40, 75]. Schermann et al. [40] identify that monoliths are the main source of pain to practice CD in industry. Another study [38] elicit a set of quality attributes such as deployability, security, modifiability, and monitorability that need more attention to gain the maximum benefits of continuous delivery. In a retrospective study on three projects adopting continuous integration and delivery, Bellomo et al. [75] reveal that the architectural decisions made in those projects played a significant role to achieve the desired state of deployment (i.e., deployability). In [194], Di Nitto et al. outlined architecturally significant stakeholders (e.g., infrastructure provider) and their concerns (e.g., monitoring) in DevOps scenarios. Then a framework called SQUID was built, which aims at supporting the documentation of DevOps-driven software architectures and their quality properties.

An alternative line of research has attempted to leverage microservice-based architectures in DevOps context [129, 147]. Based on an experience report, Balalaie et al. [147] present the architectural patterns (e.g., change code dependency to service call) and technology decisions (e.g., using containerization to support continuous delivery) employed by a case company to re-architect a monolithic architecture into microservices in the context of DevOps. In [129], Callanan and Spillane discuss that developing a standard release path and implementing independently releasable microservices through building backward compatibility with each release were the main tactics leveraged by their respective company to smooth DevOps transformation. These tactics also significantly reduced the delays in the deployment pipeline.

7.6 Conclusion

This chapter has reported a case study with two teams in a company (i.e., it has been referred to as case company in this chapter), which provides insights into the architectural decisions, their perceived implications, and the challenges that both teams faced when architecting their systems as part of their DevOps journey. The successful architectural decisions and the challenges can serve as guidelines for architecting applications to enable and support DevOps. For example, organizations that would like to adopt DevOps may augment their applications using "external configuration" described in Section 7.3. Our findings suggest that DevOps success is best associated with modular application architectures and needs to prioritize deployability, testability, modifiability, monitorability, and loggability over other quality attributes. We also found that developers often struggle with performing those operations tasks need a deep expertise in operations.

[37] https://puppet.com/

Chapter 8

Conclusions and Future Works

This thesis presents a set of empirical studies on two key practices of DevOps, i.e., continuous delivery and deployment. More specifically, the goal of this thesis is to empirically investigate the impact of continuous delivery and deployment practices on organizational structure and software architecture, those that are supposed to be fundamental limitations to adopt these practices. To that end, we first designed and conducted a systematic literature review to assess the existing literature. Second, we designed, implemented, and analyzed a large-scale mixed-methods empirical study, consisting of 21 semi-structured interviews from 19 organizations and 98 survey responses. Finally, we conducted an industrial in-depth case study with two teams in a case company. In the remainder of this chapter, we first revisit the research questions introduced in Chapter 1, and then discuss some promising areas for the future work.

8.1 Answers to the Research Questions

RQ1. What is the state of art of continuous integration, delivery and deployment research?

- We identified thirty approaches and associated tools to facilitate the implementation of continuous integration, delivery, and deployment. These approaches and associated tools (i.e., not mutually exclusive) are expected to (i) reduce build and test time in continuous integration (9 approaches), (ii) increase visibility and awareness on build and test results in continuous integration (10 approaches), (iii) support (semi-) automated continuous testing (7 approaches), (iv) detect violations, flaws, and faults in continuous integration (11 approaches), (v) address security, scalability issues in deployment pipeline (3 approaches), and (vi) improve dependability and reliability of deployment process (3 approaches).
- We have identified 20 challenges and 13 practices for continuous integration, continuous delivery and continuous deployment practices, which can guide software organizations to adopt these practices step by step and smoothly move from one practice to another.
- We found “testing (effort and time)”, “team awareness and transparency”, “good design principles”, “customer”, “highly skilled and motivated team”, “application domain”, and “appropriate infrastructure” are the most critical factors to successfully adopt and implement continuous integration, delivery, and deployment in a given organization.

RQ2. What are the organizational impacts of continuous delivery and deployment?

- We noticed that there is an indeed difference between practicing continuous delivery and continuous deployment in industry.
- We found that making a decision to move from continuous delivery to continuous deployment is influenced by both technical and socio-technical factors.
- We looked at the current state of practice of deployment pipelines in the industry and found that "version control", "build", "unit/integration testing", "continuous integration" and "production deployment" are the most common stages of the deployment pipelines.
- We observed that the last stages of the deployment pipelines including "acceptance testing", "production deployment", and "configuration and provisioning" stages are likely to be semi-automated or manual.
- We identified 11 confounding factors that demotivate or limit organizations to move from continuous delivery practice to continuous deployment (i.e., having automatic and continuous deployment). These factors include "lack of fully automated (user) acceptance test", "manual quality check", "deployment as business decision", "insufficient level of automated test coverage", "highly bureaucratic deployment process", "lack of efficient rollback mechanism", "dependency at application level", "demotivated customer", "customer environment", "domain constraints", and "manual interpretation of test results".
- We clustered development and operations teams based on types of activities they perform in adopting continuous delivery and deployment and identified four distinct clusters for their working styles. These clusters include (i) separate Dev and Ops teams with higher collaboration; (ii) separate Dev and Ops teams with facilitator(s) in the middle; (iii) small Ops team with more responsibilities for Dev team; (iv) no visible Ops team.
- We revealed that "co-locating teams", "rapid feedback", "joint work and shared responsibility", "using (common) collaboration tools more often", "increased awareness and transparency", and "empowering and engaging operations personnel" are the main strategies attempted by software-intensive organizations to improve collaboration among teams and team members for practicing continuous delivery or deployment.
- We highlighted three key high-level changes in the responsibilities of teams or team members, which are needed to effectively initiate and implement continuous delivery and deployment, including "expanding skill-set", "adopting new solutions aligned with continuous delivery/deployment", and "prioritizing tasks".

RQ3. What are the architectural impacts of continuous delivery and deployment?

- We discussed that whilst monoliths and continuous delivery/deployment are not intrinsically oxymoron, adopting continuous delivery/deployment in monoliths is more difficult, as there are hurdles for having team autonomy, fast and quick feedback, enabling automation (e.g., test automation) and scalable deployment.
- We observed that breaking down monoliths into smaller pieces brings more flexibility in continuous delivery/deployment; however, this is a challenging process for organizations.

- We found inflexibility of organizational structure (e.g., team structure) with the spirit of continuous delivery/deployment practices is the most critical challenge for implementing these practices.
- We empirically found "small and independent deployment units" is a key principle, which is widely used as an alternative to monoliths and serves as a foundation to design continuous delivery/deployment-driven architectures.
- We revealed that autonomy in terms of deployability, modifiability, testability, scalability, and isolation of the business domain are the main characteristics of this principle.
- We found adopting vertical layering and microservices to promote delivery speed comes at a cost as it necessitates considering organizational structures and highly skilled team. Ignoring this fact may negatively impact the deployment capability of an organization.
- We found that the importance of a set of quality attributes including deployability, testability, modifiability, monitorability, loggability, and resilience has increased, but overthinking about "reusability" at architecture level may negatively impact continuous delivery/deployment adoption.
- We provided empirical evidence that designing highly operations-friendly architectures are achieved by
    - prioritizing operations concerns,
    - early and continuous engagement of operations staff in decision-making,
    - leveraging logs and metric data for operational tasks.
- We found DevOps works best with modular architectures.

8.2 Opportunities for Future Research

The research presented in this thesis constitutes a step toward an empirical understanding of the impacts of continuous delivery and deployment on software architecture and organizational structure. Despite this thesis makes a significant contribution in this regard, there are still several research areas, open challenges, and gaps which require further work. Whilst some ideas for future work have already been discussed in the previous chapters, here, we summarize them in the following areas:

8.2.1 Replicating the Study

Whilst our research is the first large-scale qualitative study on the impact of DevOps on architecture, the replication of our research with larger samples in different contexts can be a line of research for future research. This provides a more reliable generalization of the findings presented in this thesis and can be used to verify which architectural principles, practices, and challenges less or more fit for a specific context (e.g., cloud environments) [43]. Achieving larger samples would enable performing dependency analysis in the survey responses to gain insights into the correlation between them.

8.2.2 Investigating Microservices Architectures in DevOps

DevOps and microservices architectures are two important trends in software research and practice. It is argued that microservices is the first architectural style after DevOps practices gathered momentum [149]. Whilst industry community mostly discusses the benefits of microservices-based architectures to embrace DevOps practices, our observation in Chapter 6 shows that software organizations are more interested in designing architectures that are much less monolithic or only adopting the microservices architectural style in a minor way for DevOps practices. There may be several reasons for this strategy such as service granularity complexity, inflexibility of the organization's structure with target architectures, and lack of strong evidence about the long-term success and benefits of microservices compared to its challenges (e.g., testing can be more complex) and drawbacks [162]. Hence, it would be interesting to empirically investigate the challenges and benefits of microservices-based architectures with DevOps practices. This may also lead to some guidelines for improving the adoption of microservices-based architectures within the context of DevOps.

8.2.3 Understanding the Role of Software Architect in DevOps

In the empirical studies presented in Chapters 5, 6 and 7, we found that the roles and responsibilities of software architects have changed in the context of DevOps as they do not only design architecture; they often must deal with and guide infrastructure architecture, test architecture, and team structure. Recent studies [21, 142, 195] also discuss why the architect is important in moving to DevOps. Future works should focus on how software architects perform their work in DevOps context to provide guidance for required job skills and education programs.

8.2.4 Studying the Impact of DevOps on Operations Responsibilities

In Chapter 5, we uncovered several changes in team members' roles and responsibilities for moving into DevOps practices. Our focus in Chapter 5 and other studies [62] was mostly on the engineering team (e.g., developer and architect) rather than operations team. However, it is argued that operators are also heavily influenced by DevOps [196]. Yet, it is unclear how DevOps changes operations responsibilities and what operations teams inside an organization would need to do/make for DevOps success. A better understanding of operations responsibilities' changes under DevOps may help improve the importance of operations teams in the software development process as it is highly emphasized that they should be treated as first-class stakeholders in DevOps.

Appendix A

Interview Guide for the Mixed-methods Study

Title: (Re-) Architecting for DevOps, Continuous Delivery and Deployment Practice

Research Team: Mojtaba Shahin, M. Ali Babar and Liming Zhu

Ethics Approval Number: H-2015-270

The University of Adelaide, Australia

General Information

This study is exploring the impacts of DevOps (development and operations) and Continuous DElivery (CDE) and Deployment (CD) practices on architecting process and how appropriate architecture design of the software can help achieve highly frequent and reliable deployment into production. We are interviewing software practitioners who either have experiences in architecting for DevOps/CDE/CD (e.g., architect) or are closely involved with or influenced by architecture (e.g., maintainer, operations engineer). We call them architecturally significant stakeholders.

Interview guide

The interview will be semi-structured. The schedule will include the following topics and questions:

Interviewee's background

- What are your role and responsibilities in the project team?
- How long have you performed that role?
- Have you had any experience as a software architect? If so, how long?
- Please talk about the size and domain of your organization.

Project's description

[**Note**: The project(s) adopting DevOps/CDE/CD practice or one of the continuous practices such as continuous integration, delivery, deployment, deployment automation or microservices were/are the major focus in this project]

- What is/was the domain and type of project?
- Team size: how many people are/were involved in this project?

- [Specific challenges for DevOps/CDE/CD context] Please give a brief overview of challenges that you might experience in this project? How could you manage these challenges?

Deployment pipeline and deployment frequency

- How often is/was the application in a releasable state?
- How often do/did you deploy the application to production or the customer?
- Was/is the deployment pipeline fully automated or not? If not, why?

Dev and Ops teams

- In this project, are/were the development and operations tasks performed by the same team?
- Could you describe your team structure before and after the adoption of DevOps/CDE/CD?
    - How adopting DevOps/CDE/CD could change your team structure?
- How could the adoption of DevOps/CDE/CD change your daily work activities?
- How do/did you consider operations requirements in the design process?
- Do/Did you involve operations stakeholders in the decision-making process at the early stage of software development? If so, how?
- Do/Did you have any difficulties in prioritizing the concerns of operations stakeholders with other stakeholders?
    - How did you manage them?

Architecture and Continuous Delivery/Deployment Practices

[**Note**: deployability is a quality attribute (non-functional requirement) which means how reliably and easily an application/component/service can be deployed to (heterogeneous) production environment]

- How would you describe the relationship between architecture and CDE/CD practice?
- How does the adoption of CDE/CD practice change the architecting process?
- How do/did you design the architecture of your system to enable and better support DevOps/CDE/CD practice e.g., improving deployability, easily changing and evolving code at the early stage of the development?
    - How do/did you predict and evaluate the deployability of your software system during the design process?
    - When you are/were designing the system, how you make/made trade-offs between design options in order to satisfy the deployability?
    - Can you provide examples of architectural decisions you made for improving deployability?
- What architecture principles and practices (e.g., patterns, styles, and tactics) do/did you employ to promote and support CD practice?

  - How do you break down the monolithic applications into independently deployable units/components/services?
    - Are you using any criteria for this purpose?
  - Are you using microservices style for this purpose?
    - What about other techniques for this purpose?
- What challenges do/did you experience whilst designing application the application architecture for CDE/CD?
  - Can you provide some examples of these sub-optimal architectural decisions that let to deployment-related limitation? i.e., architectural decisions that impeded the desired level of deployment/led to substantial technical debt
- What quality attributes are more influenced by the CDE/CD context? How is this so?
- What quality attributes are in support of or in conflict with deployability? How is this so?
- What is your opinion about logging and monitoring in DevOps/CDE/CD context?
  - Can you provide one example of these architectural decisions you made for this purpose?
- What kind of runtime data (e.g., production data, operations metrics and runtime data) did you utilize to make informed DevOps-specific (architectural) design decisions?
  - Can you provide examples of these data?

Are there any comments and issues you want us to know?

Can you please suggest any suitable persons participate in this interview?

Appendix B

Survey Instrument

Title: (Re-) Architecting for DevOps, Continuous Delivery and Deployment Practices

Research Team: Mojtaba Shahin, Prof. M. Ali Babar, Dr. Liming Zhu, and Dr. Mansooreh Zahedi

Ethics Approval Number: H-2015-270

The University of Adelaide, Australia

General Information

This survey aims at investigating how applications should be (re-) architected to enable and better support the principles of DevOps (Development and Operations) and Continuous DElivery/Deployment practices (i.e., frequently and reliably releasing new features and products with as much automation support as possible, close collaboration between Development and Operations teams, etc.). Furthermore, this survey is aimed at quantifying and augmenting the initial findings obtained from interviews with software industrial practitioners. If you are an industrial practitioner (e.g., developer, architect, consultant, program manager, tester, operations engineer, etc.) who works in DevOps context or DevOps practices, e.g., Continuous Integration (CI), Continuous DElivery (CDE), Continuous Deployment (CD), Microservices, and Deployment Automation, we would be greatly appreciative if you kindly respond to the following questions.

Useful Definitions

Architecting is a process of designing and evaluating software architecture of a system; essentially it's a decision making for devising and assessing design solutions. Both conceptual solutions (e.g., architectural patterns and tactics) and technology solutions (e.g., tools) are considered as architectural solutions and subject to architectural decisions.

Deployability is a quality attribute (non-functional requirement) which means how reliably and easily an application/component/service can be deployed to (heterogeneous) production environment

Continuous DElivery (CDE) practice is aimed at ensuring an application being always in the releasable state (i.e., an application can potentially be deployed at any time to production-like/stage environment).

Continuous Deployment (CD) practice goes a step further and automatically and steadily deploys the application or changes to a production environment without human intervention.

It is argued that while CD practice implies CDE practice, the converse is not true. The scope of CDE does not include continuous and automated release, and CD is consequently a continuation of CDE. There should be NO manual steps or decisions in CD, in which as soon as developers commit a change, the change is automatically deployed to production through a continuous deployment pipeline.

Your participation is important to enable us to correctly understand and characterize the challenges and key practices that should be considered when an application is (re-) architected for CD practice. The results will be used to develop recommendations for better practices and tool support in this regard.

Ethics Approval

This study has been approved by the Human Research Ethics Committee at the University of Adelaide (approval number H-2015-270) (you can download the Ethics Approval from this link: https://mojtabashahin.files.wordpress.com/2016/02/h-2015-270-babar-amendment-approval.pdf).

Consent

Participation in this survey is completely voluntary. You are free to decide to withdraw yourself or your information prior to the survey being submitted without any penalty. It should be noted that because participants are non-identifiable in this study, it will not be possible for participants to withdraw themselves and their information once they would have submitted the responses (as it will not be possible to determine which response is theirs). Therefore, the participants are only able to withdrawal themselves and their information prior to submitting their responses.

All data collected from this survey will be subject to confidentiality (i.e., the anonymity of individuals and their respective organizations). We will NOT attribute your responses to any participant. You will not be identified in the potential publications in the future and your personal results will not be divulged. Answering this survey may take about 20 minutes. For most parts of this survey, you only need to rate how strongly you agree or disagree on the proposed statements. If you are interested to receive the results of the survey when published, you can provide your email address at the end of the survey. We invite practitioners who have experience in DevOps, Software Architecture, Continuous DElivery (CDE), Continuous Deployment (CD), Microservices and Deployment Automation, from different organizations with divergent domains and of varying business sizes. Particularly, we would be greatly appreciative of the practitioners who work for organizations that consider software architecture as a contributing factor to adopt and implement CDE or CD practices. More information about this study can be found in "Participant Information Sheet" (https://mojtabashahin.files.wordpress.com/2016/02/participant-information-sheet_survey.pdf).

It is worth noting that the completion and submission of this survey indicate that you consent to be involved in this survey study.

Benefit

It is not expected that you will benefit from participating in this survey, apart from knowing that your contribution will help contribute an evidence-based body of knowledge to support further development and adoption of CD practice.

Take an appreciation of the participants' time, we are holding a drawing for five newly published DevOps book entitled "DevOps: A Software Architect's Perspective", SEI Series, Addison-Wesley, 2015" among all survey participants. Furthermore, we will be happy to provide with an earlier version of the final report, so you would access to the knowledge and understanding produced based on this study before a wider audience of researchers and practitioners.

Contact

If you have any question about this study, please contact us:

Mr. Mojtaba Shahin, mojtaba.shahin@adelaide.edu.au || mojtabashahin@gmail.com, Tel: +61 831 34519

Prof. Muhammad Ali Babar, ali.babr@adelaide.edu.au

Dr. Liming Zhu, liming.zhu@nicta.com.au

Dr. Mansooreh Zahedi, mzah@itu.dk.

This survey is being conducted by Mr. Mojtaba Shahin, Prof. M. Ali Babar, Dr. Liming Zhu and Dr. Mansooreh Zahedi from The University of Adelaide, Data61, Australia and IT University of Copenhagen, Denmark.

What if I have a complaint or any concerns?

If you have questions or problems associated with the practical aspects of your participation in the project or wish to raise a concern or complaint about the project, then you should consult the Principal Investigator. Contact the Human Research Ethics Committee's Secretariat on phone +61 8 8313 6028 or by email to hrec@adelaide.edu.au. If you wish to speak with an independent person regarding concerns or a complaint, the University's policy on research involving human participants, or your rights as a participant. Any complaint or concern will be treated in confidence and fully investigated. You will be informed of the outcome.

Thanks a lot in advance for your support!

You are free to forward the survey to your colleagues who are eligible for participation without our consent.

You may only participate in this survey if and only if:

- I am 18 years old or older
- I am working professionally in software/IT industry and my organization adopted or is adopting one of the following DevOps practices: Continuous DElivery (CDE), Continuous Deployment (CD), Microservices, and Deployment Automation. Or I am an independent practitioner/consultant and have experience with the above-mentioned practices.

Survey Questions

Questions	**Scale**
Demographic Questions	
Q1. How many years have you worked in software or IT industry?	0–2 / 3–5 / 6–10 / > 10 years
Q2. What is your role in the development project?	Developer / Architect / Tester / QA / …
Q3. How large is your organization?	1–100 / 101-1000 / >1000 employees
Q4. What is the domain of your organization?	Consulting and IT services / Embedded system/ …
Q5. On average, how many people work on the projects that adopted/are adopting CDE/CD practice?	Just me / 2–6 / 7–15 / 16-30 / > 30

Q6. How long ago has your organization adopted CD/CDE/Microservices/Deployment Automation practices?	0–1 / 2–4 / 5–8 / > 8 years
Continuous Delivery/Deployment Pipeline and Automation	
Q7. Regarding the design and implementation of continuous delivery/deployment pipeline in my organization:	1. A dedicated/centered team in our organization has been established to design, develop and maintain a continuous deployment pipeline that would work best for the organization. Then other teams use that pipeline. 2. The continuous deployment pipeline has been built by a temporarily established team in our organization and then other teams use that pipeline. 3. An external consulting organization has built the continuous deployment pipeline for us and the teams in our organization have been trained by the consulting organization to use that pipeline. 4. Other (Specify)
Q8. On average, how often is your application in a releasable state (i.e., production-ready)?	Multiple times a day / Once a day / A few times (e.g., one or two) a week / A few times (e.g., one or two) a month / A few times (e.g., one or two) a year
Q9. On average, how often do you deploy/release your application to production or the customer environment?	Multiple times a day / Once a day / A few times (e.g., one or two) a week / A few times (e.g., one or two) a month / A few times (e.g., one or two) a year
Q10. Imagine your application is theoretically "production-ready" (i.e., successfully adopting CDE practice), which of the following issues or factors (if any) limit or demotivate you to "continuously" and "automatically" deploy the application or changes to "customer/production environment" (CD practice):	1. Lack of fully user acceptance test automation. 2. Deployment process in our organization or our client organization is still highly bureaucratic (e.g., a large number of formal tasks, including documentations should be manually filled out and signed before each release). 3. Quality assurance team has to manually check and confirm the application before each release. 4. We need to do a lot of manual and complex configurations in a customer environment before a software release. 5. We should deliver software to our customer based on specific and predefined timeslots (i.e., calendar-based release). 6. Interpretation of test results is so time-consuming and labor-intensive process. 7. Deploying to production is considered as a business decision, which should be made by management and financial sectors. It is out of low-level stakeholders' control (e.g., developers) to deploy every change immediately to production. 8. Domain constraints (e.g., embedded system) do not allow us to deploy to customer environment frequently.

	9. The application still has a lot of dependencies with other applications (i.e., integration problem with other applications in the deployment process). 10. We don't have an efficient rollback mechanism to quickly recover issues in the deployment process. 11. Insufficient level of automated test coverage reduces our confidence in the readiness of the applications for deployment. 12. We do not face any challenge in continuously and automatically deploying software to customer/production environment at all. 13. Other (Specify):
Q11. How would you grade your continuous delivery/deployment pipeline in terms of automation?	1 (completely manual) / 2 / 3 / 4 / 5 (completely automated)
Q12. Which of the following phases constitutes your continuous delivery/deployment pipeline?	Version Control / Build / Continuous Integration / Artifact Repository Management/ Unit and Integration Testing / Acceptance Testing / Deployment / Configuration and Provisioning / Log Management and Monitoring / Containerization / Other (Specify)
Q13. Which phase(s) in your continuous delivery/deployment pipeline has the most automation support? Why?	Free text
Q14. Which phase(s) in your continuous delivery/deployment pipeline has the least automation support? Why?	Free text
Q15. What are the names of the tools and technologies (e.g., Jenkins, Chef, Docker) that you are using to set up a tool-chain for continuous delivery/deployment pipeline?	Free text
Q16. We have the right tools to set up fully automated continuous deployment/delivery pipeline.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q17. Given the increasing importance of automation in CD/CDE practice, in your understanding what are the top four things you look for/need/would like to see in automation (e.g., type of automation needs to be improved, security support)?	Free text
Team Structures and Operations Stakeholders	
Q18. My responsibility has changed after our organization adopted CDE/CD practice.	A little / Somewhat / Much / Very much / Not at all
Q19. Please explain how your responsibility has changed (e.g., what new skills you require for CDE/CD)?	Free text
Q20. The collaboration between team members (e.g., developers, quality assurance team, testers, and operations team) has increased in my organization since the adoption of CDE/CD practice.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q21. Please explain how collaboration has increased (e.g., placing operations team next to developers at the same office/location)?	Free text
Q22. Which of the following statements would best describe the structure of the development and operations teams in your organization:	1. We still have separate development and operations teams, but they have more collaboration and coordination than ever. 2. We still have separate development and operations teams, but they have

	more collaboration and coordination than ever. Furthermore, we established a team/person, for example so-called DevOps team, between development and operations teams to manage and facilitate the collaboration and communication. 3. We have a VERY small operations team (e.g., two or three members to do specific tasks) and most of responsibilities of operations team have been shifted to development teams. 4. We don't have a visible and distinct operations team at all, and the operations colleagues are completely integrated in development team (i.e., all team members have a shared responsibility and purpose). 4. Other (Specify):
Q23. Despite the adoption of CDE/CD practice in my organization, the operations team's concerns and requirements still have a lower priority than other stakeholders.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q24. To increase the amount of attention paid to operations team and their concerns, my organization has adopted the following strategies:	1. Prioritizing operations concerns (i.e., consider the operations and their requirements as being as important as others). 2. Early and continuous engagement of Ops staff in the decision-making process for the development process (i.e., design process). 3. Leveraging logs and metric data for operational activities. We collect and structure logs, metrics (e.g., CPU usage) and operational data. 4. Other (Specify):
Software Architecture and Quality Attributes in Continuous Delivery and Deployment	
Q25. How would you grade the importance of software architecture design in successfully adopting and implementing CDE/CD practice?	Very important / Important / Moderately important / Of little importance / Unimportant
Q26. When we are designing the architecture of an application, we also consider the operational aspects, requirements and concerns (e.g., to make the architecture readily supportive of CDE/CD).	Almost Always / Often / Sometimes / Rarely / Never
Q27. Operational aspects and concerns impact on our architecture design decisions.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q28. It is "possible" to successfully practice CDE/CD in "monolithic applications".	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
In order to break down (monolithic) applications into smaller and independent units/components/services as STRONGLY recommended by CDE/CD practice, how would you define small service/component/unit in your organization?	
Q29. A component/service is small if it can be scaled independently.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q30. A component/service is small if it can be deployed independently.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q31. A component/service is small if it can be tested independently.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q32. A component/service is small if it can be modified	Strongly agree / Agree / Neutral /

(changed) independently.	Disagree / Strongly disagree
Q33. Can you describe (e.g., in one sentence) how a component or service should be to be suitable for successfully practicing CDE/CD?	Free text
Q34. In the projects that have adopted or are adopting CDE/CD practice, deployability concerns impact(ed) the design of individual classes.	Almost Always / Often / Sometimes / Rarely / Never
Q35. In the projects that have adopted or are adopting CDE/CD practice, deployability concerns impact(ed) the design of individual components/services.	Almost Always / Often / Sometimes / Rarely / Never
Q36. In the projects that have adopted or are adopting CDE/CD practice, deployability concerns impact(ed) the design of interactions among components/services.	Almost Always / Often / Sometimes / Rarely / Never
Q37. In the projects that have adopted or are adopting CDE/CD practice, deployability concerns impact(ed) the design of an entire application.	Almost Always / Often / Sometimes / Rarely / Never
Q38. In order to improve deployability of an application, I can sacrifice performance, security, usability, etc.	Almost Always / Often / Sometimes / Rarely / Never
Q39. Focusing too much on reusability at component or application level can be a bottleneck to continuously deploying software.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q40. Since moving to CDE/CD practice, the need for monitoring (i.e., having a centralized monitoring system) has increased.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q41. Since moving to CDE/CD practice, the need for logging (i.e., having a centralized logging system) has increased.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q42. Since moving to CDE/CD practice, Domain Driven Design and Bounded Context patterns have been applied MORE and practiced for designing loosely coupled architectures.	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q43. Compared with less frequent releases, we avoid big upfront architectural decisions for CDE/CD practice to support evolutionary changes (i.e., architectural decisions are made as late as possible).	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Q44. The CDE/CD practice increases the need for resilience (i.e., design for failure).	Strongly agree / Agree / Neutral / Disagree / Strongly disagree
Challenges and Barriers to Adopting Continuous DElivery (CDE) and Deployment (CD) Practices How important are the following challenges (if any) during adopting and implementing CDE/CD and which may put you in trouble?	
Q45. Huge dependencies and coordination among software development team members.	Very important / Important / Moderately important / Of little importance / Unimportant
Q46. Difficulty of splitting a (monolithic) application into independently deployable and autonomous components/services.	Very important / Important / Moderately important / Of little importance / Unimportant
Q47. Inflexibility of the organization's structure with the spirit of CDE/CD practice.	Very important / Important / Moderately important / Of little importance / Unimportant
Q48. Difficulty of breaking down a single-monolithic database into smaller and continuously deployable databases (i.e., decentralized data).	Very important / Important / Moderately important / Of little importance / Unimportant
Q49. Difficulty of identifying autonomous business capabilities.	Very important / Important / Moderately important / Of little importance / Unimportant
Q50. Lack of fully test automation.	Very important / Important / Moderately important / Of little importance / Unimportant
Q51. Lack of suitable awareness of the status of the project (e.g.,	Very important / Important / Moderately

build status, release status) among team members.	important / Of little importance / Unimportant

Thank you for participating in our survey. You need to submit the form to finalize the survey completion.

(Optional) Do you want to get the results of the survey? If so, please provide your email address

(Optional) Do you want to take part in the drawing? If so, please provide your email address

(Optional) Do you have any general comments about the questions of the survey?

This survey has been approved by the Human Research Ethics Committee at the University of Adelaide (approval number H-2015-270)

If you have further questions or any complains, please contact us via:

Mr. Mojtaba Shahin, mojtaba.shahin@adelaide.edu.au, Tel: +61 831 34519

Prof. Muhammad Ali Babar, ali.babr@adelaide.edu.au

Dr. Liming Zhu, liming.zhu@nicta.com.au

Dr. Mansooreh Zahedi, mzah@itu.dk

Appendix C

Interview Guide for Case Study

Title: (Re-) Architecting for Enabling and Supporting DevOps[38]

Research Team: Mojtaba Shahin, M. Ali Babar and Liming Zhu

Ethics Approval Number: H-2015-270

The University of Adelaide, Australia

General Information

This empirical study is aimed at "*investigating the practices and tools that enable organizations of teams and architecting for successful adoption of DevOps*". We propose to carry out an in‑depth case study in a case company.

Interview guide

The interview will be semi-structured. The schedule will include the following topics and questions:

Participant background

1. What is your main role in the project team?

   - Developer
   - Architect
   - Tester
   - QA
   - Build Engineer
   - Project Manager
   - DevOps Engineer
   - Operations Engineer/Staff
   - Release Engineer
   - Software Engineer
   - Team Lead
   - Consultant

[38] Some of the interview's questions are inspired by/taken from "2017 State of DevOps Report" [35] "2017 State of DevOps Report, Available at: goo.gl/Y6sm13 [Last accessed: 10 November 2017]." 2017..

- Other:

2. How many years have you been working in software or IT industry? In the current role?

A description of the project(s) that adopting (or adopted) DevOps practices

3. Can you please briefly (e.g., in one or two sentences) describe this project, e.g., its goals?
4. What is the domain of this project?
5. Is it a green-filed project or maintenance one?
6. How many people are involved in this project?
7. Which DevOps practices (e.g., continuous delivery and deployment, infrastructure as code, etc.) are realized by your organization in this project? Why?

Continuous Delivery/Deployment (CD) Pipeline and Automation

8. How often the primary application or service you work on is in a releasable state (production ready)?
   - Multiple times a day
   - Once a day
   - A few times (e.g., one or two) a week
   - A few times (e.g., one or two) a month
   - A few times (e.g., one or two) a year
   - Less than once a year
9. For primary application or service, you work on, how often does your organization deploy/release to production or customer?
   - Multiple times a day
   - Once a day
   - A few times (e.g., one or two) a week
   - A few times (e.g., one or two) a month
   - A few times (e.g., one or two) a year
   - Less than once a year
10. How would you grade your CD pipeline in terms of automation? Is it fully automated?

    i. Do **changes** directly go to production/customers? If no, why?

    ii. On average, how long does it take between committing code and successfully placing the code into production?
      - More than six months
      - Between one month and six months
      - Between one week and one month

- Between one day and one week
- Less than one day
- Less than one hour
- I don't know or not applicable

iii. What challenges/issues/choke-points you have/had to set up [**fully automated**] **CD pipeline**?

1. Which steps are still manual in your CD pipeline? Why?
2. Automated acceptance tests and quality assurance/security checks are part of your CD pipeline?

iv. How do you design the CD pipeline stages and select tools and technologies to set up CD pipeline?

1. Any trade-off for selecting tools? Any criteria for tool choices?

v. As a practitioner, what do you want/would like to see in a CD pipeline? Any missing features in the current CD pipeline? Any limitations in current tools?

11. What benefits and costs (negative aspects) DevOps practices would bring to your organization?

i. Are you totally happy with DevOps adoption? Any dissenting opinions?

Architecture and DevOps Practices

Deployability is a quality attribute (non-functional requirement) which means how reliably and easily an application/component/service can be deployed to (heterogeneous) production environment.

12. Does your organization/team take into consideration the role of **<u>software architecture</u>** in successfully and efficiently adopting DevOps practices? If so,

i. Why do you think it is necessary to do this?

ii. How do you design/change (modernize) the application architecture for this purpose, e.g., improving deployability and supporting incremental changes?

iii. Do you have **<u>deployability</u>** and **<u>testability</u>** in mind when designing/modernizing the application?

1. How **<u>deployability</u>** and **<u>testability</u>** concerns impact your architecture design?

iv. How do you **<u>quantify</u>** and **<u>measure</u>** the deployability of your system/architecture?

1. How do you evaluate the consequence of your architectural decisions on deployability of your system?
2. Can you provide one example of **<u>architectural decisions</u>** you made for improving deployability of your application?

3. Can you provide one example of **technology decisions** you made for improving deployability of your application?

4. Can you provide some examples of the sub-optimal architectural decisions that led to problems in deployment (release) process? i.e., architectural decisions that prevented automated deployment.

v. Potential follow-up questions:

1. Are you using microservices architecture style or monolithic architecture to enable DevOps?

2. As part of (re-) architecting your application to enable DevOps, do/did you break down the application into smaller parts? If so, how? What challenges did you faced?

    a. What criteria (e.g., **business domain** and **team autonomy**) do you use to break down an application or large service/components?

3. How could you manage **database schema changes or messaging** as part of (re-) architecting your application to enable DevOps?

4. How do you deploy your application/service independently of other applications/services that it depends on?

    a. Can you make large-scale changes to the design of your application/service without depending on/permission of other teams?

5. How do you deal with multiple and heterogeneous environments when practicing **continuous** and **automatic** deployment?

    a. Can you provide examples of **architectural and technological decisions** you made for this purpose?

(Architectural) Decision Making Process in DevOps

13. How does the adoption of DevOps change decision-making process?

i. After adopting DevOps in your organization, do architects more collaborate with teams to make architecture design decisions? If so, how?

ii. After adopting DevOps in your organization, do you become more independent to make your own (design) decision? If so, how?

1. Do you need to communicate and coordinate with other teams/team members to make your own decision? Why?

iii. What kind of data (e.g., production data, operations metrics and runtime data) do you utilize to make informed/data-driven (architectural design) decisions?

1. How do you use them to inform the design of products and features?

Do you have any other comments and feedback to share with us?

Appendix D

Selected Studies in Systematic Review

Table C1. Selected studies in the review presenetd in Chapter 3

ID	Title	Author(s)	Venue	Year
S1	Introduction of continuous delivery in multi-customer project courses	S. Krusche, L. Alperowitz	International Conference on Software Engineering	2014
S2	SQA-Mashup: A mashup framework for continuous integration	M. Brandtner, E. Giger, H. Gall	Information and Software Technology	2015
S3	Vroom: Faster build processes for java	J. Bell, E. Melski, M. Dattatreya, G.E. Kaiser	IEEE Software	2015
S4	The highways and country roads to continuous deployment	M. Leppänen, S. Mäkinen, M. Pagels, V. Eloranta, J. Itkonen, M.V. Mäntylä, T. Männistö	IEEE Software	2015
S5	Challenges when adopting continuous integration: A case study	A. Debbiche, M. Diener, R.B. Svensson	International Conference on Product-Focused Software Process Improvement	2014
S6	On the journey to continuous deployment: Technical and social challenges along the way	G. Claps, R.B. Svensson, A. Aurum	Information and Software Technology	2015
S7	Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail	S. Bellomo, N. Ernst, R. Nord, R. Kazman	International Conference on Dependable Systems and Networks	2014
S8	Achieving reliable high-frequency releases in cloud environments	L. Zhu, D. Xu, A.B. Tran, X. Xu, L. Bass, I. Weber, S. Dwarakanathan	IEEE Software	2015
S9	The practice and future of release engineering: A roundtable with three release engineers	B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, K. Moir	IEEE Software	2015
S10	Climbing the "Stairway to heaven" - A mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software	H.H. Olsson, H. Alahyari, J. Bosch	Euromicro Conference on Software Engineering and Advanced Applications	2012
S11	Automated software integration flows in industry: A multiple-case study	D. Ståhl, J. Bosch	International Conference on Software Engineering	2014
S12	Hitting the target: Practices for moving toward innovation experiment systems	T. Karvonen, L.E. Lwakatare, T. Sauvola, J. Bosch, H.H. Olsson, P. Kuvaja, M. Oivo	International Conference on Software Business	2015
S13	Visualizing testing activities to support continuous integration: A multiple case study	A. Nilsson, J. Bosch, C. Berger	International Conference on Agile Software Development (XP)	2014

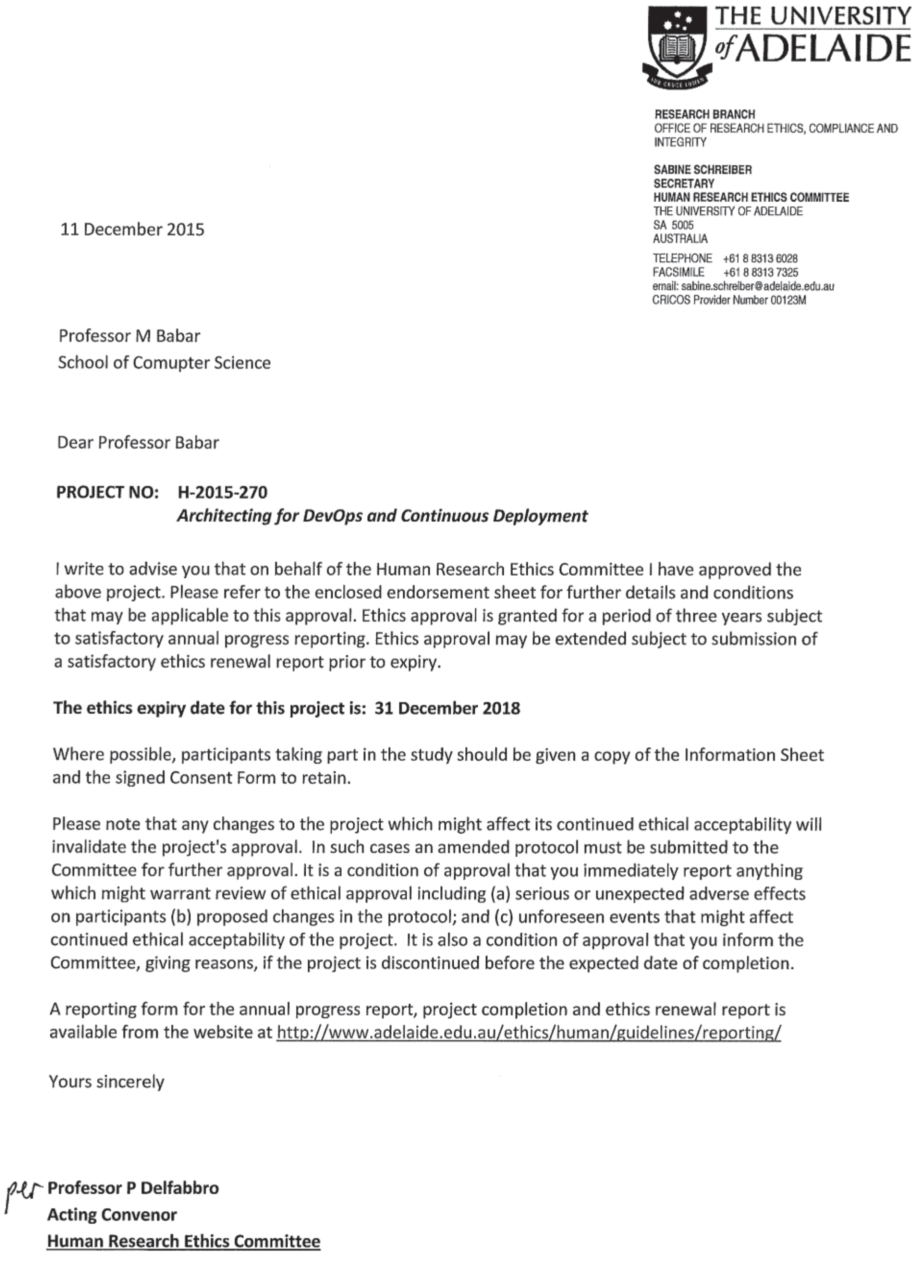
S14	Implementation of continuous integration and automated testing in software development of smart grid scheduling support system	J. Lu, Z. Yang, J. Qian	International Conference on Power System Technology	2014
S15	Implementing continuous integration software in an established computational chemistry software package	R.M. Betz, R.C. Walker	International Workshop on Software Engineering for Computational Science and Engineering	2013
S16	Making software integration really continuous	M.L. Guimarães, A.R. Silva	International Conference Fundamental Approaches to Software Engineering	2012
S17	Continuous delivery? Easy! Just change everything (well, maybe it is not that easy)	S. Neely, S. Stolt	Agile Conference (AGILE)	2013
S18	Software product measurement and analysis in a continuous integration environment	G. de Souza Pereira Moreira, R.P. Mellado, D.Á. Montini	International Conference on Information Technology: New Generations	2010
S19	UBuild: Automated testing and performance evaluation of embedded linux systems	F. Erculiani, L. Abeni, L. Palopoli	International Conference on Architecture of Computing Systems	2014
S20	Using continuous integration of code and content to teach software engineering with limited resources	J.G. Süβ, W. Billingsley	International Conference on Software Engineering	2012
S21	Backtracking incremental continuous integration	T. van der Storm	European Conference on Software Maintenance and Reengineering	2008
S22	BuildBot: Robotic monitoring of agile software development teams	R. Ablett, E. Sharlin, F. Maurer, J. Denzinger, C. Schock	International Conference on Robot & Human Interactive Communication	2007
S23	Mixed data-parallel scheduling for distributed continuous integration	O. Beaumont, N. Bonichon, L. Courtes, E. Dolstra, X. Hanin	International Parallel and Distributed Processing Symposium Workshops & PhD Forum	2012
S24	SQA-Profiles: Rule-based activity profiles for Continuous Integration environments	M. Brandtner, S.C. Muller, P. Leitner, H.C. Gall	International Conference on Software Analysis, Evolution, and Reengineering	2015
S25	Identifying and understanding header file hotspots in C/C++ build processes	S. McIntosh, B. Adams, M. Nagappan, A.E. Hassan	Automated Software Engineering	2015
S26	Practical experience with test-driven development during commissioning of the multi-star AO system ARGOS	M. Kulas, J.L. Borelli, W. Gässler, D. Peter, S. Rabien, G.O. de Xivry, L. Busoni, M. Bonaglia, T. Mazzoni, G. Rahmer	Software and Cyberinfrastructure for Astronomy III	2014
S27	Security of public continuous integration services	V.Gruhn, C. Hannebauer, C. John	International Symposium on Open Collaboration	2013
S28	Elaboration on an integrated architecture and requirement practice: Prototyping with quality attribute focus	S. Bellomo, R. L. Nord, I. Ozkaya	International Workshop on the Twin Peaks of Requirements and Architecture	2013
S29	Rapid releases and patch backouts: A software analytics approach	R. Souza, C. Chavez, R.A. Bittencourt	IEEE Software	2015
S30	Patterns for continuous integration builds in cross-platform agile software development	C. Hsieh, C. Chen	Journal of Information Science and Engineering	2015
S31	Technical dependency challenges in large-scale agile software development	N. Sekitoleko, F. Evbota, E. Knauss, A. Sandberg, M. Chaudron, H. H. Olsson	International Conference on Agile Software Development (XP)	2014

S32	A technique for agile and automatic interaction testing for product lines	M.F. Johansen, Ø. Haugen, F. Fleurey, E. Carlson, J. Endresen, T. Wien	International Conference on Testing Software and Systems	2012
S33	Ambient awareness of build status in collocated software teams	J. Downs, B. Plimmer, J. G. Hosking	International Conference on Software Engineering	2012
S34	How well do test case prioritization techniques support statistical fault localization	B. Jiang, Z. Zhang, W.K. Chan, T.H. Tse, T.Y. Chen	Information and Software Technology	2012
S35	Integrating early V&V support to a GSE tool integration platform	J.P. Pesola, H. Tanner, J. Eskeli, P. Parviainen, D. Bendas	International Conference on Global Software Engineering Workshops	2011
S36	Continuous SCRUM: Agile management of SAAS products	P. Agarwal	India Software Engineering Conference	2011
S37	Hitting the wall: What to do when high performing scrum teams overwhelm operations and infrastructure	J. Sutherland, R. Frohman	Hawaii International Conference on System Sciences	2011
S38	Test automation framework for implementing continuous integration	E.H. Kim, J. Chae Na, S.M. Ryoo	International Conference on Information Technology: New Generations	2009
S39	Using continuous integration and automated test techniques for a robust C4ISR system	H.M. Yüksel, E. Tüzün, E. Gelirli, B. Baykal	International Symposium on Computer and Information Sciences	2009
S40	A Unified test framework for continuous integration testing of SOA solutions	H. Liu, Z. Li, J. Zhu, H. Tan, H. Huang	International Conference on Web Services	2009
S41	Factors impacting rapid releases: An industrial case study	N. Kerzazi, F. Khomh	International Symposium on Empirical Software Engineering and Measurement	2014
S42	Ultimate architecture enforcement custom checks enforced at code-commit time	P. Merson	Companion of Conference on Systems, Programming, & Applications: Software for Humanity	2013
S43	Transitioning towards continuous delivery in the B2B domain: A case study	O. Rissanen, J. Münch	International Conference on Agile Software Development (XP)	2015
S44	Synthesizing continuous deployment practices used in software development	A.A.U. Rahman, E. Helms, L. Williams, C. Parnin	Agile Conference (AGILE)	2015
S45	Stakeholder perceptions of the adoption of continuous integration-A case study	E. Laukkanen, M. Paasivaara, T. Arvonen	Agile Conference (AGILE)	2015
S46	Toward agile architecture: Insights from 15 years of ATAM data	S. Bellomo, I. Gorton, R. Kazman	IEEE Software	2015
S47	Enterprise continuous integration using binary dependencies	M. Roberts	International Conference on Agile Software Development (XP)	2004
S48	Development and deployment at Facebook	D. G. Feitelson, E. Frachtenberg, K. L. Beck	IEEE Internet Computing	2013
S49	Transforming a six month release cycle to continuous flow	M. Marschall	Agile Conference (AGILE)	2007
S50	Scaling continuous integration	R.O. Rogers	International Conference on Agile Software Development (XP)	2004
S51	Architectural tactics to support rapid and agile stability	F. Bachmann, R.L. Nord, I. Ozkaya	CrossTalk: The Journal of Defense Software Engineering	2012

S52	Continuous automated testing of sdr software	J. Nimmer, B. Fallik, N. Martin, J. Chapin	Software Defined Radio Technical Conference	2006
S53	Surrogate: A simulation apparatus for continuous integration testing in service oriented architecture	H.Y. Huang, H.H. Liu, Z.J. Li, J. Zhu	International Conference on Services Computing	2008
S54	CiCUTS: Combining system execution modeling tools with continuous integration environments	J. H. Hill, D.C. Schmidt, A.A. Porter, J.M. Slaby	International Conference and Workshop on the Engineering of Computer Based Systems	2008
S55	Techniques for improving regression testing in continuous integration development environments	S. Elbaum, G. Rothermel, J. Peni	International Symposium on Foundations of Software Engineering	2014
S56	Towards DevOps in the Embedded Systems Domain: Why is It so Hard?	L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, M Oivo	Hawaii International Conference on System Sciences	2016
S57	Continuous deployment at Facebook and OANDA	T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, M. Stumm	International Conference on Software Engineering	2016
S58	DevOps making it easy to do the right thing	M. Callanan, A. Spillane	IEEE Software	2016
S59	Rondo: A tool suite for continuous deployment in dynamic environments	O. Günalp, C. Escoffier, P. Lalanda	International Conference on Services Computing	2015
S60	DevOps: A definition and perceived adoption impediments	J. Smeds, K. Nybom, I. Porres	International Conference on Agile Software Development (XP)	2015
S61	Social Testing: A framework to support adoption of continuous delivery by small medium enterprises	J. Dunne, D. Malone, J. Flood	International Conference on Computer Science, Computer Engineering, and Social Media	2015
S62	Automated testing in the continuous delivery pipeline: A case study of an online company	J. Gmeiner, R. Ramler, J. Haslinger	User Symposium on Software Quality, Test and Innovation	2015
S63	Towards post-agile development practices through productized development infrastructure	M. Leppänen, T. Kilamo, T. Mikkonen	International Workshop on Rapid Continuous Software Engineering	2015
S64	Supporting continuous integration by code-churn based test selection	E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, M. Castell	International Workshop on Rapid Continuous Software Engineering	2015
S65	Requirements to pervasive system continuous deployment	C. Escoffier, O. Günalp, P. Lalanda	International Conference on Service-Oriented Computing Workshops	2013
S66	Composing patterns to construct secure systems	P. Rimba, L. Zhu, L. Bass, I. Kuz, S. Reeves	European Dependable Computing Conference	2015
S67	Fast feedback from automated tests executed with the product build	M. Eyl, C. Reichmann, K. Müller-Glaser	International Conference Software Quality Days	2016
S68	POD-Diagnosis: Error diagnosis of sporadic operations on cloud applications	X. Xu, L. Zhu, I. Weber, L. Bass, D. Sun	International Conference on Dependable Systems and Networks	2014
S69	Feature toggles: practitioner practices and a case study	M. T. Rahman, L. P. Querel, P. C. Rigby, B. Adams	Working Conference on Mining Software Repositories	2016

Appendix E

Approved Ethics Applications



THE UNIVERSITY *of* ADELAIDE

RESEARCH BRANCH
OFFICE OF RESEARCH ETHICS, COMPLIANCE AND INTEGRITY

SABINE SCHREIBER
SECRETARY
HUMAN RESEARCH ETHICS COMMITTEE
THE UNIVERSITY OF ADELAIDE
SA 5005
AUSTRALIA

TELEPHONE +61 8 8313 6028
FACSIMILE +61 8 8313 7325
email: sabine.schreiber@adelaide.edu.au
CRICOS Provider Number 00123M

11 December 2015

Professor M Babar
School of Comupter Science

Dear Professor Babar

PROJECT NO: H-2015-270
Architecting for DevOps and Continuous Deployment

I write to advise you that on behalf of the Human Research Ethics Committee I have approved the above project. Please refer to the enclosed endorsement sheet for further details and conditions that may be applicable to this approval. Ethics approval is granted for a period of three years subject to satisfactory annual progress reporting. Ethics approval may be extended subject to submission of a satisfactory ethics renewal report prior to expiry.

The ethics expiry date for this project is: 31 December 2018

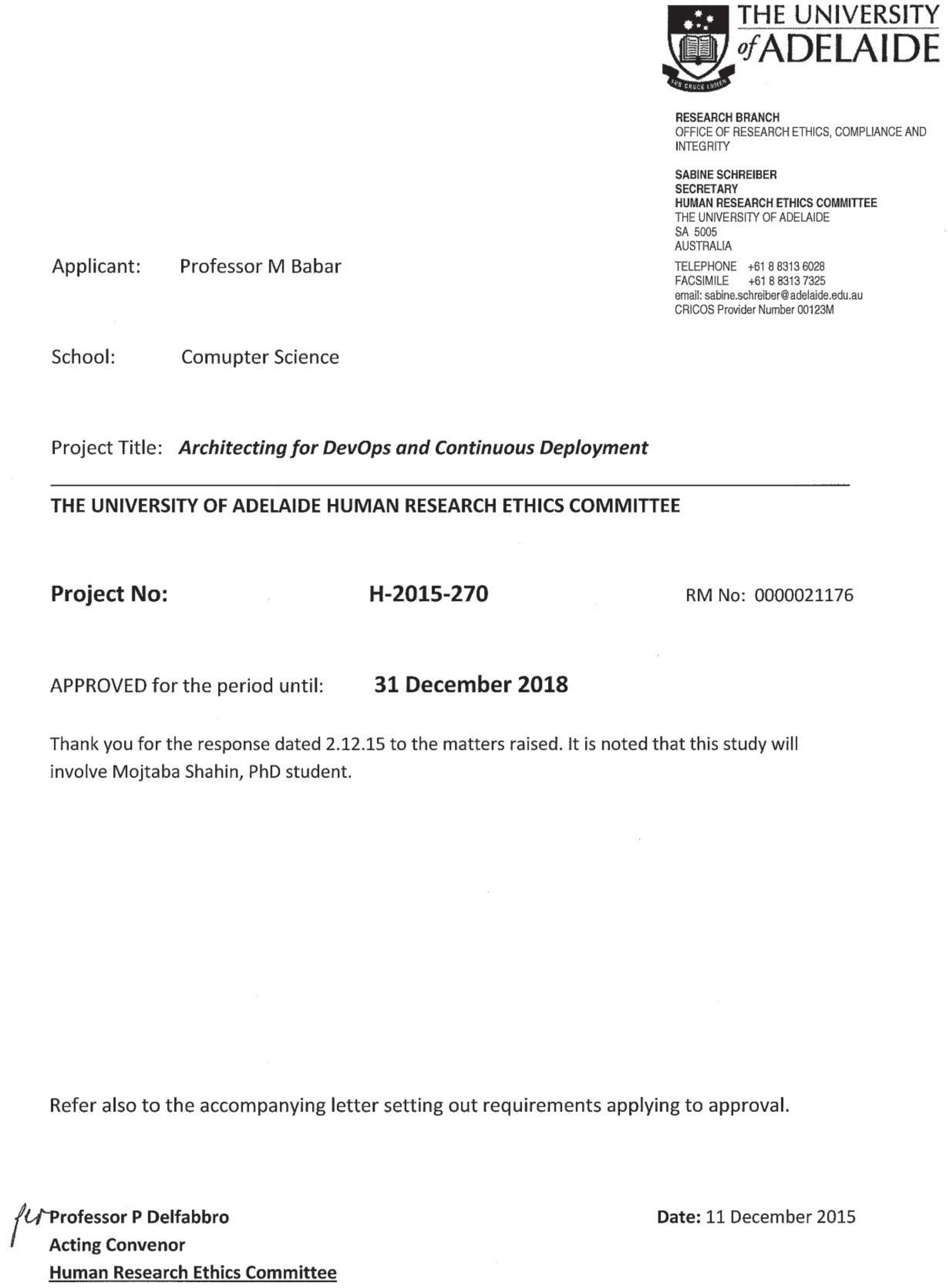
Where possible, participants taking part in the study should be given a copy of the Information Sheet and the signed Consent Form to retain.

Please note that any changes to the project which might affect its continued ethical acceptability will invalidate the project's approval. In such cases an amended protocol must be submitted to the Committee for further approval. It is a condition of approval that you immediately report anything which might warrant review of ethical approval including (a) serious or unexpected adverse effects on participants (b) proposed changes in the protocol; and (c) unforeseen events that might affect continued ethical acceptability of the project. It is also a condition of approval that you inform the Committee, giving reasons, if the project is discontinued before the expected date of completion.

A reporting form for the annual progress report, project completion and ethics renewal report is available from the website at http://www.adelaide.edu.au/ethics/human/guidelines/reporting/

Yours sincerely

per **Professor P Delfabbro**
Acting Convenor
Human Research Ethics Committee



THE UNIVERSITY
of ADELAIDE

RESEARCH BRANCH
OFFICE OF RESEARCH ETHICS, COMPLIANCE AND INTEGRITY

SABINE SCHREIBER
SECRETARY
HUMAN RESEARCH ETHICS COMMITTEE
THE UNIVERSITY OF ADELAIDE
SA 5005
AUSTRALIA
TELEPHONE +61 8 8313 6028
FACSIMILE +61 8 8313 7325
email: sabine.schreiber@adelaide.edu.au
CRICOS Provider Number 00123M

Applicant: Professor M Babar

School: Comupter Science

Project Title: ***Architecting for DevOps and Continuous Deployment***

THE UNIVERSITY OF ADELAIDE HUMAN RESEARCH ETHICS COMMITTEE

Project No: **H-2015-270** RM No: 0000021176

APPROVED for the period until: **31 December 2018**

Thank you for the response dated 2.12.15 to the matters raised. It is noted that this study will involve Mojtaba Shahin, PhD student.

Refer also to the accompanying letter setting out requirements applying to approval.

per **Professor P Delfabbro**
Acting Convenor
Human Research Ethics Committee

Date: 11 December 2015

THE UNIVERSITY *of* ADELAIDE

RESEARCH BRANCH
OFFICE OF RESEARCH ETHICS, COMPLIANCE AND INTEGRITY

AMY LEHMANN
HUMAN RESEARCH ETHICS OFFICER
THE UNIVERSITY OF ADELAIDE
LEVEL 4, RUNDLE MALL PLAZA
50 RUNDLE MALL
ADELAIDE SA 5000 AUSTRALIA

TELEPHONE +61 8 8313 4725
FACSIMILE +61 8 8313 3700
email: hrec@adelaide.edu.au
CRICOS Provider Number 00123M

16 June 2016

Professor M Babar
School of Computer Science

Dear Professor Babar

PROJECT NO: H-2015-270
Architecting for DevOps and Continuous Deployment

Thank you for the modified ethics application dated 30.4.16 submitted by Mojtaba Shahin requesting amendment to the above project. On behalf of the Human Research Ethics Committee I have approved the request to conduct a new online survey as detailed in the amended application. Thank you for forwarding the modified participant documents and survey on the 15.06.2016.

The ethics expiry date for this project is: 31 December 2018

Participants in the study are to be given a copy of the Information Sheet and the signed Consent Form to retain. It is also a condition of approval that you **immediately report** anything which might warrant review of ethical approval including:

- serious or unexpected adverse effects on participants,
- previously unforeseen events which might affect continued ethical acceptability of the project,
- proposed changes to the protocol; and
- the project is discontinued before the expected date of completion.

Yours sincerely

for **Professor Paul Delfabbro**
Convenor
Human Research Ethics Committee

References

[1] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access,* vol. 5, pp. 3909-3943, 2017.

[2] M. Shahin, M. A. Babar, and L. Zhu, "The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, 2016, pp. 1-10: ACM.

[3] M. Shahin, M. A. Babar, M. Zahedi, and L. Zhu, "Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges," in *11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Toronto, Canada, 2017: IEEE.

[4] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "Adopting Continuous Delivery and Deployment: Impacts on Team Structures, Collaboration and Responsibilities," in *21st International Conference on Evaluation and Assessment in Software Engineering*, Karlskrona, Sweden, 2017, pp. 384-393: ACM.

[5] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "An Empirical Study of Architecting for Continuous Delivery and Deployment," *submitted to Empirical Software Engineering* 2017.

[6] M. Httermann, *DevOps for developers*. Apress, 2012.

[7] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[8] G. Kim, K. Behr, and K. Spafford, *The phoenix project: A novel about IT, DevOps, and helping your business win*. IT Revolution, 2014.

[9] "2015 State of DevOps Report, Available at:goo.gl/oJ2Tvi [Last accessed: 5 October 2015]." 2015.

[10] B. Fitzgerald and K.-J. Stol, "Continuous Software Engineering: A Roadmap and Agenda," *Journal of Systems and Software,* vol. 123, 2017.

[11] M. Leppanen *et al.*, "The Highways and Country Roads to Continuous Deployment," *IEEE Software,* vol. 32, no. 2, pp. 64-72, 2015.

[12] I. Weber, S. Nepal, and L. Zhu, "Developing Dependable and Secure Cloud Applications," *IEEE Internet Computing,* vol. 20, no. 3, pp. 74-79, 2016.

[13] J. Humble. *Continuous Delivery vs Continuous Deployment, Available at: goo.gl/qE1JoM [Last accessed: 1 March 2016].*

[14] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," *IEEE Software,* vol. 32, no. 2, pp. 50-54, 2015.

[15] *What is Continuous Deployment?, Available at: goo.gl/sLxWRD, [Last accessed: 12 July 2016].*

[16] A. Thiele. (2014). *Continuous Delivery: An Easy Must-Have for Agile Development, Available at: goo.gl/ymgCSq [Last accessed: 10 July 2016].*

[17] E. Luke and S. Prince. (2016). *No One Agrees How to Define CI or CD. Available at: goo.gl/Z8Qonq [Last accessed: 1 August 2016].*

[18] M. Skelton and C. O'Dell, *Continuous Delivery with Windows and .NET*. O'Reilly 2016.

[19] J. P. Reed. *The business case for continuous delivery, Available at: goo.gl/Bq9ugb [Last accessed: 12 July 2016].*

[20] S. Prince. (2016). *The Product Managers' Guide to Continuous Delivery and DevOps, Available at: goo.gl/D8mGkH [Last accessed: 2 November 2016].*

[21] L. Bass, "The Software Architect and DevOps," *IEEE Software,* vol. 35, no. 1, pp. 8-10, 2017.

[22] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *38th International Conference on Software Engineering Companion*, Austin, Texas, 2016, pp. 21-30: ACM.

[23] G. G. Claps, R. Berntsson 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.

[24] E. Laukkanen, T. O. A. Lehtinen, J. Itkonen, M. Paasivaara, and C. Lassenius, "Bottom-up Adoption of Continuous Delivery in a Stage-Gate Managed Software Organization," in *10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Ciudad Real, Spain, 2016, pp. 1-10: ACM.

[25] M. d. Jong, A. v. Deursen, and A. Cleve, "Zero-downtime SQL database schema evolution for continuous deployment," in *39th International Conference on Software Engineering: Software Engineering in Practice Track*, Buenos Aires, Argentina, 2017, pp. 143-152: IEEE Press.

[26] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel," *Future Generation Computer Systems,* vol. 56, pp. 317-332, 2016.

[27] S. Mäkinen *et al.*, "Improving the delivery cycle: A multiple-case study of the toolchains in Finnish software intensive enterprises," *Information and Software Technology,* vol. 80, pp. 175-194, 2016.

[28] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st edition ed. Addison-Wesley Professional, 2010.

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

[30] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including Performance Benchmarks into Continuous Integration to Enable DevOps," *SIGSOFT Software Engineering Notes,* vol. 40, no. 2, pp. 1-4, 2015.

[31] P. J. Andre van Hoorn, Philipp Leitner, Ingo Weber, "Report from GI-Dagstuhl Seminar 16394: Software Performance Engineering in the DevOps World. Available at: https://arxiv.org/abs/1709.08951," 2017.

[32] L. Bass, R. Holz, P. Rimba, A. B. Tran, and L. Zhu, "Securing a deployment pipeline," in *Third International Workshop on Release Engineering*, ed. Florence, Italy: IEEE Press, 2015, pp. 4-7.

[33] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery—A systematic literature review," *Information and Software Technology,* vol. 82, pp. 55-79, 2017.

[34] P. Rodríguez *et al.*, "Continuous deployment of software intensive products and services: A systematic mapping study," *Journal of Systems and Software,* vol. 123, pp. 263-291, 2017.

[35] "2017 State of DevOps Report, Available at: goo.gl/Y6sm13 [Last accessed: 10 November 2017]." 2017.

[36] M. E. Conway, "How do committees invent?," *Datamation,* vol. 14, no. 5, 1968.

[37] L. Northrop, "Trends and New Directions in Software Architecture, Available at: goo.gl/ZAnkQp," 2015.

[38] L. Chen, "Towards Architecting for Continuous Delivery," in *12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2015, pp. 131-134.

[39] N. Vishal. (2015). *Architecting for Continuous Delivery, Available at: goo.gl/zWA5kT [Last accessed: 15 March 2016].*

[40] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. Gall, "An empirical study on principles and practices of continuous delivery and deployment," *PeerJ Preprints 4:e1889v1,* 2016.

[41] M. Shahin, "Architecting for DevOps and Continuous Deployment," in *2015 24th Australasian Software Engineering Conference*, Adelaide, SA, Australia, 2015, pp. 147-148: ACM.

[42] S. Arunachalam, "Open access to scientific knowledge," *DESIDOC Journal of Library & Information Technology,* vol. 28, no. 1, p. 7, 2008.

[43] D. Graziotin, "Towards a Theory of Affect and Software Developers' Performance," PhD Thesis, Faculty of Computer Science, Free University of Bozen-Bolzano, 2016.

[44] B. A. Kitchenham, T. Dyba, and M. Jorgensen, "Evidence-Based Software Engineering," in *26th International Conference on Software Engineering*, 2004, pp. 273-281, 999432: IEEE Computer Society.

[45] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," in "EBSE Technical Report Ver. 2.3 " 2007.

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

[47] J. W. Creswell and V. L. P. Clark, *Designing and conducting mixed methods research*. SAGE Publishing, 2007.

[48] M. G. Waterman, "Reconciling agility and architecture: a theory of agile architecture," PhD Thesis, Victoria University of Wellington, 2014.

[49] S. E. Hove and B. Anda, "Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research," in *11th IEEE International Software Metrics Symposium*, 2005, p. 23, 1092163: IEEE Computer Society.

[50] B. A. Kitchenham and S. L. Pfleeger, "Personal Opinion Surveys," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London: Springer London, 2008, pp. 63-92.

[51] V. Garousi, M. Felderer, and M. V. Mäntylä, "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature," in *20th International Conference on Evaluation and Assessment in Software Engineering*, Limerick, Ireland, 2016, pp. 1-6, 2916008: ACM.

[52] L. A. Palinkas, S. M. Horwitz, C. A. Green, J. P. Wisdom, N. Duan, and K. Hoagwood, "Purposeful Sampling for Qualitative Data Collection and Analysis in Mixed Method Implementation Research," *Administration and Policy in Mental Health and Mental Health Services Research,* journal article vol. 42, no. 5, pp. 533-544, 2015.

[53] L. A. Goodman, "Snowball Sampling," *Annals of Mathematical Statistics,* vol. 32, no. 1, pp. 148-170, 1961.

[54] D. S. Cruzes and T. Dyba, "Recommended Steps for Thematic Synthesis in Software Engineering," in *International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 275-284.

[55] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering,* vol. 25, no. 4, pp. 557-572, 1999.

[56] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology,* vol. 3, no. 2, pp. 77-101, 2006.

[57] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *38th International Conference on Software Engineering*, Austin, Texas, 2016, pp. 285-296: ACM.

[58] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The Design Space of Bug Fixes and How Developers Navigate It," *IEEE Transactions on Software Engineering,* vol. 41, no. 1, pp. 65-81, 2015.

[59] A. W. Meade and S. B. Craig, "Identifying careless responses in survey data," *Psychological methods,* vol. 17, no. 3, p. 437, 2012.

[60] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software,* vol. 12, no. 4, pp. 52-62, 1995.

[61] F. Adrian, "Response bias, social desirability and dissimulation," *Personality and Individual Differences,* vol. 7, no. 3, pp. 385-400, 1996.

[62] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: an empirical study on software development for the cloud," in *10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015, pp. 393-403: ACM.

[63] M. S. Litwin and A. Fink, *How to measure survey reliability and validity*. SAGE Publications, 1995.

[64] F. J. Fowler Jr, *Survey research methods*. Sage publications, 2013.

[65] M. Q. Patton, *Qualitative evaluation and research methods*. SAGE Publications, 1990.

[66] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. SAGE Publications, 2003.

[67] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," (in English), *Empirical Software Engineering,* vol. 14, no. 2, pp. 131-164, 2009/04/01 2009.

[68] L. Prechelt, H. Schmeisky, and F. Zieris, "Quality experience: a grounded theory of successful agile projects without dedicated testers," in *38th International Conference on Software Engineering*, Austin, Texas, 2016, pp. 1017-1027, 2884789: ACM.

[69] B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. Chicago, Aldine, 1967.

[70] R. Hoda and J. Noble, "Becoming agile: a grounded theory of agile transitions in practice," in *39th International Conference on Software Engineering*, Buenos Aires, Argentina, 2017, pp. 141-151, 3097386: IEEE Press.

[71] R. Hoda, "Self-organizing agile teams: A grounded theory," PhD Thesis, Victoria University of Wellington, 2011.

[72] E. G. Guba, "Criteria for assessing the trustworthiness of naturalistic inquiries," *Educational Technology Research and Development,* vol. 29, no. 2, pp. 75-91, 1981.

[73] B. Flyvbjerg, "Five misunderstandings about case-study research," *Qualitative inquiry,* vol. 12, no. 2, pp. 219-245, 2006.

[74] E. Kalliamvakou, C. Bird, T. Zimmermann, A. Begel, R. DeLine, and D. M. German, "What Makes a Great Manager of Software Engineers?," *IEEE Transactions on Software Engineering,* vol. PP, no. 99, pp. 1-1, 2017.

[75] 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 *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014, pp. 702-707.

[76] K. F. Tómasdóttir, M. Aniche, and A. v. Deursen, "Why and how JavaScript developers use linters," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, Urbana-Champaign, IL, USA, 2017, pp. 578-589, 3155634: IEEE Press.

[77] K.-J. Stol, P. Avgeriou, M. A. Babar, Y. Lucas, and B. Fitzgerald, "Key factors for adopting inner source," *ACM Transactions on Software Engineering and Methodology,* vol. 23, no. 2, pp. 1-35, 2014.

[78] A. Phillips, M. Sens, A. de Jonge, and M. van Holsteijn, *The IT Manager's Guide to Continuous Delivery: Delivering business value in hours, not months*. XebiaLabs, 2015.

[79] M. Fowler. *Continuous Integration, Available at: goo.gl/5EhHR7 [Last accessed: 21 October 2015].*

[80] A. A. U. Rahman, E. Helms, L. Williams, and C. Parnin, "Synthesizing Continuous Deployment Practices Used in Software Development," in *Agile Conference (AGILE)*, 2015, pp. 1-10.

[81] H. H. 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 (SEAA)*, 2012, pp. 392-399.

[82] D. Ståhl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software,* vol. 87, pp. 48-59, 2014.

[83] M. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen, "On rapid releases and software testing: a case study and a semi-systematic literature review," (in English), *Empirical Software Engineering,* vol. 20, no. 5, pp. 1384–1425, 2015.

[84] A. Eck, F. Uebernickel, and W. Brenner, "Fit for Continuous Integration: How Organizations Assimilate an Agile Practice," in *20th Americas Conference on Information Systems (AMCIS)*, 2014: Association for Information Systems.

[85] N. S. R. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, "Identification and management of technical debt: A systematic mapping study," *Information and Software Technology,* vol. 70, pp. 100-121, 2016.

[86] "Longman Dictionary of Contemporary English Online, http://www.ldoceonline.com/," ed.

[87] "Cambridge Dictionary, http://dictionary.cambridge.org/," ed.

[88] Y. Dittrich, "What does it mean to use a method? Towards a practice theory for software engineering," *Information and Software Technology,* vol. 70, pp. 220-231, 2016/02/01 2016.

[89] J. Nørbjerg and P. Kraft, "Software practice is social practice," in *Social thinking*, D. Yvonne, F. Christiane, and K. Ralf, Eds.: MIT Press, 2002, pp. 205-222.

[90] K. Schmidt, "The Concept of 'Practice': What's the Point?," in *11th International Conference on the Design of Cooperative Systems (COOP)*, Nice, France, 2014, pp. 427-444: Springer International Publishing.

[91] H. Zhang, M. A. Babar, and P. Tell, "Identifying relevant studies in software engineering," *Information and Software Technology,* vol. 53, no. 6, pp. 625-637, 2011.

[92] D. Budgen, M. Turner, P. Brereton, and B. Kitchenham, "Using mapping studies in software engineering," in *20th Annual Meeting of thePsychology of Programming Interest Group (PPIG)*, 2008, vol. 8, pp. 195-204.

[93] L. Chen, M. A. Babar, and H. Zhang, "Towards an evidence-based understanding of electronic data sources," in *14th international conference on Evaluation and Assessment in Software Engineering*, UK, 2010, pp. 135-138, 2227074: British Computer Society.

[94] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy, "Performance Analysis for Object-Oriented Software: A Systematic Mapping," *IEEE Transactions on Software Engineering,* vol. 41, no. 7, pp. 691-710, 2015.

[95] M. Daneva, D. Damian, A. Marchetto, and O. Pastor, "Empirical research methodologies and studies in Requirements Engineering: How far did we come?," *Journal of Systems and Software,* vol. 95, pp. 1-9, 2014.

[96] B. Kitchenham *et al.*, "Systematic literature reviews in software engineering – A tertiary study," *Information and Software Technology,* vol. 52, no. 8, pp. 792-805, 2010.

[97] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements engineering paper classification and evaluation criteria: a proposal and a discussion," *Requirements Engineering,* vol. 11, no. 1, pp. 102-107, 2005.

[98] B. Adams and S. McIntosh, "Modern Release Engineering in a Nutshell -- Why Researchers Should Care," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 78-90.

[99] M. Meyer, "Continuous Integration and Its Tools," *IEEE Software,* vol. 31, no. 3, pp. 14-16, 2014.

[100] V. Armenise, "Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery," in *IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*, 2015, pp. 24-27.

[101] J. A. Espinosa, S. A. Slaughter, R. E. Kraut, and J. D. Herbsleb, "Team Knowledge and Coordination in Geographically Distributed Software Development," *Journal of Management Information Systems,* vol. 24, no. 1, pp. 135-169, 2007.

[102] K. Dikert, M. Paasivaara, and C. Lassenius, "Challenges and success factors for large-scale agile transformations: A systematic literature review," *Journal of Systems and Software,* vol. 119, pp. 87-108, 2016.

[103] J. Greenberg and R. A. Baron, *Behavior in organizations*, 9th ed. Pearson/Prentice Hall, 2008.

[104] D. Sato. (2014). *Canary Release. Available at: goo.gl/mqJVy3 [Last accessed: 10 October 2015].*

[105] J. Humble. *Principle 2: Decouple Deployment and Release, Available at: goo.gl/wu7Smy [Last accessed: 22 October 2015].*

[106] (2015). *Continuous Delivery - Five Habits of Highly Successful Continuous Delivery Practitioners, Available at: goo.gl/o9Dr5c [Last accessed: 5 August 2016].*

[107] D. Kirk and S. G. MacDonell, "Investigating a conceptual construct for software context," in *18th International Conference on Evaluation and Assessment in Software Engineering*, London, England, United Kingdom, 2014, pp. 1-10: ACM.

[108] T. Dybå, D. I. K. Sjøberg, and D. S. Cruzes, "What works for whom, where, when, and why? on the role of context in empirical software engineering," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Lund, Sweden, 2012, pp. 19-28: ACM.

[109] K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 401-404.

[110] M. Zahedi, M. Shahin, and M. Ali Babar, "A systematic review of knowledge sharing challenges and practices in global software development," *International Journal of Information Management,* vol. 36, no. 6, Part A, pp. 995-1019, 2016.

[111] S. Newman, *Building Microservices.* O'Reilly Media, Inc, 2015.

[112] L. Bass, R. Holz, P. Rimba, A. B. Tran, and Z. Liming, "Securing a Deployment Pipeline," in *IEEE/ACM 3rd International Workshop on Release Engineering (RELENG)*, 2015, pp. 4-7.

[113] S. Neely and S. Stolt, "Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)," in *Agile Conference (AGILE)*, 2013, pp. 121-128.

[114] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ed. Austin, Texas: ACM, 2016, pp. 21-30.

[115] L. E. Lwakatare *et al.*, "Towards DevOps in the Embedded Systems Domain: Why is It So Hard?," in *49th Hawaii International Conference on System Sciences (HICSS)*, 2016, pp. 5437-5446.

[116] S. G. Yaman *et al.*, "Customer Involvement in Continuous Deployment: A Systematic Literature Review," Cham, 2016, pp. 249-265: Springer International Publishing.

[117] T. Fitz. (2016). *Continuous Deployment: Beyond Continuous Delivery. Available at: goo.gl/PPbTxL [Last accessed: 21 December 2016].*

[118] L. W. Richter. (2016). *Getting from Continuous Delivery to Continuous Deploymenty. Available at: goo.gl/gYtAzG [Last accessed: 18 December 2016].*

[119] K. Lankford. (2013). *Beyond Continuous Delivery—All the Way to Continuous Deployment, Available at: goo.gl/QAZ1DG [Last accessed: 13 January 2017]. .*

[120] G. Schermann, J. Cito, P. Leitner, and H. C. Gall, "Towards quality gates in continuous delivery and deployment," in *IEEE 24th International Conference on Program Comprehension (ICPC)*, 2016, pp. 1-4.

[121] (2016). *2016 State of DevOps Report, Available at:goo.gl/DrkW6U [Last accessed: 5 October 2017].*

[122] M. Mooney. *Continuous Deployment For Practical People, Available at:goo.gl/LJjBnA [Last accessed: 11 June 2017]. .*

[123] M. Brandtner, E. Giger, and H. Gall, "SQA-Mashup: A mashup framework for continuous integration," *Information and Software Technology,* vol. 65, pp. 97-113, 2015.

[124] D. Ståhl, K. Hallén, and J. Bosch, "Continuous Integration and Delivery Traceability in Industry: Needs and Practices," in *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016, pp. 68-72.

[125] A. Brown. (2015). *What's the Best Team Structure for DevOps Success? Available at: goo.gl/3Z11og [Last accessed: 13 September 2017].*

[126] *What Team Structure is Right for DevOps to Flourish, Available at: goo.gl/KM6N3p [Last accessed: 24 September 2017].*

[127] J. Wettinger, U. Breitenbücher, M. Falkenthal, and F. Leymann, "Collaborative gathering and continuous delivery of DevOps solutions through repositories," *Computer Science - Research and Development,* journal article pp. 1-10, 2016.

[128] S. Krusche and L. Alperowitz, "Introduction of continuous delivery in multi-customer project courses," in *36th International Conference on Software Engineering Companion* Hyderabad, India, 2014, pp. 335-343: ACM.

[129] M. Callanan and A. Spillane, "DevOps: Making It Easy to Do the Right Thing," *IEEE Software,* vol. 33, no. 3, pp. 53-59, 2016.

[130] K. Nybom, J. Smeds, and I. Porres, "On the Impact of Mixing Responsibilities Between Devs and Ops," in *17th International Conference on Agile Processes, in Software Engineering, and Extreme Programming*, Edinburgh, UK, 2016, pp. 131-143: Springer International Publishing.

[131] B. B. N. d. França, J. Helvio Jeronimo, and G. H. Travassos, "Characterizing DevOps by Hearing Multiple Voices," in *30th Brazilian Symposium on Software Engineering*, Maringá, Brazil, 2016, pp. 53-62: ACM.

[132] *There's No Such Thing as a "Devops Team", Available at: goo.gl/ZCTNyY [Last accessed: 5 March 2017].*

[133] K. Bittner and T. Buntel. *Overcoming Organizational Obstacles to DevOps and Continuous Delivery, Available at: goo.gl/9FpTL4 [Last accessed: 12 June 2017].*

[134] N. Ford. *Continuous Delivery for Architects, Available at:goo.gl/dyWN5a [Last accessed: 20 October 2016].*

[135] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," *Cutter IT Journal,* vol. 24, no. 8, 2011.

[136] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "An Exploratory Study of DevOps: Extending the Dimensions of DevOps with Practices," in *The Eleventh International Conference on Software Engineering Advances (ICSEA)*, 2016, pp. 91-99: IARIA.

[137] J. Iden, B. Tessem, and T. Päivärinta, "Problems in the interplay of development and IT operations in system development projects: A Delphi study of Norwegian IT experts," *Information and Software Technology,* vol. 53, no. 4, pp. 394-406, 2011.

[138] L. Lwakatare, P. Kuvaja, and M. Oivo, "Dimensions of DevOps," in *Agile Processes, in Software Engineering, and Extreme Programming*, vol. 212, C. Lassenius, T. Dingsøyr, and M. Paasivaara, Eds. (Lecture Notes in Business Information Processing: Springer International Publishing, 2015, pp. 212-217.

[139] 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," presented at the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 23-26 June 2014, 2014.

[140] P. M. Dooley, "The Intersection of DevOps and ITIL, Available at: goo.gl/tqg2hD [Last accessed: 14 June 2017]." Global Knowledge2015.

[141] L. Bass, R. Jeffery, H. Wada, I. Weber, and Z. Liming, "Eliciting operations requirements for applications," in *1st International Workshop on Release Engineering (RELENG)*, 2013, pp. 5-8.

[142] G. Hohpe, I. Ozkaya, U. Zdun, and O. Zimmermann, "The Software Architect's Role in the Digital Age," *IEEE Software,* vol. 33, no. 6, pp. 30-39, 2016.

[143] E. Woods, "Operational: The Forgotten Architectural View," *IEEE Software,* vol. 33, no. 3, pp. 20-23, 2016.

[144] "ISO/IEC/IEEE Systems and software engineering -- Architecture description," *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000),* pp. 1-46, 2011.

[145] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar, "10 years of software architecture knowledge management: Practice and future," *Journal of Systems and Software,* vol. 116, pp. 191-205, 2016.

[146] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conference on Software Architecture*, 2005, pp. 109-120.

[147] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software,* vol. 33, no. 3, pp. 42-52, 2016.

[148] T. Mårtensson, D. Ståhl, and J. Bosch, "Continuous Integration Impediments in Large-Scale Industry Projects," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 169-178.

[149] N. Ford and R. Parsons. (2016). *Microservices as an Evolutionary Architecture, Available at: goo.gl/aysZvA [Last accessed: 20 March 2016].*

[150] J. Lewis and M. Fowler. *Microservices: a definition of this new architectural term, Available at: goo.gl/me6tp5 [Last accessed: 05 January 2016].*

[151] M. Skelton. (2016). *How to break apart a monolithic system safely without destroying your team, Available at: goo.gl/pqBVm2 [Last accessed: 4 November 2016].*

[152] P. Ketan. (2015). *Monolithic vs Microservice Architecture, Available at: goo.gl/F46ptW [Last accessed: 24 October 2016].*

[153] *Self-Contained Systems: Assembling Software from Independent Systems, Available at: http://scs-architecture.org/ [Last accessed: 1 June 2017].*

[154] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195-216.

[155] R. Chris. (2014). *Pattern: Monolithic Architecture, Available at: goo.gl/royZ7i [Last accessed: 4 November 2016].*

[156] G. Arun. (2015). *Microservices, Monoliths, and NoOps, Available at: goo.gl/zou2x3 [Last accessed: 8 November 2016].*

[157] S. Gibson. *Monoliths are Bad Design... and You Know It, Available at: goo.gl/xVEbSE [Last accessed: 4 March 2016].*

[158] J. Humble. (2011). *Organize software delivery around outcomes, not roles: continuous delivery and cross-functional teams, Available at: goo.gl/MnFtJN [Last accessed: 10 August 2016].*

[159] P. Beijer and T. de Klerk, *IT Architecture- Essential Practice for IT Business Solutions.* Lulu. com, 2010.

[160] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 243-246.

[161] N. Ford. *Architecture is abstract until operationalized, Available at: goo.gl/HorpbH [Last accessed: 21 February 2016].*

[162] M. Fowler. (2015). *MicroservicePremium, Available at: goo.gl/3WVKsn [Last accessed: 31 October 2016].*

[163] I. Manotas *et al.*, "An empirical study of practitioners' perspectives on green software engineering," in *38th International Conference on Software Engineering*, Austin, Texas, 2016, pp. 237-248: ACM.

[164] B. Sokhan. *Domain Driven Design for Services Architecture, Available at: goo.gl/ftCLnR [Last accessed: 10 January 2016].*

[165] V. Gitlevich and E. Evans. *What is Domain-driven design? Available at: goo.gl/S3zMSR [Last accessed: 21 June 2016].*

[166] E. Evans, *Domain-driven design: tackling complexity in the heart of softwareT.* Addison-Wesley Professional, 2004.

[167] Alberto Brandolini. (2013). *Introducing Event Storming, Available at: goo.gl/GMzzDv [Last accessed: 8 July 2017].*

[168] M. Erder and P. Pureur, *Continuous architecture: sustainable architecture in an agile and cloud-centric world.* Morgan Kaufmann, 2015.

[169] T. d. Pauw. (2017). *Feature Branching is Evil, Available at: https://speakerdeck.com/tdpauw/xp2017-feature-branching-is-evil/ [Last accessed: 27 May 2017].*

[170] M. T. Rahman, L. P. Querel, P. C. Rigby, and B. Adams, "Feature Toggles: Practitioner Practices and a Case Study," in *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, pp. 201-211.

[171] K. Gabhart. (2014). *Resilient IT Through DevOps, Available at: goo.gl/6KwMtN [Last accessed: 1 July 2017]. .*

[172] K. Wnuk, "Involving Relevant Stakeholders into the Decision Process about Software Components," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 129-132.

[173] N. Ernst, J. Klein, G. Mathew, and T. Menzies, "Using Stakeholder Preferences to Make Better Architecture Decisions," in *IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 133-136.

[174] S. Suneja *et al.*, "Safe Inspection of Live Virtual Machines," in *13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Xi'an, China, 2017, pp. 97-111, 3050766: ACM.

[175] L. Prewer. (2015). *Smoothing the continuous delivery path – a tale of two teams, Available at: goo.gl/1oqjsP [Last accessed: 2 October 2016].*

[176] D. Schauenberg. (2014). *Development, Deployment and Collaboration at Etsy, Available at: goo.gl/umGTM2 [Last accessed: 1 September 2017].*

[177] Y. Yaniv. (2014). *Closing the Gap Between Database Continuous Delivery and Code Continuous Delivery, Available at: goo.gl/mERZcV [Last accessed: 21 August 2016].*

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

[179] A. Wallgren. (2015). *Continuous Delivery of Microservices: Patterns and Processes, Available at: goo.gl/Yk6ddH [Last accessed: 10 February 2018].*

[180] W. John *et al.*, "Service Provider DevOps," *IEEE Communications Magazine,* vol. 55, no. 1, pp. 204-211, 2017.

[181] F. Ullah, A. J. Raft, M. Shahin, M. Zahedi, and M. A. Babar, "Security Support in Continuous Deployment Pipeline," in *12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, Porto, Portugal, 2017.

[182] R. d. Feijter, "Towards the adoption of DevOps in software product organizations: A maturity model approach," Master Thesis, Utrecht University, 2017.

[183] T. Laukkarinen, K. Kuusinen, and T. Mikkonen, "DevOps in regulated software development: case medical devices," in *39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, Buenos Aires, Argentina, 2017, pp. 15-18, 3102967: IEEE Press.

[184] M. Schmidt. (2016). *DevOps and Continuous Delivery: Not the Same, Available at: goo.gl/9q1zan [Last accessed: 2 October 2017].*

[185] "Exploring Microservices: 14 Questions Answered By Experts, Available at: goo.gl/rohpzK [Last accessed: 7 January 2018]." xebialabs.

[186] L. Zhu, L. Bass, and G. Champlin-Scharff, "DevOps and Its Practices," *IEEE Software,* vol. 33, no. 3, pp. 32-34, 2016.

[187] S. Haselböck, R. Weinreich, and G. Buchgeher, "Decision guidance models for microservices: service discovery and fault tolerance," in *Fifth European Conference on the Engineering of Computer-Based Systems*, Larnaca, Cyprus, 2017, pp. 1-10: ACM.

[188] M. Shahin, P. Liang, and M. R. Khayyambashi, "Architectural design decision: Existing models and tools," in *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2009, pp. 293-296.

[189] (2011). *ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, Available at: goo.gl/KbpNbE [Last accessed: 18 January 2018].*

[190] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems," in *International Conference on Research in Adaptive and Convergent Systems*, Krakow, Poland, 2017, pp. 228-235: ACM.

[191] T. Mauro. (2015). *Adopting Microservices at Netflix: Lessons for Architectural DesignT, Available at: goo.gl/iUxqXA [Last accessed: 10 December 2017].*

[192] G. Bergman. (2016). *Serving 86 million users – DevOps the Netflix way, Available at: goo.gl/AhPRk3 [Last accessed: 5 January 2018].*

[193] G. Haff. (2017). *DevOps success: A new team model emerges, Available at: goo.gl/c35qyE [Last accessed: 10 January 2018].*

[194] E. D. Nitto, P. Jamshidi, M. Guerriero, I. Spais, and D. A. Tamburri, "A software architecture framework for quality-aware DevOps," in *2nd International Workshop on Quality-Aware DevOps*, Germany, 2016, pp. 12-17: ACM.

[195] T. Potts and E. Ort, "Keep CALM and Architect On: An Architect's Role in DevOps," 13th Software Engineering Institute Architecture Technology User Network (SATURN) 2017.

[196] N. Kerzazi and B. Adams, "Who Needs Release and DevOps Engineers, and Why?," in *IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, 2016, pp. 77-83.