Empir Software Eng (2018) 23:3535–3577
https://doi.org/10.1007/s10664-018-9616-7

Comparison of release engineering practices in a large mature company and a startup

Eero Laukkanen[1] · Maria Paasivaara[1] · Juha Itkonen[1] · Casper Lassenius[1]

Published online: 28 March 2018

Abstract Modern release engineering practices provide multiple benefits for software companies, but organizations have struggled when trying to adopt the most advanced practices, such as continuous delivery. It is not known in which contexts the most advanced practices are applicable and what can be achieved by adopting them. In this study, we discuss the effect of the organizational context on adopted release engineering practices and what outcomes are achieved with the practices. We study two organizational contexts: the startup and the large mature company context. The effect of the product context is mitigated by studying two case organizations with similar products, a rare research opportunity. We performed 18 interviews with various roles in the case organizations. The number of production environments, the number of customers, the control over the production environment, the available resources, the organization size and the distribution of the organization affected the release engineering practices and the ability to release frequently. Having less internal verification and more customer verification enabled fast feedback and customer experimentation in the startup context, but increased the number of production defects. However, having more internal verification in the large mature company context surprisingly did not prevent production defects. The organizational context had a large effect on how achievable modern release engineering practices, such as continuous delivery, were. In the startup context,

Communicated by: Hakan Erdogmus

✉ Eero Laukkanen
eero.laukkanen@alumni.aalto.fi

Maria Paasivaara
maria.paasivaara@aalto.fi

Juha Itkonen
juha.itkonen@iki.fi

Casper Lassenius
casper.lassenius@aalto.fi

[1] Department of Computer Science, PO Box 15400, 00076 Aalto, Finland

the lack of resources was the main factor hindering the improvement of release engineering practices, while in the large mature company context, the number of stakeholders and products were the main factors.

Keywords Release engineering · Startup · Continuous integration · Continuous delivery · Case study

1 Introduction

Release engineering is the process of delivering the individual contributions of developers to the end users of a software product, consisting of version control, build and test with continuous integration, additional testing, deployment and release (Adams and McIntosh 2016). In modern release engineering practices, such as continuous delivery (CD) (Humble and Farley 2010), all the steps in release engineering process are automated, meaning that in principle, individual changes can be released, if needed, immediately after building and testing them.

There are many organizations, especially in the web application domain, that have adopted modern release engineering practices (Rahman et al. 2015). However, numerous cases have been reported of difficulty in adopting the practices (Laukkanen et al. 2017). While the existence of successful cases proves that the practice is applicable in some situations, adopting modern release engineering practices outside the sweet spot might not be as feasible. To avoid fruitless attempts to adopt the practices, we need a better understanding of how the organizational context affects the feasibility of adopting modern release engineering.

In this study, we discuss the effect of the organizational context on release engineering practices. We focus on two, previously less studied contexts: the startup and the large mature company context. Software engineering in startups has been empirically studied only recently (Paternoster et al. 2014; Giardino et al. 2016), but specific studies on release engineering are missing. In the large mature company context, modern release engineering has been discussed by a few studies (Debbiche et al. 2014; Laukkanen et al. 2015, 2016; Savor et al. 2016). Previous research has investigated adoption challenges and described the implemented practices, no systematic analysis of the effect of the context has been conducted.

In addition, only a few large, mature companies have reported successful adoption of modern release engineering, and understanding the organizational context in depth could reveal why such examples are so scarce. Companies, such as Facebook (Savor et al. 2016), have successfully adopted modern release engineering practices, but it is not known what separates them from other large companies and what difficulties there might be when the target system is not a web application. Furthermore, in some of the web application companies, the practices might have been in place from the beginning, and therefore no adoption process would have been needed after the organization had transformed to a large mature company.

We study two organizations that were building directly competing products for the same market. The first organization, a large mature company, was building a product that could be used by customers world-wide. The second organization, a young startup, had built its product according to the needs of one specific customer. After gaining success with the first customer, the startup was trying to attract other customers for the product.

As the products were similar, this helps isolate the effects of the organizational context from the product context when studying the release engineering practices. In addition, the large mature company had acquired the startup organization after the startup had proved to be more successful in the market. Thus, we could gain insights from experts that represented the large organization, but had familiarized themselves with the startup. The experts could compare the organizational contexts and provide insights into why something worked in one context but not in the other. We performed a total of 18 interviews, nine in the large mature company and nine in the startup.

The main contribution of this research is the identification of organizational context factors that have an effect on the release engineering practices. This identification serves as a starting point for understanding the phenomenon of modern release engineering practices in various contexts. To practitioners, we contribute by providing insights into release engineering practices in two contexts.

This paper is structured as follows: we provide a background for the study and review previous related studies in Section 2. The case organizations and our research methods are described in Section 3. The results of the study are presented in Section 4. Finally, the results and the threats to validity are discussed in Section 5 and the study is concluded in Section 6.

2 Background

In this section, we first introduce the concept of release engineering followed by a discussion of what we mean by organizational context. Finally, we review related empirical studies on the subject.

2.1 Release Engineering

Release engineering means the process of bringing the individual changes made by developers to the end users of the software with high quality (Adams and McIntosh 2016). Release engineering combines four software development activities: *software configuration management* (Berczuk and Appleton 2002) practices that define how changes, builds and releases are identified and tracked; *software testing* (Desikan 2006), the process of testing the software to find defects; *software release management* (van der Hoek et al. 1997), which defines how software is made available to and obtained by its users; and *IT operations* (Roche 2013) that ensures that the software is functioning properly in use in the *production environment*. Release engineering views the activities as a combination in order to avoid local optimization and to improve the release process as a whole.

Typically, release engineering covers six areas (Adams and McIntosh 2016):[1] 1) *version control*: branching and merging; 2) *deployment pipeline* (Humble and Farley 2010): building and testing; 3) *build systems*: compilation, linking and packaging; 4) *infrastructure-as-code*: automation of the configuration of servers, middleware, applications, firewalls, etc.; 5) *deployment*: installing a new version of software; 6) *release*: publishing a new version of software.

Designing the release engineering process is often a trade-off between release confidence and velocity (Schermann et al. 2016). *Release confidence* measures how confidently

[1]Same list as in (Adams and McIntosh 2016), except continuous integration is replaced with deployment pipeline in order to have a consistent terminology.

changes can be deployed after passing the release engineering process, and *release velocity* measures the time from the version control commit to the deployment to production environment. For example, having a large amount of manual testing in the deployment pipeline increases release confidence, but decreases release velocity substantially. Automatic tests provide a balance between confidence and velocity.

Modern release engineering practices (Adams and McIntosh 2016) aim to increase the release velocity of individual changes, but also ensure sufficient level of release confidence. The practices emphasize five principles: 1. build artifacts, application and environment configurations are version controlled similarly to software source code (Adams and McIntosh 2016); 2. different organizational functions of development, testing and operations collaborate tightly (DevOps) (Roche 2013); 3. the build, testing and deployment processes are automated (Humble and Farley 2010); 4. changes can be released individually instead of as parts of larger releases (Humble and Farley 2010); 5. in addition to detecting defects before production deployments, attention is given to defect identification during deployments and ability to rollback failed deployments (Fitz 2009).

In the industry, there has been three common characterizations of release engineering maturity: *Continuous integration* (CI) (Fowler 2006) is a practice of integrating and testing individual changes frequently. The changes are not necessarily deployable to the production environment all the time, but instead can require additional testing before production deployment. *Continuous delivery* (CD) (Humble and Farley 2010) is CI combined with the requirement that changes, after passing automated tests, can be deployed to production environment. However, production deployments are not necessarily done for each verified change. *Continuous deployment* (CDep) (Fitz 2009) is CD combined with the requirement that verified changes are automatically deployed to production environment. Even after production deployment, the change might not be released, because it can be made hidden from the users with feature toggles (Rahman et al. 2016).

2.2 Organizational Context

Software engineering (SE) research has suffered from having too little contextual understanding of the phenomena related to the field (Dybå et al. 2012). Although non-contextualized, generalized knowledge can be applied universally, SE results are often context-specific (Dybå et al. 2012), and misinterpreting them as universal can lead to hasty generalization. To mitigate this, SE researchers emphasize the importance of linking the results to the context they are found in (Dybå et al. 2012; Kruchten 2013).

In this study, we focus on the effects of the organizational contexts on release engineering practices in the case organizations. We do not use any predefined factors to determine organizational context, but instead derive the factors inductively based on the collected data. Thus, our definition of organizational context might differ from other sources, e.g., Kruchten who divides context between organizational and project context (Kruchten 2013). In this paper, we make a logical division between the organizational and the product context: the *product context* defines who the end users of the software are, what the product provides to them, and how and why the product is used; the *organizational context* defines, who develops the software, and where, when, why and how is it developed.

Example organizational contexts are: a startup company (Giardino et al. 2016), a mid-sized company (Neely and Stolt 2013), a development unit in a larger company (Laukkanen et al. 2015), an open source software development community (Michlmayr 2007) and an engineering project course (Lehtinen et al. 2014). From these, we focus on the startup and

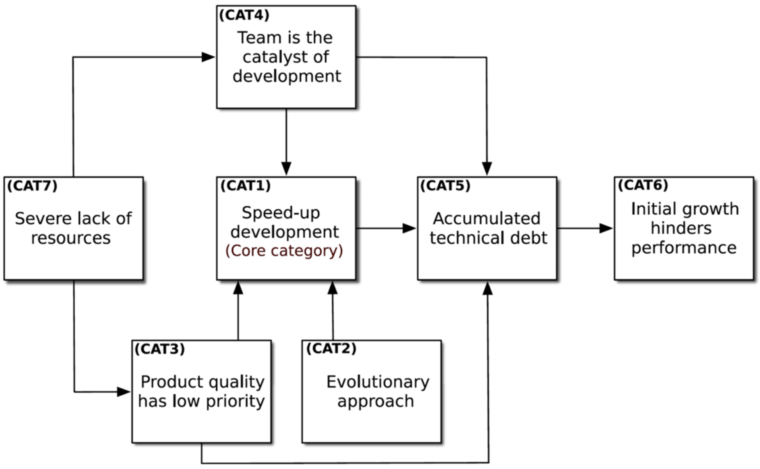
(CAT4)
Team is the catalyst of development
(CAT7)
Severe lack of resources
(CAT1)
Speed-up development
(Core category)
(CAT5)
Accumulated technical debt
(CAT6)
Initial growth hinders performance
(CAT3)
Product quality has low priority
(CAT2)
Evolutionary approach

Fig. 1 Greenfield startup model (Giardino et al. 2016)

development unit in a large company context, because those are two radically different contexts that have received little research attention regarding release engineering.

Startups are known for their ability to innovate (Yoffie and Cusumano 1999; Freeman and Engel 2007). However, the amount of empirical research about the nature of startup software development is limited (Paternoster et al. 2014). Nevertheless, a recent mapping study provides a characterization of startup software development: "*The most frequently reported contextual features of startups concern the general lack of resources, high reactiveness and flexibility, intense time-pressure, uncertain conditions and fast growth.*" (Paternoster et al. 2014). The dynamics of startup software development are captured in the greenfield startup model (Giardino et al. 2016), see Fig. 1.

Despite the fact that software development in large mature companies is common, few studies investigate what makes the large mature company software development context unique. Certain aspects of the development are investigated in individual studies: scale and responsiveness (Olsson et al. 2014), overscoping (Bjarnason et al. 2012), large-scale agile (van Waardenburg and van Vliet 2013; Dikert et al. 2016), configuration management (Fauzi et al. 2010), dependencies and communication (Sekitoleko et al. 2014), distributed development (Wagstrom and Datta 2014) and heavy processes (Laukkanen et al. 2016). The studies focus on different aspects of large mature companies: number of customers (Olsson et al. 2014), plan-driven processes (Laukkanen et al. 2016; Bjarnason et al. 2012; van Waardenburg and van Vliet 2013), number of personnel (Dikert et al. 2016), and organizational distance (Fauzi et al. 2010; Sekitoleko et al. 2014; Wagstrom and Datta 2014).

2.3 Empirical Studies on Release Engineering

Release engineering as a whole has been studied only in a few studies. Two dissertations on release engineering have been published, focusing on the open source context (Michlmayr 2007; Wright 2012). Modern release engineering practices (Adams and McIntosh 2016), like continuous integration, continuous delivery and continuous deployment have been under empirical investigation only recently, and most of the related publications have been experience reports (Laukkanen et al. 2017). A few studies have been made of individual aspects of release engineering, such as feature toggles (Rahman et al. 2016) and release stabilization (Rahman and Rigby 2015). However, the studies often lack contextual information (Laukkanen et al. 2017), while it has been acknowledged that the context plays a large role in release engineering (Wright 2012).

Release engineering has not been specifically studied in startups, but the studies of startup software development have identified some characteristics of release engineering, too (Paternoster et al. 2014; Giardino et al. 2016): lack of overall testing, lack of automated tests, manual smoke tests, user reports mitigate the lack of testing, reasonable level of automated testing for critical components, deploying multiple times per day, progressive roll-outs, direct contact and observation of users, automated feedback collection, analysis of product metrics, product quality has low priority, UX has the highest priority and development speed is emphasized over product quality. In addition, given that startups generally lack resources (Giardino et al. 2016) and the lack of resources is one key problem preventing adopting modern release engineering practices (Laukkanen et al. 2017), we can hypothesize that startups generally have not adopted modern release engineering practices.

Adoption of modern release engineering practices in large mature companies has been investigated in a few previous studies. Many adoption challenges have been revealed related to the development and release processes, system architecture and distribution of the organization (Debbiche et al. 2014; Laukkanen et al. 2015, 2016), and the adoption has been either limited (Laukkanen et al. 2015) or entirely stagnant (Debbiche et al. 2014; Laukkanen et al. 2016). However, it is not clear why the adoption is an issue in large mature companies, although it has been identified that the used product development process (Laukkanen et al. 2016), architecture and distribution of the organization (Laukkanen et al. 2015) play a critical role.

In summary, we identify a gap in the current state of the research that we aim to contribute to with this study. First, release engineering studies in general currently lack contextual understanding, although it has been acknowledged that the context has a large effect on the release engineering practices. Second, research on release engineering practices in startups has been limited to some characterizing aspects, but holistic understanding of them is missing. Finally, release engineering in large mature companies has been identified to be difficult, but the essence why the situation is different to other contexts has not been explored well enough.

3 Method

In this section, we introduce the research method used. First, we summarize the case organizations of the study briefly, as more in-depth description is provided in Section 4.3. After that, we present our research goal and questions. We continue by discussing the case selection rationale, followed by data collection and analysis methods. Finally, we present how we validated our results and mitigated the threats to validity.

3.1 Case Organizations

Both organizations were building competing software products. Case *BigCorp* was a product development unit in a large mature company, whereas Case *SmallOrg* was a small startup company (see Table 1). The company of BigCorp acquired SmallOrg after the SmallOrg product proved to be more successful in the market. In this study, we performed interviews

Table 1 Summary of the case organizations before the acquisition

Theme	Case BigCorp	Case SmallOrg
Organizational context	Product development unit in a large mature company	Startup company
Company size	>20,000	50
Case organization size	180	50
Distribution of the organization	Two sites in Europe, one site in Asia	One site in North America

in both case organizations after the acquisition (see Fig. 2). However, in our analysis we focus on the release engineering practices before the acquisition.

3.1.1 Case BigCorp

BigCorp is a product development unit in a mature, large, multi-national company. The product was a continuation of other older products, but the current development was started approximately two years before the acquisition (see Fig. 2). At the time of the acquisition, BigCorp's product had tens of existing customers, but was not getting new deals due to competition by, e.g., SmallOrg. In order to be competitive in the market, the company of BigCorp acquired SmallOrg.

3.1.2 Case SmallOrg

SmallOrg was a relatively young startup, having started the development of the product approximately four years before the acquisition (see Fig. 2). During the initial years, the number of personnel in the case was low, but after acquiring its first customer, it had scaled its software development personnel to three teams and 20 developers. As the product had gained success when demonstrated to new customers, SmallOrg realized that it would need either more investments or be acquired by a larger company to support the growth of the product. Thus, both BigCorp and SmallOrg considered the deal to be a win-win.

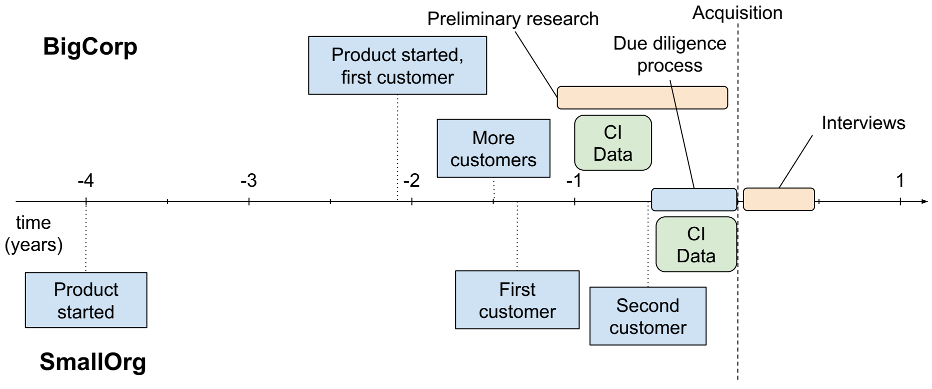

Fig. 2 Timeline of the case organizations and the research

3.2 Research Goal and Questions

In this study, our goal is to *understand how the organizational context affects release engineering practices in a startup compared to a large mature company*. We aim to reach this goal by answering the following research questions:

RQ1. What release engineering practices had the case organizations implemented?
RQ2. What outcomes did the implemented release engineering practices have for the case organizations?
RQ3. What were the reasons for differences in release engineering practices in the case organizations?

3.3 Case Selection Rationale

The case study comparing BigCorp and SmallOrg is revelatory (Yin 1994), as it is not often possible to mitigate the effect of product context when comparing software development organizations. Thus, this setting provides a rare opportunity for research. In addition, SmallOrg was acquired by BigCorp and the employees of BigCorp had acquired understanding of the SmallOrg before this study, first, during the due diligence process (see Fig. 2), and second, after the acquisition when the employees started to work with the SmallOrg product a few months before the interviews. Thus, we were able to get expert insights of the differences in the case organizations and the reasons for the differences. Finally, we were able to compare release engineering practices, as both organizations had implemented some modern practices in their work.

3.4 Data Collection

In order to justify the use of BigCorp interviews as evidence for SmallOrg practices, we summarize the due diligence process briefly. The due diligence process started half a year before the acquisition, although BigCorp had done preliminary assessments of the SmallOrg even a year before the due diligence process had been started. During the due diligence process, BigCorp employees visited the SmallOrg and asked questions, received demos and documentation of the system and could see the source code. One of the architects we interviewed took part in the due diligence process. The rest of our interviewees started to work with SmallOrg after the acquisition decision had been made. Thus, they had already been working with them for several months before the interviews and were sufficiently familiar with SmallOrg.

We performed 18 interviews using an interview guide approach (Patton 2002). The first two interviews with BigCorp managers were more flexible, as we familiarized ourselves with the cases and the situation. In the first interview, two managers participated in order to have different viewpoints when planning the subsequent interviews. After the initial interview, we designed an interview guide with topics to discuss, see Table 2 for the themes covered. An interview guide does not have explicit questions, but instead lists topics to discuss (Patton 2002) to allow flexibility during the interviews. We allowed flexibility in the later interviews too, in order to adapt to the different roles of the interviewees and to identify and discuss emerging issues.

In order to be able to compare the two contexts, we focused data collection and analysis on the time period before the acquisition. As the interviews were performed post-acquisition (see Fig. 2), we emphasized during the interviews that we wanted to discuss the time period

Table 2 Themes of the interview guide used in the interviews

Theme	Topics
Background info	Interviewee role before the acquisition, place in the organization, responsibilities, relations to others in the organization
Organizational differences	Differences between SmallOrg and BigCorp, explanations for the differences, consequences of the differences
Software development process	Customer requests, planning and prioritization, quality assurance, release
Continuous integration	Practices, test automation, tools, problems, future plans
Organizational structure	Teams, responsibilities, division of labor, communication, decision making
Product	Features, architecture, technologies
Metrics	Release cycle, CI cycle, defect counts
Other topics	Explanations for the SmallOrg's market success

before the acquisition instead of the current situation. In addition, during the data analysis we were cautious to not use any quotations that described the current situation instead of the time before the acquisition.

In BigCorp, we interviewed employees that had familiarized themselves with SmallOrg: six managers and three architects (see Table 3). The managers were responsible for managing the development of the BigCorp product and later integrating SmallOrg into BigCorp. The architects were responsible for the system design and release engineering practices.

We did not interview any developers or testers in BigCorp, as we had performed preliminary research in the case organization earlier (see Fig. 2 and Table 4). The preliminary research consisted of a background interview, CI data collection and two group interviews. In the two group interviews, we had heard the developers and testers about the release engineering practices in BigCorp and did not consider it necessary to interview more of them in this study. The preliminary research data was used for describing the release engineering process (see Fig. 4) and triangulating the other interviews. Otherwise, the interviews in Table 3 were the data source for the results of the study.

From SmallOrg, we attempted to get a holistic view of the organization by interviewing all the roles in the organization: a manager, an architect, three development team members, two service team members and two domain experts.

The number of interviewers and the length of the interviews varied, see Table 3. Most of the BigCorp interviews were performed by at least two researchers, whereas all SmallOrg interviews were performed by only one researcher due to the geographical distance to the SmallOrg offices. The BigCorp managers who were interviewed by the single researcher were visiting the SmallOrg site during the interviews. The interviews took 50–110 min.

We also collected quantitative data from CI systems, and had access to the SmallOrg wiki pages, where information about the development process and practices was stored. The wiki pages were used to gain an early understanding of the process and practices, but were not used in the actual data analysis, because written processes and practices do not always correspond the actual used processes and practices. The quantitative CI data from the CI systems contained the history of product builds, their duration and results, see Table 5 for

Table 3 Overview of the interviews. The flexible interviews without the interview guide are denoted with (*)

Inter-viewers	Interviewees	Interviewee's role (at the time of the interviews) and experience	Length (min)
3	2 BigCorp managers on Site A (*)	Business improvement manager. 23 years at BigCorp in different R&D management positions. (For the second manager see the next row.)	83
3	BigCorp manager on Site A (*)	Head of business line transformation. Manager at BigCorp for 21 years. Led the agile transformation and agile coaching. Was involved in the assessment of the SmallOrg before the acquisition.	110
2	BigCorp manager on Site A	R&D leader. Assessed SmallOrg methods and tools in the acquisition. 22 years in various roles at BigCorp, e.g., testing manager, program manager and agile and lean coach.	50
3	BigCorp manager on Site C	Manager and Coach. Was involved in the evaluation of the SmallOrg before the acquisition and after the acquisition became part of the joint integration team in the areas of IT and HR. Started 12 years ago as at BigCorp as software engineer. Various positions in R&D and R&D management.	74
1	BigCorp manager on Site D	Business PO. Responsible of the studied business area at BigCorp. One year in the project. 21 years in BigCorp, starting as a developer. Various positions such as project management, program management, product customization and pre-sales.	74
1	BigCorp manager on Site D	Program manager. 2,5 years in the program management of the studies product. Joined BigCorp 17 years ago as a developer, then architect, line manager and program manager.	57
2	BigCorp architect on Site B	Head architect. Head technical assessor in the acquisition of SmallOrg. 21 years at BigCorp, 17 years in related products.	84
2	BigCorp architect on Site B	System Architect. Was involved in the technical evaluation of the SmallCorp's product and has worked in the integration of the products since the acquisition. Joined BigCorp 11 years ago as a developer. Worked already for the predecessor product.	74
2	BigCorp architect on Site B	Technical Product Owner. Was involved in planning the integration of the two products and companies since the beginning of the acquisition, concentrating especially in continuous integration and continuous delivery aspects.	107

Table 3 (continued)

Interviewers	Interviewees	Interviewee's role (at the time of the interviews) and experience	Length (min)
1	SmallOrg manager	Head of Product Strategy at BigCorp. Before the acquisition the CEO of SmallOrg, one of the founders of the SmallOrg. Over 20 years in this business area.	81
1	SmallOrg architect	Chief architect. One of first employees at SmallOrg, started as a senior developer. 15 years in this business area.	72
1	SmallOrg domain expert	Head of systems engineering. One of the original founders of SmallOrg. Has worked in this business area over 20 years. Technical background, started in hardware design.	77
1	SmallOrg domain expert	Systems engineer. Has worked in this business area for more than 10 years in several companies, first on the customer side and then in design.	62
1	SmallOrg development team member	Technical lead. Leads an integration team.	79
1	SmallOrg development team member	Technical lead and line manager. 9 years in software development, started 8 years ago in the predecessor of SmallOrg and 3 years ago in this product as a developer.	84
1	SmallOrg development team member	Developer. Has worked at SmallOrg since graduation from the university, i.e., 1,5 years.	51
1	SmallOrg service team lead	Head of engineering services. 18 year of experience in this business area. Previously field support engineer meeting customers. One of the founders of the SmallOrg.	67
1	SmallOrg service team member	Customer support and services manager. 3 years in SmallOrg, first as a developer and moved two years ago to the customer support team to perform customer trials.	62

an overview of the collected CI data. The data was not used in the primary analysis in this study, but was employed in triangulating observations from the interviews. In addition, the CI data was used to illustrate the difference in CI discipline between the two cases by calculating descriptive statistics (see Table 9). The collected CI data was from the time before the acquisition (see Fig. 2).

BigCorp's CI data was half a year older than SmallOrg's due to data availability. BigCorp deleted old CI data after few weeks to save disk space and thus the data had to be actively collected by us every week. We think that the time difference did not bias the results, because there is no improvement visible in the collected BigCorp CI data. If there was improvement in the data, BigCorp could have performed better during the same time period when the SmallOrg data was collected. Furthermore, the difference between BigCorp

Table 4 Summary of the preliminary research we had done in BigCorp before this study

Method	Description
Background interview	An interview with a person responsible for improving release engineering practices in BigCorp. The length of the interview was 102 min. The interview consisted of the following themes: case background information, used software development methods, testing strategy, use of devops, used development tools, challenges and benefits of CD in the case organization.
CI data collection	CI data was collected for the analysis of broken builds.
Two group interviews	Two group interviews were performed with managers, testers and developers on one site of the case organization. The interviews lasted 104 and 160 min and both of them had 10 interviewees. The group interviews discussed the adoption problems of modern release engineering practices.

data and SmallOrg data is so large that we believe it cannot be explained with the difference in time period only.

3.5 Data Analysis

The interviews were audio recorded and transcribed by a professional transcription company. The transcriptions were qualitatively coded (Patton 2002) by the first author. First, quotations from the transcriptions were selected for later analysis when the quotations discussed either the organizational or product context, release engineering practices or the outcomes of the practices. Second, the first author gave descriptive codes for the individual concepts in the quotations, see Table 6 for the codes used in the analysis. The codes were categorized either to be product context codes, organizational context codes, release engineering implementation codes or outcome codes.

Third, the first author wrote an initial draft of the result section of this paper, structuring it by the identified codes. The other authors read the initial draft and gave critical comments on whether the results were valid and how could they be improved. Through many iterations, the results gained their final form when all the authors were satisfied and confident that they truthfully represent the interview material and the case reality. Finally, key stakeholders in BigCorp validated the results by reading a nearly finished version of the article. They could also verify the results from the viewpoint of SmallOrg, as they had worked closely with the organization after the acquisition.

Table 5 Collected quantitative CI data from the CI systems

Case	First data point	Time period	# of builds
BigCorp	Year before acquisition	22 weeks	1889
SmallOrg	Half a year before acquisition	24 weeks	760

Table 6 Codes used in the article

Product context	Customer-specific customization required, customer traditionally do not want frequent releases, need to react fast to customer requests, some customers ready for frequent releases, complex product, early market, customers want to test the product in their environment
Organizational context – Customers	Single customer, friendly customer, multiple customers, strict service level agreement
Organizational context – Organizational structure	Co-location, collaborative organization, functional organization, large organization, multi-site, rigid organization, small organization
Organizational context – Product development process	Flexible process, plan-driven process, risk-avoiding organization, high system quality requirements, daily prioritization, product portfolio, overtime work on weekends
Organizational context – Resources	Amazon AWS, no production-like test environment, getting test environment takes minutes, getting production-like test environment takes months
Release engineering practices – Code review	Code review gate, build in branches, no code review
Release engineering practices – Continuous integration discipline	Fast build, build kept green, build in branches, build only changed components, failing build, slow build
Release engineering practices – Internal verification scope	Low test automation, manual system tests, integration tests in CI, simulators with customer data, low verification, smoke tests, no definition of done, low performance verification, dedicated testers, definition of done, production-like test environment, unit tests
Release engineering practices – Domain expert testing	Domain expert testing
Release engineering practices – Testing with customers	Customer acceptance testing, closed-loop testing, open-loop testing, strict service level agreement, manual progressive deployments
Release engineering outcomes – Feedback from customers	Fast reacting to customer needs, two week release capability, direct feedback from production, fast development cycle, no feedback from production, slow development cycle, slow reacting to customer needs, slow release capability
Release engineering outcomes – Number of defect reports	Many defect reports
Release engineering outcomes – Quality issues with new customers	Quality problems, product not ready for general availability, technical debt, elementary programming mistakes, scalability problems

3.6 Mitigating Threats to Validity

We mitigated the threats to validity of case study research (Runeson and Höst 2009) by using the following strategies for construct validity, internal validity and external validity.

Construct validity: by using inductive data collection and analysis methods, we made sure that the used constructs are the same as used by the case organization members. This means that we did not try to investigate constructs that were external to the case organizations, but instead allowed the case organization members themselves decide which words or meanings they use when describing their release engineering practices.

Internal validity: we only used such evidence from the interviews that was logically consistent with other interviewees' experiences. This mitigated the possibility that the evidence would be an opinion of single persons and might not represent the truth. We also interviewed persons in different roles in the case organizations and therefore the results represent a holistic viewpoint. In addition, we collected quantitative data from actual CI systems (see Table 5) to triangulate the qualitative evidence. We had multiple researchers who took part in the interviews and analysis, thus the consistency is not dependent on single researchers either. In addition, we validated our results with two case organization members after the analysis.

External validity: in a case study, we do not attempt statistical generalization but instead analytical generalization (Yin 1994). To achieve this, we extracted generalizable variables and their relationships based on the two case studies (see Fig. 8).

4 Results

In this section, we present the results of the study. First, we present overview descriptions of the development processes of the case organizations. Second, we cover the product context which was common for both case organizations. Third, we introduce the organizational contexts of the case organizations. Fourth, we show what kind of release engineering practices were in place in the case organizations. Fifth, we investigate what kind of outcomes the case organizations achieved with their release engineering practices. Finally, we synthesize the results by comparing the cases to understand the reasons for the differences in release engineering practices.

4.1 Overview of the Development Processes

4.1.1 Case BigCorp

The BigCorp product had multiple world-wide customers whose feature requests went through a typical roadmapping and prioritization process. The features were specified and grouped into releases by the product management unit which was external to BigCorp. BigCorp received the detailed release plans which BigCorp implemented in a three month development cycle. During development, the continuous integration system was used to verify the product continuously. The product still required additional testing near the end of the development cycle. When the release development was finished by BigCorp, it was sent to an external unit who performed end-to-end testing for the product. After that, the release would be installed to pilot customers and finally, given that the quality of the release was sufficient, it would receive general availability status.

4.1.2 Case SmallOrg

The SmallOrg product had one friendly customer who was heavily involved in the product development. SmallOrg domain experts worked closely with the friendly customer experts to specify features for the product. After that, the domain experts discussed the features directly with the software developers who were responsible for implementing the features. Every code change to the product was reviewed by a lead developer and a continuous integration system would run automated tests for each change. Every two weeks the product was manually release tested by SmallOrg and after that the release was taken to the customer test environment for external testing. If there were no problems with the release, it would be gradually installed to the whole production environment of the friendly customer.

4.2 Product Context

The product is a software product sold to the customers of the case organizations worldwide. The product is installed to customer premises and integrates with customer-specific production systems.

The market for the product is still in its early stages, which means that there is ambiguity in product requirements, but also potential for substantial growth. The product has to integrate to different kinds of production environments, which can vary technologically between customers and regions.

> In Asia, the customer needs are different than in USA. [...] And then, Korea and Japan have their own quality and other requirements. —BigCorp Manager

Due to the requirement ambiguity and production environment differences, the product has to be customized for each customer. Thus, the business is a combination of product and service business, where a general product is customized for each customer's needs. Some customer-specific customizations can be productized later on to provide the customizations to other customers too. In addition, customers want to have a *proof of concept* (PoC) before buying the product. In a PoC, the product provider demonstrates the benefits of the product by running it in the customer's environment. During PoCs, customers often have customization requests specific to their own needs and being able to address those requests is a good way to assure the customer of the benefits of the product.

> In the end, [the business] is closer to service business than product business. You cannot make [the product] in a factory that work adequately well, but instead you have to go to the customer's [environment], or at least to a test [environment] and do little testing and fix them there. —BigCorp Manager

> Usually you have to implement some things [...] that are gonna be [customer] dependent. — SmallOrg Dev Team

> When you do a proof of concept to a customer, then there will be things that the customer wants to have really quickly. Like, "if you can put this [feature] there, then I will buy it." —BigCorp Manager

The product is installed to the customer premises, and thus each customer has its own instance of the product running. After buying the product, the customers traditionally do not want frequent updates to the system, because a malfunctioning product can cause serious harm to the customer. A typical release cycle to customers, including new features and defect fixes, can be from six months to one year. The actual product release cycle can be faster, as it was in both BigCorp and SmallOrg, but individual customers are not typically willing to accept more frequent updates. This puts pressure on the release engineering practices, since there is no possibility to easily fix defects after a release.

> If a product gets a good reputation, that typically it does not cause any hiccups, then maybe three months could be the fastest release cycle. And even then you would need to have a very willing [customer]. Half a year is the most probable release cycle, in my opinion. Because it is not a trivial thing for the [customers], ever. [...] If you release an immature [product], you can get the whole [production environment] down suddenly, totally. —BigCorp Architect

> I know there has been some customer requirements or requests to say, we want a product [version] to be supported for six months, and only be installed once a year or something like that. —SmallOrg Dev Team

However, since the customer environments differ a lot from each other, it is difficult or practically impossible to verify the release quality in a laboratory setting, and often some defects are only discovered in customer environments. Luckily, there are some customers that are more risk-tolerant and allow monthly or even more frequent releases.

> We did work with the [customer] [...] a lot to actually go ahead and, verify, what the impact of the changes will be in the [production environment]. That also uncovered [many other issues], because now you're connecting to an actual [environment], the information in the actual [environment] can be different, and, data can be missing sometimes and so, the hardening of [the product] takes a lot of time. —SmallOrg Domain Expert

> That's kind of like the biggest source of our, defects and bugs. Is just running on new sets of data that we didn't run before. That exposes new, assumptions in our system that were not correct. —SmallOrg Dev Team

4.3 Organizational Context

In this section, we introduce the organizational contexts of the case organizations. Considering what would be the most noteworthy aspects related to engineering practices, we divided the organizational context into four themes (see Table 7): customers, organizational structure, product development process and resources.

Table 7 Comparison of the organizational contexts in the case organizations

Theme	Case BigCorp	Case SmallOrg
Customers	Tens	One, Friendly
Organizational structure	Large, Distributed, Functional	Small, Co-located, Collaborative
Product development process	Long-term, Multiple stakeholders and products	Short-term, Few stakeholders, One product
Resources	Sufficient workforce and Test equipment	Limited workforce and Test equipment

4.3.1 Customers

The number of customers and customer intimacy differed a lot between the two case organizations. BigCorp had multiple existing customers, whereas SmallOrg worked mostly with only one but friendly customer.

BigCorp BigCorp had tens of existing customers for their product at the time of acquisition. Having so many customers globally makes the development processes heavier for three reasons. First, requests from different customers need to be prioritized, because it is not feasible to implement every requirement on demand. The prioritization will decrease customer satisfaction, because every request cannot be served.

> If you have five, ten or fifty customers things will change completely. Every customer will do their own requests and they are typically different from each other.
> —BigCorp Architect

> You have to have global market view what are the most important things. Which leads to unsatisfied customers because you cannot fulfill every customer's priority requests.
> —BigCorp Architect

Second, different releases to different customers need to be tracked, in order to be able to provide customer support and defect fixes to different customers. In addition, updating to newer releases has to support many previous versions, because different customers might have many different versions running.

> When you think globally, you have to think how do you release to different customers, how do you track what each customers has, what patches has gone to where.
> —BigCorp Architect

Finally, you have to be more careful with the quality of the product, because getting defect reports from many customers simultaneously would congest the development pipeline quickly, as different kinds of defects can be found by different customers.

> With many customers, you have to pay more attention to product quality. [...] If you have released the product to fifty customers and you find a defect, find a second and a third defect, then it will get harder to fix the defects in time. —BigCorp Architect

SmallOrg For most of the time before the acquisition, SmallOrg had only one paying customer. They had signed a contract with a second one a few months before the acquisition (see Fig. 2), but the second customer was not using the product before the acquisition. Thus, it is accurate to say that SmallOrg had only one customer before the acquisition when discussing release engineering practices.

SmallOrg's relationship with their first customer was intimate and friendly. The customer was geographically located near SmallOrg and they had tight communication from early on during the product development. The customer was willing to get frequent updates to the product and gave SmallOrg direct access to the production environment, which helped SmallOrg discover requirements for the product and fix issues that would only appear in the production environment.

> We already were building a product and [the SmallOrg customer] said oh, why don't you put this and that, and we changed the whole product to match what they were asking for. —SmallOrg Service Team

> [SmallOrg] set up this kind of, ways of working, that they have this weekly, project management meeting with [SmallOrg customer]. Over there basically they have access to a quite large [environment] and, kind of the daily issues with that [environment]. That is a powerful addition, for the development of the product. What we did with [BigCorp product], I think [...] we haven't been that close, to the customer.
> —BigCorp Manager

> They had a very good relationship with [SmallOrg customer], where they could develop the software, in the [SmallOrg customer environment] so you could, take a software which is half-ready, to the [SmallOrg customer environment], test, improve, test, improve, and then deploy it, throughout. —BigCorp Manager

SmallOrg could prioritize and release new content quickly compared to BigCorp, because there was only a single customer and production instance. In addition, SmallOrg had made a strict service-level agreement (SLA) with the customer so that if there were any defects in the released product, the defects would be fixed fast and the customer would still be satisfied. The SLA demanded that SmallOrg had to respond to critical requests in 5 min, although actually fixing the issues could take longer time.

> At [SmallOrg] they managed to do somehow that, the customer was quite satisfied and, the main reason for that was this really really short, feedback, cycle that, they had. So when the customer reported the issue, [...] they were, kind of, replying in the five-minute time frame. And then also the, correction package arrived [...] in a very short time. —BigCorp Manager

4.3.2 Organizational Structure

BigCorp was a large and functionally divided organization, and the company of BigCorp was developing multiple products. The studied product was one product in a larger product portfolio. In contrast, SmallOrg focused only on the studied product and was a small co-located organization (see Fig. 3).

BigCorp In order to efficiently manage multiple product portfolios and a global customer base, BigCorp was organized functionally and was distributed over the world (see Fig. 3). Functional organization meant that BigCorp had different organizations for product management, product development, customer support, sales and delivery. The product management and development organizations were specific to the product portfolio where the BigCorp product resided, whereas the customer support, sales and delivery organizations instead worked with all the products in the BigCorp company.

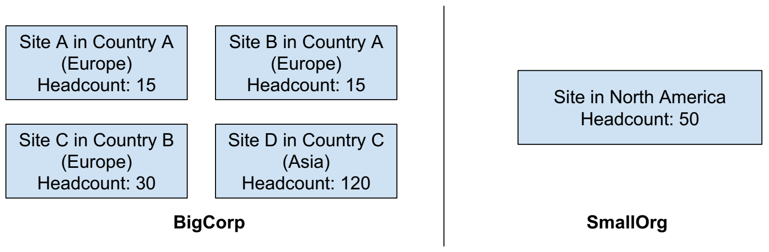
Site A in Country A (Europe) Headcount: 15
Site B in Country A (Europe) Headcount: 15
Site C in Country B (Europe) Headcount: 30
Site D in Country C (Asia) Headcount: 120
BigCorp
Site in North America Headcount: 50
SmallOrg

Fig. 3 Structural difference between the product development organizations of the studied products

> I think this is quite a typical structure what we have in BigCorp. We have a business line where is the product development. Then we have sales in another organization and delivery in a third organization. And there comes the challenge that how to work over the different functions that work with different logic, incentives and success criteria. How to get the whole to work together. —BigCorp Manager

> The power from a BigCorp point of view is to also see, we have a huge portfolio. And how do you really interplay across this portfolio in order to really get value to the customer. —BigCorp Manager

From the viewpoint of the studied product, the functional organization was a major challenge, because the product differed from the other products of the BigCorp company. Being an early market small software product, it had lower priority than other mature, large products. The prioritization prevented other organizations, such as sales, support and delivery, from adapting their ways of working to match how software could be developed efficiently from sales to delivery. In addition, as the product development was a separate organization with little interaction with the customers, it had a limited possibility to gain fast feedback from the changes done to the product.

The size and distribution of the organization also had an effect on the release engineering practices. There were both geographical and timezone differences even inside single functions, such as product development. The size and distribution made communication more difficult and decision-making took more time.

> However, when you have a big team which is distributed to many sites, and you have many customers, so you're not so, let's say flexible. So that's why, actually, I think, one reason why [BigCorp product], development was, became very slow, is that the [BigCorp product] team was growing too fast. —BigCorp Architect

> But in case of [BigCorp product] [...] lead designers or even architects [are] on different sites. And if we need to agree on something, it takes much more time so, because of time zone difference and, anyway, even if we are in the same time zone, it's a different thing when you discuss it over the phone or, just, in the same room. —BigCorp Architect

SmallOrg SmallOrg was a small, co-located and collaborative organization. The organization was functionally divided into management, domain experts (employees with years of experience in the product domain), software development, and customer service. Due to the small size and organizational culture, the functions did not become silos, as people collaborated closely over the functional borders.

The co-location and size of the organization made communication easy and decision-making fast. Only a small number of people were needed for decision-making, all of which were available directly to each other through physical interaction or electronic communication tools. There was less need for written documentation, because any information needed could be asked straight from the knowledgeable person.

The SmallOrg employees did not themselves report the small organizational size as important for their success. However, the BigCorp employees emphasized the size difference in the interviews.

> [SmallOrg] was still a small company it was a bit easier because of course they were kind of co-located in the same room. So if there were some, say, problems or some synchronization was needed, so they just talked to each other. [...] In my opinion this is a very big advantage, when the team is co-located. —BigCorp Architect

> The fact that the [SmallOrg] was in one place and that, let's say, the most, difficult parts, they were done by a few lead developers. [...] So, if you want to do some changes, or implement some urgent customer requests, it was quite easy to do because these three guys are, in the same room, and they can agree with each other how to do it quickly. —BigCorp Architect

The software development teams did not have direct contact with the customers, but the distance was nevertheless short, because the domain experts and service teams, who were in contact with the customers, had direct communication with the developers. The domain experts and customer service team members could either contact the developers face-to-face in the SmallOrg office, or use a chat application.

> I: So the connection between the [proof of concept] and software engineering is very tight?
> R: Super. Very, like direct communication. —SmallOrg Service Team

In addition to the tight collaboration, the domain experts were responsible for the customer requests from specification to delivery. Thus, there were no hand-overs between different functions, which is typical in a larger functional organization and often results in delays and lower quality.

> [SmallOrg] had domain experts and all the requirements to the developers came through the domain experts. And when a new feature goes to a customer, the experts are responsible for it end-to-end. Their responsibility does not end when the requirement is specified but only when it is used by a customer and customer verifies that it is good. —BigCorp Manager

4.3.3 Product Development Process

BigCorp had a heavy plan-driven process. SmallOrg was flexible and agile, changing priorities constantly.

BigCorp Due to the functional and distributed organization, the size of the product portfolio and number of customers, BigCorp's product development process was a heavy and plan-driven stage-gate process (Cooper 1990). While the actual software development had elements from Scrum and Kanban and was performed iteratively with cross-functional teams, other parts of the product development process were not flexible. The release scope and deadlines were fixed before starting the software development.

Customer requests went through a typical roadmapping and prioritization process before they reached the development organization. The lead-time from customer request to delivery could be as long as 18 months, although the product release cycle was three months. The features were delivered as quarterly releases, and the customers paid for each release.

> The [BigCorp] system is much more rigid and slower. We say that we can put [a customer request] on the roadmap and it will come after 18 months.
> —BigCorp Manager

Since there was a long organizational distance from the sales and marketing to the development organization, it was important that the plans would be realized completely, because otherwise there would be implications for the other organizations too. Failure to fulfill commitments would be punished with a bureaucratic process where one would need to explain what went wrong. Therefore, the development organization tried to make commitments they were totally sure they could deliver and plans would not be changed after initiation.

> In [BigCorp] the work is very Waterfall-like. You have to certainly commit to things and just those things have to be delivered. Even if only a little bit is not delivered, you will be punished enormously. —BigCorp Architect

In practice, however, software development is highly complex and unpredictable work. Even when the development organization tried to minimize the risk of missing the planned targets, it became clear that there was too much work in the plans. The plan-driven process caused burden on the product development organization, because the execution of the long-term plans was not feasible in practice. The burden was visible in being late in development and in having to do overtime work.

> For [BigCorp product], we have had to work over the weekends for a year now. —BigCorp Manager

During the development process, there were quality gates whose intent it was that certain criteria would be met and the release was proceeding as planned. In practice, however, those quality criteria did not prevent low quality software from entering the last stage, general availability, in the process. Still, the quality gates generated lots of bureaucracy to be managed for the development organization.

> We have lots of [quality gates], long lists. Now, if you think [BigCorp product], we have a customer who uses a release which has passed general availability a long time ago. The quality of that release is very low. This illustrates what our extremely heavy process can do. —BigCorp Manager

SmallOrg To be able to react to customer requests quickly, SmallOrg had a flexible product development process. Although new features were specified and implemented typically in a few months, smaller changes could be started immediately after customer requests. The developers were organized in three component teams and used elements from Scrum in their development work. New changes could be delivered every two weeks, which was the release cycle of the product. Especially during the proof of concepts, it was valuable to be able to react to customer requests fast, and even the two weeks release cycle could be too slow for reacting to the customer requests during the PoCs. Thus, during the PoCs, some changes were applied immediately after implementing them, overriding the official release cycle.

> I think it's the, start-up, let's say, mode of working that they, try to do as fast as possible what the customer requests. —BigCorp Architect

There were two downsides of the flexible process: the accumulation of technical debt and constant reprioritization. Because changes were made frequently, the technical debt kept building up. SmallOrg was aware of the issue, but having limited resources and a need to satisfy customers, it made a conscious decision to let the technical debt accumulate.

> But I think that, [SmallOrg was] investing more in feature development rather than, kind of, trying to reduce this technical debt, they were building up. —BigCorp Manager

> [SmallOrg]'s focus has been, building a product that has usefulness. Sometimes.. we take a trade-off between a highly scalable product. Versus a product that you can do things, well. And do useful things. —SmallOrg Architect

Software developers were under a constant interference due to reprioritization. It was good for the business to react quickly to customer requirements, but some developers thought that there could have been some protection measures that would have allowed better concentration and productivity for the developers.

> It seems like we just have to switch what we're working on a lot. [...] For example, the support team will just come over to developers and ask them things or email them directly. [...] The developers are being pulled in different directions a lot.
> —SmallOrg Dev Team

4.3.4 Resources

BigCorp had vast resources for product development, but adjusting the amount of resources given to the product development organization had delays. SmallOrg was struggling with limited resources, which is common for startups. However, when resources would provide direct value for the organization, it was a short decision to provide those resources.

BigCorp Having a globally recognized brand, BigCorp had high standards for their product quality and system-level requirements. The requirements included concepts of high availability, load balancing and scalability. Thus, BigCorp had resources to implement and verify that these system-level requirements are in place. In addition, BigCorp was a large organization with plenty of workforce for product development.

> As we are [BigCorp], we have to have certain system-level features, such as high availability, scalability etc. in place. —BigCorp Architect

Despite the fact that there were lots of resources, the processes of the large organization were slow to provide new hardware for test environments, for example. Thus, having resources did not necessarily mean that they could be used quickly.

> It takes two to three months to get a laboratory equipment with dedicated hardware.
> —BigCorp Manager

SmallOrg Being a startup, SmallOrg was constantly struggling with limited resources. The lack of resources was visible in that the organization did not have any production-like test environment for internal verification of the product. Thus, SmallOrg was forced to use the customer's environments for verification and testing.

> [SmallOrg] was lacking the resources of, testing the software in a, close-to-customer environment. —BigCorp Manager

Despite the overall lack of resources, SmallOrg had invested in elastic cloud from Amazon that they used for testing purposes and running their CI system. Given that there was a need for something in the organization, investments could be made rather easily if they would bring value for the product development.

> We looked into using Amazon Web Service to help us to easily, elastically scale the hardware that we need. —SmallOrg Architect

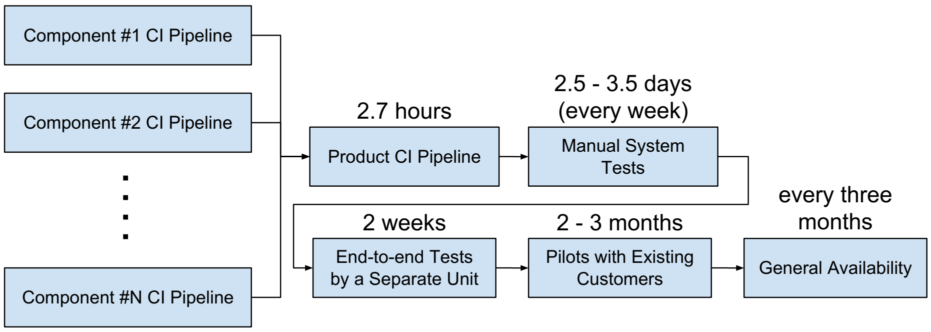

Fig. 4 Case BigCorp release engineering process

4.4 Release Engineering Practices

In this section, we show what kind of release engineering practices were in place in the case organizations. BigCorp, in general, had a sophisticated release engineering pipeline (see Fig. 4), although it had not achieved good CI discipline. SmallOrg had a faster pipeline (see Fig. 5) and had achieved good CI discipline, but its verification scope, both automated and manual, was lower than BigCorp's. SmallOrg mitigated the lower verification scope with other release engineering practices. We compare the cases by looking at five areas (see Table 8): code review, CI discipline, internal verification scope, domain expert testing and testing with customers.

4.4.1 Code Review

SmallOrg had a code review process that was both strict and fast. BigCorp, however, did not have a formal code review practice in place and was struggling with low quality code partly because of that.

BigCorp BigCorp did not have a formal code review practice in place. There was no clear reason why code review was not practiced. Due to the lack of code review and a high number of new and inexperienced developers, there were signs of bad code quality.

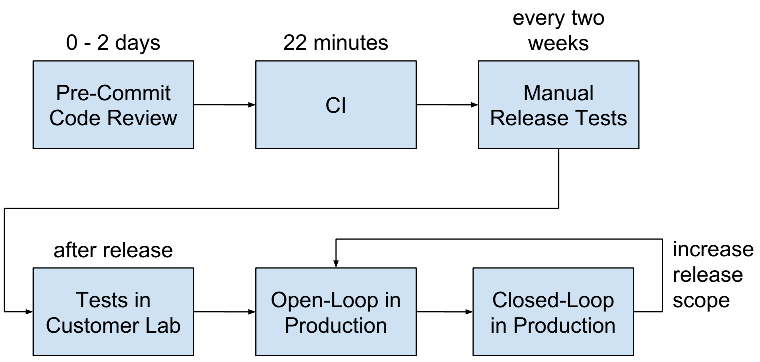

Fig. 5 Case SmallOrg release engineering process

Table 8 Comparison of release engineering practices in the case organizations

Practice	Case BigCorp	Case SmallOrg
Code review	No systematic code review	All code is reviewed through pull requests
CI discipline	Slow CI, broken most of the time	Fast CI, unbroken most of the time
Internal verification scope	Strict definition of done, dedicated testers, integration and non-functional testing in test labs	No definition of done, no dedicated testers, low unit test coverage, lack of integration and non-functional testing
Domain expert testing	No domain experts in the development organization	Domain experts test every feature
Testing with customers	Pilots with existing customers performed by other organization than development, low feedback from production use	Extensive testing in customer labs and production environment by domain experts and service team members, fast feedback to development

> In practice, I think we do not do any code review. There has been much talk about it, but it is so laborious, so no. —BigCorp Manager

> In [BigCorp product] for instance, there was, maybe another extreme that we have too many developers [...] and a lot of them were quite, fresh. And, I think that partly because of that we have quite a lot of, bad code, which was not reviewed, by more experienced guys. —BigCorp Manager

SmallOrg SmallOrg had a well functioning code review process. Developers worked in feature branches during development. When a piece of work was thought to be ready, a pull request was opened and a lead developer and a few other developers were assigned to it. The assignees reviewed the changes, gave critical feedback and suggested whether or not the changes should be accepted to the product. The developer could improve the changes in the branch based on the feedback. The branch was also continuously tested by the CI system and it would not be integrated to the product if all the tests would not pass.

> Pull request is to merge from the current branch to the main branch. At that time, the code reviewers, watch it and they comment and they disapprove if they find something wrong. Until it gets fixed they do not approve it, and there are some merge privileges only for a few people here. —SmallOrg Dev Team

The code review process had multiple benefits for SmallOrg. First, all code changes were reviewed by experienced lead developers, which helped onboarding new developers and prevented low quality code from entering the product. Second, the changes were not integrated into the product until the code review was completed and the CI build would pass for the feature branch. Thus, the product build was green almost all the time, as build-breaking changes would not be accepted for integration.

Table 9 Comparison of CI metrics in the case organizations

Metric (mean)	Case BigCorp	Case SmallOrg
Green builds / week	1.6 (min–max: 0–10)	24.3 (min–max: 8–44)
Build success rate	1.9% (weekly min–max: 0–14%)	76.7% (weekly min–max: 38–97%)
CI execution time	2.7 h (weekly min–max: 0.23–5.0 h)	22 min (weekly min–max: 16–41 min)

BigCorp metrics include only the product CI pipeline (see Fig. 4), the actual success rate and execution time are longer when including the component CI pipelines. See Table 5 for data description

4.4.2 Continuous Integration Discipline

BigCorp did not have disciplined CI usage, as the build was slow and often broken. SmallOrg, on the other hand, had achieved a disciplined use of CI, and benefited more from it.

BigCorp BigCorp had structured the CI pipeline according to the architecture of the product (see Fig. 4). First, the individual components of the product were build and tested. Successful component CI pipelines would feed into the product CI pipeline, where the components would be integrated. On a component-level, builds might have passed, but the product build was often failing.

> Everybody says okay my module is working and I am green. [...] But [when] you put them together, it's not green. —BigCorp Architect

In addition to failing, the build took multiple hours to execute (see Table 9). Thus, fixing a failing build was hard, because one had to wait multiple hours to see if the build would be functional again after fixes. The slow build also delayed the feedback on the changes made by the developers.

A major reason for the failing build was that changes were integrated into the product before the CI was run. One could integrate non-functional changes and others could start building on top of those changes even when the changes had not been verified by the CI. Learning from SmallOrg, the build could have been improved by not allowing changes that would break the build.

> In [BigCorp product] it was a problem with how it builds that if somebody breaks the, build and the whole team suffers. —BigCorp Architect

Other reasons for the failing builds were fragile third party components in the product architecture and the organizational distribution. The third party components were claimed to make the build fail randomly. Distribution of the organization made communicating changes more difficult, and conflicting changes were made because of that.

SmallOrg SmallOrg's product was continuously integrated in a single phase (see Fig. 5). The phase was relatively fast, taking approximately 22 min (see Table 9), and was kept unbroken most of the time. The build time was also monitored and if it started to take too long time, it would be optimized. SmallOrg had reduced the build time by building only components that had changed after previous run and by running tests in parallel.

> [Note: the quotation explains the situation during the interviews, not before the acquisition, although the situation is similar based on the CI data.] We've actually been pretty good about keeping [the main branch] working. So we got like last 30 days.. 84% successful so it's pretty good. I just made a change around there that reduced the duration of the build because it was getting up to like 29–30 min, so I made a change that got it back down to 17 or 18 min, which is nice.
>
> —SmallOrg Dev Team

The CI was run for all the feature branches in addition to the main branch. Running the CI in branches allowed developers to see the CI results before integrating the branch into the product. Thus, build failures rarely affected other developers' work.

> Our [CI] system will build every single branch and run all of the checked in tests. And deploy the system. Then it runs some very light integration tests with new data. Then you can see, if I merge this in, I'm not gonna break the system. Which would happen sometimes. —SmallOrg Dev Team

4.4.3 Internal Verification Scope

BigCorp had high internal quality standards and put a lot of resources on verifying the quality before giving the product to the customers. SmallOrg had only limited internal verification scope, somewhat due to the lack of resources.

BigCorp BigCorp had put plenty of resources to verify the quality of their product before giving it to the customers. First, the organization had dedicated testers that created automated test cases but also performed manual testing on the product. Second, the organization had a definition of done that included test coverage criteria for automated tests. Third, the organization had invested in production-like laboratory environments that allowed in-depth functional and non-functional testing of the product.

> In [BigCorp product] I think our [definition of done] was quite strict. That you should [create] all the unit tests or the automated tests if you have [made UI changes].
>
> —BigCorp Architect

> So in [BigCorp] we have these labs, which we call [environment] verification or system verification. But that requires quite serious.. upfront investment.
>
> —BigCorp Manager

However, after certain production deployments, it became clear that even after putting a lot of resources on internal verification, there would be issues that would reveal themselves only in the production system. Thus, even a rigorous internal verification can be insufficient if the production environment cannot be simulated totally.

> We have a customer that is really taking the product into use. And it shows that we have slipped in the test system verification a lot. That we have not tried to test the system against the load that is present in a customer environment.
>
> —BigCorp Architect

SmallOrg The internal verification scope of SmallOrg was limited compared to BigOrg's. First, the test automation coverage of SmallOrg was lower than BigCorp's. The unit test code coverage varied from 20 to 50% between different modules of the product. The reason for low test code coverage was that early in the development, SmallOrg did not focus on unit testing at all and they had increased the coverage only afterwards. Instead of aiming at a high test coverage, SmallOrg had created smoke tests which allowed fast verification that nothing was critically broken.

> I was quite amazed that how low [SmallOrg's] test automation is. They are having, some automated tests but, by far not the level we are requiring at [BigCorp].
> —BigCorp Architect

> I: How much do you rely on manual testing?
> R: Fairly heavily. —SmallOrg Dev Team

> The smoke tests essentially just run most modules and verify that there's no errors and, that does actually catch quite a bit. Surprisingly. —SmallOrg Dev Team

Second, the CI included some light integration tests, but there was no integration to the real production-like environments. The reason for this was that the environments are expensive and SmallOrg lacked the resources to acquire such test systems. Nevertheless, SmallOrg had acquired test data from customer systems, so at least that data could be used for partially simulating the production environments. However, that data was used only in the manual release tests, not in the CI integration tests.

> Yeah the integration tests so we.. start a virtual machine on AWS and then we install our software. And then we run tests on that software. So it's.. you can test things like database access and things like that but there isn't usually [production-like data].
> —SmallOrg Dev Team

> Since the earliest days, ever since we connected into the, first time into the [SmallOrg customer environment], we have a snapshot of actual configuration, actual data from the [SmallOrg customer environment]. —SmallOrg Domain Expert

Third, there were no performance, high availability or scalability tests. These would again require investments that SmallOrg did not have resources for. Instead, when performance issues were discovered during other verification activities, they were fixed reactively.

> Then another big, problem in my opinion is that they did not perform at all the performance testing, performance and capacity testing. When you asked them for instance, how big [environment] the [SmallOrg product] [...] can handle. So they [don't even know] the answer. —BigCorp Architect

Fourth, SmallOrg did not have resources for dedicated testing personnel and relied on developers and domain experts for manual release testing that took a few days. Some parts of the manual testing were still semi-automated; they were executed automatically but required a human to verify the results. Finally, SmallOrg did not have any formal definition of done in place. Thus, developers created automated test cases when they considered it useful.

> [SmallOrg's] definition of done, that I mentioned earlier that they were, kind of it was not so strict so, they were able to cut some corners, here and there. [...] So, you implement the code, the code is reviewed, it is done. And test automation I think, used to be more or less, voluntary. So if you want to do it, you do it. If not, you don't do it, in [SmallOrg product]. —BigCorp Architect

4.4.4 Domain Expert Testing

BigCorp had dedicated testing personnel, but they did not have any domain expertise. SmallOrg had domain experts who both specified and tested the product functionality.

BigCorp BigCorp had dedicated testing personnel for internal verification. They could verify that the software worked as specified. However, they lacked domain expertise and could not verify the software from the viewpoint of the users of the software.

> In [BigCorp product] we have a set of people who only do test. They don't do development at all. —BigCorp Architect

SmallOrg SmallOrg had domain experts that both specified features and could verify them from the viewpoint of the users of the product. Domain experts also leveraged collected data from the customer and could simulate some parts of the production environments internally.

> A software engineer develops a prototype that they then hand off to the [domain expert who] will do their own validation, their own testing, [to] see if the expected functionality is there. [...] A lot of times [...] software developer [does not understand] fully the requirements or [...] what needs to be implemented.
> —SmallOrg Domain Expert

4.4.5 Testing with Customers

BigCorp had little interaction with the customers. Instead, they relied on production-like test environments that they used internally. SmallOrg did not have resources for such internal test environments, and was forced to perform additional verification at the customer premises.

BigCorp BigCorp relied only on internal verification with production-like test environments. They did not perform any testing with the customers. After finishing a release internally, the release would be sent to specific existing customers for pilots of new the functionality (see Fig. 4). The pilot customers were those who had requested the new features in the release and were willing to do piloting. However, the pilots were performed by a separate delivery organization and did not involve product development.

SmallOrg SmallOrg relied on multiple ways to test the product with its friendly customer. First, the customer had a test laboratory where SmallOrg could for the first time integrate with real production-like environments. Second, when the software was actually deployed to production-use, SmallOrg used open-loop, closed-loop and progressive deployments, which are introduced after the next paragraph.

> Another thing is, another way that we do testing, it's the monitoring of the production server. So it's, that becomes a field testing. Sometimes, because the product nature itself, it is very costly to build a test lab to mimic the production [environment]. Some of the test cases you really cannot cover in the test lab. What you can do, it's monitoring production [environment] and trying to detect issues, as it goes. That's another way to compromise, [complement] our testing effort. —SmallOrg Architect

Without own production-like test environments, SmallOrg had to verify the integration in the customer's laboratory together with the customer's domain experts. Thus, integration testing was sufficiently fulfilled only after the product had passed internal verification and passed to customer testing.

> It was more, the integration at the beginning was the most challenging part. Once it's integrated with one system, because it's the same [integration point] everywhere it's just a matter of tweaking a little pieces, when it's a different software on a lease. [...] But once those interfaces were developed then it was very straightforward.
> —SmallOrg Service Team

Open-loop verification means running new functionality in a production environment, but while the functionality receives data from the production environment, all the effects it would have to the production environment are turned off. Thus, its behavior can be safely observed in a real environment. In addition, the service team always had a rollback plan if something would go wrong. After verifying that the product worked in open-loop mode, it was switched to closed-loop mode where the product had an actual effect on the production system.

> [After internal verification we] run the module [in the production environment] first and open loop, then when they're happy with the output, the recommendations from the module then to proceed with closed loop. —SmallOrg Domain Expert

> The, open-loop, again, so, depends on the scenario. If it's a new customer, we're doing a [proof of concept], we always do open-loop and so we can review the changes with the [customer]. That gives them a better sense for what the tool is doing, helps to understand it, and also more confidence that, OK, it's making the right types of decisions. And then, closed-loop obviously, as we get more, field experience with modules then, we'll become a lot more comfortable [...] that there's no, unexpected behavior or surprises. —SmallOrg Domain Expert

> We would [be] monitoring very very very closely any changes being made, making sure that, there's no, issues being caused by, [...] errors in the software. [...] So that's a process that can take some time to complete. —SmallOrg Domain Expert

Progressive deployments meant that a release was not applied straight away to the whole customer's production environment. Instead, the release was first verified in a small part of the environment, and after successful verifications the release scope was progressively increased to cover the whole production environment.

> We didn't deploy to even all the servers at the same time. So, normally we would, have two servers, [one for different technologies], where we did the first, upgrade, and we would test it. Always we find some kind of issue that gets solved and a new build is made before we start deploying [to] the other servers. So it's like a soaking period of two three days. Once it's stable enough, now we go ahead and upgrade all of them. So, it's pretty time consuming. —SmallOrg Service Team

Due to the friendly relationship and SLA with the customer, SmallOrg could perform and monitor the releases directly. Thus, any problems that were encountered in the production system could be fixed immediately. However, the cost of that was the strict SLA (see Section 4.3.1).

> And they had managed to develop, a product which, was allowed to be kind of tested, inside the [SmallOrg customer], environment. But the price for that was that the, level of service, they, kind of agreed with [SmallOrg customer] was a really crazy one. [...]

> If a critical, issue came up, related to [SmallOrg product], the, response time, from, [SmallOrg] was 5 min. —BigCorp Manager

4.5 Outcomes of the Release Engineering Practices

In this section, we investigate what kind of outcomes the case organizations achieved with the release engineering practices. We summarize three outcomes (Table 10): feedback from the customers, the number of defect reports and quality issues with new customers.

4.5.1 Feedback from the Customers

The amount and granularity of customer feedback received by the BigCorp product development unit was low. In contrast, SmallOrg received direct feedback from customers and from the production-use of the product and could react fast to the customer requests.

BigCorp Although the customer service, sales and delivery organizations were in direct contact with the customers, BigCorp did not receive much feedback about the product use. The only channel for feedback was defect reports that came through the customer service organization. However, BigCorp could not verify whether the lack of defect reports from modules meant that the module quality was good, or simply that the module was not used by the customers.

> Because we have a long distance to the customers, the developers do not have information on what modules the customers really use. The only way to deduce that is to see which modules generate defect reports from the customers.
> —BigCorp Manager

SmallOrg SmallOrg received direct feedback from its customer. Since the product was released every two weeks, SmallOrg quickly received feedback on recent changes, both by seeing it running in the production environment, and by discussing with the customer. The service team and the domain experts worked closely with both the customer and the software developers. In addition, the service team could monitor the production environment directly. Thus, SmallOrg received insights from the customer production environments and the product usage. This allowed them to learn from production issues and improve the customer value of the product.

> The customer support team, they're in charge of testing the module making sure it runs properly, before doing closed-loop execution. Sometimes we'll find problems there. We document all the issues and pass it to R&D. —SmallOrg Service Team

Table 10 Comparison of release engineering outcomes in the case organizations

Outcome	Case BigCorp	Case SmallOrg
Feedback from the customers	Low feedback from the production use	Frequent feedback from the production use
Number of defect reports	Low number of defect reports	High number of defect reports
Quality issues with new customers	Elementary programming mistakes revealed	Scalability and other issues

> We basically manage the servers ourselves. So we manage the release. And I think in [SmallOrg customer], [...] we're still the ones that do the deployment and monitor the system. —SmallOrg Dev Team

4.5.2 The Number of Defect Reports

When the BigCorp and SmallOrg products were compared by the BigCorp organization, it was noticed that the number of defect reports was higher for the SmallOrg than for the BigCorp product. This was caused by the fact that the internal verification scope was not high in SmallOrg. However, the number of defect reports was not an issue for the SmallOrg customer, because the defects were fixed quickly.

> [SmallOrg] gets over thirty defects per week from the customer. Which is a lot... We cannot deliver to fifty customers, with that kind of quality. —BigCorp Manager

4.5.3 Quality Issues with New Customers

Both BigCorp and SmallOrg had observed quality issues when deploying the product to new customers.

BigCorp Even though BigCorp paid high attention to internal verification, there were quality issues when trying to use the product with new customers. The experienced architects believed that this was caused by inexperienced new software developers and the lack of code review by more experienced developers.

> And we have revealed, from the software, so many shocking things [...] Due to, maybe inexperienced developers, it is not understood that, when there is a small error, some temporary problem in customer environment, the software just crashes. [...] And this is the problem we have with [BigCorp customer] at the moment.
> —BigCorp Architect

SmallOrg SmallOrg had quality issues during proof of concepts. This was caused by lack of internal verification, having only one customer that had used the product and not having high quality requirements for the product. When quality problems were discovered, they were fixed fast during proof of concepts.

> It was quite a large [environment] and, basically the [SmallOrg product] was, not working as expected, and it was not scaling, as it was, promised to. Then, the, normal ways of, kind of normal ways of working at [SmallOrg] was that, they were, sending back this information to the R&D and the R&D was fixing it really fast, and sending it back to the customer. —BigCorp Architect

4.6 Synthesis

In this section, we first synthesize the case findings individually, and finally combine the evidence into relationships between contextual variables.

BigCorp's release engineering practices were affected by the high quality standards and organizational structure (see Fig. 6 and Table 11). High quality requirements made CI more complex and combined with large and distributed organization, CI became undisciplined. Thus, BigCorp required additional manual verification, which slowed the release capability.

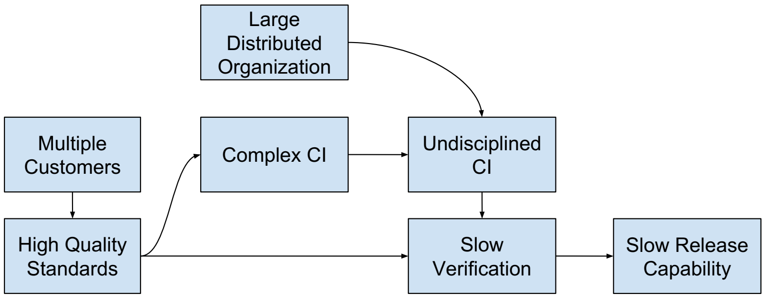

Fig. 6 Case BigCorp release engineering explained

SmallOrg's release engineering practices were affected by the lack of resources. However, being a co-located small organization, keeping the CI discipline was easier (see Fig. 7 and Table 12). The lack of resources caused lack of test automation and internal verification which was mitigated by the code review and disciplined CI practices. The lack of internal verification allowed fast release capability and gaining feedback from production. The product was verified externally in customer environments, in order to avoid critical defects. Having a friendly customer that allows external verification enabled lower internal verification.

To synthesize the two cases, we found five variables that affected release engineering practices (see Fig. 8 and Table 13): number of production environments (instances where different versions of the product are run simultaneously), number of customers, control over

Table 11 Explanations for the arrows in Fig. 6

Arrow	Explanation
Multiple customers → High quality standards	BigCorp had a large number of customers, which required high quality standards. See Sections 4.3.1 and 4.3.4.
High quality standards → Complex CI	High quality standards meant that the system had to be tested more rigorously in the CI system. See Section 4.4.2.
High quality standards → Slow verification	High quality standards also required additional testing after the CI pipeline. See Section 4.4.3.
Complex CI → Undisciplined CI	The complexity of the CI system made the CI practice more undisciplined. See Section 4.4.2.
Large distributed organization → Undisciplined CI	In addition, the distribution of the organization made the CI practice more undisciplined. See Section 4.4.2.
Undisciplined CI → Slow verification	As the CI process was not functioning well enough, additional manual verification had to be performed, which slowed down the verification. See Sections 4.4.2 and 4.4.3.
Slow verification → Slow release capability	Due to slow verification, BigCorp had slow release capability. See Fig. 4.

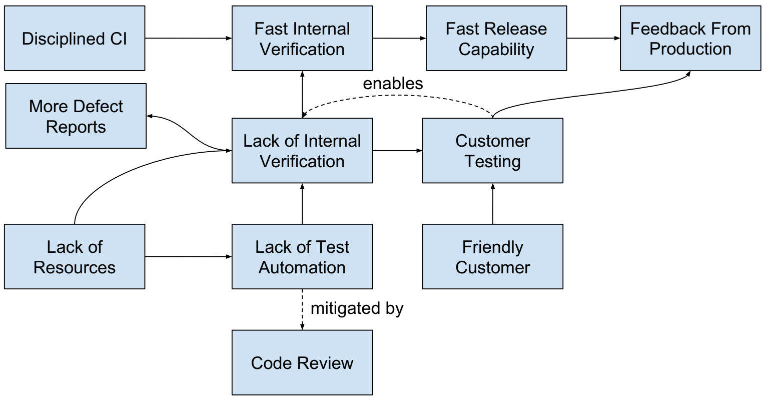

Fig. 7 Case SmallOrg release engineering explained

production, available resources and organization size and distribution. The four first variables affect the requirement for internal quality standards. The quality standards affect the complexity of the CI system. CI system complexity and organizational size and distribution affect the possibility to achieve CI discipline. Together, CI discipline and internal quality standards affect the release capability and the ability to gain feedback from production.

5 Discussion

In this section, we answer the research questions of the study, compare the results to previous studies, discuss implications to practitioners and threats to validity.

5.1 What Release Engineering Practices Had the Case Organizations Implemented?

There were five major differences between the release engineering practices implemented in the case organizations (see Table 8). First, SmallOrg used a code review practice, while BigCorp did not. Second, the CI discipline was better in SmallOrg than in BigCorp. Third, the scope of the verification performed internally by the organization was higher in BigCorp than in SmallOrg. Fourth, SmallOrg made use of domain experts when internally verifying the product, but BigCorp did not have such domain experts available. Finally, SmallOrg focused more on the external verification in the customer environment, whereas BigCorp did not perform such external verification themselves, but it was done by a separate organization in the company.

The release engineering practices of BigCorp were structurally similar to common practice (Adams and McIntosh 2016), although the achieved CI discipline was not good. Slow and failing builds are a common problem when adopting modern release engineering practices (Laukkanen et al. 2017).

The approach of SmallOrg was different. Their focus was not on the automated testing scope, but instead on code review, domain expert testing and customer testing. Similar findings have been reported in previous startup studies (Paternoster et al. 2014; Giardino et al. 2016) that identified the same characteristics: lack of overall testing, lack of

Table 12 Explanations for the arrows in Fig. 7

Arrow	Explanation
Disciplined CI → Fast internal verification	SmallOrg's Disciplined CI kept the software continuously in a good condition and allowed fast internal verification. See Section 4.4.2.
Fast internal verification → Fast release capability	Fast internal verification allowed fast release capability. See Fig. 5.
Fast release capability → Feedback from production	Frequent releases enabled getting feedback from production use. See Section 4.5.1.
Lack of internal verification → Fast internal verification	Due to SmallOrg's small internal verification scope, the internal verification was faster. See Section 4.4.3.
Lack of internal verification → More defect reports	Lack of internal verification caused more defect reports from customers. See Section 4.5.2.
Lack of internal verification → Customer testing	Due to the lack of internal verification, more testing had to be done with customers. See Section 4.4.5.
Customer testing enables lack of internal verification	Being able to test with customers reduced the need for internal verification. See Section 4.4.5.
Customer testing → Feedback from production	Through customer testing, SmallOrg gained feedback from the production use. See Section 4.4.5.
Friendly customer → Customer testing	SmallOrg's friendly customer allowed performing customer testing. See Sections 4.3.1 and 4.4.5.
Lack of resources → Lack of internal verification	Due to the lack of resources, SmallOrg's internal verification scope was smaller. See Sections 4.3.4 and 4.4.3.
Lack of resources → Lack of test automation	Lack of resources also restricted SmallOrg's amount of test automation. See Sections 4.3.4 and 4.4.3.
Lack of test automation → Lack of internal verification	Small amount of test automation meant that less testing was performed internally in SmallOrg. See Section 4.4.3.
Lack of test automation mitigated by code review	SmallOrg's functional code review practice mitigated the lack of test automation, because errors were caught by more experienced developers. See Section 4.4.1.

automated testing, manual smoke tests, user reports mitigate the lack of testing and progressive roll-outs.

Only code review was not identified as an important aspect in the earlier studies (Paternoster et al. 2014; Giardino et al. 2016), which can be explained with that the companies in the earlier studies were smaller than SmallOrg; previously studied cases had 3–20 employees, whereas SmallOrg had 50 employees. We speculate that code review becomes necessary after the development team size grows and more experienced developers need to guide the more inexperienced ones.

In general, our findings illustrate the differences in mindsets regarding the role of the customer. In the startup, the practices emphasized the capability of releasing early and often to

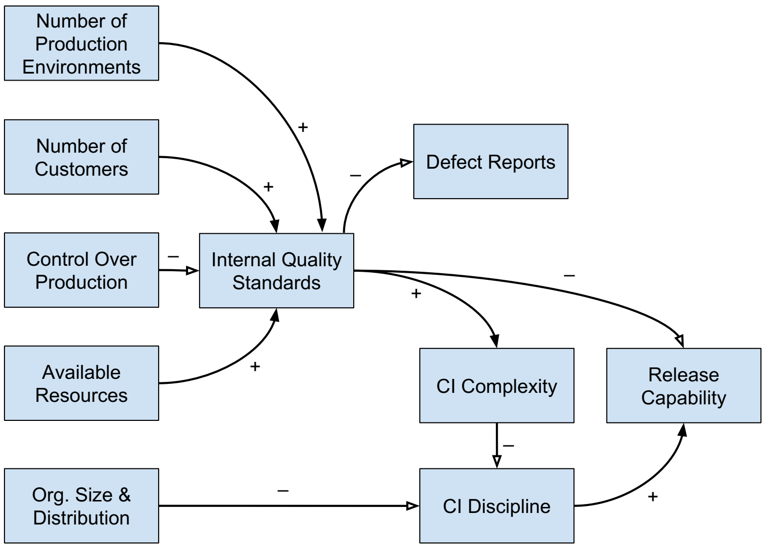

Fig. 8 Synthesis of the release engineering driving forces in the two cases

get customer feedback. The large corporation, instead, emphasized the internal verification before releasing to the customers and avoiding negative customer feedback.

5.2 What Outcomes Did the Implemented Release Engineering Practices Have for the Case Organizations?

Release engineering practices have a trade-off between confidence and velocity (Schermann et al. 2016). BigCorp achieved good release confidence on its product, but release velocity was not as good due to not being able to continuously verify the product quality. In addition, BigCorp had encountered problems when deploying their product to new customers and therefore release engineering confidence did not remove the need to fix problems in production environments. SmallOrg had better velocity, but the release confidence was not as good after internal verification. Thus, SmallOrg relied heavily on customer testing, which allowed learning from production issues.

Having the capability to release the product more frequently and performing additional testing in the customer environment allowed SmallOrg to iterate the product development more quickly than BigCorp and discover valuable features that were required in the customer environments. Generally, more flexible processes suit better situations that are complex and hard to predict (Ralph and Narros 2013).

However, release engineering outcomes were not the only factor that explained the market success of SmallOrg. A small and collaborative organization, a high level of domain expertise and tight customer collaboration seem to have played a large role in the market success, as having release engineering practices in place does not ensure that developed features are valuable. Nevertheless, the release engineering practices enabled continuous experimentation with the customer, which would not have been possible through tight communication alone. In addition, without having sufficient release engineering practices in

Table 13 Explanations for the arrows in Fig. 8

Arrow	Explanation
Number of production environments + → Internal quality standards	BigCorp had a large number of customers and production environments, which required higher internal quality standards. See Sections 4.3.1 and 4.4.3.
Number of customers + → Internal quality standards	Similarly, BigCorp's larger number of customers increased the internal quality standards. See Sections 4.3.1 and 4.4.3.
Control over production − → Internal quality standards	SmallOrg could control its friendly customer's production environment and thus did not need as high internal quality standards as BigCorp, whose control over production environments was limited. See Sections 4.3.1, 4.4.3 and 4.4.5.
Available resources + → Internal quality standards	SmallOrg did not have enough resources to have internal quality standards as high as BigCorp. See Sections 4.3.4 and 4.4.3.
Internal quality standards − → Defect reports	BigCorp had verified its product more rigorously internally than SmallOrg, and thus SmallOrg received more defect reports for its product. See Sections 4.4.3 and 4.5.2.
Internal quality standards + → CI complexity	BigCorp's internal quality standards were higher than SmallOrg's, which added complexity to the CI system. SmallOrg did not have as high internal standards and had a simple CI system. See Figs. 4 and 5 and Sections 4.4.3 and 4.4.2.
Internal quality standards − → Release capability	BigCorp's larger internal quality standards increased the time and resources needed for the release engineering process, whereas SmallOrg's lower internal quality standards allowed more frequent releases. See Figs. 4 and 5 and Section 4.4.3.
CI complexity − → CI discipline	BigCorp had more complex CI, which made it more difficult to keep CI usage disciplined. SmallOrg had a simple CI system and also good CI discipline. See Section 4.4.2.
Organization size and distribution −→ CI discipline	BigCorp's large and distributed organization structure made it more difficult to have disciplined CI. SmallOrg was small and co-located, which made it easier to keep CI discipline. See Sections 4.3.2 and 4.4.2.
CI discipline − → Release capability	SmallOrg had a good CI discipline and was able to do releases frequently, whereas BigCorp's undisciplined CI required additional release stabilization efforts, which reduced release capability. See Figs. 4 and 5 and Section 4.4.2.

place, the risk of regression defects increases and the quality threshold of customers might not be exceeded anymore.

As a more general implication of our findings, we hypothesize that *disciplined and automated CI practice and team level development quality practices are important enablers of fast deployment and release capability*. Practices, such as CI automation and code reviews, help establishing good enough low level code quality that enables sufficiently fast deployment to production environment. Fast customer feedback and domain expert validation can ensure more valuable deliveries and mitigate the lack of comprehensive automated tests, if recoverability from release failures is also high.

5.3 What Were the Reasons for Differences in Release Engineering Practices in the Case Organizations?

BigCorp's more complex situation required stringent release engineering practices. It turned to be difficult to fulfill such stringent requirements continuously, because the large and distributed organization structure made the communication between different parts of the organization more arduous. SmallOrg's situation was simpler, but its lack of resources played a significant role in determining the release engineering practices. SmallOrg had mitigated the difficulties caused by the lack of resources with close customer collaboration, which reduced the need for internal verification.

We identified that the organizational structure, size and distribution affected the communication between different parts of the organization. Communication is a key element in modern release engineering, because to keep software releasable most of the time, the whole development organization has to be able to keep the CI disciplined (Laukkanen et al. 2015). In addition, in a situation that a product cannot be fully verified internally, it is important to have working communication between different functional parts, development and deployment, or more traditionally, development and operations (DevOps) (Dyck et al. 2015).

We also identified that complexity factors, such as having multiple customers with different versions, increases the complexity of release engineering practices, too. Thus, more resources are needed for release engineering practices in more complex situations. If such resources are not available in an organization, it can be that having frequent release cycle is not possible, and the organization in question has to use other means, such as lengthy testing periods, instead of automatic verification.

Finally, we identified that while the lack of resources can prevent comprehensive internal verification, it is still possible to verify the product with the customer directly. A similar approach was used by Netscape (Yoffie and Cusumano 1999): "By downloading the beta, trying it out, and filing their complaints, customers served – sometimes unwittingly – as Netscape's virtual quality-assurance team." Customer testing can be leveraged if it can be ensured that encountered bugs do not cause any actual harm or that they can be fixed sufficiently fast. Thus, fast release capability can enable the use of customer testing.

In general, we hypothesize that *a complicated product and customer environment requires higher internal standards for definition of done and test automation coverage, but challenges the release capability*. We also hypothesize that *high test automation coverage levels are not required for fast deployment capability in the range of 1-2 weeks, but test automation becomes crucial when approaching one-day release capability*, as a similar finding was made in an earlier study (Mäkinen et al. 2016).

Table 14 Pros and cons of the release engineering practices in the case organizations

	Case BigCorp	Case SmallOrg
Pros	Internal verification, capable to serve multiple customers	Customer verification, release velocity, capable to serve a single customer well
Cons	Release velocity, new customer satisfaction	Internal verification

5.4 Implications for Practitioners

For practitioners, it is important to understand the pros and cons of the release engineering practices of the two organizational contexts (see Table 14). BigCorp was more successful in internal verification and was capable of serving multiple customers. SmallOrg had a good customer verification processes, better release velocity and was capable to serve a single customer excellently. As cons, BigCorp had lower release velocity and could not satisfy potential new customers in proof of concepts. SmallOrg had less mature internal verification processes, which caused more defects to slip to the customers.

To mitigate the cons of the organizational contexts, some strategies can be applied. First, for BigCorp, release velocity could be increased by automating the release engineering processes. Some parts of the process had been automated, but the automation was not well-functioning according to the CI metrics (see Table 9). Second, BigCorp's customer satisfaction could be improved by mimicking the SmallOrg strategy to have the development team work directly with the customer service team. Olsson et al. (Olsson et al. 2014) describe how large-scale software development organizations can achieve both scale and responsiveness by implementing customer-specific software development teams. Finally, SmallOrg's internal verification processes could be improved by providing more resources for the organization. After the acquisition, BigCorp started to increase the test automation scope for the SmallOrg product in order to be able to support more customers.

As more general implications from this study we present the following lessons learned for small start-ups and large mature organizations. First, when beginning the building of release engineering practices it is important to start with disciplined and well functioning CI practices and deployment automation to ensure fast release capability. For growing start-ups, fast customer feedback is more achievable and valuable than comprehensive internal verification. Because there will always be customer problems, it is more important to build fast feedback loops and recovery capability than long internal verification phases. The level and coverage of the internal verification should be increased gradually as the environment and product becomes more complex, continuously maintaining the fast release capability.

Second, for large mature organizations the key implications are improving the discipline and cycle time of the CI and deployment practices. The rigorous team level review and testing practices are crucial to build consistent continuous delivery capability. The delivery pace is dependent on the length of the feedback loops that can be shortened by bringing the separated roles closer together, improving collaboration and breaking the silos. Testing can be moved closer to the production by phased testing and open / closed loop testing and other modern release engineering practices, such as dark launches (Feitelson et al. 2013) and feature toggles (Rahman et al. 2016). Working closely with selected pilot customers would enable fast customer feedback to develop right features and value for customers. On the other hand, a large number of customers requires sufficient internal verification and can be supported by organizing the development into customer specific teams (Olsson et al. 2014).

5.5 Threats to Validity

In this section, we use the classification by Runeson and Höst (2009) to discuss the threats to validity of our results.

Construct validity addresses whether the constructs that are operationalized represent what the researcher has in mind and what is investigated according to the research questions. In this study, we used constructs such as organizational context, product context and release engineering. As the data collection and analysis were performed inductively, it is possible that the used constructs are not the same as used in other research. Furthermore, the constructs are not mature, as release engineering has gained research attention only recently and there has been disagreement about contextual constructs (Dybå et al. 2012).

Internal validity addresses whether the causal relations identified in a study are correct. In this study, we found several relations between release engineering constructs (see Figs. 6, 7 and 8). However, the relations are not causal in nature, but rational explanations given by the interviewees. Thus, the validity of the relations is limited, because they rely on the subjective views provided by the interviewees. In addition, we could not interview all parts of BigCorp and did not employ other data collection methods than interviews and CI data analysis. Using other methods, such as observation, would have strengthened the results. Finally, we performed the interviews after the acquisition and the interviewees might have remembered the situation before the acquisition differently than it actually was, since the situation had changed considerably after the acquisition.

External validity addresses whether the results are generalizable to other cases than the ones that were studied. Some of the results are specific only to the case organizations, e.g., we could not analytically generalize, why code review was used in SmallOrg and not in BigCorp. Furthermore, the results are generalizable only to similar situations, considering the product context and organizational contexts.

Reliability addresses whether the data collection and analysis could be conducted by other researchers. The procedures used in this study contained interpretative elements. The interviews were flexible and allowed interviewers to ask questions based on their intuition. The data analysis was performed by the first author and relied on interpretations, although the interpretations were reviewed by other authors too. Nevertheless, the interpretative elements reduce the reliability of the results.

6 Conclusions

Modern release engineering practices have been known to the industry for some years (Humble and Farley 2010) and have recently gained academic attention (Adams and McIntosh 2016). However, there is a clear gap of empirical research concerning how applicable the modern release engineering practices are in different contexts (Laukkanen et al. 2017). In this study, we have addressed this gap by comparing the release engineering practices in two differing organizations that were developing similar software products.

We found that the number of production environments and customers, control over the production and available resources affected the release engineering practices through higher quality standards (see Fig. 8). In addition, the lack of resources substantially affected the test automation and internal verification scope, which was mitigated with customer testing practices in the smaller organization (see Fig. 7). Furthermore, with proper techniques in place, such as open- and closed-loop testing and progressive deployments, customer testing could be performed safely even in the context where production defects are harmful. Code review and CI discipline were achievable in a low-resource context, because CI requirements were not set too high. Otherwise, CI discipline was difficult to achieve even in a high-resource context (see Table 9).

Faster release capability allowed faster customer feedback, although the number of defect reports increased (see Table 10). However, the lack of defect reports did not necessarily mean that the software product was of good quality. Indeed, trying to achieve good release confidence with just internal verification might not be possible in all contexts, and verification in production environment was required for detecting certain defects. The organization with faster release capability could fix these production-only defects faster. Fast release capability did not explain market success directly, but it was a critical enabler for frequent customer experimentation while still assuring the quality. Not all customers were ready for fast release cycles, and there was still market for high quality products with slower release cycles.

We contribute to research by introducing a revelatory comparison case study. In software engineering research, it is difficult to draw conclusions from case study research, because the product context affects the development practices substantially. In this study, we achieved to mitigate this difficulty by having two cases that build similar products. Our results can be validated in other case studies or even surveys addressing the gap of empirical research on release engineering.

We contribute to practitioners by showing that neither of the organizational contexts was perfect in all situations. The large mature organization was better in serving multiple customers, while the small startup was better serving a few customers with exceptional customer satisfaction. Employing the strategies of startups can help larger organizations to improve release engineering velocity and customer responsiveness, while startups can benefit from additional resources to release engineering processes.

As future work, the recognized constructs and their relations (see Fig. 8) can be verified and extended in future case studies and surveys. For example, the following research questions could be pursued:

- How does organizational size and distribution affect continuous integration discipline?
- How does continuous integration discipline affect release capability?
- How could internal quality standards be measured?

In addition, it would be interesting to study how the release engineering practices evolve after the acquisition.

Acknowledgments The authors would like to thank the case companies and in particular our interviewees. This work has been supported by TEKES as part of the Need for Speed research program of DIMECC (Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

References

Adams B, McIntosh S (2016) Modern release engineering in a nutshell—why researchers should care. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 5, pp 78–90, https://doi.org/10.1109/SANER.2016.108

Berczuk SP, Appleton B (2002) Software configuration management patterns: effective teamwork, practical integration. Addison-Wesley Longman Publishing Co. Inc, New York

Bjarnason E, Wnuk K, Regnell B (2012) Are you biting off more than you can chew? A case study on causes and effects of overscoping in large-scale software engineering. Inf Softw Technol 54(10):1107–1124. https://doi.org/10.1016/j.infsof.2012.04.006

Cooper RG (1990) Stage-gate systems: a new tool for managing new products. Bus Horiz 33(3):44–54

Debbiche A, Dienér M, Berntsson Svensson R (2014) Challenges when adopting continuous integration: a case study. In: Product-focused software process improvement. Springer International Publishing, Lecture Notes in Computer Science, vol 8892, pp 17–32

Desikan S (2006) Software testing: principles and practice. Pearson Education India

Dikert K, Paasivaara M, Lassenius C (2016) Challenges and success factors for large-scale agile transformations: a systematic literature review. J Syst Softw 119:87–108. https://doi.org/10.1016/j.jss.2016.06.013

Dybå T, Sjøberg DI, Cruzes DS (2012) What works for whom, where, when, and why?: On the role of context in empirical software engineering. In: Proceedings of the ACM-IEEE international symposium on empirical software engineering and measurement. New York, pp 19–28. https://doi.org/10.1145/2372251.2372256

Dyck A, Penners R, Lichter H (2015) Towards definitions for release engineering and DevOps. In: Proceedings of the third international workshop on release engineering. RELENG '15. IEEE Press, Piscataway, pp 3–3

Fauzi SSM, Bannerman PL, Staples M (2010) Software configuration management in global software development: a systematic map. In: Asia Pacific software engineering conference, pp 404–413, https://doi.org/10.1109/APSEC.2010.53

Feitelson D, Frachtenberg E, Beck K (2013) Development and deployment at Facebook. IEEE Internet Comput 17(4):8–17

Fitz T (2009) Continuous deployment. http://timothyfitz.com/2009/02/08/continuous-deployment/

Fowler M (2006) Continuous integration. http://martinfowler.com/articles/continuousIntegration.html

Freeman J, Engel JS (2007) Models of innovation: startups and mature corporations. Calif Manag Rev 50(1):94–119

Giardino C, Paternoster N, Unterkalmsteiner M, Gorschek T, Abrahamsson P (2016) Software development in startup companies: the greenfield startup model. IEEE Trans Softw Eng 42(6):585–604. https://doi.org/10.1109/TSE.2015.2509970

Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation, 1st edn. Addison-Wesley Professional, Upper Saddle River

Kruchten P (2013) Contextualizing agile software development. J Softw: Evol Process 25(4):351–361. https://doi.org/10.1002/smr.572

Laukkanen E, Paasivaara M, Arvonen T (2015) Stakeholder perceptions of the adoption of continuous integration—a case study. In: Agile conference. Washington, DC, vol 2015, pp 11–20

Laukkanen E, Lehtinen TO, Itkonen J, Paasivaara M, Lassenius C (2016) Bottom-up adoption of continuous delivery in a stage-gate managed software organization. In: Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement. ESEM '16, pp 45:1–45:10. ACM, New York, https://doi.org/10.1145/2961111.2962608

Laukkanen E, Itkonen J, Lassenius C (2017) Problems, causes and solutions when adopting continuous delivery—a systematic literature review. Inf Softw Technol 82:55–79. https://doi.org/10.1016/j.infsof.2016.10.001

Lehtinen TOA, Virtanen R, Viljanen JO, Mäntylä MV, Lassenius C (2014) A tool supporting root cause analysis for synchronous retrospectives in distributed software teams. Inf Softw Technol 56(4):408–437. https://doi.org/10.1016/j.infsof.2014.01.004

Mäkinen S, Leppänen M, Kilamo T, Mattila AL, Laukkanen E, Pagels M, Männistö T (2016) Improving the delivery cycle: a multiple-case study of the toolchains in finnish software intensive enterprises. Inf Softw Technol 80:175–194

Michlmayr M (2007) Quality improvement in volunteer free and open source software projects. PhD thesis

Neely S, Stolt S (2013) Continuous delivery? Easy! Just Change everything (well, maybe it is not that easy). In: Proceedings of the Agile Conference. AGILE '13, pp 121–128. IEEE Computer Society, Washington, DC, p 2013, https://doi.org/10.1109/AGILE.2013.17

Olsson H, Sandberg A, Bosch J, Alahyari H (2014) Scale and responsiveness in large-scale software development. IEEE Softw 31(5):87–93. https://doi.org/10.1109/MS.2013.139

Paternoster N, Giardino C, Unterkalmsteiner M, Gorschek T, Abrahamsson P (2014) Software development in startup companies: a systematic mapping study. Inf Softw Technol 56(10):1200–1218. https://doi.org/10.1016/j.infsof.2014.04.014

Patton MQ (2002) Qualitative research & evaluation methods, 3rd edn. SAGE Publications, published. Hardcover

Rahman MT, Rigby PC (2015) Release stabilization on Linux and Chrome. IEEE Softw 32(2):81–88

Rahman AAU, Helms E, Williams L, Parnin C (2015) Synthesizing continuous deployment practices used in software development. In: Agile conference (AGILE), vol 2015, pp 1–10. https://doi.org/10.1109/Agile.2015.12

Rahman MT, Querel LP, Rigby PC, Adams B (2016) Feature toggles: practitioner practices and a case study. In: Proceedings of the 13th international conference on mining software repositories. ACM Press, pp 201–211, https://doi.org/10.1145/2901739.2901745

Ralph P, Narros JE (2013) Complexity. In: PACIS, p 154

Roche J (2013) Adopting DevOps practices in quality assurance. Commun ACM 56(11):38–43. https://doi.org/10.1145/2524713.2524721

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empi. Softw Eng 14(2):131–164. https://doi.org/10.1007/s10664-008-9102-8

Savor T, Douglas M, Gentili M, Williams L, Beck K, Stumm M (2016) Continuous deployment at Facebook and OANDA. In: Proceedings of the 38th international conference on software engineering companion. ACM Press, pp 21–30, https://doi.org/10.1145/2889160.2889223

Schermann G, Cito J, Leitner P, Gall HC (2016) Towards quality gates in continuous delivery and deployment. In: 2016 IEEE 24th international conference on program comprehension (ICPC), pp 1–4, https://doi.org/10.1109/ICPC.2016.7503737

Sekitoleko N, Evbota F, Knauss E, Sandberg A, Chaudron M, Olsson HH (2014) Technical dependency challenges in large-scale agile software development. In: Cantone G, Marchesi M (eds) Agile processes in software engineering and extreme programming, no. 179 in Lecture Notes in Business Information Processing. Springer International Publishing, pp 46–61. https://doi.org/10.1007/978-3-319-06862-6_4

van der Hoek A, Hall RS, Heimbigner D, Wolf AL (1997) Software release management. In: Proceedings of the 6th European software engineering conference held jointly with the 5th ACM SIGSOFT international symposium on foundations of software engineering. ESEC '97/FSE-5, pp 159–175. Springer Inc., New York, https://doi.org/10.1145/267895.267909

van Waardenburg G, van Vliet H (2013) When agile meets the enterprise. Inf Softw Technol 55(12):2154–2171. https://doi.org/10.1016/j.infsof.2013.07.012

Wagstrom P, Datta S (2014) Does latitude hurt while longitude kills? Geographical and temporal separation in a large scale software development project. In: Proceedings of the 36th international conference on software engineering. ICSE 2014. ACM, New York, pp 199–210. https://doi.org/10.1145/2568225.2568279

Wright HK (2012) Release engineering processes, their faults and failures. PhD thesis, The University of Texas at Austin

Yin RK (1994) Case study research: design and methods, 2nd edn. Sage, Newbury Park

Yoffie DB, Cusumano MA (1999) Judo strategy: the competitive dynamics of Internet time. Harv Bus Rev 77:70–82

Eero Laukkanen is a Software Designer at Solita, a digital business consulting and services company. He works in a team with software designers, user experience designers and data scientists to create new digital services that augment human capabilities with artificial intelligence. He is interested in modern release engineering and digital service design. He has a D.Sc. degree from Aalto University.

Maria Paasivaara is an Associate Professor at the IT University of Copenhagen and an Adjunct Professor at Aalto University. Her research interests include software engineering processes and practices, continuous software engineering, agile software development, large-scale agile, DevOps, software project management, global software engineering and software engineering educational research. She performs empirical research in close collaboration with industrial and academic partners and aims at solving real-world problems that are important to the software industry. She has a D.Sc. degree from Helsinki University of Technology.

Juha Itkonen worked as a post doctoral researcher at Aalto University. His research focuses on empirical work in industrial context on the topics of exploratory software testing, continuous software engineering practices, quality assurance in large-scale agile context, and human issues in software engineering. He has a D.Sc. degree in software engineering from Aalto University.

Casper Lassenius is an Associate Professor at Aalto University. His research is in the field of empirical software engineering, with recent interests including agile software development in the small and large, continuous software engineering, DevOps, quality assurance, and global software engineering. He has a D.Sc. degree from Helsinki University of Technology.