Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study

Pilar Rodríguez[a,*], Alireza Haghighatkhah[a], Lucy Ellen Lwakatare[a], Susanna Teppola[b], Tanja Suomalainen[b], Juho Eskeli[b], Teemu Karvonen[a], Pasi Kuvaja[a], June M. Verner[c], Markku Oivo[a]

[a]*Department of Information Processing Sciences. Box 3000, 90014 University of Oulu, Finland.*
[b]*VTT Technical Research Centre of Finland P.O. Box 1100, FI-90571 Oulu, Finland.*
[c]*Keele University, UK.*

Abstract

BACKGROUND - The software intensive industry is moving towards the adoption of a value-driven and adaptive real-time business paradigm. The traditional view of software as an item that evolves through releases every few months is being replaced by the continuous evolution of software functionality. OBJECTIVE - This study aims to classify and analyse the literature related to continuous deployment in the software domain in order to scope the phenomenon, provide an overview of the state-of-the-art, investigate the scientific evidence in the reported results and identify areas suitable for further research. METHOD - We conducted a systematic mapping study and classified the continuous deployment literature. The benefits and challenges related to continuous deployment were also analysed. RESULTS - The systematic mapping study includes 50 primary studies published between 2001 and 2014. An in-depth analysis of the primary studies revealed ten recurrent themes that characterize continuous deployment and provide researchers with directions for future work. In addition, a set of benefits and challenges of which practitioners may take advantage were identified. CONCLUSION - Overall, although the topic area is very promising, it is still in its infancy, thus offering a plethora of new opportunities for both researchers and software intensive companies.

Keywords: continuous deployment, continuous delivery, rapid release, software development, continuous software engineering, software product, software intensive system, software service, systematic mapping study, secondary study, literature survey

1. Introduction

The software intensive industry is evolving towards a value-driven and adaptive real-time business paradigm [39]. *The Age of Information* [19], which is strongly based on the *Internet speed-of-things*, has shaped a digital economy and a knowledge-based society. Digital resources are constantly available for everyone, information flows are accelerated and individuals can explore their personal needs more easily. Consequently, fast-changing and unpredictable markets have shifted the competitive software development landscape. The traditional view of software as a static item that can be bought and owned is giving way to software services in which customers expect a continuous evolution of product functionality that provides additional value [14]. These market features enable new business opportunities. However, they also exert pressure to develop *dynamic capabilities* [28]. To maintain their competitive advantage, software intensive companies need to deliver valuable product features to customers considerably faster than before, if not near to real-time, while embracing business changes and pursuing economic efficiency.

Agile software development (ASD) emerged in 2001 [6] as a ground breaking foundation for new software development processes. Iterative development, continuous integration and short feedback cycles were advocated as a replacement for the traditional engineering stage-gate models. The ultimate aim of ASD was to improve the organization's capability to adapt to market fluctuations and customer needs. Although ASD was initially considered a fad, and caused some controversies [11], it became progressively mainstream. Thus, software practitioners have increasingly adopted ASD [69], and the research in the area has become well-established [25].

A recent evolutionary step from agile and lean software development is rapid and continuous software engineering. Rapid and continuous software engineering refers to the *'organizational capability to develop, release and learn from software in rapid parallel cycles, such as hours, days or very few weeks'* (ICSE 2014, http://continuous-se.org/). ASD is extended to approaches where the step between development and deployment is minimalized in order to deploy code immediately to production environment for customers to use. Continuous deployment (CD) is the term used to refer to this phenomenon [36, 61, 32, 39, 21]. Although the concept of deploying software to customers as soon as new code is developed is not new and is based on ASD principles, CD extends ASD by moving from cyclic to continuous value delivery. This evolution requires not only agile processes at the team level but also integration of the complete R&D organization, parallelization and

*Corresponding author. Tel.: +358 40 1602 179.
E-mail address: pilar.rodriguez@oulu.fi (P. Rodríguez).

automation of processes that are sequential in the value chain and constant customer feedback.

Leading organizations, such as Facebook, Microsoft and IBM, have provided examples of CD implementation [21], which has led to the emergence of studies related to CD in the scientific literature (e.g. [61, 24, 31, 47]). Mäntylä et al. [52] recently published a semi-systematic literature review as part of their research on rapid releases and testing. Although their findings are significant and shed light on the grey area of CD, the CD research field remains dispersed among different research areas and a structured understanding of the main factors that characterize CD is not provided. Therefore, the goal of this study is to identify the state of the art of the phenomenon of CD in the context of software development. This systematic mapping study [44] aims at the following:

1. To establish the body of knowledge of CD by identifying and categorizing the available research on the topic
2. To assess the quality of the existing research in terms of industrial relevance and research rigour
3. To identify the most relevant articles in the field of CD
4. To determine the underlying factors that characterize CD as both a concept and a phenomenon
5. To provide baselines to assist with further research

This study combines the process of a systematic literature review (SLR), as established by Kitchenham and Charters [43], with the mapping study, as suggested by Petersen et al. [66]. We wish to present a a logical organization of the CD literature in order to provide researchers with a structured body of knowledge on which to base their studies. We also want to furnish practitioners with the main factors they should consider when deciding whether or not to migrate to CD, the benefits that they can expect, as well as the potential risks and challenges they might face with CD.

The remainder of this paper is organized as follows: background and related work are presented in Section 2. Section 3 describes the research methodology, including a discussion on threats to validity and countermeasures taken to minimize their effects. In Section 4, we present the results of the mapping study. Sections 5 and 6 provide an analysis of the factors, benefits and challenges characterizing CD. Opportunities for future research are discussed in Section 7. Section 8 presents a comparison of our findings with related work and, concretely, to the semi-systematic literature review conducted by Mäntylä et al. [52]. Finally, we present our conclusions in Section 9.

2. Background and Related Work

The roots of CD began fifteen years ago with the formulation of the Agile Manifesto [6]. Some argued that ASD was just *old wine in new bottles* [55], in reference to the roots of ASD in iterative models, such as the spiral model [13]. Undoubtedly, today's software engineering is the result of the evolutions of previous software development models [12]. This section describes the phenomenon of CD, providing a historical view and identifying its core ideas. We then summarize previous literature reviews related to CD and justify the need for this review and its research questions.

2.1. Continuous deployment

The *Internet-speed of things* has changed the way software companies deliver value to their customers. The widespread adoption of lean principles and agile methodologies [69] has provided evidence of the need for and value of flexibility and adaptation in the current environment of software development [32]. ASD established some foundations for CD. For example, agile principles, such as *'our highest priority is to satisfy the customer through early and continuous delivery of valuable software'* and *'deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale'* make clear reference to CD. However, ASD has mainly focused on speeding up the development process at the team level through methods such as eXtreme Programming [5] and Scrum [71]. CD moves beyond the concept of ASD towards a situation in which software functionality is continuously deployed to the final customers (production environment), and where customer input is the main driver for innovation [62]. Instead of working for months on a major new release, companies limit their cycle time (i.e., the time between two subsequent releases) to a couple of weeks, days or, in some cases, even hours [52]. A plethora of evidence related to CD exists in organizational white papers and on-line blogs, where practitioners have voiced their experiences and expectations in moving to CD (e.g. Facebook[1], IBM[2], Microsoft[3], Google[4], Adobe[5], Netflix[6] and IMVU[7]). Accordingly, a body of academic literature is also emerging on this topic.

Humble and Farley [36], in their book on CD published in 2010, state that continuous delivery provides enterprises with the ability to deliver rapidly, reliably and repeatedly value to customers at low risk with minimal manual overhead. While continuous integration, which is a core ASD practice, mainly focuses on the automation of the build process (the code is built, and a set of unit tests are run when it is checked into version control repository), continuous delivery is a logical progression that automates the entire workflow simplifying the rapid release of software. The central concept in Humble and Farley's approach is a deployment pipeline that establishes an automated end-to-end process to ensure that the system works at technical level, executes fairly automated acceptance tests and lastly deploys to a production or staging environment.

[1] https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920 (accessed 09, 2015)

[2] http://www.ibm.com/developerworks/rational/library/continuous-deployment-rational-alm/ (accessed 09, 2015)

[3] http://blogs.msdn.com/b/bharry/archive/2012/06/07/announcing-continuous-deployment-to-azure-with-team-foundation-service.aspx (accessed 09, 2015)

[4] https://air.mozilla.org/continuous-delivery-at-google/ (accessed 09, 2015)

[5] http://steveblank.com/2014/01/06/15756/ (accessed 09, 2015)

[6] http://steveblank.com/2014/01/06/15756/ (accessed 09, 2015)

[7] http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/ (accessed 09, 2015)

The concepts underlying CD have also attracted the attention of researchers. Recent editions of the ICSE[8] conference (2013, 2014, 2015) have offered workshops focused on the topic such as the International Workshop on Release Engineering (RELENG), which emphasized *'the recent trend to reduce the release cycle to days or even hours'*, as well as the International Workshop on Rapid and Continuous Software Engineering (RCoSE). Drawing upon the concept of rapid and continuous software engineering, Fitzgerald and Stol [32] propose a conceptual model that presents a set of continuous activities across the software development lifecycle with a holistic view of business, development, operations and innovation activities. In a similar vein, Järvinen et al. [39] propose a model to characterize the evolution of enterprises towards and beyond real-time value delivery. The model is based on concepts borrowed from the *Elastic Enterprise* [80] and the *Lean Startup* framework [68]. The authors argued that in the current economic climate, software companies require new capabilities to: 1) deliver value in real-time; 2) increase customer insight through active customer involvement and rapid experimentation; and 3) seek new ways of executing their existing businesses to enable them to move into completely new business areas when needed. The last stage is called *mercury business*.

According to Olsson et. al 's model [62], companies evolve from traditional development to ASD through the careful adoption of agile practices and a shift to smaller cross-functional teams. When an organization matures in the use of agile, and uses automated system integration and verification, then that organization can take the next step, which is the adoption of continuous integration. When continuous integration is in place, customers often express an interest in receiving enhancements and bug fixes more frequently, so the organization migrates to CD. The final step occurs when the organization not only releases software continuously but also develops mechanisms to conduct rapid experimentation in order to drive innovation.

Although similarities and differences exist among the emerging models of CD, three major themes characterize the models: 1) deployment, 2) continuity and 3) speed. Hence, continuous deployment means the ability to bring valuable product features to customers on demand and at will (deployment), in series or patterns with the aim of achieving continuous flow (continuity) and in significantly shorter cycles than traditional lead-times, from a couple of weeks, to days or even hours (speed). In addition, some authors distinguish between delivery and deployment. For example, Humble and Farley [36] describe continuous deployment as the automatic deployment of every change to production, whilst continuous delivery is an organizational capability that ensures that every change can be deployed to production, if it is targeted (the organization may choose not to do it, usually due to business reasons). However, most of the scientific literature uses the terms continuous deployment and continuous delivery interchangeably.

[8]International Conference on Software Engineering, http://icse-conferences.org/

2.2. Related work

This section clarifies the need for our CD study. Several studies have systematically analysed the literature on areas related to CD. However, no review has specifically focused on actually structuring the body of CD knowledge [83].

ASD and its practices have been the topics of diverse SLRs. Dybå and Dingsøyr [26] conducted a well-known SLR on empirical studies of ASD published up to and including 2005. In this review, 33 relevant primary studies were identified and classified into four thematic groups: introduction and adoption, human and social factors, perceptions of agile methods and comparative studies. They found a steady increase in the number of studies on ASD. However, they also observed poor quality with regard to the research methods used in most studies, and they suggested an increase in both the number and the quality of empirical studies on ASD. However, the focus in [26] is different from a review of CD as most primary studies included focus on agile methods at the team level instead of integration of the complete R&D organization. As noted by Olsson et al. [62], transition to CD involves evolving agile practices beyond the R&D organization to ensure that other functions such as product management and sales are functioning in an agile manner as well.

As the body of knowledge on ASD has matured, other authors focused their analyses on specific ASD practices. Test-driven development (TDD) is one of the most reviewed areas [20, 77]. However, the described benefits and drawbacks of TDD were inconclusive. Similarly, continuous integration was the subject of several SLRs [73, 27]. Ståhl and Bosch [73] performed a SLR in order to investigate how the practice of continuous integration is implemented in practice. Twenty-two descriptive themes, including build duration, build frequency and pre-integration procedures, were extracted from 46 publications. The authors concluded that a multitude of continuous integration variants exist in practice, but there was no consensus on continuous integration as a single homogeneous practice. Similarly, Eck et al. [27] conducted a SLR to examine how organizations assimilate continuous integration. Based on 43 studies, the authors presented a conceptual framework illustrating the assimilation stages of continuous integration, which included acceptance, routinization and infusion. Again, although continuous integration is a key aspect of CD (as our findings also confirm), continuous integration merely focuses on automating the build process. In general, although ASD and its practices are within the scope of this research, because they represent the origins of CD [52], we are interested in ASD only as far as it explicitly supports CD.

Recently, Mäntylä et al. [52] published a semi-systematic literature review as part of a study focused on rapid releases and testing. They analysed the current tendency towards rapid releases, as well as its benefits, enablers and problems. This review showed that rapid release is a prevalent practice in industry that originated with the agile, open source, lean and Internet speed development movements. Parallel development, strong tool infrastructure for automatic deployment and testing, as well as pro-active customers and product managers, were found

to be enablers of rapid release, whilst time pressure, increased technical debt, customer unwillingness to update, as well as conflicting goals between rapid release and achieving high reliability and test coverage, were found to potentially cause problems. In addition, shorter time-to-market, rapid feedback, customer satisfaction, increased efficiency, improved quality focus and easier monitoring of progress and quality, were identified as benefits. Although Mäntylä et al.'s study shed light on the grey area of CD, what characterizes and constitutes the phenomenon of rapid releases was unclear.

In summary, CD has attracted a lot of interest from the software industry within a short time, and the notion of CD among practitioners is growing. In addition, research related to CD is emerging in the scientific literature. However, the literature is not well structured, and there is no clear understanding of the main factors that characterize CD. The existing literature reviews, which were presented in the previous section (i.e., ASD literature reviews and Mäntylä et al.'s semi-systematic literature review [52]), only partially cover the existing scientific studies on CD. Furthermore, the validity of the evidence in the CD literature is unclear because no previous review evaluated the quality of the published studies.

3. Research Methodology

A systematic mapping study was conducted to obtain an overview of the research on CD. The main difference between systematic mapping studies and SLRs is that while SLRs aim to *'identify best practice with respect to specific procedures, technologies, methods or tools by aggregating information from comparative studies'*, mapping studies focus on *'classification and thematic analysis of literature on a software engineering topic'* [44][9]. In the case of CD, although the term is frequently used in industrial and academic circles (see Section 2), its meaning and the main factors that are part of CD have remained undefined. Therefore, before aggregating information in terms of research outcomes, we need to provide a comprehensive definition of CD, that is, identify, categorize and analyse the available research on the topic of CD in order to describe the phenomenon, obtain an overview of its state-of-the-art, determine the scientific evidence in the reported results and determine areas that are suitable for more detailed study.

In our mapping study, we followed the process of a SLR as established by Kitchenham and Charters [43], but we adapted it to a mapping study, as suggested by Petersen et al. [66]. The research process is outlined in Figure 1. The process was iterative, with feedback loops between the steps that helped to focus the mapping study as we learned more about the phenomenon itself. For example, the search string was piloted until we found one that ensured the most complete access to the body of knowledge about CD. Furthermore, additional research questions were incorporated when we learned more about the data available in the primary studies. However, in order to achieve a legible description of how the research was conducted, the research process is described in a sequential manner, emphasizing the important design decisions taken during the process.

This section describes the design and execution of our systematic mapping study. The complete research package, which includes the research protocol that helped maintain the chain of evidence and the transparency between each step in the process, is available on request.

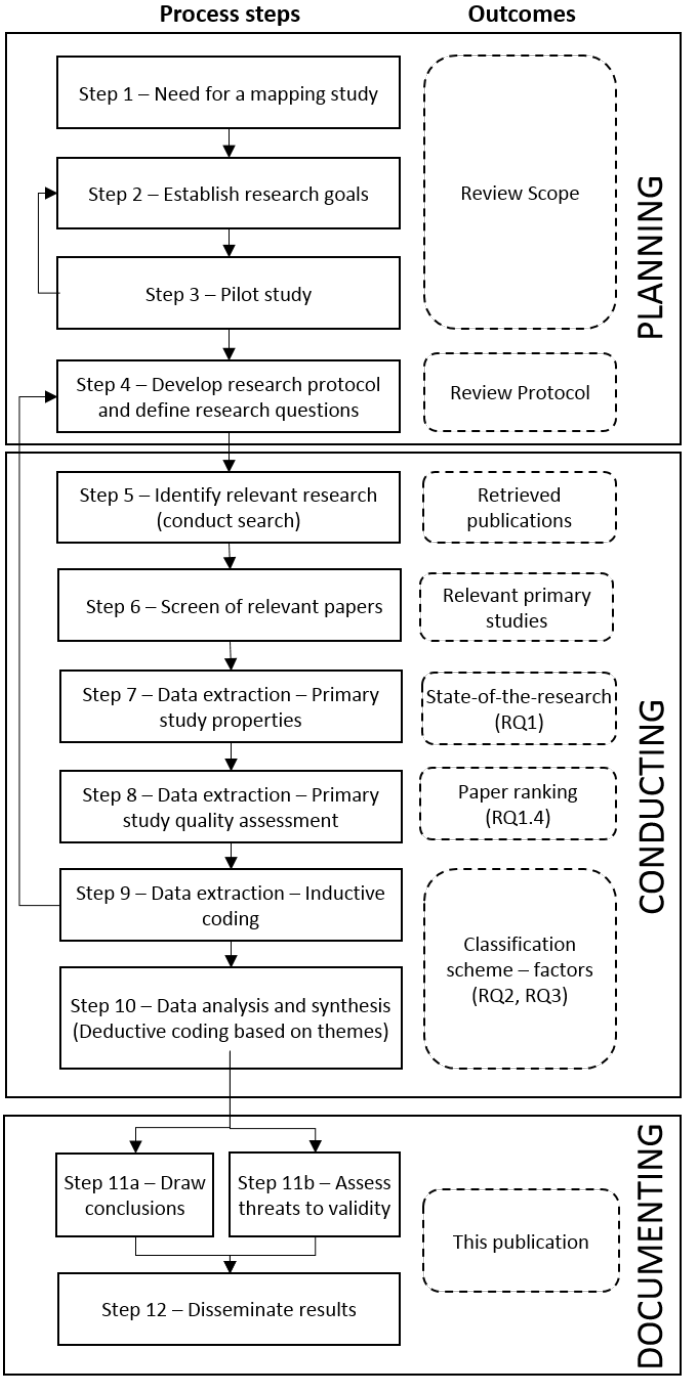

Figure 1: Mapping study steps.

[9] A more detailed description of when to use systematic mapping studies and a discussion of its main differences from SLRs is presented by [44].

Table 1: Research Questions for the Mapping Study

ID	Question	Aim
RQ1	*What is the current state of the research pertaining to CD in the context of software intensive products and services?*	Providing an overview of the studies on CD in the context of software intensive products and services.
RQ1.1	What research methods have been used in studies related to CD?	To categorize available CD research according to research type (industry report, case study, experiment, survey questionnaire, theory development, etc.).
RQ1.2	What kinds of contributions are provided by studies related to CD?	To categorize available CD research according to its contribution facet (model, theory, framework/method, guidelines, lessons learned, advice/implication or tool).
RQ1.3	What are the publication channels used to publish studies related to CD?	To gain an overview of the publication channels used for CD studies (conference or journal) and publication years' frequency distribution.
RQ1.4	What are the levels of relevance and rigour in the published articles?	To assess the quality of the CD studies by examining two perspectives: their industrial relevance and scientific rigour.
RQ2	*What are the main factors that characterize CD in the context of software intensive products and services?*	To structure the state of the art of CD by identifying and analysing the underlying themes associated to this phenomenon discussed in the literature and that, therefore, define it.
RQ2.1	What do researchers mean when they refer to the term CD in the context of software intensive products and services?	To identify and analyse the different definitions of CD available in the literature.
RQ3	*What are the reported benefits and challenges in association with CD in the context of software intensive products and services?*	To identify the reported benefits and challenges experienced when using CD.
RQ4	*What are the research gaps in the area of CD in the context of software intensive products and services?*	To identify research gaps in the field of CD and opportunities for further research.

3.1. The need for the study and definition of research questions

The need for this mapping study (step 1, Fig.1) emerged in the context of the large Finnish research program, *Need for Speed* (N4S, 85 M€, 2014-2017)[10]. The main objective of this study (step 2, Fig.1), which is mentioned in section 1, is to identify the relevant CD literature in order to: 1) create a knowledge base in the area; 2) understand the factors that characterize the nature of CD in the software intensive industry; 3) critically determinate the scientific evidence reported in the CD literature; and 4) identify areas that should be addressed in future research. Our objectives are expressed in the form of the research questions presented in Table 1 (step 4, Fig.1). The research questions (RQ) were defined from a broad perspective. RQ1 concerns the state of the research pertaining to CD; RQ2 concerns the main factors of CD; and RQ4 concerns the research gaps in the area of CD. After reading the entire set of primary studies and gaining a better understanding of the data (step 9, Fig.1), we discovered that different definitions of CD were present in the literature, as well as benefits and challenges regarding its adoption. Accordantly, RQ2 was supplemented with RQ2.1 and RQ3 was formulated to complete the set of research questions. The research questions defined the data to be extracted from the primary studies (see Section 3.5).

[10]http://www.n4s.fi/en/

3.2. Search strategy and databases

Several experimental searches were piloted from April to June 2014 (Fig.1, step 3) in order to better scope the research and determine the search string that most appropriately describes the phenomenon. The following search string was the first used in the review:

Pilot search string: *("continuous delivery" OR "continuous deployment") AND "software"*

Using this string, 28 hits were found in Scopus, and 17 of the 28 were closely studied. Eleven were excluded because they were duplicated, did not focus on the software domain or were non-scientific. Based on an initial reading of the 17 studies, the search string was revised to include the keywords that best described the phenomenon of CD. Table 2 presents the final search string used to retrieve the primary studies together with the rational for the selection of those terms. The search string is composed of terms that represent the population AND intervention, as proposed by Kitchenham and Charters [43]. Our research focus is on scientific studies that discuss software (population) AND have the intention of deployment [OR closely related terms] in a continuous [OR closely related terms] manner (intervention).

Table 2: Search Keywords

	Search term	Rational
Population	software	Studies discussing software, software development, software engineering or software intensive products/services/systems.
AND		
Intervention	deploy*	Studies discussing "deployment" in the context of software (e.g. deploy, deployment, deploying).
	deliver*	Studies discussing "delivery" in the context of software (e.g. deliver, delivery, delivering).
	release*	Studies discussing "release" in the context of software (e.g. release, releasing).
	AND	
	continuous*	Studies discussing "continuous" in the context of software delivery or deployment (e.g. continuous, continuously).
	rapid*	Rapid is a closely related term to "continuous" (e.g. rapid, rapidly).
	fast*	Fast is a closely related term to "continuous" (e.g. fast, faster).
	real time	Real-time is a closely related term to "continuous" (e.g. "real-time delivery").
	agil*	Agile is a closely related term to "continuous" (e.g. agile, agility).
	iterat*	Iterative is a closely related term to "continuous" (e.g. iterative, iteration, iterations).
	increment*	Incremental is a closely related term to "continuous" (e.g. incrementally, increment, incremental).

To increase publication coverage, we decided to use the broad term *software* to ensure that we would not miss relevant references. In addition, during the pilot study, we discovered that various terms referred to the concept of continuous, such as real-time or rapid; therefore, we added these terms to the search string. We also learned that relevant studies, such as [29] and [37] were missed when the terms *continuous* and *delivery* or *deployment* were used together (e.g. 'continuous deployment'). Therefore, we decided to allow the term *continuous* (or its closely related terms) to be separated from the terms *deployment*, *delivery* and *release*. These design decisions, together with the extensive usage of closely related terms, increased the number of studies retrieved (i.e., introduced noise in the search) but reduced the risk of missing relevant studies. We used this search string to search within keywords, titles and abstracts.

The selected databases in which we performed the search are shown in Table 3, in addition to the number of studies retrieved from each database (up to and including June 2014)[11]. The databases were selected considering their coverage of the software engineering literature. The individual search strings in the study protocol are available by request.

3.3. Primary study selection criteria

Studies were eligible for inclusion in the mapping study if they presented a scientific contribution to the body of CD knowledge in the context of the software-intensive industry. Concretely, the inclusion criteria were defined as *'any study that is a scientific article and clearly states that it focuses on software development or software intensive products, systems or services AND includes any software development activity as primary subject with the intention of continuous deployment or delivery of a software product, system or service'*.

Because our goal was to analyse trends in the area of CD rather than aggregate empirical evidence gathered from individual studies, both theoretical and empirical studies were included in the mapping study [44]. Similarly, studies conducted in both industry and in academia were included. Three aspects were considered during the screening process used to evaluate whether the content of an article was relevant to CD: deployment, continuity and speed. To be included in the review, the study had to: 1) show intention/ability of bringing a software product/system/service to the production environment in order to be used by the customer (deployment); 2) emphasise continuous series or patterns so that the software is deployed repeatedly, providing a continuous evolution of software functionality (continuity); and 3) focus on significantly shorter cycles than traditional development lead-times, preferably near to real time, at will or on-demand (speed).

We excluded search results that:

1. Did not clearly discuss the continuous deployment or delivery of software intensive product/systems/services.
2. Were not related to the software domain (e.g., medicine, biology, physics, etc.).
3. Were not peer-reviewed scientific articles (e.g. presentations, call for papers, keynote speeches, prefaces, etc.) or book and book chapters.
4. Were short papers.
5. Were not written in English.
6. Were duplicate articles.

For example, because the focus of this mapping study is on the phenomenon of CD, general discussions about ASD, lean software development or any of their practices and tools without an explicit application to CD were excluded. In addition, we imposed no limitations with regard to quality (rigour and relevance) in selecting primary studies.

[11] Performed on 27 June 2014

Table 3: Selected databases and retrieved papers

Database	Filter	Papers
ACM Digital Library	None	933
Scopus	Only conference papers and journal articles in English in the following subject areas: computer science, engineering, business management and accounting	7997
IEEE Xplore	Only conference papers and journal articles	5303
ISI Web of Science	Only articles in the following research areas: engineering, computer science and telecommunications	5215
Science Direct	Only conference papers and journal articles	1934
		Total 21382

3.4. Primary study selection procedure

To screen the retrieved publications (steps 5 and 6, Fig.1) we followed the process shown in Figure 2. Due to the high number of publications found (21,382) and the inconsistency between the meta-data format stored in different databases, we decided to use the reference management system RefWorks[12] that automated the task of aggregating papers into a consistent list of candidate papers in a unified format. The selection procedure comprised the following steps:

First, two researchers together went through the list of 21,382 candidate publications in order to eliminate duplicate, non- English publications, non-relevant software engineering studies and non-peer review scientific articles (exclusion criteria 2-6). Non-relevant software engineering studies were identified by checking the publication forum and the publication title. Obviously non-scientific peer review publications, such as those titled *'A call for...'* or *'Proceedings of...'* as well as introductions of workshops and editorials were also identified and removed. At the end of this stage, the number of remaining papers was 9,924. In the second stage, based on exclusion criterion 1, we screened candidate papers by conducting a conservative in-depth review. We excluded papers only when it was clear that they were not within the scope of our research. In unclear cases, the paper was passed to the next screening phase. First, two researchers (simultaneously) read the titles of the remaining papers. Generally, it was difficult to identify whether the focus of the study was on CD based only on its title. Therefore, papers that clearly did not focus on CD were excluded in this step; 7,217 papers were excluded. Then two researchers (individually) went through the list of remaining publications (2,707) to screen their abstracts. When they crosschecked the outcome, they agreed to exclude 2,377 papers and include 16 papers. However, based on abstracts they could not decide on 314 papers, which were categorized as *'unsure'*. Introduction, conclusions and, when needed, the whole paper was read in order to resolve *'unsure'* cases. In case of conflict between two researchers a third researcher helped to resolve the conflict. Finally, 34 studies were added to the list of the 16 papers already included (based on title). Therefore, 50 papers remained in the final pool of primary studies. The primary studies (PS) are included in the reference list at the end of this paper (identified by the symbol *[PS]).

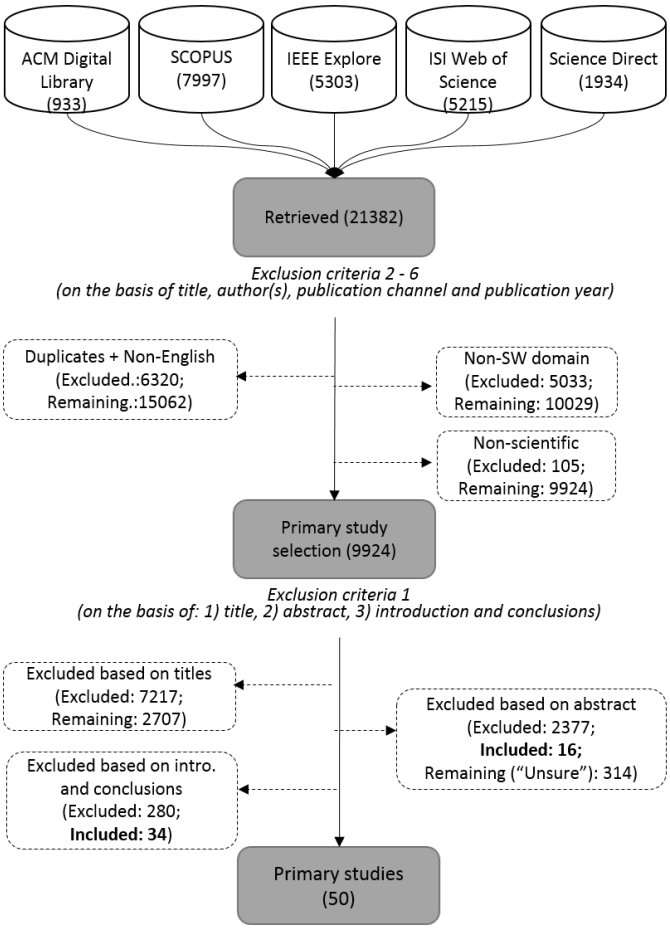

Figure 2: Screening of papers.

3.5. Data Extraction

Table 4 lists the properties collected during the data extraction phase. Based on the RQs and using the qualitative analysis tool NVivo[13], three categories of data were extracted from the primary studies: 1) primary study properties (P1 to P6); 2) quality assessment information (P7 and P8); and 3) recurrent themes (P9 to P12). The next subsections describe each extracted property.

[12] http://www.refworks.com/

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

Table 4: Data Extraction Form

ID	Property	Research question(s)
P1	General type of paper	RQ1.1
P2	Research method	RQ1.1
P3	Contribution	RQ1.2
P4	Domain	RQ1.2
P5	Pertinence	RQ1.2
P6	Publication year and forum	RQ1.3
P7	Quality - rigour	RQ1.4
P8	Quality - relevance	RQ1.4
P9	CD factor	RQ2
P10	CD definition	RQ2.1
P11	Benefit from using CD	RQ3
P12	Challenge for adopting CD	RQ3

3.5.1. Primary study properties: P1-P6

Six primary study properties, which are briefly described below, were defined in order to answer RQ1.1 - RQ1.3. The detailed definitions of these categories are presented in Appendix A.

1. *General type of paper*: represents the type of research (empirical, theoretical or both). Definitions of these values were adapted from [42].
2. *Research method*: categorizes the studies according to the applied research method. Ten categories were considered: case study, industry report, experiment, survey, action research, mixed methods, grounded theory, design science, opinion paper and not stated. When possible, we used the definitions provided by Unterkalmsteiner et al. [78]. Otherwise we created our own definition[14].
3. *Contribution*: maps different types of study outcomes (inspired by Paternoster et al. [65]). Seven categories of contribution (adapted from [72]) were chosen: model, theory, framework or method, guidelines, lessons learned, advice or implications, and tools.
4. *Domain*: maps the different types of domain in which CD is used. Four categories were used to classify the domains of the primary studies: 'embedded systems', 'web/internet based applications or services', 'desktop applications' and 'not stated/not clear' (to indicate that the domain was not clearly stated in the paper).
5. *Pertinence*: this property (inspired by Paternoster et al. [65]) was designed to distinguish between studies entirely devoted to CD and studies with a broader perspective. The values of pertinence were defined as fully, partially or marginally focused on CD.
6. *Publication year and channel*: categorizes primary studies according to publication channel (conference or journal) and provides the frequency distribution of the publication years.

[14]For those research methods that Unterkalmsteiner et al. [78] did not consider in their classification.

3.5.2. Primary study quality assessment: P7 and P8

Primary study quality assessment (e.g. evaluating how reliable the study is) is critical in SLRs because the aim of SLRs is to aggregate the results of primary studies in order to discover whether the research outcomes are consistent or contradictory. Therefore, ensuring that results are comparable and based on the best evidence is critical in SLRs. However, analysing the quality of the primary studies is not essential in the case of mapping studies, which aim to classify the relevant literature [44]. In our case, we analysed the quality of our primary studies as one characteristic of the state of the art of CD (RQ1.4). Therefore, the primary studies were not filtered (excluded) based on their quality but all primary studies were considered in the analysis phase independently of the quality assessment results (i.e., categorizing the topic area according to their meta-data [RQ1], counting the number of studies in those categories and identifying and analysing the factors that characterize CD [RQ2 and RQ3]).

To assess the quality of our primary studies (Fig.1, step 8), we applied the method proposed by Ivarsson and Gorschek [38]. Accordingly, we considered two perspectives: scientific rigour and industrial relevance. Three aspects were used to evaluate scientific rigour: 1) context description: to what degree the context of the study is described so that it can be understood and compared to another context, allowing the replication of the study; 2) study design: to what degree the design of the study is properly described and guarantees the rigour of the research; and 3) validity discussion: to what extent the validity of the study is considered and evaluated. Rigour was evaluated using a three-point scale: strong description (1), medium description (0.5) and weak description (0). Thus, the assessment of rigour ranged from 3 to 0. On the other hand, industrial relevance was evaluated according to four aspects: 1) subjects: whether the subjects of the study were representative of CD practitioners (i.e., they were not students or researchers); 2) context: whether the study was performed in a representative setting (i.e., industrial setting); 3) scale: whether the study size was realistic (i.e., not based on a 'toy' example); and 4) research method: whether the research method used in the study contributes to an investigation of real situations (typically empirical research conducted in representative settings contributes in this sense). In accordance to Ivarsson and Gorschek's method, relevance was measured using two values: 1 if the aspect contributed to industrial relevance and 0 otherwise. Therefore, the assessments of industrial relevance ranged from 4 to 0. To analyse the quality of theoretical papers, we adapted Ivarsson and Gorschek's model [38] to the characteristics of theoretical studies. Industrial relevance was measured in the same way. Therefore, most theoretical papers had low industrial relevance if they did not include any empirical evaluation of their proposed model, framework or tool. The criteria for rigour were adapted as follows: 1) context description: description of the context in which the theory or model could be applied; 2) study design: the degree to which the theoretical contribution used sound theoretical bases to guarantee the quality of the research; and 3) validity discussion: the extent to which the limitations of the theoretical ap-

proach were discussed. For a detailed discussion of the criteria used in Ivarsson and Gorschek's model, the reader is referred to their extensive study on this subject [38].

3.5.3. Continuous deployment factors: P9 and P12

We used the concept *'factor'* (P9) to identify and categorize the recurrent themes in the literature related to CD and create our classification schema (RQ2). During the primary study coding process (see Section 3.6), we identified factors defining aspects that, according to our primary studies, are relevant in achieving CD. That is, aspects that are frequently considered in the literature in order to implement CD in practice. The classification schema grew with the extraction of the data from the primary studies, and it was consolidated in workshops attended by the first seven authors. Similarly, we identified diverse definitions of CD (P10) and compared them in order to understand what researchers mean when they refer to the concept of CD (RQ2.1). Finally, in order to answer RQ3, we coded the benefits that the studies claimed were gained from the use of CD (P11), as well as challenges or aspects that were difficult to implement in the context of CD (P12).

3.6. Data analysis and interpretation

Descriptive statistics were used to answer RQ1. Quantitative descriptions of frequencies were used to analyse research methods, types of contributions, publication channels, publication years, and quality of primary studies.

In addition, thematic synthesis [23] was used to answer RQ2 and RQ3. In order to identify CD factors and create a reliable classification schema, we analysed each primary study in two phases, adapting the five thematic synthesis steps recommended by Cruzes and Dybå [23]. Both inductive and deductive coding was used in the analysis. First, primary studies were coded using an inductive approach (Fig.1, step 9). The goal of this step was to identify key CD aspects. Codes emerged from labelling relevant segments of text that referred to factors important in the context of CD. For example, many papers made reference to *automating the deployment* and a correspondent code was created. Each primary study was coded by one researcher. Then two full-day workshops were conducted in order to review the generated codes and the coding process itself. After a stable list of *'free-codes'* was created, the codes of all primary studies were compared and then organized into higher-order interpretation themes in order to form a high-level set of categories (descriptive themes). For example, we found codes that referred to automating different activities in the development process such as *test automation*, *build automation*, *integration automation*, *deployment automation*, *automation and configuration management*, and we created the theme *automation* in order to consider this aspect of CD. Two full-day workshops were held in order to translate the codes into themes. When the themes were defined, the second coding phase, which was deductive, was implemented (Fig. 2, step 10). The goal of this phase was to check the themes back to the original primary study data. The 50 primary studies were read again and coded according to the themes identified in order to ensure that each theme was taken into account in the analysis of each primary study. During this phase, the themes were consolidated and organized according to the classification schema described in Section 4.5. The themes then were synthesized (results presented in Sections 5 and 6). A theme owner was nominated for each theme, which was the researcher responsible for synthesizing the content of that theme. A second researcher reviewed the synthesis of the theme to improve its reliability. Definitions, benefits and challenges were identified and analysed in a similarly deductive manner.

3.7. Validity threats and limitations of the study

A strength of this study is that a number of researchers were closely involved allowing for triangulation in all phases of the research. Seven researchers (the first seven authors) actively participated in the review, and three researchers with experience in conducting SLRs (the last three authors) acted as external reviewers to validate the research protocol and guide the research process. Nonetheless, there are some potential threats to the validity of this mapping study that need to be considered when interpreting the results (Fig.1, step 11b). We next describe these threats together with the strategies that were applied in order to mitigate their effects.

3.7.1. Identification of primary studies

The process of identifying the primary studies that constitute a mapping study is critical for the success of the research. The search string was built on three main attributes: deployment, continuity and speed in the context of the software industry (and their closely related terms). Nonetheless, the threat of missing relevant articles remains. The attempt to identify the entire body of knowledge in an emerging topic, such as CD, is very challenging. Inconsistency or the use of different terminology with respect to the search string (see Table 2) might have biased the identification of primary studies. This, however, is a minor threat because of the large volume of retrieved studies (21,382). In addition, the search string was used to search in keywords, titles and abstracts. We did not attempt to design a very precise search string that avoided noise because of the blurred nature of the phenomenon itself and because we were able to manage the number of retrieved studies. Thus, our strategy focused on retrieving as many documents as possible that were related to CD. For example, as explained in Section 3.2, we decided to separate the terms deployment and continuous, which created noise in the studies retrieved. From 15,062 publications (after the removal of duplicates and non-English documents), only 50 were selected as final primary studies. This low precision represents a moderate threat to the validity of the mapping study because it induced a significantly higher level of effort when selecting the final primary studies. However, seven researchers actively participated in the research, which minimizes the influence of this threat. In addition, three researchers individually piloted the inclusion and exclusion criteria in order to check their validity. Every step in the selection process was conducted by pairs of researchers in order to minimize subjectivity. When opinions within the pair conflicted, a third researcher helped to find agreement. In unclear cases, we were conservative and always included the paper in the next screening step.

3.7.2. Data extraction

When the primary studies were selected, they were subject to in-depth analysis. The analysis phase also posed threats to validity, which also need to be considered. These threats are largely because of researcher bias. The main countermeasures taken to address this threat were researcher triangulation and explicit definitions of the data to be extracted. Three aspects were analysed during the data extraction: study properties, quality assessment and factor analysis (including definitions of CD, benefits and challenges). Regarding the properties, each property was explicitly defined in the protocol, as indicated in Section 3.5. In addition, each paper was analysed by one researcher and then reviewed by a second researcher, who double-checked that the properties were properly collected. In cases of disagreement, a third researcher worked to achieve a resolution. The same process was applied in assessing both aspects of the study's quality: industrial relevance and scientific rigour. It is important to note that in the case of study quality, the evaluation depended on the reporting quality and not on the intrinsic quality of the study itself. With regard to the factor analysis, different countermeasures were used in the inductive coding, deductive coding and study synthesis. A single researcher inductively coded each primary study. This phase was deliberately unrestricted to produce *'free-codes'*. However, two full-day workshops consolidated the inductive coding to generate a reliable classification schema (themes identification), which involved seven researchers. Once themes were identified, they were formally defined for inclusion in the protocol by two researchers in order to ensure that all researchers involved had the same understanding of the themes. Based on the themes identified, the primary studies were deductively coded at the theme level. Deductive coding was conducted by a single researcher, who was nominated as the owner of that specific theme, with a second researcher reviewing the theme level coding. When the primary studies were coded at the theme level, synthesis was carried out by the theme owner and then reviewed by two other researchers. In addition, one researcher, who had a global vision of the study, reviewed every step in the analysis in order to consolidate the results and ensure consistency of the analysis.

3.7.3. Publication bias

Publication bias refers to *'the problem that positive research outcomes are more likely to be published than negative ones'* [78]. This problem occurs in any literature review or mapping study. In our case, its effect was moderate because our study does not aim to compare research outcomes but to draw a map of the state of the art of CD. Nonetheless, publication bias may have affected our results regarding the benefits and challenges experienced when migrating to CD. The benefits may be overemphasized, compared to the possible risks. In addition, publication bias is affected by the sources of information considered in the study. However, we did not restrict publishers, journals or conferences. We also used five electronic databases, of which two were specialized in the field (ACM Digital Library and IEEE Xplore) and three that offered wide coverage of diverse sciences (ISI Web of Science, Scopus and Science Direct). Although our results were limited by scientific studies published in these databases, they covered a wide range of the software engineering literature. In addition, we excluded non-peer reviewed scientific studies, book, book chapters and short papers because we did not consider that they would provide reliable information for our study.

4. Results: Overview of the State-of-the-Art of CD

From the initial set of 21,382 publications (see Table 3), 50 studies were identified as contributing to the topic of CD in the software domain. This section presents an overview of the body of CD knowledge found from their review. We structure the section according to the research questions presented in Table 1. Sections 4.1 to 4.4 present the research methods used, the kind of contributions provided, publications channels and the results of the quality assessment. Section 4.5 then provides a list of the main factors (i.e., recurrent themes in the literature) that characterize CD (classification schema). Finally, Section 4.6 presents an overview of the benefits and challenges of CD identified in the literature. The main factors, benefits and challenges are further elaborated in Sections 5 and 6, respectively. Table B.10 in Appendix B presents a detailed overview of each primary study.

4.1. RQ1.1: Research methods

The primary studies were classified according to the research method used in the study, as defined in Appendix A. Figure 3 shows the distribution of the research methods. Most studies on the state-of-the-art of CD were empirical in nature (72 percent comprising industry report as well as case study, action research, grounded theory, design science, mixed method and experiment). However, theoretical studies (painted area in the chart), mainly in the form of models and frameworks, and methods, were also significant (24 percent, from which 16 percent were also empirically evaluated). Opinion papers completed the distribution of the research methods (4 percent). Interestingly, a high percentage of primary studies shared practitioners' experiences regarding the application of CD in the form of industry reports (36 percent). Even so, 48 percent of the studies used quite rigorous empirical research methods such as case studies, action research, grounded theory, design science, experiments and mixed methods. Case studies constituted a clear majority of this group (36 percent, of which 70 percent were pure empirical research that applied the case study research method, and 30 percent were combinations of theoretical research, mainly frameworks and models, evaluated by case studies). Action research (4 percent), mixed method, design science, grounded theory and experimentation (2 percent each) completed the list of empirical research methods. In general, the results showed that from a scientific point of view, the body of CD knowledge is still at an exploratory stage as a high percentage of the studies presented the views of practitioners regarding CD. This is natural in a phenomenon that has been especially driven by industry instead of resulting from the context of a research lab. However, as shown in Figure 4, the percentage of studies that

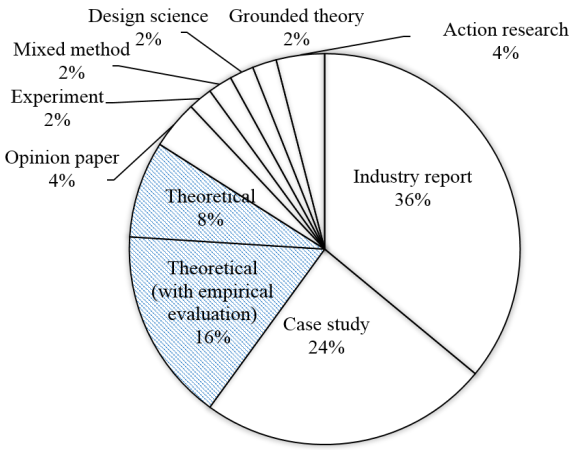

Figure 3: Publication distribution-research method.

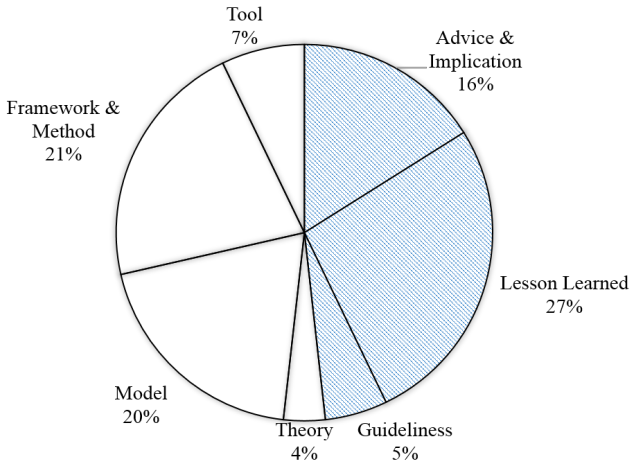

Figure 5: Publication distribution-contribution.

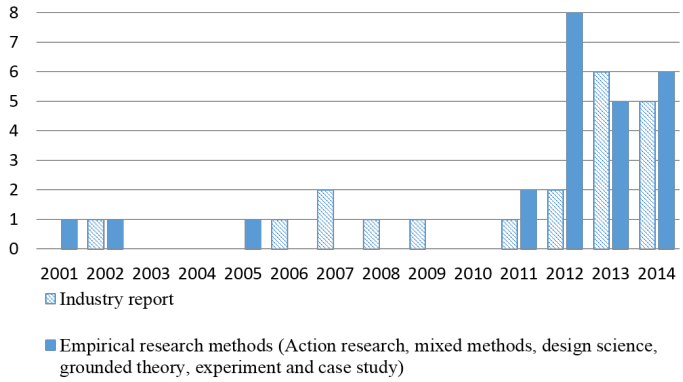

Figure 4: Distribution of research method and publication year.

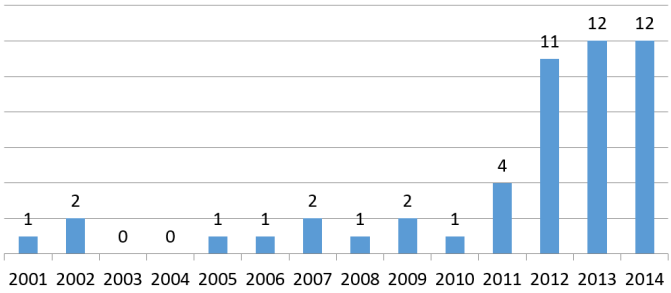

Figure 6: Publication distribution by year.

apply more rigorous research methods has been increasing during recent years.

4.2. RQ1.2: Contributions

Figure 5 shows the distribution of the research contributions. Many contributions were in the form of CD advice and implications (16 percent), lessons learned (27 percent) and guidelines (5 percent) when applying CD. Nonetheless, 48 percent of the contributions provided concrete approaches that could be used to support CD. These approaches included methods and frameworks for implementing CD (21 percent) (e.g. continuous Scrum [1] and methods for identifying risk areas in the context of CD [22] [2]), models representing relevant concepts of CD (20 percent) (e.g., models for continuous experimentation [30] and [29]), and tools supporting the technical infrastructure of CD (7 percent) (e.g. Gamma tool for assisting developers to monitor deployed systems [63]). The complete list of frameworks and methods, models and tools identified in the mapping study is listed in Table B.11, Appendix B.

4.3. RQ1.3: Publication years' frequency distribution and publication channels

As showed in Figure 6, the papers reviewed were published between 2001 and 2014, which indicates the novelty of the phenomenon. Hence, the research on CD is still in its infancy when compared with the history of the software engineering discipline. Although some studies were published between 2001 and 2011, most were published within the last three years (68 percent). The publication of the book on CD by Humble and Farley in 2010 [36] has probably influenced the emergence of scientific studies on the topic. Still, the large number of recently published studies indicates an increasing interest in CD and points to the relevance of the area.

An interesting result was the distribution of the study domain by year (See Figure 7) which was categorized in embedded systems, web/Internet based applications or services, and desktop applications. The category 'not stated' (N/S) was used for cases in which the domain was not clearly defined (see Section 3.5.1). As shown in Figure 7, a clear majority of the primary studies were conducted in the web applications/services domain (42 percent). Twenty four percent of the studies were in the embedded systems and four studies (8 percent) were conducted in the context of desktop applications (i.e. Firefox). Finally, a high percentage of studies (26 percent) did not clearly describe the domain in which the research was conducted. Interestingly, the study domain and their frequency distribution by year indicated that the first studies published in the area focused only on web applications or services (primary studies published be-

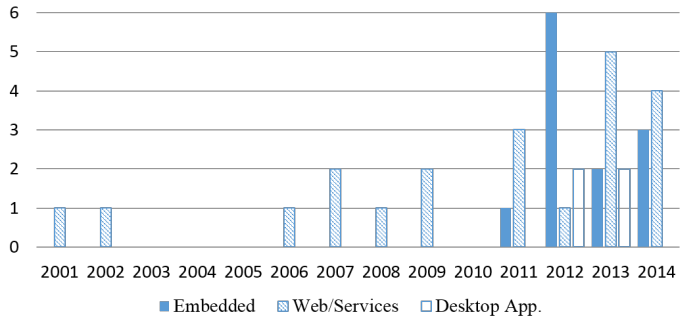

Figure 7: Distribution by publication year and domain.

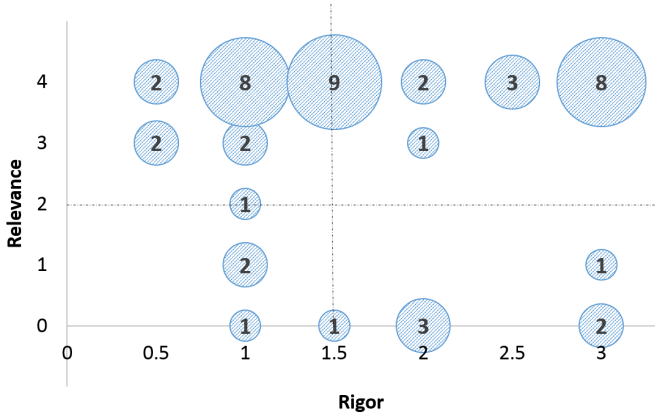

Figure 8: rigour-relevance overview.

tween 2001 and 2009). Thus, the first of our primary studies in the embedded domain was not published until 2011. Furthermore, the results of the pertinence of these studies (whether the study was devoted to CD or had a broader perspective, see Section 3.5.1), showed that 66 percent of studies conducted in the web domain fully focused on CD (14 studies), whereas this number decreased to 41 percent in the case of embedded systems (5 studies). These results indicate that apparently CD is used more often in web-based applications. Some organizations in the area of web applications are currently able to deploy many new versions per day (see the references to organizational white papers and on-line blogs in Section 2.1). However, this goal is still a challenge for systems in other domains and organizations, such as in the field of embedded systems. Hence, primary studies in the embedded domain are more dedicated to specific aspects of CD. The challenges encountered in applying CD in embedded systems are further elaborated in Section 6.

Regarding publication channels, 84 percent of the primary studies were published in conference proceedings (42 papers), while 16 percent (8 papers) were published in journals or magazines. Overall, journal publications are subject to a more rigorous review process to ensure the quality of the research. However, they also take a longer time to publish papers and this might discourage authors in a field that is developing rapidly and is largely driven by industry. Although the publication channels cannot be used to provide direct measure of the quality of the studies, it can be interpreted as an early indicator. The next section further elaborates on the scientific quality of the primary studies.

4.4. RQ1.4: Primary study quality

The assessment of the primary studies quality regarding scientific rigour and industrial relevance, as described in Section 3.5.2, resulted in Figure 8. The two opinion papers ([67] and [9]) were excluded from this analysis as they do not suit the assessment dimensions considered in Ivarsson and Gorschek's approach [38]. The raw data for this figure is available in Table B.10, Appendix B.

Regarding industry relevance, 37 primary studies (74 percent) had a relevance rating higher than two. Accordingly, most of the studies were conducted in industrial settings involving practitioners on an industrial scale. From these studies, 14 (28 percent) lie in the upper right quadrant of the chart (rigour >= 2, relevance >= 3). However, 23 studies (46 percent) exhibited high industry relevance (relevance >= 3) but showed low scientific rigour (rigour < 2). In addition, theoretical contributions without any empirical evaluation and research conducted with students constituted most of the studies with low relevance. Overall, it can be said that the topic of CD is highly relevant from the perspective of the industry; therefore, CD appears to be a promising research area because in an applied research field such as software engineering, it is the industry that ultimately determines the relevance of the research results.

Regarding scientific rigour, as many as 28 of the 50 primary studies have a rigour value of < 2 (56 percent). Consequently, the global scientific evidence of the body of CD knowledge can be considered as medium-low. With regard to context description, in the majority of the studies, the context in which the research is performed is not described to a degree such that it can be fully understood and compared with other contexts. Study design, data collection and data analysis were, in general, not well described, as many of the publications were industry reports. In addition, only twelve primary studies included a proper validity discussion, (i.e. issues of bias, validity and reliability), whilst four primary studies mentioned validity, it was not described in detail. Thus, there is no description of threats to validity in 34 of the primary studies. Thus, it can be concluded that there are important limitations to the scientific quality of the studies we retrieved and this inevitably reduces the reliability of the results.

4.5. RQ2: Continuous deployment factors

The 50 primary studies cover a wide range of research topics. We categorised them into ten main themes, which together characterise the phenomenon of CD. A brief description of each theme and the primary studies that make reference to each theme are presented in Table 5. These factors are further elaborated in Section 5.

4.6. RQ3: Benefits and challenges

The benefits and challenges reported in the primary studies were identified and synthesised, as reported in Tables 6 and 7.

Table 5: Continuous deployment factors - Mapping study recurrent themes

Factor	Description	Frequency	Primary studies
1. Fast and frequent releases	Ability to release software whenever the organisation wants to (on demand or at will) based on need and with preference given to shorter cycles or even continuous flow (weekly or daily).	28	[8, 76, 75, 79, 58, 16, 17, 29, 1, 31, 32, 34, 2, 46, 47, 48, 50, 54, 59, 62, 67, 18, 22, 31, 40, 51, 57, 41, 49, 53, 74]
2. Flexible product design and architecture	CD requires evolutionary and robust software architecture with the aim of balancing speed and stability.	9	[2, 7, 8, 15, 51, 62, 79, 16, 17]
3. Continuous testing and quality assurance	Ensuring the quality of the software at all times without compromises despite the need for fast and continuous deployment.	31	[1, 2, 8, 10, 15, 22, 31, 32, 34, 37, 40, 47, 48, 51, 54, 57, 56, 59, 61, 75, 76, 3, 9, 16, 17, 41, 53, 60, 74, 49, 4]
4. Automation	Automating the delivery pipeline from building and testing to deployment and monitoring.	24	[1, 2, 10, 31, 32, 34, 37, 40, 47, 48, 51, 54, 57, 56, 59, 61, 62, 76, 75, 79, 9, 67, 3, 70]
5. Configuration management	Version control branching strategies and system configuration management approaches to enable CD.	12	[34, 46, 47, 54, 59, 31, 56, 37, 40, 9, 51, 79]
6. Customer involvement	Mechanisms to involve customers in the development process and collect customer feedback from deliveries as early as possible (even near real-time) to drive design decisions and innovation.	12	[79, 29, 15, 31, 45, 46, 47, 51, 54, 62, 49, 82]
7. Continuous and rapid experimentation	Systematical design and execution of small field experiments to guide product development and accelerate innovation.	10	[15, 30, 31, 59, 61, 62, 9, 29, 67, 33]
8. Post-deployment activities	Activities that are conducted once the product (or a new feature or enhancement of the product) has been deployed to support fast business and technical decision making.	12	[1, 24, 31, 32, 45, 46, 51, 59, 61, 63, 9, 33]
9. Agile and Lean	Extending Agile and Lean software development towards continuous flow to support CD.	22	[7, 8, 76, 75, 10, 16, 1, 30, 34, 35, 2, 46, 47, 54, 59, 62, 67, 3, 18, 31, 57, 18]
10. Organizational factors	Organizational factors that enable CD (integrated corporative functions, transparency and innovative and experimental organizational culture).	17	[7, 24, 31, 32, 35, 40, 46, 47, 50, 54, 57, 59, 62, 16, 67, 64, 74]

Table 6: Benefits of continuous deployment

Benefit	PS
Shorter time-to-market	[1, 10, 31, 34, 46, 50, 57, 61, 62, 59, 76, 75, 9, 48, 54, 59, 67, 30, 41, 49]
Continuous feedback	[35, 46, 47, 50, 51, 59, 75, 62, 61, 24, 31, 32, 9, 41]
Improved release reliability	[1, 37, 46, 59, 9]
Increased customer satisfaction	[1, 10, 59, 76, 41]
Improved developer productivity	[1, 35, 37, 9]
Rapid innovation	[31, 34, 61, 62]
Narrower test focus	[22, 31, 59, 41]

Table 7: Challenges of continuous deployment

Challenge	PS
Transforming towards CD	[16, 54, 10, 57, 59, 61, 64]
Customer unwillingness	[62, 61, 10, 1, 63, 82]
Increased QA effort	[1, 40, 46, 54, 56, 59, 41, 53]
CD in embedded domain	[75, 3, 15, 49]

The benefits experienced when applying CD include shorter time-to-market, instant feedback, especially from customers when using proper monitoring and experimentation techniques, improved release reliability, partially as a result of narrower test focus, and improved customer satisfaction and developer productivity. Overall, CD benefits are more often mentioned in primary studies than are the challenges; this may be a consequence of authors and practitioners willingness to report positive rather than negative results. Still, important challenges were identified regarding the change that moving towards CD implies in to the whole organisation. These include customers' unwillingness to receive continuous product updates, increased QA efforts and difficulties applying CD in the embedded domain. CD benefits and challenges are further elaborated in Section 6.

5. Analysis of Continuous Deployment Factors (RQ2)

First, we were interested in understanding what researchers mean when they refer to the term 'CD' (RQ2.1). Table 8 shows descriptions of continuous deployment and continuous delivery as they appear in the primary studies. We did not find any formal definition of CD in any of the primary studies; however, there appears to be some level of agreement amongst authors that CD refers to the ability of an organisation to release software functionality directly to customers on demand and at will (deployment), faster and more frequently than traditional software development. We observed that there exists a tendency to use the two concepts interchangeably (except [32]).

In addition, the CD literature encompasses diverse recurrent themes or factors, as listed in Section 4.5. In the remainder of this section, we analyse each of the identified factors based on a synthesis of the primary studies.

5.1. Fast and frequent releases

Several papers in the mapping study discuss fast and frequent release as shown in Table 5. Fast release is the ability to release software whenever the organization wants to, based on their need, which could be weekly or daily [59, 46]. Almost all of our primary studies make reference in one way or another to accelerating the release cycle by shortening the release cadence and turning it into a continuous flow e.g. from months to weeks [75, 76, 41, 49], or from six months [54] or eight-weeks [59] to a continuous flow. However, achieving fast release in the form of a continuous flow is not free of charge. For example, Rally Software [59] began to shrink the release cycles down to fortnightly, weekly, semi-weekly and finally at-will, which took months of preparatory work to streamline and automate the deployment. In addition, a case study at Mozilla Firefox [41] points out the question whether the quality of the software product improves as the shorter release cycles results in shorter testing periods. Also, Lavoie and Merlo [49], claim that accelerating the release cycle can make it harder to perform re-engineering activities.

5.1.1. Continuous planning

The CD literature emphasizes two aspects related to planning fast and frequent releases: continuity (e.g. [32, 67]) and taking a holistic view of planning [30, 58, 17, 8, 47]. Traditional planning tends be performed cyclically and is usually triggered by the annual financial year. However, CD challenges and changes traditional planning towards continuous planning in order to achieve fast and frequent releases [32]. CD requires that planning activities are done more frequently to ensure alignment between the needs of the business context and software development, requiring tighter integration between planning and execution. Fitzgerald and Stol [32] define plans as dynamic open-ended artifacts that evolve in response to changes in the business environment and require multiple stakeholders to be involved both from business and software functions. Hence, tighter integration between planning and execution is required in order to achieve a more holistic view of planning. Poppendieck and Cusumano [67] suggest considering software as a flow system where software is designed, developed, and delivered with a steady flow of small changes. A view that is fundamentally different from thinking of software development as a completed project, or even thinking about software as a series of annual or semi-annual releases. Rapid delivery should not be isolated to the software development alone, and the flow should happen within the overall product development cycle, of which software is just one aspect. Therefore, continuous planning includes all activities from strategic and business planning to product, portfolio and release planning. Similarly, Fagerholm et al. [30] point out that according to software development based on continuous experimentation (common in CD, see section 5.7), the experimental results should be continuously linked with the product roadmap as well as managed within a flexible business strategy in order to provide guidance for planning activities.

Table 8: Descriptions of continuous delivery and continuous deployment available in the primary studies

PS	Description
[46]	'Continuous delivery is a set of practices and principles to release software faster and more frequently'.
[59]	'Officially, we describe continuous delivery as the ability to release software whenever we want. This could be weekly or daily deployments to production; it could mean every check-in goes straight to production. The frequency is not our deciding factor. It is the ability to deploy at will'.
[61]	'The concept of continuous deployment, i.e. the ability to deliver software functionality frequently to customers and subsequently, the ability to continuously learn from real-time customer usage of software'.
[31]	' . . . the practices that Internet companies use are known as continuous deployment. This reects the habit of deploying new code as a series of small changes as soon as its ready'.
[62]	'Continuous deployment is the idea that you push out changes to the code all the time instead of doing large builds and having planned releases of large chunks of functionality'.
[40]	' . . . continuous delivery [1] that is, to continuously deploy the environment in a test environment that is reasonably similar to the actual production environment as part of development and testing efforts and to promote it to production when appropriate'.
[32]	'These concepts are related in that continuous deployment is a prerequisite for continuous delivery, but the reverse is not necessarily the case. That is, continuous delivery refers to re- leasing valid software builds to users automatically, whereas continuous deployment refers to the practice of deploying the software to some environment, but not automatically delivering to customers'.
[1]	'SaaS products provide an opportunity to provide consumers with continuous deployment of new features, as opposed to scheduled version upgrades as is the norm for products installed on-premise . . . continuous deployment of new versions of a software product in production'.

5.1.2. Mechanisms for achieving fast and frequent release

Besides continuous planning, other mechanisms are proposed in the literature to achieve fast and frequent release. Many of these mechanisms are important enough to become themes in their own right and are further developed in the following sections. For example, most studies highlight automation as essential to achieving fast and frequent release (e.g. [1, 59, 54, 79]). Close interaction with customers (e.g. [75, 29, 58]), having a clear release process (e.g. [50, 47]), a release management workflow [47] or a continuous delivery workflow [46, 47] appear also in the literature as enablers of CD. Staron et al. [74] present a release readiness indicator, a mechanism to predict in which week the release would be possible given the defect history. In addition, some studies discuss fast release in the context of ASD. For example, Krusche and Alperowitz [46] distinguish between time-based and event-based delivery. They claim that fast release, as the teams' ability to deliver a potentially shippable product increment at any time in the project, is particularly useful at the end of the sprint (time-based delivery) when delivering the increment to the customer, but it also helps to obtain rapid feedback during the sprint (event-based delivery). In a similar vein, Agarwal [1] proposes Continuous Scrum as a mechanism to achieve and sustain a rapid release cycle (one week product deployment) through parallel development.

5.1.3. Effects of fast and frequent release on product quality

The effects of fast and frequent release on the quality of the delivered software are also a focus in the literature. The CD literature highlights that a faster and more frequent release cycle should not compromise quality (e.g. [1, 59, 31, 41]). Thus, it is important for the engineering and QA teams to ensure backward compatibility of enhancements, so that users perceive only improvements rather than experience any loss of functionality [1]. Thus, the ability to release quickly does not mean that the development should be rushed into without a full understanding of what is actually being done [59]. Neely and Stolt [59] advises monitoring everything to know the exact state of the system at every moment. Furthermore, based on a case study conducted on the effect of rapid releases upon quality at Firefox [41], Khomh et al. found that even though users do not experience significantly more post-release bugs in comparison with the traditional release model, program crashes occur earlier and users experience bugs earlier during execution. In another similar empirical study in the context of Mozilla Firefox, the rapid release model makes re-engineering activities harder to achieve and even though the code changes are smaller, they become a more important risk with a fast release cycle [49]. In addition, the authors found that code change activities tend to focus more on bug fixing and maintenance than functionality expansion. However, there is no significant difference concerning the volume of changes among rapid release and a traditional model.

5.2. Flexible product design and architecture

Several primary studies make reference to product architecture and design [2, 7, 8, 15, 51, 62, 79, 16, 17]. CD demands a software architecture in which the product and its underlying infrastructure continuously evolve and adapt to changing requirements [16, 51]. Thus, it is essential that the underlying architecture is flexible and is able to accommodate rapid feedback [7, 8, 51]. However, at the same time, the architecture must be robust enough to allow the organization to invest its resources in offensive initiatives (e.g. new functionality, product enhancements and innovation) rather than defensive efforts (e.g.

bug fixes) [8, 7, 51, 16]. To achieve this, the software architecture and design have to be highly modular and loosely coupled [51, 62, 8]. In addition, what seems essential in the context of CD is that the software architecture accommodates mechanisms to rollback unsuccessful deployment [62], supports independent deployment of a particular component rather than the entire system [62, 79] and enables experimentation through runtime variation of functionality as well as data collection mechanisms [15, 62].

The main challenge with regards to software design and architecture, in the context of CD, is the ability to maintain the right balance between speed (quickly delivering functionality to the users) and stability (providing reliable and flexible architecture) [8, 17]. To overcome this challenge, some approaches propose a focus on measuring and monitoring source code and architectural quality. For instance, [8, 7] suggest extending prototyping to include quality attributes, such as performance or security-related issues (i.e. prototyping with a quality attribute focus), as a method of incorporating both functional and non-functional requirements in the context of CD. Rapid architecture trade-off analysis to accommodate rapid feedback and evaluate design options [7], quantifying architectural dependencies by combining Design Structure Matrix (DSM) and Domain Mapping Matrix (DMM) techniques [17] and identifying and assessing risky areas of the source code based on diverse metrics [2] are also mechanisms proposed to maintain speed and stability. These mechanisms provide systematic feedback, bring more visibility and awareness to stakeholders and finally trigger re-factoring and re-architecture initiatives when needed [8, 17, 2].

5.3. Continuous testing and quality assurance

Empirical evidence shows that most companies typically perform testing activities late in the development process causing unpredictable additional development efforts [60] and significant delays in releases [74]. Many primary studies emphasise the importance of employing testing and quality assurance (QA) practices throughout the whole development process in the context of CD, as features are rolled out, and not just at the end of the development (e.g. [1, 59, 10, 32, 75, 34, 54]. Thus, continuous testing aims to bring testing practices as close as possible to developers in order to avoid leaving testing activities only at the end of development [32].

In addition, a common problem in the testing activities is that many individual developers do not have an end-to-end overview of all the testing activities that are conducted during the software development process, other than their own individual activities [60, 4]. As a consequence several problems such as duplication of test efforts and slow feedback loops are further testing challenges. To help alleviate these problems in the particular context of CD, authors emphasise the need to make all testing activities transparent to individual developers [4] and using different techniques that help to describe and give a holistic overview of all testing activities such as CIViT - Continuous Integration Visualization Technique [60].

An important observation from studies conducted in open source software, particularly Firefox, was a slight increase on the number of reported bugs observed during testing in rapid release mode compared to traditional development [41, 53, 49]. The studies on Firefox's transition to rapid releases also revealed that CD allows less time for testing activities but enables fast and thorough investigation of software features with the highest regressions risk at a relatively narrower scope [41, 53]. The implication of the latter is that when transitioning to CD tremendous changes in terms of testing resources and strategies need to take place to make the testing process more sustainable in CD. Therefore, it remains crucial to assess whether such changes have significant impacts to the quality of the product [53].

On the other hand, improvements to existing testing practices in terms of fast feedback of code changes to developers during testing activities is expected in CD regardless of whether the software is open source or not [4, 76]. Reported strategies to continuously ensure the quality of the software in the context of CD include not only testing strategies but also creating a company culture of quality [31].

5.3.1. Test automation

Well-known testing techniques in ASD, such as test automation, are also crucial in CD in order to achieve continuous testing [1, 31, 34, 54, 59, 61, 32, 37, 47, 56, 48, 76, 9, 60]. According to our primary studies, test automation ensures: (a) quality of software through extensive test coverage [34, 54], (b) continuous integration and release of quality software [59, 37, 47, 48] and (c) provide early feedback to the development team [37] so that issues are resolved and root causes eliminated [32]. In test automation, a variety of test suites: unit, functional, integration, and performance tests are executed at different phases [37] and with different scope: component, subsystem, partial product, product, release and customer [60]. These test suites use practices such as automated execution of test scripts on a build server after each code commit to test and check the status of the code or the build [1, 48, 54, 76], automating acceptance testing [37, 57], automatic testing of the production environment in non-embedded software [40] and simulation to demonstrate and test the quality of software much earlier for embedded software [3]. In addition, efficient prioritisation and the ordering of automated test execution are emphasised in order to ensure fast feedback to the development team as well as the proper use of resources [32, 37].

Two primary approaches that facilitate test automation, code driven testing (e.g. TDD and test planning [59]) and GUI testing are discussed. Code driven testing practices are mostly preferred by practitioners working in CD [1, 8, 57, 10, 59, 16]. However, due to limitations such as high costs and the effort required to train and manage test-programs, Agarwal [1] recommends the use of automated GUI testing. GUI testing is seen as a lead indicator of bugs that typically appear in a production environment [59]. More details on how GUI testing is implemented in practice are given in the Automation theme (Section 5.4).

5.3.2. Testing with users

Testing new features while in real use and using a small fraction of actual end users, is also emphasized when aiming to CD [31, 48, 59, 15, 51, 41]. This approach provides immediate feedback about the feature quality as perceived by users, allowing developers to quickly discover new bugs to fix [48, 51, 41]. For example, Facebook reports testing new features internally by employees and later by a subset of real world users [31], before making them available to all users. Similarly, in [48] testing new features in real use is done with beta users who are also actual users. When the result of testing features in real use is satisfactory, features are deployed to the entire user base. Otherwise, alternative mechanisms such as rollback are executed [31, 59]. Furthermore, although this approach is mainly used in web applications [31, 48], there is also evidence of testing in real use in the embedded domain, but with the aim of establishing a proof-of-concept as the system is not intended for mass production [15]. See Customer involvement (Section 5.6) and post-deployment activities (Section 5.8) themes for more details.

5.3.3. Creating a culture of quality

It is also suggested that continuous testing and QA embody a culture of developer responsibility in which developers bear the responsibility for writing good code, perform thorough tests as well as support the operational use of their software [31, 54, 76]. As systems become larger and more complex, such culture complements test automation systems and allows the quality of the software to be maintained at scale.

5.3.4. Technical debt

Finally, another important aspect observed and related to quality assurance in the context of CD is the concept of technical debt. It was noted that as a consequence of trade-offs between the fast deployment of software and poor development, testing and quality assurance practices, organisations acquire technical debt over time [57, 8, 17]. As technical debt causes architecture quality degradation over time, organizations applying CD need to continuously monitor and measure the quality of the degrading architecture [8, 17]. Interestingly, two authors in our review have separately studied measurement techniques to determine risky files in embedded software development [2] and the risk associated with deploying certain features [22]. The method proposed by Antinyan et al. [2] is based on measuring a set of code properties, such as McCabes cyclomatic complexity, in order to identify areas of source code that may be faulty and difficult to maintain and provide quick feedback to developers. Risk assessment scoring heuristics for software deployment is another method proposed by Comas et al. [22]. The method proposes decomposing web application into different functionality tiers to identify changes and the requirements needed to implement the changes. Each change implemented in the software is analysed along with risk. The result of the analysis is used to provide information to system integrators about the risky areas on which to focus their efforts e.g. testing in appropriate places.

5.4. Automation

In the context of CD, the focus is on automating the entire delivery pipeline. Prior to CD only parts of the pipeline were automated. However, the CD literature highlights the importance of eliminating all manual steps from build to deploy [34, 59], extending continuous integration with release and deploy automation, and automated configuration management of deployment environments. Humble et al. [37] recommend automating build, testing, and deployment in the early stage of the project and to evolve the automation along with the application. Furthermore, automation is also needed for measuring and improving the work, which concerns the whole delivery pipeline.

5.4.1. Continuous Integration

Continuous integration (CI) is one of the main enablers of CD [61, 62, 34, 32, 75, 3, 1, 37, 47, 48, 54]. The main advantage of CI is that it automates tasks such as compiling code, running unit and acceptance tests, monitoring and validating code coverage, checking compliance with coding standards, static code analysis, automatic code review and building deployment packages [32, 1, 54, 76, 3]. Therefore, CI provides mechanisms to ensure that there is always a shippable product that has passed all of the testing phases [32, 62]. Fitzgerald and Stol [32] report that the frequency of integration is as important as automation itself. Thus, the frequency should be high enough to ensure quick feedback to developers. CI is usually coupled with feedback mechanisms (e.g. dashboards) [34, 37, 48, 54] that enable rapid feedback on source code and triggers for immediate problem resolutions [32, 51]. Regarding the application domain, the main challenge of using CD in embedded systems is that physical assets and hardware equipment should also support CI in order to benefit from CD as a whole. To overcome this problem, in the avionics domain, Ard et al. [3] propose a simulated integrated system that facilitates CI of embedded systems.

The level of automation of CI system varies depending on the primary study [76, 75, 10, 1, 34, 37, 48, 54, 59, 31, 57, 9, 67]. Guidance and working practices on structuring the CI pipeline and what possible stages and tools can be used in the pipeline are reported in several primary studies [34, 37, 31, 76, 75, 48, 1, 2]. Goodmand and Elbaz [34] suggests automating quick builds in order to provide developers with instant feedback. [76, 34] perform nightly builds. In [75, 48], the CI system automatically closes the source control repository from further commits in case of a build failure. [37] suggests using several testing stages in which different types of testing should be independent from each other. Facebook [31] integrates a code review stage as part of the build process using the tool Phabricator. Antinyan et al. [2] develop an automatic measurement system to identify risky files that may need refactoring.

Automated GUI-testing seems to be a popular practice in our primary studies [1, 75, 76, 31, 59]. In GUI testing *'test-automation software is used to record mouse movements and key-presses, and replay these when needed'* [1]. However, implementing automated GUI testing in practice is challenging [1] because it needs to be flexible enough to test different scenarios [76] and do so at a fast-pace [59]. Agarwal [1] also notes the

need for manual observation during automatic GUI testing because the testing software is not able to identify the occurrence of (unexpected) errors due to difficulties in identifying errors automatically. Watir and Webdriver are used in Facebook for automated GUI testing [31].

5.4.2. Release and deploy automation

Many of our primary studies extend their system beyond traditional CI into release and deploy automation [1, 34, 37, 59, 31, 79]. As an example Agarwal [1] suggests an automation system to integrate the configuration management, build, release and testing processes. The system provides developers with a means to migrate changes from one environment to another (e.g. development to production). Thus, the system would allow a release manager to perform a release with selected content, and to upgrade the software in the target environment. In a similar vein, Facebook [31] has a tool (Gatekeeper) for deployment that allows the developers to turn features on and off in the code, and to select which user groups see which features.

Various types of guidance, working practices, and tools for automated deployment and monitoring are provided in [37, 40, 56, 31, 59, 70]. [37, 40, 56] support automated deployment to all types of environments (development, test, staging, production). Feitelson et al. [31] report on how Facebook performs multi-stage deployment where internal testing and performance testing are performed before deployment to production. [59] reports the experiences of Rally Software when mimicking test environments to production environment. This lowers the risk in deployment and provides advice to identify the barriers that prevent delivery from a commit. DevOps also provided practices for automatically linking operations with development and QA functions. For example, with DevOps configuration management becomes another form of source code that can be automatically managed using standard source development techniques [56]. In addition, deployment monitoring tools for global deployment are discussed in [59, 31] (e.g. [31] uses BitTorrent). Although most of the primary studies acknowledge the benefits that automation brings in the context of CD, there were also studies that noted the possible limitations that automation might imply in terms of flexibility for adapting and configuring systems to particular organizations or development contexts [70].

5.4.3. Automation of configuration management for deployment environments

The configuration management system (CM) is also automated in CD, which enables automated provisioning and deployment to various target environments. Meyer et al. [56] describe automated CM tools that specify configuration actions using a high-level declarative language stored on a central server. Using this tool, client machines compare their configuration state to the central configuration specifications after which configuration actions are applied as necessary to bridge the gap between the current configuration and the desired configuration. [40, 37, 9] also describe automated CM systems. For example, Kalantar et al. [40] developed Weaver, a system to manage the configuration of an entire environment, including software components that span systems, and the infrastructure elements that are needed to support them. To do this, they defined a Domain Specific Language to specify blueprints environments, and at runtime execute the blueprints to create or modify environments. The system allows validation of blueprints at design, deployment, and runtime. However, it is not intended to replace low level automation building blocks to install and configure individual software components (e.g. scripting languages, Chef, Puppet). Benefield [9] suggests a similar system to [40] in which they can manage configuration and deployment to a system. In addition, [37, 40, 56] treat deployment scripts (or blueprints) as code which is stored under version control and can also be subject to automated tests as noted in [56].

5.5. Configuration Management

Besides automating CM, two main topics emerge in the review regarding CM and CD: version control branching strategies and system configuration management.

5.5.1. Version control branching strategies

Version control branching strategies aim for regular and incremental delivery, lower integration risk (due to more frequent merges) and improved coordination between teams. [34, 46, 47, 54, 59, 31] discuss version control branching strategies that are suitable for CD. Using a single branch, without any private branches, in order to keep the continuous deployment of new functionality simple is reported in [34, 59]. However, most studies in the review report as using both main (or master) and separate branches. For example, [54, 31] report the use of a main branch that is kept releasable all the time. In addition, separate branches are used for each user story that are merged into the main branch after passing quality gates. In a similar vein, [46, 47] report experiences with a branching model called git-flow[15]. In their variation of git-flow, features are developed in separate feature branches, after which they are merged into the development branch to be shared with other developers. Internal releases are triggered when developers merge feature branches into the main development branch (under the release manager's supervision). Neely and Stolt [59] describe the use of a master branch, which encourages small size stories in order to make merges easier. Thus, long running branches or feature branches are rarely employed in this case. However, instead of feature branches the organization uses feature toggle through an administration interface to switch the toggles and clean them after a story is completed.

5.5.2. System configuration

As described in Section 5.4.3, CD bases CM on automation [56, 37, 40]. In addition, different strategies are proposed in the literature to manage system configuration in CD.

In order to make system CM easier, Humble et al. [37] suggest deploying the same software binaries in every environment

[15] http://nvie.com/posts/a-successful-git-branching-model/

and maintaining runtime configuration separately from binaries. To facilitate the identification of problems and solutions, frequent deployments, where each deployment introduces only a limited amount of new code, characterise the software development process at Facebook [31]. In case of problems they roll back single commits and any of their dependencies, or if that is not possible, they revert the whole binary (consisting of possibly multiple commits) to the previous working version [31]. They also suggest reverting back commits of developers who are not present during the delivery in order to minimize deployment problems. In a similar vein, MacCormack [51] highlights the importance of being able to trace feedback from a release to a particular revision of software. Benefield [9] suggests an atomic packaging scheme where versioned self-contained packages can be independently released and rolled back. Finally, [79] develops a theoretical model for upgrading component-based software in which every release references all of its dependencies, and where releases can be tracked back to source code.

5.6. Customer involvement

One characteristic of CD is collecting customer feedback from deliveries as early as possible (even near real-time) in order to base design decisions on real customer usage and thus use customer input as the main driver for innovation [62, 29]. The importance of customer feedback is highlighted not only in requirements elicitation, prioritisation, definition of user stories and 'definition of done' (DoD) [47, 15, 31] but also in other phases of the product development such as acceptance testing. For example, Marschall [54] suggests developing customer tests in a way that the customer is required to sign off on each user story (or product feature) before it can be considered complete.

Customer involvement in CD concerns the following tasks: 1) determining from whom feedback is collected, 2) what issue feedback addresses, 3) how feedback is collected and in which format, 4) how feedback is processed and 5) how feedback is taken into account in the development process. Some approaches can be found in the CD literature regarding the three first items; however, approaches for processing feedback and taking it into account are scarce.

Determining from whom feedback is collected depends on the kind of feedback that is required or interesting to gather. Olsson et al. [62] propose locating lead customers who serve as role models for other customers. MacCormack [51] suggests that a valuable avenue for identifying lead (beta) customers is through exploring the company's customer-support database. Ko et al. [45] also warn about the importance of selecting a sample of customers that is representative of the user community and discusses challenges when using a vocal minority of existing users.

The second and third tasks refer to how to get customer feedback and limit it so that it targets only the specific issue at hand. Regarding getting feedback for bug fixes, van der Storm [79] introduces a component-based system in which specific components are automatically delivered after fixing the bug and customer feedback is accurately collected. This allows for getting fast feedback for a specific issue on which a developer is focused at the time [46, 47]. In Facebook, Gatekeeper is used to control which parts of the code are actually active for customers [31]. With Gatekeeper, engineers can turn tests on and off at will and also apply them to selected user groups. In this way, the feedback is collected only from the active parts of the code. Gatekeeper can also be used to turn off new code that is causing problems, thereby reducing the need to immediately deploy a correction [31]. Moreover, automated tools such as a delivery server or deployment pipeline allow customers to give feedback directly within the software in a structured way [46, 47]. In addition, monitoring customer usage scenarios (even without the user knowing about it) [15]; A/B testing as an experimental approach to find out what users want [31]; and prototypes and mock-ups as the first visualisation of user interface [51] are useful for collecting feedback and helping with an accurate understanding of customer expectations.

However, the literature does not provide significant solutions for tasks four and five, i.e., how the feedback is processed and how the feedback is taken into account in the development process. When collecting customer feedback, especially when using structured feedback channels, there needs to be mechanisms in place to process incoming feedback and to interpret the information quickly. Excepting [51], which proposes a procedure where the first thing that developers have to do in the morning is to check if there are problems in their latest submission regarding feedback for daily builds, mechanisms for systematically processing feedback were not elaborated in the primary studies. Nevertheless, the literature does emphasise close collaboration with the customer, especially during requirements elicitation, prioritisation and the definition of user stories and DoD [47, 15]. Moreover, there are warnings about the effects that continuous changes in a product might have upon customers. For example, [49] notes that when doing fast releases, how much and what to change between releases must be seriously considered, as this has a direct impact on the interactions between users and developers. In the same vein, Zade and Choppella [82] highlight that in fast releases, changes in user interfaces need to be provided with serious attention. If the way a user interacts with the software changes considerably between releases, then a negative impact on customer experience is likely.

5.7. Continuous and rapid experimentation

Ten primary studies, published during recent years, make reference to continuous and rapid experimentation [31, 29, 15, 67, 61, 62, 9, 30, 59, 33]. Although the literature lacks a unified definition, continuous and rapid experimentation in the context of CD refers to systematically designing and executing small field experiments to guide product development and accelerate innovation; thus, it aims to base the business and design decisions of product enhancements and new functionality on data rather than on stakeholder opinions, even if they are experts in the area [29].

The 'Stairway to Heaven' model suggested R&D as an experimental system as the last step of its evolutionary path of

software organisations from traditional methods to CD and beyond [62, 61]. It describes a situation where the organisation constantly conducts experiments to guide product development and accelerate innovation and decision-making. To achieve this objective, companies need to adopt a short-cycle innovation process centred on customer feedback and usage data [62, 61]. This information is used to guide the evolution of the system and the actual deployment of new software functionality [62, 61]. For example, Facebook uses A/B (split) testing as an experimental approach to immediately identify user needs and values rather than trying to elicit requirements following the traditional requirements engineering approach [31]. When using A/B testing, randomized experiments are conducted over two or more variants of an enhancement (or similar feature) in order to compare how they are perceived by end-users through statistical hypothesis testing [31, 9]. This experimental approach is largely facilitated by the fact that CD significantly reduces the gap between the company and its customers (see customer involvement theme, Section 5.6) [9, 31].

In order to enable continuous and rapid experimentation, architectural infrastructure for runtime variability of functionality, mechanisms for data collection and rollback mechanisms to revert changes are required [62]. For instance, Rally Software [59] suggested A/B testing with Feature Toggle as a technique to manage and support run-time variability of functionality. Goel et al. [33] describe the development of an infrastructure for fast upgrade of database systems which enabled Facebook to deploy experimental software builds and improvements on a large scale of machines without degrading the systems uptime and availability. Apart from technological requirements, organisational functions, which includes release and product management as well as innovation and R&D, must be well aligned and tightly integrated [62, 61].

It is interesting to observe that continuous experimentation has been proposed not only in the context of web applications in companies such as Facebook or Rally Software, where innovation cycles are naturally shorter, but also in the context of embedded systems. For example, [29] and [15], in the automotive industry, present the innovation experiment system (IES). IES is an evolution of current R&D practices moving from considering innovation as a process internally guided and assessed by the original system manufacturer to a process in which innovation is actually evaluated by real users at scale. However, the literature also recognises the limitations of applying the experimental paradigm in the context of safety critical or other systems that require certification and heavy verification and validation processes [15, 30].

Regarding how continuous and rapid experiments are actually conducted, the CD literature is quite scarce. Fagerholm et al. [30] present an initial model for continuous experimentation composed of an experimentation cycle based on build-measure-learn blocks and their underlying infrastructure. The build-measure-learn blocks structure the activity of conducting experiments and connect product vision, business strategy and technological product development through experimentation. On the other hand, the underlying infrastructure of the model comprises three layers, including roles involved in running the experiment, enabling technical infrastructure and information artefacts that are needed for conducting the experiments.

5.8. Post-Deployment activities

The theme of post-deployment activities refers to those activities that are conducted once the product (or a new feature or enhancement of the product) has been deployed [61]. CD has created a large number of new opportunities not just for observing user behaviours and monitoring how systems and services are being used [24, 9], but also for identifying unexpected patterns and runtime issues [24, 46], monitoring system quality attributes [24, 31, 32] and collecting real-time data to feed both business and technical planning [9]. For instance, Orso et al. [63] proposed an approach to perform different monitoring tasks and collect useful information on their software's behaviour. In addition, continuous monitoring might also aim to monitor metrics related to service-level agreements and system quality attributes, including performance, system availability and operational infrastructure. Goel et al. [33] reports on a very fast, distributed, in-memory database at the heart of Facebook that is extensively used for post deployment activities including advertisement revenue monitoring, performance debugging, as well as real-time analysis of user behaviour and service logs.

As acknowledged by various studies [24, 31, 32, 9], the main objective of continuous monitoring is to constantly monitor and measure both business indicators and infrastructure-related metrics in order to facilitate and improve business and technical decision-making. More importantly, continuous monitoring must always be unobtrusive to users, thought it needs to be visible and accessible to all relevant stakeholders, including development, operation and business people [24, 9]. One of the most significant activities in this sense is post-release testing to ensure successful deployment [1] and also performing critical validation and testing on real users at scale [31]. Dark deployment is used where enterprises deliver new features or services that are invisible to customers and have no impact on the running system. This technique can be used to test system quality attributes and examine them under simulated workload in a real production environment [31, 59]. Another relevant practice, canary deployment, allows enterprises to deliver a new version to a limited user population to test the system under real production traffic and use. The new version is then delivered to the whole user population once it reaches a high enough level of quality [31, 59].

5.9. Agile and lean software development

CD goes beyond agile and lean software development; thus, agile and lean software development methods and practices are the first steps the organisation can take toward CD, e.g. [32, 74]. Hence, ASD methods and practices can be considered as an enabler for CD. However, CD scales ASD practices throughout the whole organisation instead of focusing only on team-level activities. For example, this is noticed in the continuous experimentation approach, in which software is developed based on field experiments with relevant stakeholders, i.e. customers or users [30]. The experimental results are linked throughout

the organisation with a product roadmap as well as managed within a flexible business strategy [30]. Along the same lines, lean software development promotes the consideration of the whole organisation as part of CD: *'If you deliver daily, waste is exposed almost immediately . . . optimizing just a part of the system simply is not an option with daily deployment'* [67]. To accelerate continuous software delivery and achieve agility at scale, Cantor and Royce [18] describe the IBM transformation from conventional engineering governance to economic governance and Bayesian analytics, which integrate governance with agility aspects.

Most of the primary studies discuss in one way or another aspects related to ASD. For example, [9, 67, 57] explain that lean software development supports delivery of a continuous flow of small features into production, as is the aim of CD. Benefield [9] focuses on lean techniques for the SaaS delivery model. Using specific Agile and Lean software development methods appears frequently in the primary studies as well (e.g. using Scrum [7], continuous Scrum [1], pair programming [10], eXtreme Programming [35], or Rugby which is an agile process model including workflows for the continuous delivery of software [47]).

CD also changes the traditional ASD practices and methods into a continuous flow. Continuous ways of working are described in the ASD literature as follows: continuous improvement and employee empowerment, e.g. [57]; CI, e.g. [8] and [47]; continuous delivery, e.g. [47, 46, 59]; continuous delivery of features, e.g. [7]; or the transforming of a release cycle into a continuous flow, e.g. [54] and [1]. For instance, Agarwal [1] reports results from continuous scrum in which bug fixes, minor enhancements and major features are released continuously on a weekly basis by a single development team. Each sprint has three phases, creating a triple-sprint overlap pattern: planning, development and QA. The development team is divided into team members who are responsible and capable of executing each of these phases; hence, there are sub-teams each with a different function, i.e. planning, development and QA. Each sprint is time-boxed into a three-week period. During the first week of a sprint, the product owner and scrum master together with the inputs from the development team formulate a plan for the remainder of the sprint. Thereafter, the planning subteam starts planning the next sprints and the development team starts development on the sprint that was planned by the subteam in the prior week. Similarly, the QA team performs QA on what was developed by the development team in the prior week. Thus, sequential sprints overlap with a phase-lag of a one-week release cycle [1].

5.10. Organizational factors

Many factors related to organisational aspects were also highlighted in the CD literature. We classified these aspects into three groups: integrated corporate functions, transparency and innovative and experimental organisational culture.

5.10.1. Integrated corporate functions

Both empirical and non-empirical studies stressed the integration of the R&D organisation with other corporate functions such as sales, marketing, product management, QA, release and operations. This integration is crucial for fast delivery and deployment. It enables transparency and understanding of the whole picture of product development activities and overcomes corporate constraints that often cause delays in product deliveries, e.g. hand-over delays and communication gaps [35, 7, 50, 67, 40, 31, 32]. A flat R&D organisational structure is common when applying ASD [35, 59]. However, CD demands a greater alignment of the R&D organisation with other corporate functions [62]. For instance, integration of R&D with the operations/maintenance team, also referred to as DevOps, is noted in several studies [31, 16, 32, 46]. Similar to DevOps is the emphasis on the integration between software development and business strategy, termed BizDev [32]. Both DevOps and BizDev focus on achieving a shorter cycle time with increased feedback loops [24, 32].

Several strategies describing how to integrate corporate functions are depicted in the CD literature; however, they are mostly initial proposals. For instance, Neely and Stolt [59] describe the journey that Rally Software followed for tracking the status of work in real time to sales and marketing teams in order to integrate them with R&D. One challenge here is that in addition, marketing strategy needs to be adapted to the CD approach. Using cross-domain competences amongst team members to ensure effective communication and integration with other corporate functions has also been stressed by studies conducted in non-embedded domains [31, 24]. Feitelson et al. [31] highlight that at Facebook, software developers are trained to possess multiple skills, such as abilities in testing and operations, in order to ensure good-quality software all the time without the need for a supporting QA function [31]. Similarly, in [24], software developers also perform activities related to operations functions, in addition to performing QA function activities. However, cross-domain competence may be more challenging in the embedded domain, where extensive knowledge of hardware-related aspects is needed. Additionally, several studies have proposed the use of common practices and platforms for software development and maintenance/operation/release as mechanisms to integrate corporate functions with the R&D organisation [16, 32, 35]. Using shared models such as roadmaps and a feature dependency matrix has also been identified as a mechanism that facilitates effective coordination and interrelationships between solution designers and release managers of complex services [50].

5.10.2. Transparency

Organisational transparency, which is mentioned above as one of the main targets of integrated corporate functions, is a predecessor for building CD into an organisation. Organisational transparency intends to show the bigger picture of scattered development activities in different parts of the organisation, building a common understanding among stakeholders about the development progress and goals [46, 47, 35, 7, 50, 67, 40, 31, 32]. In CD, transparency has an important role in creating the ability to foresee, trace and understand important aspects of product development in real time [47, 59]. Thus, transparency is also an enabler for identifying early risks that

may harm product delivery and for reacting proactively to these risks. For example, it is important to provide mechanisms for visibility in the sales and marketing activities so that it is possible to track the status of the work in real time allowing for appropriate planning activities [59]. As described in Section 5.3, the importance of visualizing testing activities from end to end is highlighted in the literature [60].

In addition, it is noted that making the development progress transparent enables better awareness for team members of their contribution on the delivery of value [74] as well as allowing them to decide what is needed [57] and to take personal responsibility [54]. An important way for build transparency in the context of CD is the use of key performance indicators (KPIs) as metrics to visualise the performance of the organisation. For instance, Staron et al. [74] conducted a case study on a large agile and lean software development project at Ericsson in Sweden, in which KPIs were used to visualise the release readiness across the distributed teams in order to predict the time in weeks to release the product. Visualising the quality of the outcomes by KPIs provided a way to build common awareness of the status among all stakeholders, from designers to unit managers [74]. Therefore, it is important to identify useful metrics in the context of CD when building transparency. Other approaches for building transparency into an organisation are proposed in the CD literature, e.g. traffic lights visualisation is commonly used to display the status of production at any time [46]. Moreover, information radiators and Kanban boards are often used around the work space to ensure that daily progress on a project is completely transparent and available for all to see [57].

5.10.3. Innovative and experimental organizational culture

Many primary studies stress the importance of people-driven development and the need to create an innovative and experimental organisational culture to enable CD. According to the primary studies, learning from experience is more important and benecial than chastising those responsible for a failure. Because humans make errors, some distrust is natural, but attention should be focused on honest communication and learning from mistakes; this improves trust among stakeholders, which enables greater efficiency and also innovativeness. In CD, failures are treated as opportunities for improvement rather than as occasions for assigning blame [31]. Marschall [54] emphasises the importance of the role of individuals in taking responsibility for completing their tasks when producing value to a customer. Personal responsibilities can be used even to substitute specialisation, methodology and formalised procedures created for blaming and self-protection, which have no place in a team of engineers willing to take responsibility for the entire system [31].

The study by Papatheocharous et al. [64] highlights the importance of human factors as a basis for increasing the capabilities of continuous software engineering. In their study they identified that providing tools for developers to improve themselves, and designing and assigning work tasks based on personal qualities may lead to situations where team members are willing to accept more responsibility in managing themselves, and, thus, take responsibility for the project outcome.

Regarding innovation, innovations should be encouraged by breaking the routine with frequent activities aimed toward new innovations. There should be flexibility and breakout times in the daily routines. For instance, hackathons are commonly used in Facebook in order to encourage interaction among different organizational functions from engineers to financial, legal and other departments [31].

6. Analysis of Reported Benefits and Challenges for Continuous Deployment (RQ3)

6.1. Benefits

The literature highlights several benefits from applying CD. The most referenced benefits are shorter time-to-market, increased customer satisfaction, continuous feedback, rapid innovation, narrower test focus, improved release reliability and quality and, improved developer productivity. However, the strength and quality of evidence for these benefits is limited as many claims are based on industry reports (i.e. practitioners' perceptions) or discussed in non-empirical studies. Furthermore, in many cases, benefits are claimed, but no rational or more detailed explanation of the reasons for these benefits is provided in the papers. Nonetheless, the benefits found in primary studies are detailed in the following paragraphs.

The most immediate benefit of applying CD is *shorter time-to-market* through fast and frequent releases [1, 10, 31, 34, 46, 50, 57, 61, 62, 76, 75, 9, 48, 54, 59, 67, 41, 49]. For instance, [31, 54, 59] shortened their delivery cycles from months or weeks to continuous flow or daily deliveries. Similarly, in the context of embedded systems, [76, 75] reduced their release cycles significantly from three months to three weeks. Shorter release cycles enable companies to constantly develop, learn and improve their offerings based on instant customer feedback [62, 48, 59, 67, 41] and thus, companies can quickly learn what customers value and focus on deploying relevant functionalities that meet customers' expectations [67, 62, 61]. Shorter release cycles enable faster feedback about new features and bug fixes, which makes release planning slightly easier (short term v.s long term planning) [41]. Moreover, a higher number of releases provides more marketing opportunities for companies [41].

CD has also been found to increase *customer satisfaction* and enable *continuous customer feedback*. CD allows continual product enhancement and immediate access to new features and bug fixes, which increases customer satisfaction [1, 10, 59, 76, 41]. For example, Neely and Stolt [59] reported, *'We received an email from a customer saying that they had noticed the defect fix and wanted to say a huge thank you for resolving a pain point in the application'*. According to Khomh et al. [41], under a rapid release model, users can adopt new versions of the product faster, bugs are fixed faster and users do not experience significantly more post-release bugs in comparison with the traditional release model.

In addition, customers can evaluate the enhancements and provide feedback immediately and in a continuous way (i.e. continuous customer feedback), which improves communication between the company and its customers [35, 46, 47, 50, 51,

59, 75, 62, 61]. Furthermore, closer interaction with customers enables enterprises to monitor and collect instant field data on their customers and the software's behaviour [24, 31, 32, 9]. The main advantage is that companies have the chance to rapidly sense, understand and improve their offerings based on actionable metrics and data [45, 51].

Apart from customer feedback, continuous and immediate feedback from CI and an automated infrastructure helps to identify and resolve issues more rapidly [31, 34, 37, 51, 59, 32]. For instance, Goodman and Elbaz [34] observed that the CI process shortens the feedback cycle time substantially. Similarly, Humble et al. [37] also reported that build and deployment scripts accelerate rapid feedback not just on the integration of modules of source code but also on problems integrating with the deployment environment and its external dependencies.

Closer relationships with customers further can facilitate *rapid innovation*. Continuous and instant customer feedback allows companies to invest their resources in developing relevant functionalities and innovation initiatives [31, 34]. [61, 62] observed that faster feedback means cheaper development since the R&D organisation can then spend time developing the right things rather than correcting mistakes in functionality.

Narrower test focus appears also in our primary studies [22, 31, 59, 53]. From a technical point of view, CD implies that each deployment introduces only limited amounts of new code. From this perspective, frequent releases with a smaller scope reduces risk [59, 31] and provides a narrower test focus, which more accurately guides quality assurance activities [22, 31, 59, 53]. A narrow scope further allows deeper investigation of the product's active parts and makes issues easier to fix and debug [22, 31, 59, 53].

Several studies [1, 37, 46, 59, 9] also reported that the deployment infrastructure, coupled with intensive automated testing and fast rollback mechanisms, *improves release reliability and quality*. Neely and Stolt [59] reported that automated deployment along with scrutiny of monitoring systems provided safer environments for shipping code. Furthermore, intensive automated tests ensure that new improvements pass all quality assurance procedures, thus leading to a higher quality of releases [1, 9, 59]. Finally, automated deployment processes are reported as also leading to *improve developer productivity* since they allow a single engineer to develop and deploy a new improvement instantly to several services, to verify immediately and rollback to the previous stable version, if required [1, 35, 37, 9].

6.2. Challenges

Regarding challenges in CD, the process of transformation towards CD, customers' unwillingness to receive continuous product updates, increased QA efforts and the challenges of applying CD in embedded domains were often referenced in the literature.

Transforming towards CD is an evolutionary process and requires investment in deployment processes, as well as changes in people's mindset and the general organisation way of working. For example, Neely and Stolt [59] describes how months of preparatory work were needed to get the deployment process streamlined and automated. [59, 16] observed that, if an organisation is not experienced in ASD, a direct transition to CD requires too many changes to handle at one time. Furthermore, this transition requires a change in mindset as people may be afraid to release the new code directly into production environments [54, 59]. One company manager in [54] noted, *'When the release team and I confronted the developers with our new process, releasing a story as soon as it is signed off it scared the hell out of them'*. Moving to CD requires a change in organisational culture, buy-in from all key stakeholders and transparency in the organisation [10, 57, 59, 61]. Lavoie and Merlo [49] point out that fast release cycles might stress third-party developers because of the risk of non-compatibility of extension modules. Thus, human factors, including personality and cognitive aspects, plays a fundamental role in truly achieving continuous delivery. As acknowledged by Papatheocharous et al. [64], in the context of continuous software engineering where organizations are required to develop, deliver and learn in fast and parallel cycles, it is profoundly important to establish an agile thinking culture from the individuals, to teams as well as upper management levels.

Even though customers seem to be more satisfied (see the previous section on benefits), the literature notes *customer unwillingness* to accept CD as another challenge [62, 61, 10, 1, 63]. According to Agarwal [1], typically, customers are reluctant to accept new functionalities mainly because of poor quality of releases. Orso et al. [63] also identified privacy and security concerns about information collected from customers as inhibitors of CD. To address privacy concerns, which also suits monitoring purposes, they suggested that organisations should seek permission from users to gather information. One other inhibitor from a customer point of view is the learning curve that continuous changes (either to the functionality or to the user interface) request from the end-user perspective. Zade and Choppella [82] empirically investigated the impact of changes on end-users. Their results suggest that the learning gap that changes in user interfaces may produce is a crucial factor to be considered when changes are continuously delivered to end-users.

Several studies [1, 40, 46, 54, 56, 59] reported *increased QA efforts* due to difficulties in managing the test automation infrastructure. Similarly, a case study conducted on Mozilla Firefox [53] found that while rapid release has numerous benefits and strongly supports shorter release times, at the same time it increases the test efforts. This stems from the fact that more specialized testers are required to sustain the testing effort in a rapid release model. In addition, CD requires establishing an effective QA process and new mechanisms to ensure backward compatibility of enhancements.

In addition, in the context of the *embedded domain*, Ard et al. [3] reported that physical assets and hardware equipment should also support automation, in general, and CI, in particular, to get benefits from CD as a whole. Bosch and Eklund [15] reported challenges concerning experimentation in software-intensive embedded systems. In particular, the architecture of embedded devices must support mechanisms to add or ex-

change applications when running experiments with minimal impact on the rest of the system. More importantly, the memory and processing footprint, as well as connectivity aspects, need to be considered carefully. Infrastructure requires the support of rollback mechanisms and immediate reversion to safe versions. Furthermore, since, in experimental scenarios, the infrastructure needed to keep track of individual devices, security and privacy issues require extra consideration [15]. Finally, Trimble and Webster [75] elaborated in the domain of mission critical systems. Safety issues require updates be planned and this prevents near real-time value delivery (e.g. users need to be notified and trained for new capabilities beforehand, to use them in mission-operation environments).

Finally, other challenges found in the literature, although they do not appear very frequently, include: lack of trust in software quality [62], difficulty in managing various configurations and run-time environments [40, 56] and natural tensions between the desire to deliver functionalities quickly and the need for reliable products [8, 17]. In addition, further difficulties exist in release planning and managing the roadmap in a fast-paced environment [50] and risks associated with gathering user feedback from a limited population (i.e. minority) that may constrain the software's evolution or even mislead product development [45].

7. Research Gaps and Opportunities for Future Research (RQ4)

The topic of CD appears to be highly relevant to the industry. Practitioners have contributed heavily to its body of knowledge, and the results of the quality assessment demonstrate the great significance of CD to the industry. However, with regard to empirical rigor, the quality assessment's results are low (see section 4.4). In addition, looking at the pertinence facet (see Table B.9, Appendix B), less than half of the studies (22 papers) are entirely dedicated to CD. Six of these produced contributions in the form of advice and implications, and three in the form of lessons learned. This is not necessarily negative as CD encompasses many different aspects. Still, more research that fully focused on CD is needed. In general, although the topic appears to be very promising, research on CD seems to be still in its infancy, which promises a range of new opportunities for researchers.

Each of the 10 identified themes represents opportunities for future research. However, the themes are explored at different levels, offering different research opportunities. The research to date has tended to focus on factors such as continuous testing and QA (31 papers), fast and frequent release (28 papers), automation (24 papers, mainly in CI), and agile and lean software development (22 papers). However, only a small number of studies have dealt with aspects such as flexible design and architecture (9 papers), continuous and rapid experimentation (10 papers), and customer involvement (12 papers) in the context of CD. More concrete opportunities for future research include:

- Continuous and rapid experimentation is an emerging research topic with many possibilities for future work. Most papers in this area offer theoretical proposals that have yet to be adequately validated (e.g. [15, 30]). In particular, technical challenges [15], challenges with large scale experimentation [15, 30], business implications of continuous and rapid experimentation [15], privacy issues when running experiments with customers [15], and the process of transforming towards continuous experimentation itself [61] are identified as areas for future research.

- Technical infrastructure for supporting CD. Incremental deployment is referred to in some of the industry reports in companies such as Facebook [33] and Rally [59]. However, how it is done in practice is unclear. Although some techniques and tools are mentioned in the literature, such as canary deployment and dark deployment [31, 59], they are briefly introduced without going into much detail on how they are actually used in practice. This topic is especially relevant when considering the importance of system availability and quality aspects related to the version of the system that is deployed at any given moment. Another example is Scuba as a solution to support monitoring of post-deployment user behaviour at Facebook [33]. Although, the Scuba database is briefly introduced, the kind of data that is collected as well as mechanisms to analyse it and feed it back to the development process are not described. Moreover, why certain technologies are selected over other similar existing solutions in the market has not been analysed. Thus, which are the most suitable technologies under certain conditions is unclear.

- Although customer involvement is emphasised in many primary studies, we discovered that the tasks required for making continuous and effective use of customer feedback are underdeveloped. Besides the need for mechanisms to identify representative customers and infrastructure for collecting customer data, a clear research gap appears in solutions for processing incoming feedback and quickly interpreting the information. Further investigation is needed on new approaches to express and validate assumptions from user feedback, as the software evolves [45], and privacy and security concerns that may inhibit customer feedback collection [63].

- Regarding flexible and robust software architectures, some mechanisms for balancing stability and speed have been proposed in the literature (e.g. continuous assessment of quality attributes through prototyping or automated approaches and rapid architecture trade-off analysis). Nonetheless, the natural tension between the desire to deliver functionalities quickly and the need for reliable products is still a challenge for CD [8, 17]. Investigating measurement systems for managing technical debt [8], risk assessment methods for CD [22], and stability-triggers-speed scenarios (i.e. stability causes a refocus on speed) [8] are areas proposed by the primary studies for future research.

- Continuous planning also seems to play an important role

in achieving CD. Based on our results, continuous planning is not commonly adopted and applied throughout the entire organisation. It is currently connected to prioritisation [76] or involves only a certain level of planning, mainly release planning (e.g. Continuous Scrum [1]). Fitzgerald and Stol [32] found that the current focus of continuous planning in CD is mainly on what emerges from ASD, which it is related to sprint iterations or, at best, software releases. Thus, empirical research rarely describes how continuous planning is conducted at different organizational levels and how the information from plans is visible at different levels of planning.

- Similarly, despite the identified need for, and importance of, integrating R&D with other corporate functions, there is very little empirical research that evaluates emerging approaches such as DevOps and BizDez [32]. Research is needed to identify and empirically evaluate mechanisms to facilitate collaboration between different organizational functions, not just at the organisational level but also at the technical level.

- We also believe that automation is an important area for future research. The goal of automation is to eliminate all manual steps from build to deployment processes. The literature shows different research gaps regarding automation. For example, GUI testing [1] is identified as a lead indicator of bugs that typically appear in production environments. However, GUI testing still requires manual observation because of difficulties in automatically identifying errors. This is an area for possible future research. In particular, Agarwal [1] pointed out the possibility of utilising advanced techniques in video analytics to enable fully automated GUI testing. Similarly, dealing with variability is considered an area that calls for future work, particularly when automating the deployment of software components [79].

- The application of CD in contexts that are different from web applications (i.e. embedded systems) presents a clear opportunity for future research as different domains have different constraints when applying CD. As identified in Section 4, when reviewing at the study domain, the majority of studies have been conducted in the web application domain. Organisations in the area of web applications are currently able to implement CD and even deploy many new versions per day (e.g. [31, 54, 59]). However, this goal is still a formidable challenge for domains such as in embedded systems. Although the application of CD in embedded systems has attracted significant interest over the last four years (see Figure 7), many challenges are still apparent for CD in embedded systems. For example, in the area of post-deployment activities, those primary studies that have used or discussed post-deployment activities focus on cloud- or web-based domains. However, unlike web-based companies, not all software development companies have access to a huge amount of customers. Another example is the case of software ecosystems, which are becoming increasingly popular. The implications of using CD with different contributors or when interdependent systems need to be coordinated are not clear.

- The implications of human aspects with CD are also underdeveloped. With the adoption of CD, individuals are given more responsibility for systems under development. For example, in an optimal situation, an individual developer could deploy a new version of the system directly to customers in just a couple of automated steps. However, the individual should be also ready to take on that responsibility. Papatheocharous et al. [64] argue, *'models considering the individuals personal traits to accommodate the objective of holistic continuous software engineering'* are scarce. Models such as Stairway to Heaven [61] provide guidance on organizational levels, but how CD relates to personality and cognitive factors remains an unexplored research area.

To conclude, besides specific areas that require future research, we find that CD needs an increase in both the number and rigour of empirical studies. Recent case studies, such as those that focus on Mozilla Firefox [4, 41, 49, 53] or the experiment conducted by Zade and Choppella [82], provide rigorous contributions to illustrate the impact of CD on product/process quality as well as on the end customer. However, these studies explicitly mention limitations in generalizing results. Thus, similar studies are needed. Industry participation is essential in order to obtain the right data set that allows to produce reliable results (particularly in research aspects that are the hardest to research in terms of data availability such as those that involve customers). Moreover, although several benefits related to CD are mentioned in the literature, they are mainly referred to in industry reports and non-empirical research. Thus, empirical studies and scientific evidence that confirm or refute these benefits as well as studies that analyse cause-effect relationship between identified factors might be investigated in future work. Similarly, emerging approaches for implementing CD require more rigorously empirical evaluations that can help to generalize results where possible. In particular, the lack of context descriptions in the primary studies makes comparing different studies and providing more generalizable results difficult.

8. Comparison to Related Reviews

In order to put the findings of this work into the context of literature and highlight the contributions made by the present work, we compare our results with related literature reviews as presented in Section 2.2. Since ASD and specific agile practices are not the center of our work, the comparison focuses mainly on our findings and the findings of the semi-systematic literature study conducted by Mäntylä et al. [52].

Both studies differ slightly in terms of focus and scope. Still, the benefits and challenges of CD, are at the heart of both reviews. The focus of Mäntylä et al. [52] is mainly on the impact of rapid releases on testing processes; additionally, the results

are extended with a semi-systematic literature review, which includes 24 primary studies, to investigate the prevalence of rapid release, spanning its origination, enablers, benefits and problems. Our study, on the other hand, seeks to collect and synthesize all relevant studies on the topic of CD in a systematic manner in order to characterize this phenomenon by identifying recurrent themes and exploring the multifaceted nature of CD. We found 50 primary studies relevant in the context of CD. Our study extends Mäntylä et al.'s semi-systematic literature review in that its focus is entirely on CD and its recurrent themes; it includes scientific studies published up to June 2014 (comprising theoretical studies as well), and takes into account all the aspects of the systematic literature review method in order to ensure reliable results, such as peer reviews of each research step, systematic data extraction and data synthesis, and quality assessment of primary studies. These aspects are absent in the study conducted by Mäntylä et al. [52].

In a similar vein to Mäntylä et al. [52], we found that CD has had an important impact on the software industry in recent years (see Figure 8). Indeed, many of our primary studies, especially those published in recent years, discuss specific cases of real companies, such as Facebook [33, 31], Firefox [4, 41, 49, 53], IBM [18, 35, 40, 50, 16], Rally Corporation [59], Ericsson [74, 2], Volvo [2] or NASA [76, 75]. The fact that CD has been adopted in such a diverse set of companies provides testimony that CD is being applied in many different domains. Our primary studies include evidence of CD being adopted even in domains such as safety critical (e.g., [76, 75, 3]) and science systems [81]. Thus, our results confirm the claim made by Mäntylä et al. [52] that CD can be part of any software development domain. In addition, our study presents an overview of the state-of-the-art of CD (Section 4), which, as presented in Section 7, reveals that the body of knowledge of CD is in an exploratory stage and its scientific evidence must be improve.

Mäntylä et al. [52] also discuss enablers as *'accelerating factors that facilitate the adoption of rapid releases.'* We did not specifically focus on enablers as preconditions of CD, as we had a wider scope to investigate recurrent factors in the literature related to CD. Mäntylä et al. [52] found that *'parallel development with tools enabling easy automatic deployment and testing, and with proactive customers and product managers'* are enablers of rapid releases. However, the actual mechanisms to achieve frequent releases, tools employed to support automation, and strategies for involving customers in the product development process were not further elaborated. Our study extends Mäntylä et al.'s preliminary identification of important aspects enabling rapid software releases by creating a schema that classifies recurrent aspects in the CD literature and identifies concrete frameworks, methods and tools that support CD in practice.

Our classification schema comprises 10 underpinning factors that define CD. Factors already identified in the study by Mäntylä et al. remain in our classification schema under the following themes: fast and frequent releases, automation, continuous testing and quality assurance, and customer involvement. However, we also found other aspects that are relevant in the context of CD, such as flexible product design and architecture, configuration management, continuous and rapid experimentation, post-deployment activities, agile and lean software development and organizational factors, such as integrated corporative functions, transparency, and the need of creating an innovative and experimental organizational culture. Indeed, these factors are not independent of each other, but have overlaps and synergies between them that help support CD in practice. For example, continuous testing and quality assurance is critical in CD because the product is continuously deployed to the end customer. Thus, mechanisms that assure quality, such as automated testing, are essential in the context of CD. However, automation is much more than just automating the testing process; it also aims to minimise the manual overhead by automating the entire end-to-end workflow, including aspects such as automation of the delivery process, configuration management, etc. In a similar vein, experimentation, as a way of making decisions based on objective data rather than 'gurus' opinions that may lead to incorrect interpretations of the reality, is emphasized in the CD body of knowledge. Moreover, it is considered to be a common-sense approach to proactively involve customers and customers' views in the development process.

Similar to the findings of Mäntylä et al., we found that tool support is very important. Table B11 in Appendix B summarizes the concrete tools and methods that, according to our primary studies, software intensive companies use to apply CD. Moreover, our findings are aligned with the claim that *'release length simply appeared as a variable in the release models without providing insights regarding rapid releases'* [52]. Thus, the release length in our primary studies varied from a few hours to a few weeks, depending mainly on the application domain.

With regard to the benefits and challenges of using CD, our findings are well aligned with the results already reported by Mäntylä et al. [52]. Shorter time-to-market, increased customer satisfaction, continuous rapid feedback, and narrower test focus were among the benefits identified in both reviews. According to the primary studies in our review, developers are more productive when using CD as a direct consequence of the automation of deployment processes. According to the review conducted by Mäntylä et al., efficiency also increases, but as a result of time-pressure. In addition, we found that the deployment infrastructure supporting automation and fast rollback mechanisms improves release reliability and quality. Mäntylä et al. note that using CD makes it easier to monitor progress and quality.

Finally, with regard to the challenges of using CD, we found four main aspects were reported in the literature: the process of transforming towards CD, customers' unwillingness to accept continuous updates, increased QA efforts, and the application of CD in the embedded domain. Although Mäntylä et al. found customer unwillingness to be a challenge with rapid releases, the challenges identified in their review mainly focus on testing areas, such as conflicting goals between rapid release and achieving high reliability and test coverage. Still, as described above, these findings need to be confirmed through more rigorous scientific studies as, at the moment, they are mainly based

on practitioner perceptions.

9. Conclusion

This study provides a structured understanding of the body of CD knowledge, together with a systematically collected list of references relevant to CD. By using the systematic mapping method, we identified, classified and analysed primary studies related to CD based on a survey of the literature conducted in June 2014. The most important findings of this review, which are organized according to the study's research questions, are summarised below.

- *RQ1: What is the current state of the research pertaining to CD in the context of software intensive products and services?* From the 21,382 retrieved documents, we identified 50 primary studies relevant to CD. Most of these primary studies are industry reports (36 percent) and case studies (24 percent). Other research methods that were used include action research (4 percent), and grounded theory, mixed method, design science and experiment, each with just two percent of the studies. Overall, 42 percent of the primary studies contributed to CD in the context of web/Internet-based services and applications, and 24 percent were related to embedded systems. 8 percent of the studies focused on desktop applications and the domains of the remaining 26 percent of the primary studies were not clearly described. In addition, 8 percent of primary studies contributed to CD theoretically (without empirical evidence). A rigour and relevance analysis indicated that 37 primary studies exhibited high relevance; however, of these, only 14 studies showed high rigour and 23 studies had less than moderate rigour. This provides clear evidence that scientific contributions in the literature on CD are currently of high relevance but medium-low rigour, which calls for more meticulous research.

- *RQ2: What are the main factors that characterise CD in the context of software intensive products and services (sub: what do researchers mean when they refer to CD)?* A general consensus exists among most authors of the literature surveyed that CD refers to the ability of organisations to release software functionalities to customers quickly and frequently, soon after each new functionality is developed. However, these authors tended to use the concept interchangeably with continuous delivery (although as it is described in Section 2.1, they have different meanings).

  As a result of our analysis of the 50 primary studies, we identified 10 recurrent themes or factors related to CD. Each of the 10 factors was analysed and presented in detail in Section 5. The factors include: (1) fast and frequent release, (2) flexible product design and architecture, (3) continuous testing and quality assurance, (4) automation (of build and test (CI), deployment/delivery/release processes and configuration of deployment environments), (5) configuration management, (6) customer involvement, (7) continuous and rapid experimentation, (8) post-deployment activities, (9) agile and lean software development and (10) organisational factors, including integrated corporate functions, transparency and an innovative and experimental organisational culture.

- *RQ3. What are the reported benefits and challenges in association with CD in the context of software intensive products and services?* A number of benefits and challenges related to CD were identified. Transforming towards CD is identified as challenging, requiring significant investment in deployment processes, as well as in changes in people's mindsets and organisations' general way of working. In addition, an unwillingness by some customers to accept new functionality, a need to increase efforts in QA and the application of CD in the context of embedded systems were identified as significant challenges. However, CD also poses potential benefits for organisations, such as shortening their time-to-market by reinforcing the organisations' capabilities to release software functionalities to customers more quickly and frequently, an increase in customer satisfaction with the continual deployment of valuable product enhancements and obtaining immediate feedback during the development process, particularly from customers, which helps to guide software development activities and quickly identifies potential problems. In addition, CD also appears to facilitate rapid innovation through experimentation and continuous and instant customer feedback and to improve release reliability and quality, in part, due to a narrower test focus and the extensive use of automation. However, these findings have to be carefully interpreted, as the empirical evidence is limited mainly to practitioners' perceptions.

- *RQ4. What are the research gaps in the area of CD in the context of software intensive products and services?* Finally, a plethora of venues for future research, due to the topic's freshness and its industrial relevance were identified. Rigorous scientific contributions are clearly needed, particularly those based on empirical evidence evaluating the benefits of CD. In addition, we identified a number of research gaps within the 10 themes identified. Continuous and rapid experimentation is an emerging research topic with many avenues for future work. Similarly, a clear research gap exists for mechanisms to use customer feedback in the most appropriate way so that information can be quickly interpreted. In continuous testing and QA, research contributions on mechanisms for implementing automated GUI testing are required, as well as investigations assessing technical debt in the context of CD. In addition, topics such as continuous planning, automation and integrated corporate functions (e.g. DevOps and BizDez), all appear to be especially relevant. In addition to the above specific areas of future research, CD research needs to increase in both number and, especially, rigour of empirical studies.

This research has been carried out within the Digile Need for Speed program, and it has been partially funded by Tekes (the Finnish Funding Agency for Technology and Innovation).

10. References

[1] Agarwal, P., 2011. Continuous scrum: agile management of saas products. In: Proceedings of the 4th India Software Engineering Conference. ACM, pp. 51–60. ***[PS1]**.

[2] Antinyan, V., Staron, M., Meding, W., Osterstrom, P., Wikstrom, E., Wranker, J., Henriksson, A., Hansson, J., 2014. Identifying risky areas of software code in agile/lean software development: An industrial experience report. In: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on. IEEE, pp. 154–163. ***[PS2]**.

[3] Ard, J., Davidsen, K., Hurst, T., 2014. Simulation-based embedded agile development. Software, IEEE 31 (2), 97–101. ***[PS3]**.

[4] Baysal, O., Kononenko, O., Holmes, R., Godfrey, M. W., 2012. The secret life of patches: A firefox case study. In: Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE, pp. 447–455. ***[PS4]**.

[5] Beck, K., 2000. Extreme programming explained: embrace change. Addison-Wesley Professional.

[6] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al., 2001. The agile manifesto.

[7] Bellomo, S., Nord, R. L., Ozkaya, I., 2013. Elaboration on an integrated architecture and requirement practice: Prototyping with quality attribute focus. In: Twin Peaks of Requirements and Architecture (TwinPeaks), 2013 2nd International Workshop on the. IEEE, pp. 8–13. ***[PS5]**.

[8] Bellomo, S., Nord, R. L., Ozkaya, I., 2013. A study of enabling factors for rapid fielding combined practices to balance speed and stability. In: Software Engineering (ICSE), 2013 35th International Conference on. IEEE, pp. 982–991. ***[PS6]**.

[9] Benefield, R., 2009. Agile deployment: Lean service management and deployment strategies for the saas enterprise. In: System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on. IEEE, pp. 1–5. ***[PS7]**.

[10] Blotner, J. A., 2002. Agile techniques to avoid firefighting at a start-up. In: OOPSLA 2002 Practitioners Reports. ACM, pp. 1–ff. ***[PS8]**.

[11] Boehm, B., 2002. Get ready for agile methods, with care. Computer 35 (1), 64–69.

[12] Boehm, B., 2006. A view of 20th and 21st century software engineering. In: Proceedings of the 28th international conference on Software engineering. ACM, pp. 12–29.

[13] Boehm, B. W., 1988. A spiral model of software development and enhancement. Computer 21 (5), 61–72.

[14] Bosch, J., 2012. Building products as innovation experiment systems. In: Software Business. Springer, pp. 27–39.

[15] Bosch, J., Eklund, U., 2012. Eternal embedded software: Towards innovation experiment systems. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change. Vol. 7609. Springer Berlin Heidelberg, pp. 19–31. ***[PS9].**

[16] Brown, A. W., Ambler, S., Royce, W., 2013. Agility at scale: economic governance, measured improvement, and disciplined delivery. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 873–881. ***[PS10]**.

[17] Brown, N., Nord, R. L., Ozkaya, I., Pais, M., 2011. Analysis and management of architectural dependencies in iterative release planning. In: Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on. IEEE, pp. 103–112.***[PS11]**.

[18] Cantor, M., Royce, W., Jan 2014. Economic governance of software delivery. Software, IEEE 31 (1), 54–61. ***[PS12]**.

[19] Castells, M., 2011. The rise of the network society: The information age: Economy, society, and culture. Vol. 1. John Wiley & Sons.

[20] Causevic, A., Sundmark, D., Punnekkat, S., 2011. Factors limiting industrial adoption of test driven development: A systematic review. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on. IEEE, pp. 337–346.

[21] Claps, G. G., Svensson, R. B., Aurum, A., 2015. On the journey to continuous deployment: Technical and social challenges along the way. Information and Software Technology 57, 21–31.

[22] Comas, J., Mostashari, A., Mansouri, M., Turner, R., 2011. A software deployment risk assessment heuristic for use in a rapidly-changing business-to-consumer web environment. International Journal of Software Engineering & Its Applications 5 (4), ***[PS13]**.

[23] Cruzes, D. S., Dyba, T., 2011. Recommended steps for thematic synthesis in software engineering. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. IEEE, pp. 275–284.

[24] Cukier, D., 2013. Devops patterns to scale web applications using cloud services. In: Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity. ACM, pp. 143–152. ***[PS14]**.

[25] Dingsøyr, T., Nerur, S., Balijepally, V., Moe, N. B., 2012. A decade of agile methodologies: Towards explaining agile software development. Journal of Systems and Software 85 (6), 1213–1221.

[26] Dybå, T., Dingsøyr, T., 2008. Empirical studies of agile software development: A systematic review. Information and software technology 50 (9), 833–859.

[27] Eck, A., Uebernickel, F., Brenner, W., 2014. Fit for continuous integration: How organizations assimilate an agile practice.

[28] Eisenhardt, K. M., Martin, J. A., 2000. Dynamic capabilities: what are they? Strategic management journal 21 (10-11), 1105–1121.

[29] Eklund, U., Bosch, J., 2012. Architecture for large-scale innovation experiment systems. In: Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on. IEEE, pp. 244–248. ***[PS15]**.

[30] Fagerholm, F., Guinea, A. S., Mäenpää, H., Münch, J., 2014. Building blocks for continuous experimentation. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE 2014), Hyderabad, India. ***[PS16]**.

[31] Feitelson, D., Frachtenberg, E., Beck, K., 2013. Development and deployment at facebook. IEEE Internet Computing, 1. ***[PS17]**.

[32] Fitzgerald, B., Stol, K.-J., 2014. Continuous software engineering and beyond: trends and challenges. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, pp. 1–9. ***[PS18]**.

[33] Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Ul Haq, F., Wiener, J., 2014. Fast database restarts at facebook. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, pp. 541–549. ***[PS19]**.

[34] Goodman, D., Elbaz, M., 2008. " it's not the pants, it's the people in the pants" learnings from the gap agile transformation what worked, how we did it, and what still puzzles us. In: Agile, 2008. AGILE'08. Conference. IEEE, pp. 112–115. ***[PS20]**.

[35] Gotel, O., Leip, D., 2007. Agile software development meets corporate deployment procedures: stretching the agile envelope. In: Agile Processes in Software Engineering and Extreme Programming. Springer, pp. 24–27. ***[PS21]**.

[36] Humble, J., Farley, D., 2010. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education.

[37] Humble, J., Read, C., North, D., 2006. The deployment production line. In: Agile Conference, 2006. IEEE, pp. 6–pp. ***[PS22]**.

[38] Ivarsson, M., Gorschek, T., 2011. A method for evaluating rigor and industrial relevance of technology evaluations. Empirical Software Engineering 16 (3), 365–395.

[39] Järvinen, J., Huomo, T., Mikkonen, T., Tyrväinen, P., 2014. From agile software development to mercury business. In: Software Business. Towards Continuous Value Delivery. Springer, pp. 58–71.

[40] Kalantar, M., Rosenberg, F., Doran, J., Eilam, T., Elder, M., Oliveira, F., Snible, E., Roth, T., 2014. Weaver: Language and runtime for software defined environments. IBM Journal of Research and Development 58 (2), 1–12. ***[PS23]**.

[41] Khomh, F., Dhaliwal, T., Zou, Y., Adams, B., 2012. Do faster releases improve software quality? an empirical case study of mozilla firefox. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, pp. 179–188. ***[PS24]**.

[42] Kitchenham, B., 2010. Whats up with software metrics?–a preliminary

mapping study. Journal of systems and software 83 (1), 37–51.

[43] Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Tech. rep., Technical report, EBSE Technical Report EBSE-2007-01.

[44] Kitchenham, B. A., Budgen, D., Pearl Brereton, O., 2011. Using mapping studies as the basis for further research–a participant-observer case study. Information and Software Technology 53 (6), 638–651.

[45] Ko, A. J., Lee, M. J., Ferrari, V., Ip, S., Tran, C., 2011. A case study of post-deployment user feedback triage. In: Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM, pp. 1–8. ***[PS25]**.

[46] Krusche, S., Alperowitz, L., 2014. Introduction of continuous delivery in multi-customer project courses. In: Companion Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 335–343. ***[PS26]**.

[47] Krusche, S., Alperowitz, L., Bruegge, B., Wagner, M. O., 2014. Rugby: an agile process model based on continuous delivery. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, pp. 42–50. ***[PS27]**.

[48] Lacoste, F. J., 2009. Killing the gatekeeper: Introducing a continuous integration system. In: Agile Conference, 2009. AGILE'09. IEEE, pp. 387–392. ***[PS28]**.

[49] Lavoie, T., Merlo, E., 2013. How much really changes?: a case study of firefox version evolution using a clone detector. In: Proceedings of the 7th International Workshop on Software Clones. IEEE Press, pp. 83–89. ***[PS29]**.

[50] Ludwig, H., Cappi, J., Becker, V., Stewart, B., Meade, S., 2014. Integrating service release management with service solution design. In: Service-Oriented Computing–ICSOC 2013 Workshops. Springer, pp. 28–39. ***[PS30]**.

[51] MacCormack, A., 2001. How internet companies build software. MIT Sloan Management Review 42 (2), 75–84. ***[PS31]**.

[52] Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., Petersen, K., 2014. On rapid releases and software testing: a case study and a semi-systematic literature review. Empirical Software Engineering, 1–42.

[53] Mantyla, M. V., Khomh, F., Adams, B., Engstrom, E., Petersen, K., 2013. On rapid releases and software testing. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE, pp. 20–29. ***[PS32]**.

[54] Marschall, M., 2007. Transforming a six month release cycle to continuous flow. In: Agile Conference (AGILE), 2007. IEEE, pp. 395–400. ***[PS33]**.

[55] Merisalo-Rantanen, H., Tuunanen, T., Rossi, M., 2005. Is extreme programming just old wine in new bottles: A comparison of two cases. Journal of Database Management (JDM) 16 (4), 41–61.

[56] Meyer, S., Healy, P., Lynn, T., Morrison, J., 2013. Quality assurance for open source software configuration management. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on. IEEE, pp. 454–461. ***[PS34]**.

[57] Middleton, P., Joyce, D., 2012. Lean software management: Bbc worldwide case study. Engineering Management, IEEE Transactions on 59 (1), 20–32. ***[PS35]**.

[58] Nagy, A., Njima, M., Mkrtchyan, L., 2010. A bayesian based method for agile software development release planning and project health monitoring. In: Intelligent Networking and Collaborative Systems (INCOS), 2010 2nd International Conference on. IEEE, pp. 192–199. ***[PS36]**.

[59] Neely, S., Stolt, S., 2013. Continuous delivery? easy! just change everything (well, maybe it is not that easy). In: Agile Conference (AGILE), 2013. IEEE, pp. 121–128. ***[PS37]**.

[60] Nilsson, A., Bosch, J., Berger, C., 2014. Visualizing testing activities to support continuous integration: A multiple case study. In: Agile Processes in Software Engineering and Extreme Programming. Springer, pp. 171–186. ***[PS38]**.

[61] Olsson, H. H., Alahyari, H., Bosch, J., 2012. Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on. IEEE, pp. 392–399 ***[PS39]**.

[62] Olsson, H. H., Bosch, J., Alahyari, H., 2013. Towards r&d as innovation experiment systems: A framework for moving beyond agile software development. In: IASTED Multiconferences-Proceedings of the IASTED International Conference on Software Engineering, SE 2013. pp. 798–805. ***[PS40]**.

[63] Orso, A., Liang, D., Harrold, M. J., Lipton, R., 2002. Gamma system: Continuous evolution of software after deployment. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA '02. pp. 65–69. ***[PS41]**.

[64] Papatheocharous, E., Belk, M., Nyfjord, J., Germanakos, P., Samaras, G., 2014. Personalised continuous software engineering. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, pp. 57–62. ***[PS42]**.

[65] Paternoster, N., Giardino, C., Unterkalmsteiner, M., Gorschek, T., Abrahamsson, P., 2014. Software development in startup companies: A systematic mapping study. Information and Software Technology.

[66] Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. EASE'08. British Computer Society, Swinton, UK, UK, pp. 68–77.

[67] Poppendieck, M., Cusumano, M. A., 2012. Lean software development: A tutorial. Software, IEEE 29 (5), 26–32. ***[PS43]**.

[68] Ries, E., 2011. The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses. Random House LLC.

[69] Rodríguez, P., Markkula, J., Oivo, M., Turula, K., 2012. Survey on agile and lean usage in finnish software industry. In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, pp. 139–148.

[70] Scacchi, W., Alspaugh, T. A., 2013. Processes in securing open architecture software systems. In: Proceedings of the 2013 International Conference on Software and System Process. ACM, pp. 126–135. ***[PS44]**.

[71] Schwaber, K., 2004. Agile project management with Scrum. Vol. 7. Microsoft press Redmond.

[72] Shaw, M., 2003. Writing good software engineering research papers: minitutorial. In: Proceedings of the 25th international conference on software engineering. IEEE Computer Society, pp. 726–736.

[73] Ståhl, D., Bosch, J., 2014. Modeling continuous integration practice differences in industry software development. Journal of Systems and Software 87, 48–59.

[74] Staron, M., Meding, W., Palm, K., 2012. Release readiness indicator for mature agile and lean software development projects. In: Agile Processes in Software Engineering and Extreme Programming. Springer, pp. 93–107. ***[PS45]**.

[75] Trimble, J., Webster, C., 2012. Agile development methods for space operations. In: The 12th International Conference on Space Operations. p. ***[PS46]**.

[76] Trimble, J., Webster, C., 2013. From traditional, to lean, to agile development: Finding the optimal software engineering cycle. In: System Sciences (HICSS), 2013 46th Hawaii International Conference on. IEEE, pp. 4826–4833. ***[PS47]**.

[77] Turhan, B., Layman, L., Diep, M., Erdogmus, H., Shull, F., 2010. How effective is test-driven development. Making Software: What Really Works, and Why We Believe It, 207–217.

[78] Unterkalmsteiner, M., Gorschek, T., Islam, A. M., Cheng, C. K., Permadi, R. B., Feldt, R., 2012. Evaluation and measurement of software process improvementa systematic literature review. Software Engineering, IEEE Transactions on 38 (2), 398–424.

[79] Van Der Storm, T., 2005. Continuous release and upgrade of component-based software. In: Proceedings of the 12th international workshop on Software configuration management. ACM, pp. 43–57. ***[PS48]**.

[80] Vitalari, N., Shaughnessy, H., 2012. The Elastic Enterprise: the new manifesto for business revolution. Olivet Publishing.

[81] Wicenec, A., Parsons, R., Kitaeff, S., Vinsen, K., Wu, C., Nelson, P., Reed, D., 2012. Distributed agile software development for the ska. In: SPIE Astronomical Telescopes+ Instrumentation. International Society for Optics and Photonics, pp. 845106–845106. ***[PS49]**.

[82] Zade, H., Choppella, V., 2012. Functionality or user interface: Which is easier to learn when changed? In: Intelligent Human Computer Interaction (IHCI), 2012 4th International Conference on. IEEE, pp. 1–6. ***[PS50]**.

[83] Zhang, H., Ali Babar, M., 2013. Systematic reviews in software engineering: An empirical investigation. Information and Software Technology 55 (7), 1341–1354.

Table A.9: Extracted Data - Primary Study Properties

Category	Description
P1: General type of paper (adapted from [42]).	
Empirical	If the paper bases its findings on empirical evidence (exploration of the phenomenon of CD, explanation of some aspect related to CD, evaluation of a CD technique, etc.). Therefore, the source of knowledge of the paper is acquired by means of observation or experimentation. Observations can be carried out by using different data collection methods such as direct observation of the phenomenon or interviews, surveys, focus groups, etc.
Theoretical	If the paper is descriptive and discusses some issue (theories, frameworks, or underlying concepts) and may (but not always) consider some theoretical issues concerning CD. It does not include an empirical study of the issue being discussed. Typically tools and framework that at not empirically validated as well as conceptual and mathematical analyses are included in this category.
Both	The paper is a mixed theoretical and empirical paper. Typically papers that develop techniques or frameworks with the intention of CD and provide some empirical evaluation or demonstration of the technique are included in this category.
P2: Research method (adapted from [78]).	
Case study	If one of the following criteria applies: 1) The study declares one or more research questions which are answered (completely or partially) by applying a case study. 2) The study empirically evaluates a theoretical concept by applying it in a case study (without necessarily explicitly stating research questions, but having a clearly defined goal).
Industry report	If the focus of the study is directed toward reporting industrial experiences without stating research questions or a theoretical concept which is then evaluated empirically. Usually these studies do not mention any research method explicitly.
Experiment	If the study conducts an experiment and clearly defines its design (variables, control group, treatment, etc.).
Survey	If the study collects quantitative and/or qualitative data by means of a questionnaire or interviews. In a survey study a sample that is representative of the population is studied in order to generalize results from the sample to the whole population (opposite to a case study in which only one or a limited number of cases is considered).
Action research	If the study states this research method explicitly.
Design science	If the study states this research method explicitly.
Grounded theory	If the study states this research method explicitly.
Mixed method	If the study uses multiple methods for data collection.
Not stated	If the study does not define the applied research method and it cannot be derived or interpreted from reading the paper.
Opinion paper	If the paper expresses the personal opinion of an author about CD or whether a certain aspect of CD is good or bad, or how it should be applied. The paper does not rely on related work and research methodologies and does not explicitly describe industrial experiences.
P3: Contribution (adapted from [65] and [72]).	
Model	Representation of an observed reality by concepts or related concepts after a conceptualization process.
Theory	Construct of cause-effect relationships of determined results.
Framework/Method	Method or technique related to constructing software or managing development processes. Commonly it involves better ways to do some task.
Guidelines	List of advises, synthesis of the obtained research results.
Lessons learned	Set of outcomes, directly analysed from the obtained research results.
Advice/Implications	Discursive and generic recommendation, deemed from personal opinions.
Tool	Technology, program or application used to create, debug, maintain or support software development processes.
P5: Pertinence (inspired by [65]).	
Full	The main focus of the study is CD. The three characteristic of CD are noticeable in the study (deployment, continuity and speed).
Partial	The study is partially related to CD. The study supports CD or focuses on an aspect that is important in the context of CD but CD as a whole is not its main focus.
Marginal	The study is marginally related to CD. CD is mentioned in the study but the main research focus of the study is different from CD.

Appendix A. Extracted Data - Primary Study Properties

Appendix B. Systematic Map Overview

Table B.10: Systematic map overview

PS	Research Method	Contribution	Domain	Pertinence	Rigour	Relev.
[1]	Industry report	Framework/method	Web/services	Full	1.5	4
[8]	Grounded theory	Theory	Multiple domains	Partial	2.5	4
[18]	Industry report	Framework/method	N/S	Marginal	1	4
[22]	Theoretical	Framework/method	Web/services	Partial	2	0
[24]	Industry report	Guidelines	Web/service	Full	1.5	4
[31]	Industry report	Advice/implications	Web/Service	Full	1.5	4
[46]	Case study	Lesson learned	N/S	Full	3	0
[50]	Theoretical + industry report	Framework/method	Web/service	Partial	0.5	3
[51]	Mixed methods	Lesson learned	Web/service	Full	2	4
[54]	Industry report	Advice/implications	Web/service	Full	1.5	4
[59]	Industry report	Lesson learned	Web/service	Full	1	4
[75]	Industry report	Lesson learned	Embedded, safety critical	Partial	0.5	4
[76]	Industry report	Lesson learned	Embedded, safety critical	Partial	1	4
[79]	Theoretical + case study	Framework/method	N/S	Partial	1	0
[16]	Industry report	Framework/method	N/S	Full	1	4
[17]	Theoretical + case study	Framework/method	Embedded	Partial	1	2
[7]	Industry report	Guidelines	Web/service	Partial	1.5	4
[67]	Opinion paper	Advice/implications	N/S	Partial	N/A	N/A
[2]	Action research	Framework/method	Embedded	Partial	3	4
[10]	Industry report	Advice/implication	Web/service	Marginal	1	4
[15]	Theoretical + case study	Model	Embedded	Full	1	3
[30]	Design science	Model	Web/service	Full	2	3
[32]	Theoretical	Model	N/S	Full	2	0
[34]	Industry report	Advice/implications	Web/service	Full	0.5	4
[35]	Industry report	Advice/implications	Web/service	Full	1.5	4
[37]	Industry report	Guidelines	Web/service	Full	1	4
[40]	Industry report	Tool	Web/service	Full	1	4
[45]	Case study	Lesson learned	Web/service	Partial	2.5	4
[47]	Theoretical + mixed methods	Model	N/S	Full	3	0
[48]	Industry report	Advice/implications	Web/service	Partial	1.5	4
[56]	Theoretical	Tool	N/S, open source	Partial	1.5	0
[57]	Case study	Lesson learned	Web/service	Partial	3	4
[61]	Case study	Model, lesson learned	Embedded	Full	3	4
[62]	Case study	Model, lesson learned	Embedded and web/services	Full	3	4
[63]	Theoretical + case study	Tool	N/S	Partial	1	1
[70]	Case study	Lesson learned	N/S	Marginal	1	1
[81]	Industry report	Advice/implications	Embedded	Full	0.5	3
[29]	Theoretical + case study	Model	Embedded	Full	1	3
[3]	Theoretical + industry report	Lesson learned, model	Embedded	Partial	1	4
[58]	Theoretical	Model	N/S	Partial	2	0
[9]	Opinion paper	Advice/implications	Web/service	Full	N/A	N/A
[41]	Case study	Lesson learned	Desktop application	Partial	3	4
[33]	Industry report	Framework/method	Web/service	Full	1.5	4
[49]	Case study	Lesson learned	Desktop application	Partial	2	4
[53]	Case study	Model	Desktop application	Partial	3	4
[74]	Action research	Lesson learned, framework/method	Embedded	Partial	3	4
[60]	Case study	Framework/method	Embedded	Partial	2.5	4
[64]	Case study	Framework/method	N/S	Partial	1.5	4
[4]	Case study	Lesson learned, model	Desktop application	Partial	3	4
[82]	Experiment	Theory	N/S	Marginal	3	1

Table B.11: Frameworks and Methods, Models and Tools

Primary study	Description	Pertinence
	Frameworks and Methods	
[1]	Continuous SCRUM, an approach based on Scrum to achieve fast-paced continuous product evolution and deployment.	Full
[33]	Solution applied at Facebook to enable frequent software upgrades.	Full
[16] and [18]	*Economic governance*, *Measured improvement* and *Disciplined agile delivery* frameworks to accelerate software delivery at an enterprise scale.	Full and marginal
[22]	Risk assessment heuristic approach to quantify software deployment risks in context of fast-paced continuous release environment.	Partial
[2]	Method to identify risky areas of source code and assess risks in the context of fast and incremental delivery.	Partial
[17]	Approach to quantify architecture quality with measurable criteria to guide continuous and iterative release planning.	Partial
[79]	Approach to continuous release and upgrade of component-based software.	Partial
[60]	Visualization technique of the testing activities involved from unit and component level to product and release level that support the identification of improvement areas.	Partial
[64]	Two level approach of how human factors can influence continuous software engineering.	Partial
[74]	Release readiness indicator to predict the time in weeks to release the product.	Partial
[50]	Approach to manage dependences between service design and release management.	Partial
	Models	
[61] and [62]	Stairway to Heaven model.	Full
[30]	Continuous experimentation model.	Full
[15] and [29]	Architecture for large-scale innovation experiment system.	Full
[32]	Continuous software engineering model.	Full
[47]	Rugby: Agile process model for continuous delivery.	Full
[58]	Project health measurement model based on Bayesian networks.	Partial
[53]	Model explaining the relationship between release model, release length and test effort.	Partial
[4]	Model of the Mozilla's patch lifecycle for rapid releases.	Partial
[3]	Software in Simulation (SiS) architecture to practice continuous integration and continuous deployment in the embedded domain.	Partial
	Tools	
[40]	Weaver. A domain-specific language for continuous validation of the deployed environment.	Full
[56]	Automated quality assurance service to validate configuration management scripts across a range of environments.	Partial
[63]	GAMMA. Tool for remotely monitoring the deployed software.	Partial
[49]	Clone detector as a tool to understand differences between releases such as how many changes were done between releases, how many bugs were made, etc.	Partial