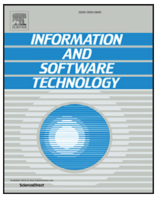
Contents lists available at ScienceDirect

Information and Software Technology

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



Problems, causes and solutions when adopting continuous delivery—A systematic literature review

Eero Laukkanen[a,*], Juha Itkonen[a], Casper Lassenius[b,a]

[a] Department of Computer Science, PO Box 15400, FI-00076 AALTO, Finland
[b] Massachusetts Institute of Technology, Sloan School of Management, USA

ARTICLE INFO

Article history:
Received 2 December 2015
Revised 11 October 2016
Accepted 11 October 2016
Available online 12 October 2016

Keywords:
Continuous integration
Continuous delivery
Continuous deployment
Systematic literature review

ABSTRACT

Context: Continuous delivery is a software development discipline in which software is always kept releasable. The literature contains instructions on how to adopt continuous delivery, but the adoption has been challenging in practice.

Objective: In this study, a systematic literature review is conducted to survey the faced problems when adopting continuous delivery. In addition, we identify causes for and solutions to the problems.

Method: By searching five major bibliographic databases, we identified 293 articles related to continuous delivery. We selected 30 of them for further analysis based on them containing empirical evidence of adoption of continuous delivery, and focus on practice instead of only tooling. We analyzed the selected articles qualitatively and extracted problems, causes and solutions. The problems and solutions were thematically synthesized into seven themes: build design, system design, integration, testing, release, human and organizational and resource.

Results: We identified a total of 40 problems, 28 causal relationships and 29 solutions related to adoption of continuous delivery. Testing and integration problems were reported most often, while the most critical reported problems were related to testing and system design. Causally, system design and testing were most connected to other themes. Solutions in the system design, resource and human and organizational themes had the most significant impact on the other themes. The system design and build design themes had the least reported solutions.

Conclusions: When adopting continuous delivery, problems related to system design are common, critical and little studied. The found problems, causes and solutions can be used to solve problems when adopting continuous delivery in practice.

1. Introduction

Continuous delivery (CD) is a software development discipline in which the software is kept in such a state that in principle, it could be released to its users at any time [1,2]. Fowler [2] proposes that practicing CD reduces deployment risk, allows believable progress tracking and enables fast user feedback.

While instructions on how to adopt CD have existed for a couple of years [1], the industry has not still adopted the practice at large [3], and those who have taken steps towards CD have found it challenging [4,5]. This raises the question whether the industry is lagging behind the best practices or whether the implementation difficulty is higher and the payoff lower than speculated by the proponents of CD. In this literature study, we look at problems in adopting CD, their causes and related solutions. We do not attempt to understand the cost-benefit ratio of CD implementation, since currently there are not enough primary studies about the cost-benefit ratio in order to create a meaningful literature study on the subject.

In this study, we attempt to create a synthesized view of the literature considering CD adoption problems, causes and solutions. Our mission is not just to identify different problem concepts, but also to understand their relationships and root causes, which is reflected in the three research questions of the study:

RQ1. What continuous delivery adoption problems have been reported in major bibliographic databases?

[*] Corresponding author.
E-mail addresses: eero.laukkanen@aalto.fi (E. Laukkanen), juha.itkonen@aalto.fi (J. Itkonen), casper.lassenius@aalto.fi (C. Lassenius).

http://dx.doi.org/10.1016/j.infsof.2016.10.001

RQ2. What causes for the continuous delivery adoption problems have been reported in major bibliographic databases?
RQ3. What solutions for the continuous delivery adoption problems have been reported in major bibliographic databases?

Summarizing the literature is valuable for practitioners for whom the large amount of academic literature from various sources is not easily accessible. In addition, for research community our attempt provides a good starting point for future research topics.

We believe this study provides an important contribution for the field, because while CD has been successfully adopted in some pioneering companies, it is not known how generally applicable it is. For example, can testing and deployment be automated in all contexts or is it not feasible in some contexts? Furthermore, in many contexts where CD has not been applied, there are signs of problems that CD is proposed to solve. Organizations who are developing software in contexts other than typical CD adoption would be eager to know what constraints CD has and would it be possible to adopt it in their context. We aim to address the decision-making challenge of whether to adopt CD or not and to what extent.

To understand the current knowledge about the problems of CD adoption, we conducted a systematic literature review (SLR). Previous literature studies have focused on characteristics [6,7] benefits [7,8], technical implementations [9], enablers [6,10] and problems [6,7] of CD or a related practice. Thus, there has been only two literature studies that investigated problems, and they studied the problems of the practice itself, not adoption of the practice. Furthermore, one of the studies was a mapping study instead of an SLR, and another one focused on rapid releases, the strategy to release with tight interval, instead of CD, the practice to keep software releasable. Therefore, to our knowledge, this is the first SLR which studies CD adoption problems, their causes and solutions.

This paper is structured as follows. First, we give background information about CD and investigate the earlier SLRs in Section 2. Next, we introduce our research goal and questions, describe our methodology, and asses the quality of the study in Section 3. In Section 4, we introduce the results, which we further discuss in Section 5. Finally, we present our conclusions and ideas for future work in Section 6.

2. Background and related work

In this section, we first define the concepts related to the subject of the study: continuous integration (CI), continuous delivery (CD) and continuous delivery adoption. CI is introduced before CD, because it is a predecessor and requirement of CD. After defining the concepts, we introduce previous literature studies that are related to the subject.

2.1. Continuous integration

According to Fowler [11], continuous integration (CI) is a software development practice where software is integrated continuously during development. In contrast, some projects have integrated the work of individual developers or teams only after multiple days, weeks or even months of development. When the integration is delayed, the possibility and severeness of conflicts between different lines of work increase.

Good practice of CI requires all developers to integrate their work to a common code repository on a daily basis [11]. In addition, after each integration, the system should be built and tested, to ensure that the system is still functional after each change and that it is safe for others to build on top of the new changes. Typically, a CI server is used for tracking new changes from a version control system and building and testing the system after each change [12]. If the build or tests fail due to a change, the developer who has made the change is notified about the failure and either the cause of the failure should be fixed or the change reverted, in order to keep the software functional.

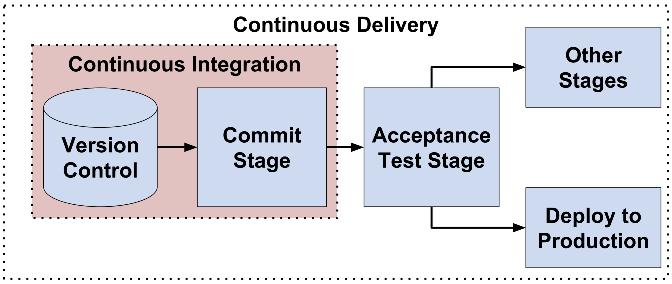


Fig. 1. The conceptual difference between CI and CD in this study.

There exist only a few scientific studies that have investigated how widely CI is practiced in the industry. Ståhl and Bosch [3] studied the CI practices in five Swedish software organizations and found that the practices were not really continuous: “activities are carried out much more infrequently than some observers might consider to qualify as being continuous”. In addition, Debbiche et al. [4] studied a large organization adopting CI and found multiple challenges. Based on these two studies, it seems that adopting CI has proven to be difficult, but why it is difficult is not known at the moment.

2.2. Continuous delivery

Continuous delivery (CD) is a software development discipline in which software can be released to production at any time [2]. The discipline is achieved through optimization, automatization and utilization of the build, deploy, test and release process [1].

CD extends CI by continuously testing that the software is of production quality and by requiring that the release process is automated. The difference between CI and CD is further highlighted in Fig. 1 where it is shown that while CI consists of only a single stage, CD consists of multiple stages that verify whether the software is in releasable condition. However, one should be aware that the terms are used differently outside this study. For example, Eck et al. [10] use the term CI while their definition for it is similar to the definition of CD in this study.

The proposed benefits of CD are increased visibility, faster feedback and empowerment of stakeholders [1]. However, when trying to adopt CD, organizations have faced numerous challenges [5]. In this study, we attempt to understand these challenges in depth.

Continuous deployment is an extension to CD in which each change is built, tested and deployed to production automatically [13]. Thus, in contrast to CD, there are no manual steps or decisions between a developer commit and a production deployment. The motivation for automating the deployment to production is to gain faster feedback from production use to fix defects that would be otherwise too expensive to detect [13]. One should also note that continuous delivery and deployment are used as synonyms outside this study. For example, Rodriguez et al. [7] use the term continuous deployment while they refer to the practice of continuous delivery, since they do not require automatic deployments to production. While it would be interesting to study continuous deployment, we did not find any reports of continuous deployment implementations from the scientific literature. Therefore, we have chosen to use continuous delivery as a primary concept in this study.

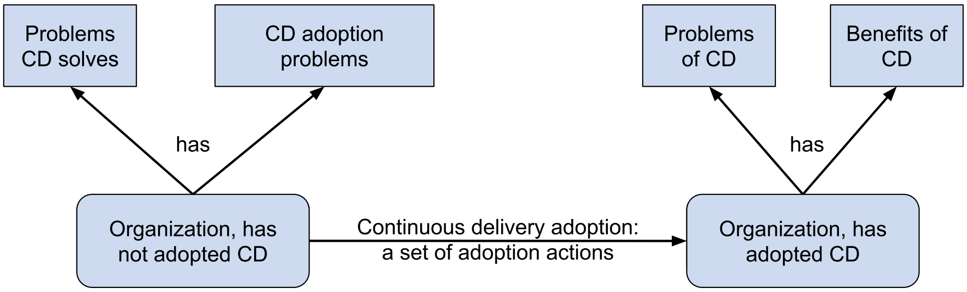

Problems CD solves
CD adoption problems
Problems of CD
Benefits of CD
has
has
Organization, has not adopted CD
Continuous delivery adoption: a set of adoption actions
Organization, has adopted CD

Fig. 2. Distinction between problems CD solves, problems preventing CD adoption, problems of CD and benefits of CD.

Table 1
Comparison of previous literature studies and this study.

Study	Focus	Type	Findings
Ståhl and Bosch [8]	CI	SLR	Benefits
Ståhl and Bosch [9]	CI	SLR	Variation points where implementations differ
Eck et al. [10]	CD	SLR	Adoption actions
Mäntylä et al. [6]	Rapid releases	Semi-systematic literature review	Benefits, adoption actions and problems
Rodriguez et al. [7]	CD	Systematic mapping study	Characteristics, benefits and problems
Adams and McIntosh [15]	Release engineering	Research agenda	Characteristics
This study	CD	SLR	Problems preventing adoption, their causes and solutions

2.3. Continuous delivery adoption

We define CD adoption as the set of actions a software development organization has to perform in order to adopt CD (see Fig. 2). The actual set of actions depends on the starting point of the organization and thus the adoption can be different from case to case. While some sources provide specific actions that need to be done during the adoption [1,6,10], or simplistic sequential models of the adoption [14], these models are prescriptive in nature and in real life the adoption more likely requires iteration and case-specific actions, as even CI implementations differ among the practitioners [3].

To be able to discuss previous literature and the focus of our study, we have constructed the following concepts related to the CD adoption:

- **Problems CD solves** are problems that are not directly related to CD, but CD is promised to solve them. These include, e.g., slow feedback of changes and error-prone releases.
- **CD adoption problems** are problems that are directly preventing CD adoption and additional actions need to be done to solve them. These problems are the focus of this study.
- **Adoption actions** need to be performed by an organization to adopt CD.
- **Problems of CD** are problems that emerge when CD is adopted.
- **Benefits of CD** are positive effects that are achieved after CD has been adopted.

The definitive source for CD [1] describes the problems CD solves, benefits of CD and suggests needed adoption actions. It briefly mentions problems preventing CD adoption, but there is no detailed discussion of them. Next, we introduce related academic literature studies and show that neither they focus on the subject of this study, the problems preventing CD adoption.

2.4. Previous related literature studies

To our knowledge, there have been six literature studies related to the subject of this study (Table 1). These studies have reviewed the characteristics [7,15], benefits [6–8], variation in implementations [9], adoption actions [6,10], and problems [6,7] of CD or a related practice such as CI, rapid releases or release engineering (Table 2). Rapid releases is a practice of releasing software frequently, so that the time between releases is hours, days or weeks instead of months [6]. Release engineering means the whole process of taking developer code changes to a release [15]. We see CD as a release engineering practice and as an enabler for hourly or daily releases, but we do not see it necessary if the time between releases is measured in weeks. In addition, practicing CD does not imply releasing rapidly, since one can keep software releasable all the time but still perform the actual releases more seldom.

The identified characteristics of CD are fast and frequent release, flexible product design and architecture, continuous testing and quality assurance, automation, configuration management, customer involvement, continuous and rapid experimentation, post-deployment activities, agile and lean and organizational factors [7]. To avoid stretching the concept of CD and keep our study focused, we use the definition in the book by Humble and Farley [1], which does not include all the factors identified by [7]. For example, we investigate CD as a development practice where software is kept releasable, but not necessarily released frequently. In addition, our definition does not necessarily imply tight customer involvement or rapid experimentation. Instead, the focus of our study is on the continuous testing and quality assurance activities, especially automated activities. Our view is more properly described by the characteristics of release engineering as defined by Adams and McIntosh: branching and merging, building and testing, build system, infrastructure-as-code, deployment and release [15].

Proposed benefits of CI, CD or rapid releases are automated acceptance and unit tests [8], improved communication [8], increased productivity [6–8], increased project predictability as an effect of finding problems earlier [6–8], increased customer satisfaction [6,7], shorter time-to-market [6,7], narrower testing scope [6,7] and improved release reliability and quality [7]. The claimed benefits vary depending on the focus of the studies.

Variation in implementations of CI can be in build duration, build frequency, build triggering, definition of failure and success, fault duration, fault handling, integration frequency, integration on broken builds, integration serialization and batching, integration target, modularization, pre-integration procedure, scope, status communication, test separation and testing of new functionality [9]. However, the variations listed here are limited only to the CI systems that automate the activities of building, testing and

Table 2
Summary of the results from previous literature studies.

Results	Results
Benefits	Automated acceptance and unit tests [8], improved communication [8], increased productivity [6–8], increased project predictability as an effect of finding problems earlier [6–8], increased customer satisfaction [6,7], shorter time-to-market [6,7], narrower testing scope [6,7], improved release reliability and quality [7].
Variation points	Build duration, build frequency, build triggering, definition of failure and success, fault duration, fault handling, integration frequency, integration on broken builds, integration serialization and batching, integration target, modularization, pre-integration procedure, scope, status communication, test separation, testing of new functionality [9].
Adoption actions	Devising an assimilation path, overcoming initial learning phase, dealing with test failures right away, introducing CD for complex systems, institutionalizing CD, clarifying division of labor, CD and distributed development, mastering test-driven development, providing CD with project start, CD assimilation metrics, devising a branching strategy, decreasing test result latency, fostering customer involvement in testing, extending CD beyond source code [10]. Parallel development of several releases, deployment of agile practices, automated testing, the involvement of product managers and pro-active customers, efficient build, test and release infrastructure [6].
Problems	Increased technical debt [6], lower reliability and test coverage [6], lower customer satisfaction [6,7], time pressure [6], transforming towards CD [7], increased QA effort [7], applying CD in the embedded domain [7].
Characteristics	Fast and frequent release, flexible product design and architecture, continuous testing and quality assurance, automation, configuration management, customer involvement, continuous and rapid experimentation, post-deployment activities, agile and lean, organizational factors [7]. Branching and merging, building and testing, build system, infrastructure-as-code, deployment and release [15].

deploying the software. In addition, there should be variations in the practices how the systems are used, but these variations are not studied in any literature study. Our focus is not to study the variations, but we see that because there is variation in the implementations, the problems emerging during the adoption must vary too between cases. Thus, we cannot assume that the problems are universally generalizable, but one must investigate them case-specifically.

CD adoption actions are devising an assimilation path, overcoming initial learning phase, dealing with test failures right away, introducing CD for complex systems, institutionalizing CD, clarifying division of labor, CD and distributed development, mastering test-driven development, providing CD with project start, CD assimilation metrics, devising a branching strategy, decreasing test result latency, fostering customer involvement in testing and extending CD beyond source code [10]. Rapid releases adoption actions are parallel development of several releases, deployment of agile practices, automated testing, the involvement of product managers and pro-active customers and efficient build, test and release infrastructure [6]. The intention in this study is to go step further and investigate what kind of problems arise when the adoption actions are attempted to be performed.

Proposed problems of CD or rapid releases are increased technical debt [6], lower reliability and test coverage [6], lower customer satisfaction [6,7], time pressure [6], transforming towards CD [7], increased QA effort [7] and applying CD in the embedded domain [7]. Interestingly, previous literature studies have found that there is the benefit of improved reliability and quality, but also the problem of technical debt, lower reliability and test coverage. Similarly, they have identified the benefit of automated acceptance and unit tests and narrower testing scope, but also the problem of increased QA effort. We do not believe that the differences are caused by the different focus of the literature studies. Instead, we see that since the benefits and problems seem to contradict each other, they must be case specific and not generalizable. In this study, we do not investigate the problems of the CD practice itself, but we focus on the problems that emerge when CD is adopted. One should not think these problems as general causal necessities, but instead instances of problems that may be present in other adoptions or not.

As a summary, previous literature studies have identified what CD [7] and release engineering [15] are, verified the benefits of CD [7], CI [8] and rapid releases [6], discovered differences in the implementations of CI [9], understood what is required to adopt CD [10] and rapid releases [6] and identified problems of practicing CD [7] and rapid releases [6] (see Table 2). However, none of the previous studies has investigated why the adoption efforts of CD are failing in the industry. One of the studies acknowledged the problem with the adoption [7], but did not investigate it further, as it was a systematic mapping study. At the same time there is increasing evidence that many organizations have not adopted CD yet [3]. To address this gap in the previous literature studies, we have executed this study.

3. Methodology

In this section, we present our research goal and questions, search strategy, filtering strategy, data extraction and synthesis and study evaluation methods. In addition, we present the selected articles used as data sources and discuss their quality assessment.

3.1. Research goal and questions

The goal of this paper is to investigate what is reported in the major bibliographic databases about the problems that prevent or hinder CD adoption and how the problems can be solved. Previous software engineering research indicates that understanding complex problems requires identifying underlying causes and their relationships [16]. Thus, in order to study CD adoption problems, we need to study their causes too. This is reflected in the three research questions of this paper:

RQ1. What continuous delivery adoption problems have been reported in major bibliographic databases?
RQ2. What causes for the continuous delivery adoption problems have been reported in major bibliographic databases?
RQ3. What solutions for the continuous delivery adoption problems have been reported in major bibliographic databases?

We answer the research questions using a systematic literature review of empirical studies of adoption and practice of CD in real-world software development (see Section 3.3 for the definition of real-world software development).

We limit ourselves to major bibliographic databases, because it allows executing systematic searches and provides material that, in general, has more in-depth explanations and neutral tone. The bibliographic databases we used are listed in Table 3. The databases include not only research articles, but also, e.g., some books written by practitioners and experience reports. However, the databases do not contain some of the material that might be relevant for the subject of study, e.g., technical reports, blog posts and video presentations. While the excluded material might have provided additional information, we believe that limiting to the major bibliographic databases provides a good contribution on its

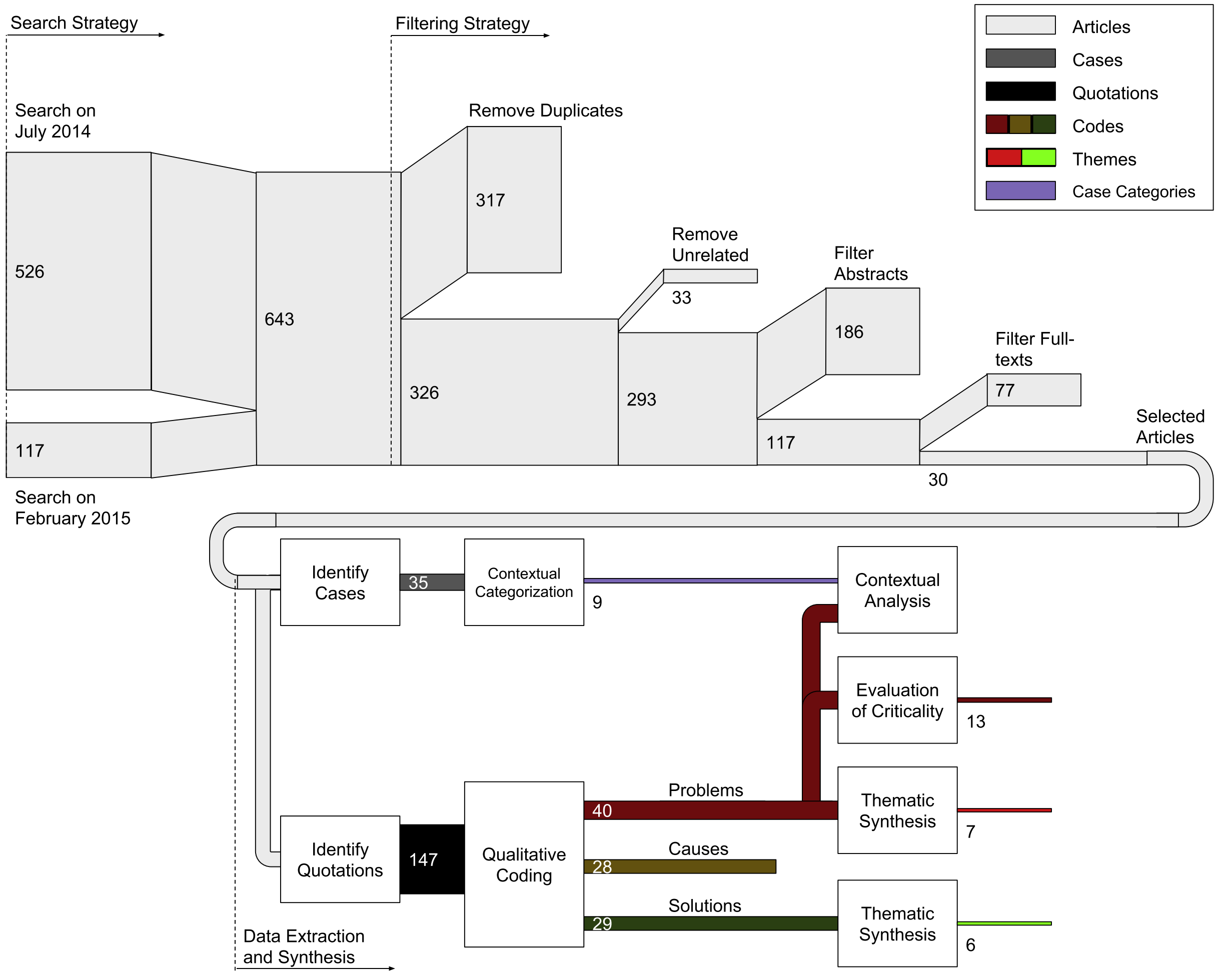


Fig. 3. An overview of the research process used in this study.

Table 3
Search results for each database in July 2014 and in February 2015. Search was executed for all years in July 2014, but only for years 2014–2015 in February 2015.

Database	July 2014	February 2015	**Total**
Scopus	197	35	232
IEEE Explore	98	30	128
ACM Digital Library	139	30	169
ISI Web of Science	79	11	90
ScienceDirect	13	11	24
Total	526	117	643

own and this work can be extended in future. This limitation increases the reliability and validity of the material, but decreases the amount of reports by practitioners [17].

We limit our investigation to problems that arise when adopting or practicing CD. We thus refrain from collecting problems that CD is meant to solve—an interesting study on its own. Furthermore, we do not limit ourselves to a strict definition of CD. The reasons are that CD is a fairly new topic and there does not exist much literature mentioning CD in the context of our study. Since it is claimed that CI is a prerequisite for CD [1], we include it in our study. Similarly, continuous deployment is claimed to be a extension of CD, and we include it too. We do this by including search terms for continuous integration and continuous deployment. This way, we will find material that considers CD adoption path beginning from CI adoption and ending in continuous deployment.

We followed Kitchenham's guidelines for conducting systematic literature reviews [18], with two exceptions. First, we decided to include multiple studies of the same organization and project, in order to use all available information for each identified case. We clearly identify such studies as depicting the same case in our analysis, results and discussion. The unit of analysis used is a case, not a publication. Second, instead of using data extraction forms, we extracted data by qualitatively coding the selected articles, as most of the papers contained only qualitative statements and little numerical data. The coding is described in more detail in Section 3.4.2.

The overall research process consisted of three steps: search strategy, filtering strategy and data extraction and synthesis (see Fig. 3). Next, we will introduce the steps.

3.2. Search strategy

The search string used was "(``continuous integration'' OR ``continuous delivery'' OR ``continuous deployment'') AND software". The first parts of the string were the subject of the study. The "software" string was included to exclude studies that related to other fields than software engineering; the same approach was used in an earlier SLR [9]. The search string was applied to titles, abstracts and keywords. The search was executed first in July 2014 and again in February 2015. The second search was executed because there had been recent new publications in the area. Both searches provided a total of 643 results (Table 3). After the filtering strategy was applied and an article was selected for inclusion, we used backward snowballing [19], which did not result in the identification of any additional studies.

3.3. Filtering strategy

We used two guiding principles when forming the filtering strategy:

- **Empirical:** the included articles should contain data from real-life software development.
- **CD practice:** the included articles should contain data from continuous delivery as a practice. Some articles just describe toolchains, which usually is separated from the context of its use.

With real-life software development, we mean an activity producing software meant to be used in real-life. For example, we included articles discussing the development of industrial, scientific and open source software systems. We also classified development happening in the context of engineering education as real-life, if the produced software was seen to be usable outside the course context. However, software development simulations or experiments were excluded to improve the external validity of the evidence. For example, [20] was excluded, because it only simulates software development.

First, we removed duplicate and totally unrelated articles from the results, which left us with 293 articles (Fig. 3). Next, we studied the abstracts of the remaining papers, and applied the following inclusion and exclusion criteria:

- *Inclusion Criterion*: a real-life case is introduced or studied.
- *Exclusion Criterion 1*: the practice or adoption of continuous integration, delivery or deployment is not studied.
- *Exclusion Criterion 2*: the main focus of the article is to evaluate a new technology or tool in a real-life case. Thus, the article does not provide information about the case itself or CD adoption.
- *Exclusion Criterion 3*: the text is not available in English.

A total of 107 articles passed the criteria.

Next, we acquired full-text versions of the articles. We did not have direct access to one article, but an extension of it was found to been published as a separate article [P11]. We applied the exclusion criteria discussed above to the full-text documents, as some of the papers turned out not to include any real-world case even if the abstracts had led us to think so. For example, the term case study can indeed mean a study of a real-world case, but in some papers it referred to projects not used in real-life. In addition, we applied the following exclusion criteria to the full-texts:

- *Exclusion Criterion 4*: the article only repeats known CD practice definitions, but does not describe their implementation.
- *Exclusion Criterion 5*: the article only describes a technical implementation of a CD system, not practice.

Out of the 107 articles, 30 passed our exclusion criteria and were included in the data analysis.

3.4. Data extraction and synthesis

We extracted data and coded it using three methods. First, we used qualitative coding to ground the analysis. Second, we conducted contextual categorization and analysis to understand the contextual variance of the reported problems. Third, we evaluated the criticality of problems to prioritize the found problems. Next, these three methods are described separately in depth.

3.4.1. Unit of analysis

In this paper, the unit of analysis is an individual case instead of an article, as several papers included multiple cases. A single case could also be described in multiple articles. The 30 articles reviewed here discussed a total of 35 cases. When referring to a case, we use capital *C*, e.g. [C1], and when referring to an article, we use capital *P*, e.g. [P1]. If an article contained multiple cases, we use the same case number for all of them but differentiate them with a small letter, e.g. [C9a] and [C9b]. The referred articles and cases are listed in a separate bibliography in Appendix A.

3.4.2. Qualitative coding

We coded the data using qualitative coding, as most of the studies were qualitative reports. We extracted the data by following the coding procedures of grounded theory [21]. Coding was performed using the following steps: conceptual coding, axial coding and selective coding. All coding work was done using ATLAS.ti [22] software.

During *conceptual coding*, articles were first examined for instances of problems that had emerged when adopting or doing CD. We did not have any predefined list of problems, so the precise method was open coding. Identifying instances of problems is highly interpretive work and simply including problems that are named explicitly problems or with synonyms, e.g. challenges, was not considered inclusive enough. For example, the following quote was coded with the codes “problem” and “Ambiguous test result”, even if it was not explicitly mentioned to be a problem:

> *Since it is impossible to predict the reason for a build failure ahead of time, we required extensive logging on the server to allow us to determine the cause of each failure. This left us with megabytes of server log files with each build. The cause of each failure had to be investigated by trolling through these large log files.*
>
> –Case C4

For each problem, we examined whether any solutions or causes for that problem were mentioned. If so, we coded the concepts as solutions and causes, respectively. The following quote was coded with the codes “problem”, “large commits”, “cause for” and “network latencies”. This can be translated into the sentence “network latencies caused the problem of large commits”.

> *On average, developers checked in once a day. Offshore developers had to deal with network latencies and checked in less frequently; batching up work into single changesets.*
>
> –Case C13

Similarly, the following quote was coded with the codes “problem”, “time-consuming testing”, “solution”, and “test segmentation”. This can be read as “test segmentation solves the problem of time-consuming testing”.

> *We ended up running several different CI builds largely because running everything in one build became prohibitively slow and we wanted the check-in build to run quickly.*
>
> –Case C13

During *axial coding*, we made connections between the codes formed during conceptual coding. We connected each solution code to every problem code that it was mentioned to solve. Similarly, we connected each problem code to every problem code that it was mentioned causing. The reported causes are presented in Section 4.2. We did not separate problem and cause codes, because often causes could be seen as problems too. On the other hand, we divided the codes strictly to be either problems or solutions, even if some solutions were considered problematic in the articles. For example, the solution “practicing small commits” can be difficult if the “network latencies” problem is present. But to code this, we used the problem code “large commits” in the relation to “network

Table 4
Case categories and categorization criteria.

Category	Criteria	Category	Criteria
Publication time		**Number of developers**	
Pre 2010	*year* ≤ 2010	Small	*size* < 20
Post 2010	*year* > 2010	Medium	20 ≤ *size* ≤ 100
		Large	*size* > 100
CD implementation maturity		**Commerciality**	
CI	CI practice.	Non-commercial	E.g., open source or scientific development.
CD	CD or advanced CI practice.	Commercial	Commercial software development.

latencies". The code "system modularization" was an exception to this rule, being categorized as both a problem and a solution, because system modularization in itself can cause some problems but also solve other problems.

During *selective coding*, only the already formed codes were applied to the articles. This time, even instances, that discussed the problem code but did not consider it as a faced problem, were coded to ground the codes better and find variance in the problem concept. Also some problem concepts were combined to raise the abstraction level of coding. For example, the following quote was coded with "effort" during selective coding:

> *Continually monitoring and nursing these builds has a severe impact on velocity early on in the process, but also saves time by identifying bugs that would normally not be identified until a later point in time.*
> –Case C4

In addition, we employed the code "prevented problem" when a problem concept was mentioned to having been solved before becoming a problem. For example, the following quote was coded with the codes "parallelization", "prevented problem" and "time-consuming testing":

> *Furthermore, the testing system separates time consuming high level tests by detaching the complete automated test run to be done in parallel on different servers. So whenever a developer checks in a new version of the software the complete automated set of tests is run.*
> –Case C1

Finally, we employed the code "claimed solution" when some solution was claimed to solve a problem but the solution was not implemented in practice. For example, the following quote was coded with the codes "problem", "ambiguous test result", "claimed solution" and "test adaptation":

> *Therefore, if a problem is detected, there is a considerable amount of time invested following the software dependencies until finding where the problem is located. The separation of those tests into lower level tasks would be an important advantage for troubleshooting problems, while guaranteeing that high level tests will work correctly if the lower level ones were successful.*
> –Case C15

3.4.3. Thematic synthesis

During thematic synthesis [23], all the problem and solution codes were synthesized into themes. As a starting point of themes, we took the different activities of software development: *design, integration, testing* and *release*. The decision to use these themes as a starting point was done after the problem instances were identified and coded. Thus, the themes were not decided beforehand; they were grounded in the identified problem codes.

If a problem occurred during or was caused by an activity, it was included in the theme. During the first round of synthesis, we noticed that other themes were required as well, and added the themes of *human and organizational* and *resource*. Finally, the *design* theme was split into *build design* and *system design*, to separate these distinct concepts.

3.4.4. Contextual categorization and analysis

We categorized each reported case according to four variables: publication time, number of developers, CD implementation maturity and commerciality, as shown in Table 4. The criteria were not constructed beforehand, but instead after the qualitative analysis of the cases, letting the categories inductively emerge from the data. When data for the categorization was not presented in the article, the categorization was interpreted based on the case description by the first author.

The CD implementation maturity of cases was determined with two steps. First, if a case described CD adoption, its maturity was determined to be CD, and if a case described CI adoption, its maturity was determined to be CI. Next, advanced CI adoption cases that described continuous system-level quality assurance procedures were upgraded to CD maturity, because those cases had more similarity to CD cases than to CI cases. The upgraded cases were C1, C4 and C8.

After the categorization, we compared the problems reported between different categories. The comparison results are presented in Section 4.2.

3.4.5. Evaluation of criticality

We selected the most critical problems for each case in order to see which problems had the largest impact hindering the CD adoption. The number of the most critical problems was not constrained and it varied from zero to two problems per case. There were two criteria for choosing the most critical problems. Either, the most severe problems that prevented adopting CD, or, the most critical enablers that allowed adopting CD.

Enabling factors were collected because, in some cases, no critical problems were mentioned, but some critical enablers were emphasized. However, when the criticality assessments by different authors were compared, it turned out that the selection of critical enablers was more subjective than the selection of critical problems. Thus, only one critical enabler was agreed upon by all authors (unsuitable architecture in case C8).

The most critical problems were extracted by three different methods:

- **Explicit**: If the article as a whole emphasized a problem, or if it was mentioned explicitly in the article that a problem was the most critical, then that problem was selected as an explicit critical problem. E.g, in case C5, where multiple problems were given, one was emphasized as the most critical:

  > *A unique challenge for Atlassian has been managing the online suite of products (i.e. the OnDemand products) that are deeply integrated with one another...Due to the complexity of cross-product dependencies, several interviewees believed this was the main challenge for the company when adopting CD.*
  > – Case C5

- **Implicit**: The authors interpreted which problems, if any, could be seen as the most critical. These interpretations were compared between the authors to mitigate bias, detailed description of the process is given in Section 3.5.
- **Causal**: the causes given in the articles were taken into account, by considering the more primary causes as more critical. For example, in case C3a, the complex build problem could be seen as critical, but it was actually caused by the inflexible build problem.

3.5. Validity of the review

The search, filtering, data extraction and synthesis were first performed by the first author, causing single researcher bias, which had to be mitigated. The search bias was mitigated by constructing the review protocol according to the guidelines by Kitchenham [18]. This review protocol was reviewed by the two other authors.

We mitigated the paper selection bias by having the two other authors make independent inclusion/exclusion decisions on independent random samples of 200 articles each of the total 293. The random sampling was done to lower the effort required for assessing the validity. This way, each paper was rated by at least two authors, and 104 of the papers were rated by all three.

We measured inter-rater agreement using Cohen's kappa [24], which was 0.5–0.6, representing moderate agreement [25]. All disagreements (63 papers) were examined, discussed and solved in a meeting involving all authors. All the disagreements were solved through discussion, and no modifications were made to the criteria. In conclusion, the filtering of abstracts was evaluated to be sufficiently reliable. The data extraction and synthesis biases in the later parts of the study were mitigated by having the second and third authors review the results.

Bias in the criticality assessment was mitigated by having the first two authors assess all the cases independently of each other. From the total of 35 cases, there were 12 full agreements, 10 partial agreements and 13 disagreements, partial agreements meaning that some of the selected codes were the same for the case, but some were not. All the partial agreements and disagreements were assessed also by the third author and the results were then discussed together by all the authors until consensus was formed. These discussions had an impact not only on the selected criticality assessments but also on the codes, which further improved the reliability of the study.

3.6. Selected articles

When extracting data from the 30 articles (see Appendix A), we noted that some of the articles did not contain any information about problems related to adopting CD. Those articles are still included in this paper for examination. The articles that did not contain any additional problems were P3, P17, P19, P21 and P26. Article P3 contained problems, but they were duplicate to Article P2 which studied the same case.

All the cases were reported during the years 2002–2014 (Fig. 4). This is not particularly surprising, since continuous integration as a practice gained most attention after publication of extreme programming in 1999 [26]. However, over half of the cases were reported after 2010, which shows an increasing interest in the subject. Seven of the cases considered CD (C5, C7, C14, C25a, C25b, C25c, C26). The other cases focused on CI.

Not all the articles contained quotations about problems when adopting CI or CD. For example, papers P21 and P26 contained detailed descriptions of CI practice, but did not list any problems. In contrast, two papers that had the most quotations were P6 with 38 quotations and P4 with 13 quotations. This is due to the fact that these two articles specifically described problems and challenges. Other articles tended to describe the CI practice implementation without considering any observed problems. Furthermore, major failures are not often reported because of publication bias [18].

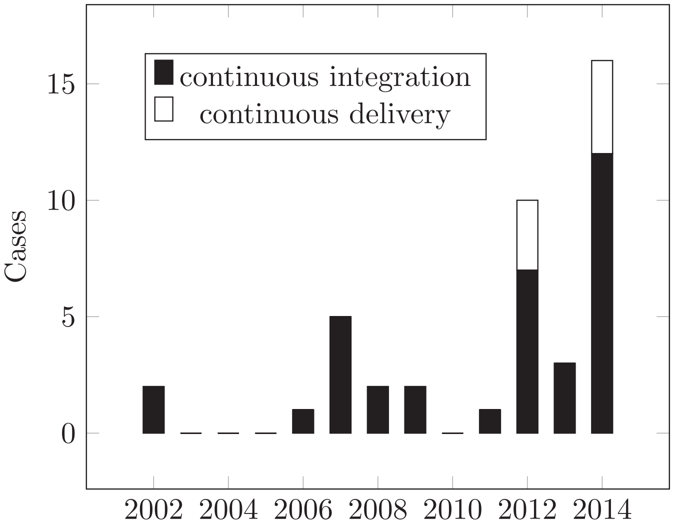


Fig. 4. Number of cases reported per year. The year of the case was the latest year given in the report or, if missing, the publication year.

3.7. Study quality assessment

Of the included 30 articles, we considered nine articles to be scientific (P6, P7, P8, P11, P19, P20, P21, P28, P30), because they contained descriptions of the research methodology employed. The other 21 articles were considered as descriptive reports. However, only two of the selected scientific articles directly studied the problems or challenges (P6, P30), and therefore, we decided not to separate the results based on whether the source was scientific or not. Instead, we aimed at extracting the observations and experiences presented in the papers rather than opinions or conclusions. Observations and experiences can be considered more valid than opinions, because they reflect the reality of the observer directly. In the context of qualitative interviews, Patton writes:

> *Questions about what a person does or has done aim to elicit behaviors, experiences, actions and activities that would have been observable had the observer been present.*
>
> –Patton [27, p. 349–350]

4. Results

In total, we identified 40 problems, 28 causal relationships and 29 solutions. In the next subsections, we explain these in detail. The results are augmented with quotes from the articles. An overview of the results can be obtained by reading only the summaries at the beginning of each subsection and a richer picture of the findings is provided through the detailed quotes.

4.1. Problems

Problems were thematically synthesized into seven themes. Five of these themes are related to the different activities of software development: build design, system design, integration, testing, and release. Two of the themes are not connected to any individual part: human and organizational and resource. The problems in the themes are listed in Table 5.

The number of cases which discussed each problem theme varied (Fig. 5). Most of the cases discussed integration and testing problems, both of them being discussed in at least 16 cases. The

Table 5
Problem themes and related problems. Cases where a problem was prevented with a solution are marked with a star (*).

Theme	Problems
Build design	Complex build [C2, C3a], inflexible build [C3a]
System design	System modularization [C2, C17e, C21, C25a, C25b], unsuitable architecture [C3a, C8, C22, C26, C25c], internal dependencies [C5], database schema changes [C5, C7(*), C25c(*)]
Integration	Large commits [C3a, C5, C7(*), C13, C14(*), C22], merge conflicts [C2(*), C3a, C5, C14, C20(*), C21, C24], broken build [C3a, C5, C6, C8, C9a, C14, C17a], work blockage [C3a, C5(*), C11, C17a, C27], long-running branches [C7(*), C14, C24, C27], broken development flow [C3a], slow integration approval [C17a]
Testing	Ambiguous test result [C2, C4, C6, C15, C17a, C27], flaky tests [C4, C6, C8, C11, C14, C22, C27], time-consuming testing [C1(*), C2(*), C3a, C3b(*), C11, C13, C14, C27], hardware testing [C1(*), C8], multi-platform testing [C2, C9b, C21], UI testing [C8, C14], untestable code [C22, C25b(*)], problematic deployment [C25a, C25b(*), C25c], complex testing [C21, C25c]
Release	[all in C5] customer data preservation, documentation, feature discovery, marketing, more deployed bugs, third party integration, users do not like updates, deployment downtime [C25c(*)]
Human and organizational	Lack of discipline [C1(*), C6, C10, C11, C12, C14], lack of motivation [C5, C6, C19, C27], lack of experience [C5, C12, C27], more pressure [C5, C27], changing roles [C5], team coordination [C5], organizational structure [C26]
Resource	Effort [C2, C3a, C4, C19, C17e, C26], insufficient hardware resources [C4, C5], network latencies [C13]

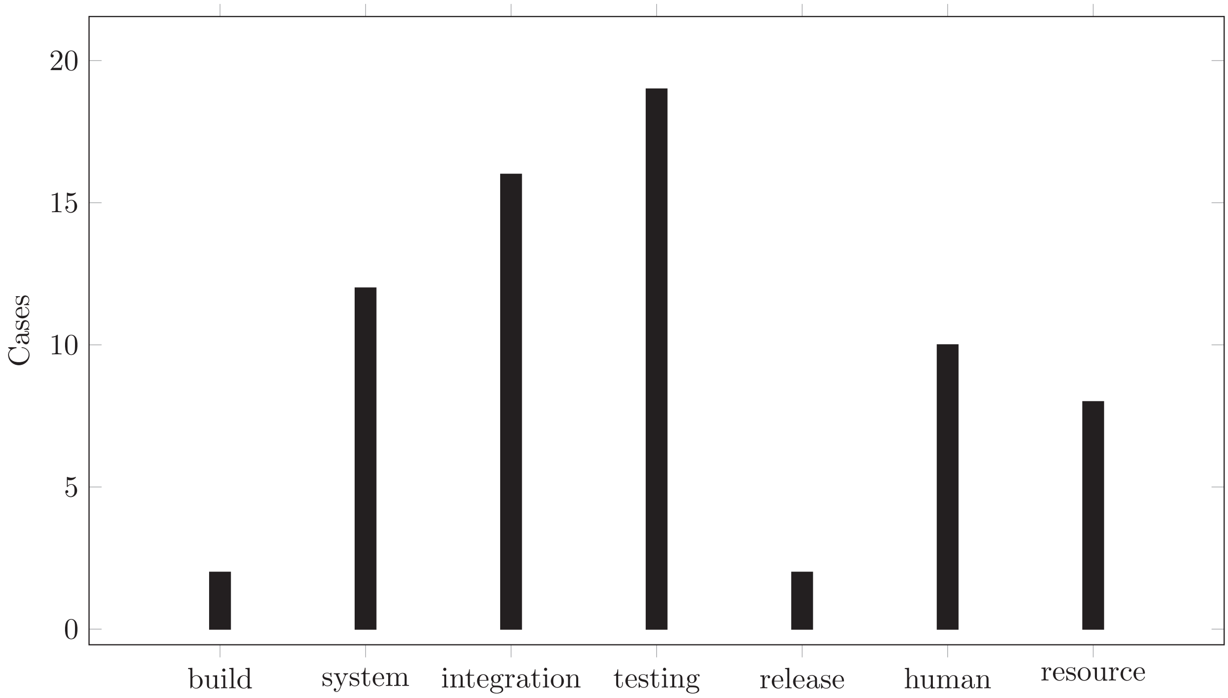


Fig. 5. Number of cases per problem theme.

Table 6
Build design problems.

Problem	Description
Complex build	Build system, process or scripts are complicated or complex.
Inflexible build	The build system cannot be modified flexibly.

second most reported problems were system design, human and organizational and resource problems, all of them handled in at least 8 cases. Finally, build design and release problems were discussed in two cases only. Most of the release problems were discussed only in one case [C5].

4.1.1. Build design problems

The build design theme covered problems that were caused by build design decisions. The codes in the theme are described in Table 6. The codes in the theme were connected and were concurrent in Case C3a. From that case, we can infer that the inflexible build actually caused the complexity of the build:

> *The Bubble team was just one team in a larger programme of work, and each team used the same build infrastructure and build targets for their modules. As the application developed, special cases inevitably crept into individual team builds making the scripts even more complex and more difficult to change.*
>
> –Case C3a

In another case, it was noted that system modularization of the application increased the complexity of the build:

> *Finally, the modular nature of AMBER requires a complicated build process where correct dependency resolution is critical. Developers who alter the build order or add their code are often unfamiliar with GNU Makefiles, especially at the level of complexity as AMBER's.*
>
> –Case C2

Complex builds are difficult to modify [C2] and significant effort can be needed to maintain them [C3a]. Complex builds can cause builds to be broken more often [C3a].

4.1.2. System design problems

The system design theme covered problems that were caused by system design decisions. The codes in the theme are described in Table 7.

Table 7
System design problems.

Problem	Description
System modularization	The system consists of multiple units, e.g., modules or services.
Unsuitable architecture	System architecture limits continuous delivery.
Internal dependencies	Dependencies between parts of the software system.
Database schema changes	Software changes require changes of database schema.

Table 8
Integration problems.

Problem	Description
Large commits	Commits containing large amount of changes.
Merge conflicts	Merging changes together reveals conflicts between changes.
Broken build	Build stays broken for long time or breaks often.
Work blockage	Completing work tasks is blocked or prevented by broken build or other integrations in a queue.
Long-running branches	Code is developed in branches that last for long time.
Broken development flow	Developers get distracted and the flow [28] of development breaks.
Slow integration approval	Changes are approved slowly to the mainline.

System modularization. System modularization was the most discussed system design problem: it was mentioned in five articles. While the codes *system modularization* and *unsuitable architecture* can be seen to overlap each other, system modularization is introduced as a separate code because of its unique properties; it was mentioned to be both a problem and a solution. For example, in the Case C2, it was said to cause build complexity, but in another quote it was said to prevent merge conflicts:

> *Merge conflicts are rare, as each developer typically stays focused on a subset of the code, and will only edit other subsections in a minor way to fix small errors, or with permission and collaboration.*
> –Case C2

In another case, system modularization was said to ensure testability of independent units:

> *They designed methods and classes as isolated services with very small responsibilities and well-defined interfaces. This allows the team to test individual units independently and to write (mocks of) the inputs and outputs of each interface. It also allows them to test the interfaces in isolation without having to interact with the entire system.*
> –Case C25a

System modularization was not a problem on its own in any instance. Rather its effects were the problems: increased development effort [C17e], testing complexity [C21] and problematic deployment [C25a].

Unsuitable architecture. An architecture unsuitable for CD was the second most discussed system design problem by being mentioned in four articles. Again, unsuitable architecture was not a problem on its own but its effects were the problems: time-consuming testing [C3a], development effort [C8], testability [C22, C25c] and problematic deployment [C25c]. Cases mentioned that architecture was unsuitable if it was monolithic [C22, C26], coupled [C3a], consisted of multiple branches of code [C8] or there were unnecessary service encapsulation [C25c].

Other system design problems were discussed lightly in a couple of cases only and thus are not included here for deeper analysis.

4.1.3. Integration problems

The integration theme covered issues that arise when the source code is integrated into the mainline. The problems in this theme are described in Table 8.

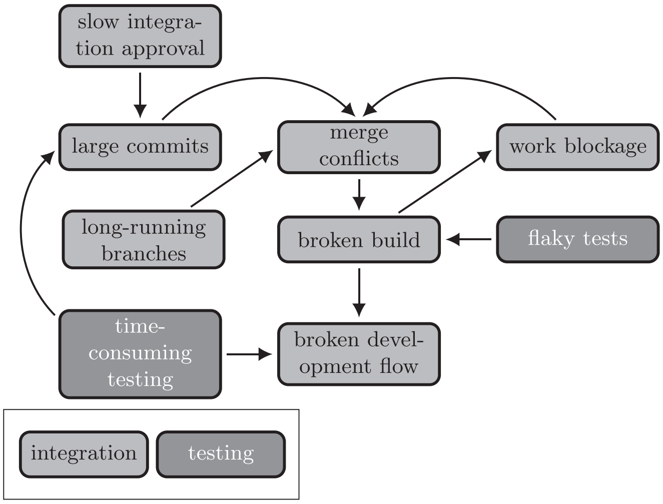


Fig. 6. Reported causal relationships between integration problems and related testing problems.

All the codes in this theme are connected through reported causal relationships, see Fig. 6. Some have tried to avoid integration problems with branching, but long-living branches are actually mentioned to make integration more troublesome in the long run:

> *...as the development of the main code base goes on, branches diverge further and further from the trunk, making the ultimate merge of the branch back into the trunk an increasingly painful and complicated process.*
> –Case C24

Another interesting characteristic in the integration theme is the vicious cycle between the codes broken build, work blockage and merge conflicts. This is emphasized in Case C3a:

> *Once the build breaks, the team experiences a kind of “work outage”. And the longer the build is broken, the more difficult it is for changes to be merged together once corrected. Quite often, this merge effort results in further build breaks and so on.*
> –Case C3a

Large commits. Large commits are problematic, because they contain multiple changes that can conflict with changes made by others:

> *These larger change sets meant that there were more file merges required before a check-in could be completed, further lengthening the time needed to commit.*
>
> –Case C3a

However, there are multiple reasons why developers do large commits: time-consuming testing [C3a], large features [C7, C14], network latencies [C13] and a slow integration approval process [C17a]. Thus, to deal with large commits, one must consider these underlying reasons behind.

Merge conflicts. Merge conflicts happen when changes made by different developers conflict. Solving such a conflict can take substantial effort:

> *We have felt the pain of merging long running branches too many times. Merge conflicts can take hours to resolve and it is all too easy to accidentally break the codebase.*
>
> –Case C14

Merge conflicts can be caused by long-running branches [C14, C24] or large commits [C3a]. Also delays in the committing process, such as lengthy code reviews, can cause merge conflicts [C21]. In some situations, merge conflicts can be rarer: if developers work on different parts of source code [C2] or if there is a small amount of developers [C20].

Broken build. Broken build was the most mentioned problem by being discussed in ten articles. Broken builds become a problem when it is hard to keep a build fixed and it takes a significant effort to fix the build:

> *The Bubble team build would often break and stay broken for some time (on one occasion for a full month iteration) so a significant proportion of developers time was spent fixing the build.*
>
> –Case C3a

If a broken build is not fixed immediately, feedback from other problems will not be gained. In addition, problematic code can spread to other developer workstations, causing trouble:

> *Some noted that if code was committed after a build failure, that new code could conceivably be problematic too, but the confounding factors would make it difficult to determine exactly where the problem was. Similarly, other developers may inadvertently obtain copies of the code without realizing it is in a broken state.*
>
> –Case C6

However, if developers are often interrupted to fix the build, it will break their development flow and take time from other tasks:

> *The Bubble teams other problem was being often interrupted to fix the build. This took significant time away from developing new functionality.*
>
> –Case C6

Reasons for broken builds were complex build [C3a], merge conflicts [C3a] and flaky tests [C14].

Work blockage. When the completion of a development task, e.g. integration, is delayed, it causes a work blockage:

> *It should also be noted that the SCM Mainline node affords no parallelism: if there is a blockage, as interviewees testify is frequently the case, it effectively halts the entire project.*
>
> –Case C17a

The reason can be that a broken build must be fixed [C3a, C11] or that there are other integrations in the queue [C27]. In addition to delays, work blockages can cause further merge conflicts [C3a].

Long-running branches. Long-running branches easily lead to merge conflicts, and developing code in branches slows the frequency of integration. However, some cases still insist on working with multiple branches:

> *Compared to smaller products, where all code is merged to a single branch, the development makes use of many branches which adds to the complexity.*
>
> –Case C27

There is not much evidence whether there are situations when multiple branches are necessary. Those who have chosen to work with a single branch have been successful with it [C7, C14]. Nevertheless, a working CI environment can help with solving large merge conflicts by providing feedback during the merge process [C24].

Broken development flow. When the CI system does not work properly and failures in the system distract developers from writing the software, the development flow [28] might get broken [C3a]. Broken development flow decreases development productivity.

Slow integration approval. The speed of integration can be slowed down by too strict approval processes:

> *Each change...must be manually approved by a project manager before it is allowed onto the SCM Mainline. The consequence of this is a queuing situation, with an elaborate ticket system having sprung up to support it, where low priority "deliveries" can be put on hold for extended periods of time.*
>
> –Case C17a

A slow integration approval process is detrimental to CD, because it leads to larger commits and delays feedback. Code review processes should be designed so that they do not cause extensive delays during integration.

4.1.4. Testing problems

The testing problem theme includes problems related to software testing. The problems are described in Table 9. The most discussed testing problems were ambiguous test result, flaky tests and time-consuming testing, all of them being mentioned in at least six cases.

Ambiguous test result. An ambiguous test result means that the test result does not guide the developer to action:

> *...several of the automated activities do not yield a clear "pass or fail" result. Instead, they generate logs, which are then inspected in order to determine whether there were any problems—something only a small minority of project members actually do, or are even capable of doing.*
>
> –Case C17a

Reasons for ambiguity can be that not every commit is tested [C2], analyzing the test result takes large amount of time [C4, C17a], the test results are not communicated to the developers [C6], there are no low-level tests that would pin point where the problem is exactly [C15] and that the tests may fail regardless of the code changes [C27]. In addition to increased effort to investigate the test result, ambiguity makes it also difficult to assign responsibility to fix issues and thus leads to lack of discipline [C6].

Flaky tests. Tests that cannot be trusted because they fail randomly can cause problems:

> *Test cases are sometimes unstable (i.e. likely to break or not reflecting the functionality to be tested) and may fail regardless of the code.*

Table 9
Testing problems.

Problem	Description
Ambiguous test result	Test result is not communicated to developers, is not an explicit pass or fail or it is not clear what broke the build.
Flaky tests	Tests that randomly fail sometimes.
Time-consuming testing	Testing takes too much time.
Hardware testing	Testing with special hardware that is under development or not always available.
Multi-platform testing	Testing with multiple platforms when developers do not have access to all of them.
UI testing	Testing the UI of the application.
Untestable code	Software is in a state that it cannot be tested.
Problematic deployment	Deployment of the software is time-consuming or error-prone.
Complex testing	Testing is complex, e.g., setting up environment.

Table 10
Release problems.

Problem	Description
Customer data preservation	Preserving customer data between upgrades.
Documentation	Keeping the documentation in-sync with the released version.
Feature discovery	Users might not discover new features.
Marketing	Marketing versionless system.
More deployed bugs	Frequent releases cause more deployed bugs.
Third party integration	Frequent releases complicate third party integration.
Users do not like updates	Users might not like frequent updates.
Deployment downtime	Downtime cannot be tolerated with frequent releases.

–Case C27

The flakiness can be caused by timing issues [C4], transient problems such as network outages [C6], test/code interaction [C8], test environment issues [C11], UI tests [C14] or determinism or concurrency bugs [C14, C22]. Flaky tests have caused lack of discipline [C14] and ambiguity in test results [C22, C27].

Time-consuming testing. Getting feedback from the tests can take too long:

> *One common opinion at the case company is that the feedback loops from the automated regression tests are too long. Regression feedback times are reported to take anywhere from four hours to two days. This highlights the problem of getting feedback from regression tests up to two days after integrating code.*
> –Case C27

If tests take too long, it can lead to larger commits [C3a], broken development flow [C3a] and lack of discipline [C11, C14]. Reported reasons for too long tests were unsuitable architecture [C3a] and unoptimized UI tests [C14].

Specific testing problems. From the rest of testing problems, *hardware testing, multi-platform testing* and *UI testing* problems are related in the sense that they refer to problems with specific kinds of tests. These tests make the testing more complex and require more effort to setup and manage automated testing:

> *...because the UI is the part of the system design that changes most frequently, having UI-based testing can drive significant trash into automated tests.*
> –Case C8

Other testing problems. Untestable code, problematic deployment and *complex testing*, are more general problems that relate to each other via system modularization. For example, system modularization was claimed to make testing more complex:

> *To simplify the development process, the platform was modularised; this meant that each API had its own git repository. This also made testing more complex. Since the APIs and core components are under continuously development by groups which apply rapid development methodology, it would be very easy for certain API to break other components and even the whole platform despite having passed its own unit test.*
> –Case C21

System modularization was reported to cause problematic deployment [C25a, C25c] and complex testing [C21, C25c]. On the other hand, system modularization was claimed to make testing and the deployment of the individual parts of the system independent of other parts [C25b]. Thus, system modularization can remove the problem of untestable code and make deployment easier. Therefore one needs to find a balance between these problems when designing modularity.

4.1.5. Release problems

Release problems (see Table 10) cause trouble when the software is released. Release problems were reported only in one article [C5], with the exception of deployment downtime which was mentioned in two articles [C5, C25c].

The lack of evidence about release problems is a result on its own. Most of the articles focused on problems that were internal, whereas release problems might be external to the developers. The exceptional article [C5] focused more on the impact of CD externally, which is one of the reasons it included multiple release challenges. To get more in-depth understanding of the release problems, readers are encouraged to read the Article P6.

4.1.6. Human and organizational problems

Human and organizational problems are not related to any specific development activity, but are general problems that relate to human and organizational aspects in CD adoption. These problems are described in Table 11. The most reported problems in this theme were lack of discipline, lack of motivation and lack of experience.

Lack of discipline. Sometimes the software organization as a whole cannot keep to the principles defined for the CD discipline:

> *The second limitation is that violations reported by automated checks can be ignored by developers, and unfortunately often they are.*
> –Case C12

Table 11
Human and organizational problems.

Problem	Description
Lack of discipline	Discipline to commit often, test diligently, monitor the build status and fix problems as a team.
Lack of motivation	People need to be motivated to get past early difficulties and effort.
Lack of experience	Lack of experience practicing CI or CD.
More pressure	Increased amount of pressure because software needs to be in always-releasable state.
Changing roles	Different roles need to adapt for collaboration.
Team coordination	Increased need for team coordination.
Organizational structure	Organizational structure, e.g., separation between divisions causes problems.

Table 12
Resource problems.

Problem	Description
Effort	Initially setting up continuous delivery requires effort.
Insufficient hardware resources	Build and test environments require hardware resources.
Network latencies	Network latencies hinder continuous integration.

This can mean discipline to committing often [C1], ensuring sufficient automated testing [C1], fixing issues found during the integration immediately [C6, C10, C12, C14] and testing changes on a developer machine before committing [C11]. Weak parts of a CI system can cause lack of discipline: ambiguous test result can make it difficult to determine who should fix integration issues [C6], time-consuming testing can make developers skip testing on their own machines [C11] and having flaky tests or time-consuming testing can lead to ignoring tests results [C14].

Lack of motivation. Despite the proposed benefits of CD, everyone might not be motivated to adopt it. But in order to achieve discipline, one must involve the whole organization to practice CD [C5]. This is especially difficult when there seems to be no time for improvement:

> *But it was hard to convince them that we needed to go through our implementation "hump of pain" to get the pieces in place that would allow us to have continuous integration. I worked on a small team and we didn't seem to have any "extra" time for me to work on the infrastructure we needed.*
> –Case C19

In addition to required effort [C19], lack of motivation can be caused by skepticism about how suitable CD is in a specific context [C27].

Lack of experience. Having inexperienced developers can make it difficult to practice CD:

> *[Challenge when adopting CD:] a lack of understanding of the CD process by novice developers due to inconsistent documentation and a lack of industry standards.*
> –Case C5

Lack of experience can cause lack of understanding [C5, C12] and people easily drift into using old habits [C27]. Lack of experience can lead to a feeling of *more pressure* when the change is driven in the organization:

> *Despite the positive support and attitude towards the concept of CI, teams feel that management would like it to happen faster than currently possible which leads to increased pressure. Some developers feel that they lack the confidence and experience to reach desired integration frequencies.*
> –Case C27

Other human and organizational problems. *Changing roles*, *team coordination* and *organizational structure* were mentioned only briefly in single cases and little evidence for them is presented. Thus they are not discussed here in depth.

4.1.7. Resource problems

The resource problems were related to the resources available for the adoption. The problems are listed in Table 12. Effort was reported in six cases, insufficient hardware resources in two cases and network latencies in one case.

Effort. Effort was mentioned with two different meanings. First, if the build system is not robust enough, it requires constant effort to be fixed:

> *The Bubble team expended significant effort working on the build. The build was very complex with automated application server deployment, database creation and module dependencies.*
> –Case C3a

Second, at the start of the adoption, an initial effort is needed for setting up the CD system and for monitoring it:

> *Continually monitoring and nursing these builds has a severe impact on velocity early on in the process, but also saves time by identifying bugs that would normally not be identified until a later point in time. It is therefore extremely important to get the customer to buy into the strategy (...) While initially setting up the framework is a time-consuming task, once this is accomplished, adding more such builds is not only straightforward, but also the most natural approach to solving other "non-functional" stories.*
> –Case C4

Effort is needed for implementing the CI system [C2, C4, C19, C26], monitoring and fixing broken builds [C3a, C4], working with a complex build [C3a], working with multiple branches [C8], working with multiple components [C17e] and maintaining the CI system [C26]. According to one case, the perceived initial effort to implement the CI system can cause a situation where it is difficult to motivate stakeholders for the adoption [C19].

Hardware resources. Hardware resources are needed for test environments, especially robustness and performance tests:

> *Robustness and performance builds tend to be resource-intensive. We chased a number of red-herrings early on due to a poor environment. It is important to get a good environment to run these tests.*
> –Case C4

Also *network latencies* cannot be tolerated, if present, they disrupt committing small changes [C13].

Table 13
Reported causal explanations.

Theme	Causes
Build design	inflexible build → complex build [C3a] system modularization → complex build [C2]
System design	–
Integration	complex build → broken build [C3a] broken build → work blockage [C3a] broken build → broken development flow [C3a] work blockage → merge conflicts [C3a] large commits → merge conflicts [C3a] time-consuming testing → broken development flow [C3a] time-consuming testing → large commits [C3a] network latencies → large commits [C13] slow integration approval → large commits [C17a] merge conflicts → broken build [C3a] long-running branches → merge conflicts [C14] flaky tests → broken build [C6]
Testing	unsuitable architecture → untestable code [C22] unsuitable architecture → time-consuming testing [C3a] system modularization → complex testing [C21, C25c] system modularization → problematic deployment [C25a, C25c] flaky tests → ambiguous test result [C22, C27]
Release	–
Human & organizational	time-consuming testing → lack of discipline [C11, C14] flaky tests → lack of discipline [C14] effort → lack of motivation [C19] ambiguous test result → lack of discipline [C6] lack of experience → more pressure [C27]
Resource	complex build → effort [C3a] broken build → effort [C3a] unsuitable architecture → effort [C8] system modularization → effort [C17e]

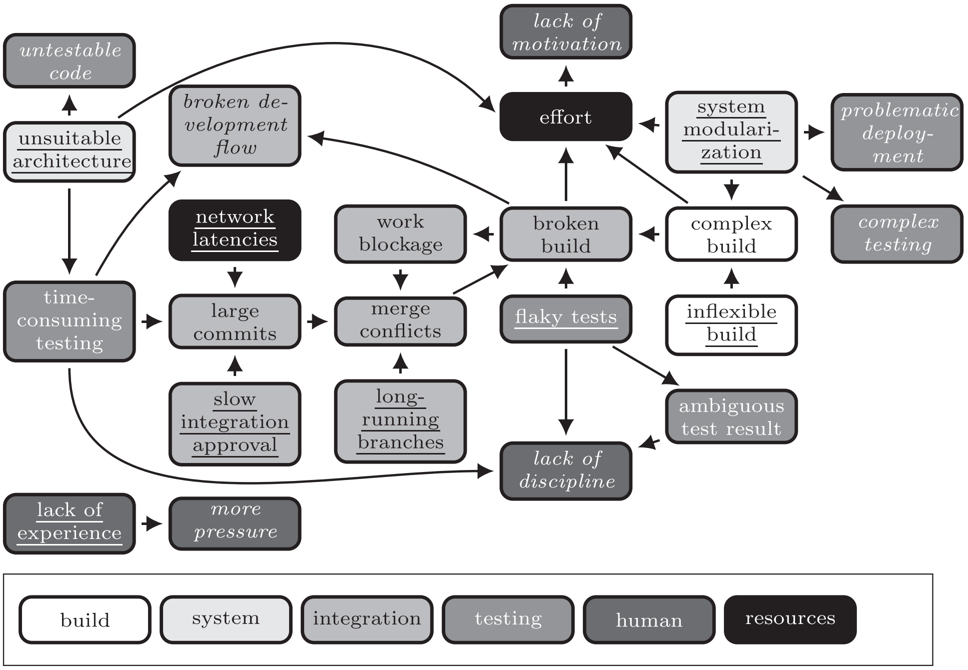


Fig. 7. All reported causal explanations. Different themes are highlighted with colors. In addition, *roots* that do not have any underlying causes are underlined and *leafs* that do not have any effects are in italics.

4.2. Causes of problems

To study the causes of the problems, we extracted reported causal explanations from the articles, see Table 13 and Fig. 7.

4.2.1. Causes of build design problems

There were two reported causes for build design problems: inflexible build and system modularization. The first problem was synthesized under the build design problem theme and the second under the system design problem theme. This indicates that the build design is affected by the system design.

4.2.2. Causes of system design problems

No reported causes for system design problems were reported. This indicates that system design activity is one of the root causes

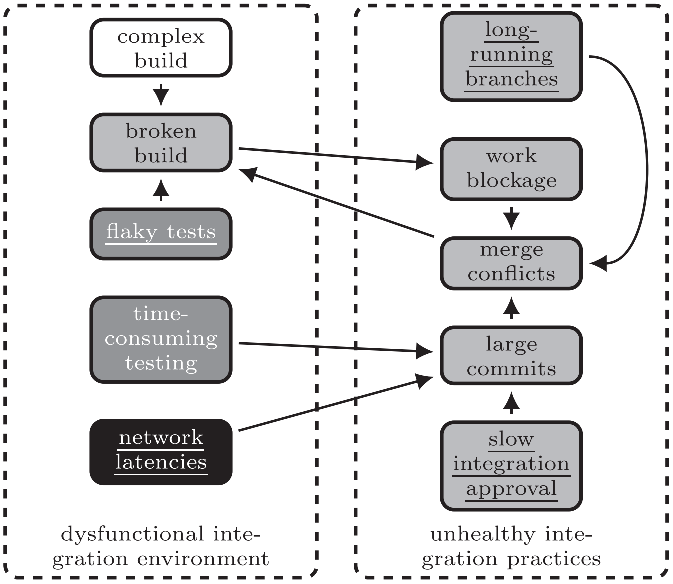


Fig. 8. Causes for integration problems from Fig. 7, grouped into dysfunctional integration environment and unhealthy integration practices.

for CD adoption problems or at least there are no known causes for the system design problems.

4.2.3. Causes of integration problems

Integration problems were caused by three problem themes: build design problems [C3a], integration problems [C3a, C13, C14, C17a] and testing problems [C3a, C6]. Especially interesting is the vicious cycle inside the integration problem theme between the problems merge conflicts, broken build and work blockage.

The causes of the integration problems could be separated to two higher level root causes (Fig. 8): dysfunctional integration environment (complex build, broken build, time-consuming testing, network latencies) and unhealthy integration practices (work blockage, large commits, merge conflicts, long-running branches, slow integration approval). However, since there is a causal relationship both ways, e.g., time-consuming testing causing large commits and merge conflicts causing broken builds, one cannot solve any of the high-level causes in isolation. Instead, a holistic solution has to be found.

4.2.4. Causes of testing problems

Testing problems were caused by system design problems [C3a, C21, C22, C25a, C25c] and other testing problems [C22, C27]. The relationship between system design and testing is common knowledge already and test-driven development (TDD) is a known solution for developing testable code. The new finding here is that system design also has an impact on testing as a part of CD.

4.2.5. Causes of release problems

No reported causes for release problems were mentioned. This was not surprising, given that only two articles discussed release problems. Further research is needed in this area.

4.2.6. Causes of human and organizational problems

Human and organizational problems were caused by testing problems [C6, C11, C14], resource problems [C19] and other human and organizational problems [C27]. Interestingly, all testing problems that were causes of human and organizational problems caused lack of discipline. Those testing problems were time-consuming testing, flaky tests and ambiguous test result. If testing activities are not functioning properly, there seems to be an urge to stop caring about testing discipline. For example, if tests are time-consuming, running them on developer's machine before committing might require too much effort and developers might skip running the tests [C11]. Furthermore, if tests are flaky or test results are ambiguous, then test results might not be trusted and ignored altogether [C6, C14].

Another interesting finding is that human and organizational problems did not cause problems in any other problem theme. One explanation considering some of the problems is that the problems are not root causes but instead symptoms of other problems. This explanation could apply to, e.g., lack of discipline problem. An alternative explanation for some of the problems is that the problems cause other problems, but the causal relationships have not been studied or reported in the literature. This explanation applies to, e.g., organizational structure, because it is explicitly claimed to cause problems when adopting CD [C26], but the actual effects are not described.

4.2.7. Causes of resource problems

The only resource problem that had reported causes was effort. Build design problems [C3a], system design problems [C8, C17e] and integration problems [C3a] were said to increase effort.

4.3. Contextual variance of problems

We categorized each case based on publication time, number of developers, CD implementation maturity and commerciality, as shown in Appendix B. There are some interesting descriptive notions based on the categorization:

- All cases with large number of developers were both post 2010 and commercial.
- Almost all (10/11) non-commercial cases had a medium number of developers.
- Almost all (9/10) CD cases were commercial cases.
- Most (8/10) of the CD cases were post 2010, but there were also many (15/25) post 2010 CI cases.
- Most (18/24) of the commercial cases were post 2010, while the majority (6/11) of the non-commercial cases were pre 2010.

For each case category, we calculated the percentage of cases that had reported distinct problem themes (Table 14). Next, we summarize the findings individually for each of our grouping variables (Figs. 9 and 10). We emphasize that these are purely descriptive measures and no statistical generalization is attempted to be made based on the measures. Thus, no conclusion regarding popularity can be made based on these measures.

Publication time. Based on the time of reporting, the only clear difference between pre 2010 and post 2010 cases is seen on the system design problem theme: post 2010 cases reported over four times more often system design problems than pre 2010 cases. A smaller difference is on the resource theme where pre 2010 cases reported 50% more often problems than post 2010 cases.

Number of developers. Integration and testing problems are reported more often by cases with larger number of developers. In contrast, cases with small number of developers reported resource problems more often.

Continuous delivery implementation maturity. CD cases reported problems more often in every other theme than build design and integration. The clearest differences are in the system design, human and organizational and resource themes. In addition, the CI cases reported problems more often in the testing theme.

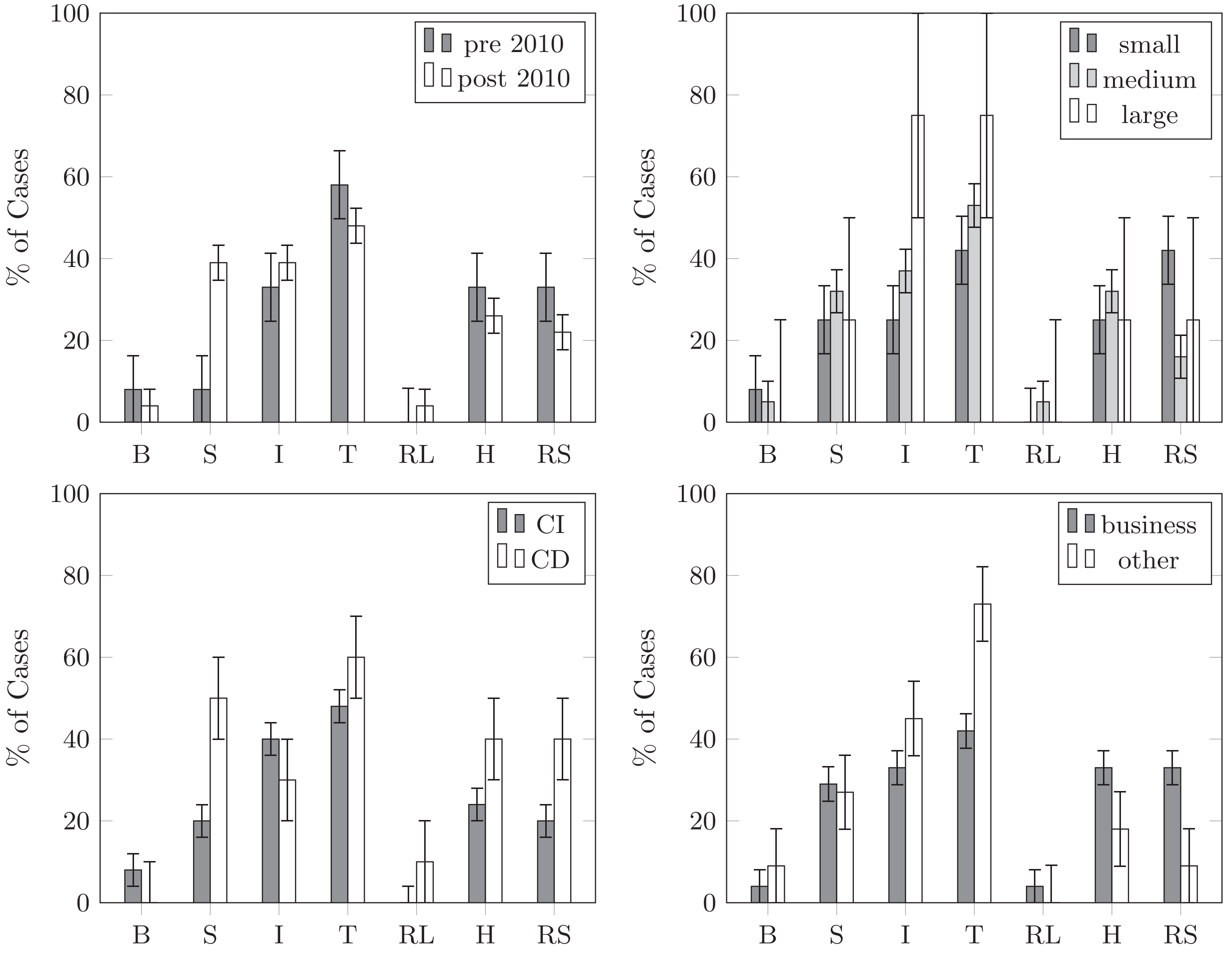


Fig. 9. Comparison of reported problems in different case categories. **B** = Build Design, **S** = System Design, **I** = Integration, **T** = Testing, **RL** = Release, **H** = Human and Organizational, **RS** = Resource. Error bars visualize an error of ± 1 case.

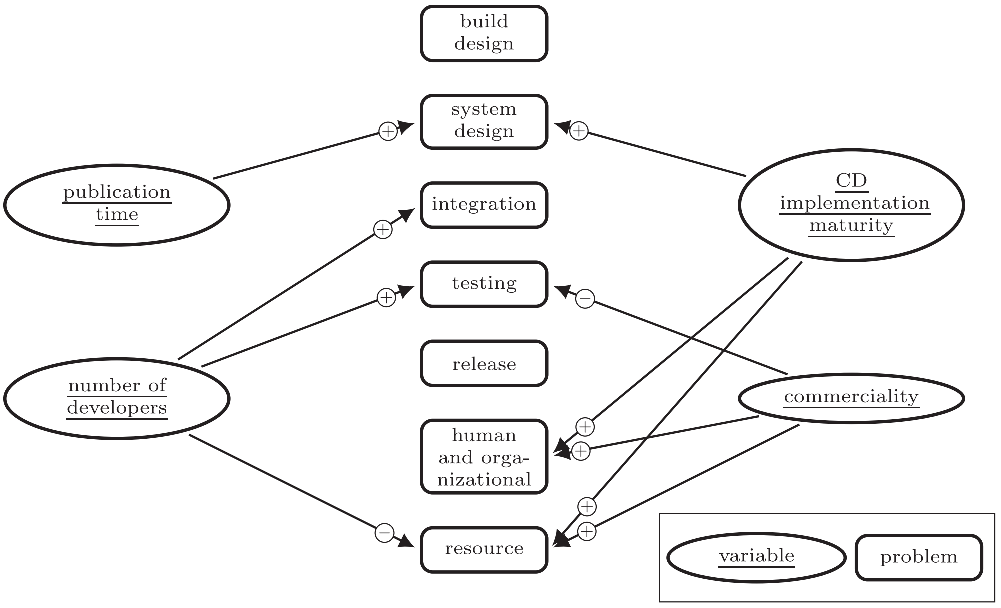


Fig. 10. Contextual differences of different problem themes based on Fig. 9. The '+'-sign denotes that problems were reported more often and the '−'-sign denotes that problems were reported less often in cases where the contextual variable was higher.

Table 14
Percentage of cases in a category that reported problems in a theme. For example, the percentage "58%" in the crossing of "Pre2010" and "Testing" means that 58% of the pre 2010 cases reported at least one testing problem.

Case	Theme						
category	Build	System	Integration	Testing	Release	Human	Resource
Pre 2010	8%	8%	33%	**58%**	0%	33%	33%
Post 2010	4%	39%	39%	**48%**	4%	26%	22%
Small	8%	25%	25%	**42%**	0%	25%	**42%**
Medium	5%	32%	37%	**53%**	5%	32%	16%
Large	0%	25%	**75%**	**75%**	0%	25%	25%
CI	8%	20%	40%	**48%**	0%	24%	20%
CD	0%	50%	30%	**60%**	10%	40%	40%
Non-commercial	9%	27%	45%	**73%**	0%	18%	9%
Commercial	4%	29%	33%	**42%**	4%	33%	33%

Table 15
The most critical problems in each case where there was any. The method for determining different kinds of critical problems is described in Section 3.4.5.

Case	Explicit	Implicit	Causal
C3a			Inflexible build, time-consuming testing
C4	Ambiguous test result		
C5	Internal dependencies		
C6		Broken build, ambiguous test result	
C8		Unsuitable architecture, broken build	
C11	Time-consuming testing		
C14		Flaky tests, time-consuming testing	
C17a		Slow integration approval	
C17e		System modularization	
C19		Lack of motivation	
C21	Multi-platform testing		
C25a		Problematic deployment	System modularization
C25c		Unsuitable architecture	
C26	Organizational structure		

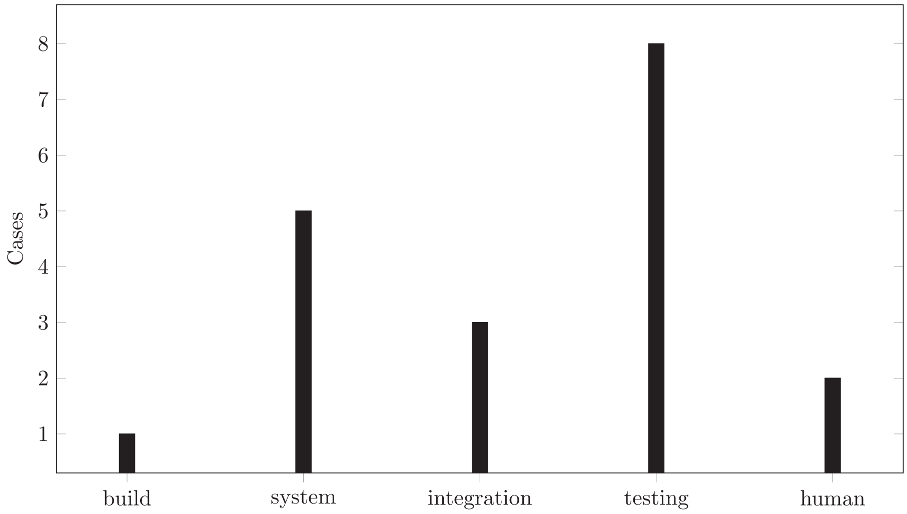


Fig. 11. Number of cases with critical problems in problem themes.

Commerciality. Commercial cases reported more often human and organizational and resource problems. Non-commercial cases reported more often testing problems than commercial cases.

4.4. Criticality of problems

The most critical problems for each case are listed in Table 15 and summarized by problem theme in Fig. 11. The most critical themes are system design and testing problems. Human and organization and integration problems were reported critical in a smaller number of cases. Build design problems were reported critical in one case and no critical release or resource problems was found.

Inflexible build was a critical build design problem in a single case [C3a], where the case suffered from build complexity caused by sharing the build system over multiple teams. The complexity required extensive build maintenance effort. One should pay attention to build design when adopting CD, in order to avoid large build maintenance effort.

The most critical system design problems were internal dependencies, unsuitable architecture and system modularization. Thus, the architecture of the system as a whole can be seen as critical for successful CD adoption. Dependencies cause trouble when

Table 16
Solutions given in articles.

Theme	Solutions
System design	System modularization, hidden changes, rollback, redundancy
Integration	Reject bad commits, no branches, monitor build length
Testing	Test segmentation, test adaptation, simulator, test parallelization, database testing, testing tests, comprehensive testing, commit-by-commit tests
Release	Marketing blog, separate release processes
Human and organizational	Remove blockages, situational help, demonstration, collaboration, social rules, more planning, low learning curve, training, top-management strategy, communication
Resource	Tooling, provide hardware resources

a change in one part of the system conflicts with other parts of the system [C5]. Architecture can be unsuitable if different configurations are developed in branches instead of using configuration properties [C8], or if web services are causing latencies, deployment and version synchronization issues [C25c]. Finally, system modularization taken into too granular level causes additional overhead [C17e] and consolidating multiple modules together can simplify a complicated deployment process [C25a].

Broken build and slow integration approval were the most critical integration problems. In all of the cases broken build caused the problem work blockage, that no further work could be delivered because of broken build. Broken build also switches off the feedback mechanism of CD; developers do not receive feedback about their changes anymore and technical debt can accumulate. Slow integration approval was a critical problem in case C17a, because it slowed down the integration frequency.

The most critical testing problems were time-consuming testing, ambiguous test result, flaky tests, multi-platform testing and problematic deployment. Out of these, time-consuming testing was the most critical in three cases, and ambiguous test result was the most critical in two cases. The rest were critical in single cases. Time-consuming testing, ambiguous test result and flaky tests are, similar to critical integration problems, related to the feedback mechanism CD provides. Either feedback is slowed down or its quality is weakened. Multi-platform testing makes testing more complex and it requires more resources to be put into testing, in terms of hardware and effort [C21]. Finally, problematic deployment can be error-prone and time-consuming [C25a].

The most critical human and organizational problems were organizational structure and lack of motivation. Organizational structure was explicitly said to be the biggest challenge in an organization with separate divisions [C26]. Finally, lack of motivation was a critical problem in a case where the benefits needed to be demonstrated to the developers [C19].

4.5. Solutions

Solutions were thematically synthesized into six themes. The themes were the same as for the problems, except that build design theme did not have any solutions, probably because build problems were discussed in two articles only. The solutions in the themes are listed in Table 16.

4.5.1. System design solutions

Four system design solutions were reported: *system modularization, hidden changes, rollback* and *redundancy* (Table 17). The design solutions considered what kind of properties the system should have to enable adopting CD.

System modularization. System modularization was already mentioned to be a problem, but it was also reported as a solution. System modularization can prevent merge conflicts, because developers work on different parts of the code [C2]. Also, individual modules can be tested in isolation and deployed independently [C25b]. However, because of the problems reported with system modularization, it should be applied with caution.

Hidden changes. Hidden changes include techniques how to develop large features and other changes incrementally, thus solving the problem of large commits. One such technique is feature toggles: parts of new features are integrated frequently, but they are not visible to the users until they are ready and a feature toggle is switched on in the configuration [C7, C14]. Another technique is branch by abstraction, which allows doing large refactoring without disturbing other development work [C7]. Instead of creating a branch in version control, the branch is created virtually in source code behind an abstraction. This method can be also used for database schema changes [C7].

Rollback and redundancy. Rollback and redundancy are properties of the system and are important when releasing the system. Rollback means that the system is built so that it can be downgraded automatically and safely if a new version causes unexpected problems [C5]. Thus, rollback mechanism reduces the risk of deploying more bugs. Redundancy means that the production system contains multiple copies of the software running simultaneously. This allows seamless updates, preserving customer data [C5] and reducing deployment downtime [C5, C25c].

4.5.2. Integration solutions

Three integration solutions were reported: *reject bad commits, no branches* and *monitor build length* (Table 18). The integration solutions are practices that take place during integration.

Reject bad commits. Reject bad commits is a practice where a commit that is automatically detected to be bad, e.g., fails some tests, is rejected from entering the mainline. Thus, the mainline is always functional, builds are not broken [C8] and discipline is enforced [C12].

No branches. No branches is a discipline that all the development is done in the mainline and no other branch is allowed. This prevents possible problems caused by long-running branches [C7, C14]. To make the no branch discipline possible, the hidden changes design solution has to be practiced to make larger changes.

Monitor build length. Monitor build length is a discipline where keeping the build length short is prioritized over other tasks. A certain criteria for build length is established and then the build is monitored and actions are taken if the build length grows too long [C3b].

4.5.3. Testing solutions

Eight testing solutions were reported: *test segmentation, test adaptation, simulator, test parallelization, database testing, testing tests, comprehensive testing* and *commit-by-commit tests* (Table 19). Testing solutions are practices and solutions applied for testing.

Table 17
System design solutions reported in articles.

Solution	Solves	Description
System modularization	Merge conflicts [C2], untestable code [C25b], problematic deployment [C25b]	Modularize the system to units that can be independently tested and deployed.
Hidden changes	Large commits [C5, C7, C14], database schema changes [C7]	Enable incremental development of large features and changes with feature toggles and branch by abstraction.
Rollback	More deployed bugs [C5]	Build a rollback mechanism to revert updates if critical bugs emerge.
Redundancy	Customer data preservation [C5], deployment downtime [C5, C25c]	Employ redundancy in production systems to allow seamless upgrades.

Table 18
Integration solutions reported in articles.

Solution	Solves	Description
Reject bad commits	Broken build [C8], lack of discipline [C12]	Automatically reject commits that would break the build.
No branches	Long-running branches [C7, C14]	To prevent long-running branches causing problems, use a no-branch policy.
Monitor build length	Time-consuming testing [C3b]	Team actively monitors build length and takes action when it grows too long.

Table 19
Testing solutions reported in articles. Claimed solutions are marked with a star (*).

Solution	Solves	Description
Test segmentation	Time-consuming testing [C2, C3a, C13]	Segment tests based on speed, criticality and functionality. Solves time-consuming testing by running the most critical tests first and others later only if the first tests pass.
Test adaptation	Hardware testing [C1, C8], ambiguous test result [C15(*)]	Tests are adapted so that later/manual tests are run earlier/automatically or vice versa. Hardware tests can be run with simulator. Solves ambiguous test result problem when earlier tests point to the root cause of failure faster than in later end-to-end tests.
Simulator	Hardware testing [C1, C8]	Custom hardware can be tested efficiently with a software simulator.
Test parallelization	Time-consuming testing [C1, C14]	Parallelizing tests to run simultaneously and on multiple machines speeds up testing.
Database testing	Database schema changes [C5]	Database schema changes can be tested similarly to other changes.
Testing tests	Flaky tests [C14]	Tests can be tested for flakiness.
Comprehensive testing	Multi-platform testing [C2]	Ensure that every platform is tested.
Commit-by-commit tests	Ambiguous test result [C2]	When tests are run for every commit, it is possible to know which change was responsible for a failure.

Test segmentation and adaptation. Two solutions were related to the organization of test cases: test segmentation and test adaptation. Test segmentation means that tests are categorized to different suites based on functionality and speed. This way, the most critical tests can be run first and other and slower tests later. Developers get fast feedback from the critical and fast tests [C2, C13]. Thus, test segmentation partially solves time-consuming testing problem. One suggested solution was to run only the tests that the change could possibly have an effect on. However, this does not solve the problem for holistic changes that have an effect on the whole system [C3a].

Test adaptation is a practice where the segmented test suites are adapted based on the history of test runs. For example, a manual test that has revealed a defect should be, if possible, automated [C1]. Also an automated test that is run later but fails often should be moved to be run earlier to provide fast feedback [C8]. Another way test adaption is claimed to help is solving the problem of ambiguous test result. When a high-level test fails, it might be difficult and time-consuming to find out why the fault occurred. Therefore it is advised that low-level tests are created which reproduce the fault and give an explicit location where the cause of the fault is [C15].

Together with test adaptation, *simulator* solution can be used for hardware testing. The benefits of the simulator are running otherwise manual hardware tests automatically and more often [C1, C8]. In addition, a simulator can run tests faster and more test combinations can be executed in less time than with real hardware [C1].

Test parallelization. Test parallelization means executing automated tests in parallel instead of serially, decreasing the amount of time to run the tests [C1, C14]. Tests can be run concurrently on a single machine or they can be run on several machines. This solution requires enough hardware resources for testing.

Database testing and testing tests. Database testing means that database schema changes are tested in addition to source code changes [C5]. Thus, they do not cause unexpected problems in the production environment. Testing tests means that even tests can be tested for flakiness [C14].

Comprehensive testing and commit-by-commit tests. Finally, comprehensive testing means that every target platform should be tested [C2]. Commit-by-commit tests means that every change should be tested individually, so when confronted with failing tests it can be directly seen which change caused the failure [C2]. It is often instructed that tests should be run for every commit in the commit stage of CD (see Fig. 1). However, the further stages can be more time-consuming and it might not be feasible to run the

Table 20
Release solutions reported in articles.

Solution	Solves	Description
Marketing blog	Feature discovery [C5], marketing [C5]	Instead of marketing individual versions, concentrate on features and blog about them.
Separate release processes	Users do not like updates [C5]	Let users decide whether they receive new updates or not.

Table 21
Human and organizational solutions reported in articles. Claimed solutions are marked with a star (*).

Solution	Solves	Description
Remove blockages	Broken build [C5, C6(*)], merge conflicts [C5], work blockage [C5]	Keeping the build unbroken and removing any blockages is the responsibility and highest priority for whole team.
Situational help	Lack of experience [C12]	Providing help based on the situation at hand.
Demonstration	Lack of motivation [C6, C19]	Demonstrate the value of continuously running test suite.
Collaboration	Changing roles [C5], organizational structure [C26]	Instead of individual responsibility, the organization as a whole should be responsible for delivery.
Social rules	Lack of experience [C5]	Adopt social rules that are easy to follow even by novices.
More planning	Team coordination [C5]	Apply more planning to coordinate teams.
Low learning curve	Lack of experience [C5]	Organize the adoption of continuous delivery so that no leap of expertise is needed.
Training	Lack of discipline [C1]	Make sure that the whole team is trained to practice continuous delivery.
Top-management strategy	Lack of motivation [C5]	Top-management can give a sense of direction for larger groups of people.
Communication	More pressure [C5]	Communicate feelings of pressure to relieve it.

stages for every commit. Comprehensive testing and commit-by-commit tests ensure testing completeness and granularity. However, achieving both is tricky because comprehensive tests take more time and it might not be feasible to run them for each commit. Thus, test segmentation becomes necessary; certain tests are executed for each commit but more comprehensive tests are executed more seldom.

4.5.4. Release solutions

There were two reported release solutions: *marketing blog* and *separate release processes* (Table 20). A marketing blog can be used for marketing a versionless product and users can discover new features at the blog [C5]. There might be certain user groups that dislike the frequent updates, and a *separate release processes* could be used for them [C5].

4.5.5. Human and organizational solutions

There were ten reported human and organizational solutions: *remove blockages, situational help, demonstration, collaboration, social rules, more planning, low learning curve, training, top-management strategy* and *communication* (Table 21).

Remove blockages. Remove blockages is a practice that when a specific problem occurs, the whole team stops what they are doing and solves the problem together. The problem can be either broken build [C5, C6], merge conflicts [C5] or any other work blockage:

> *"Atlassian ensures that its OnDemand software is always deployable by immediately stopping the entire team from performing their current responsibilities and redirecting them to work on any issue preventing the software from being deployed."*
> –Case C5

Organizational culture change. The rest of the human and organizational solutions are related to the adoption as an organizational culture change. The organization should support more closer *collaboration* to adopt CD [C5, C26]. The change should be supported with a *top-management strategy* [C5] and with *more planning* how to organize the work [C5].

To reduce learning anxiety, *low learning curve* should be achieved during the adoption [C5]. *Situational help* can be provided, meaning that personal help is given when needed [C12]. The system and value of it can be *demonstrated* to further motivate and train stakeholders [C6, C19]. More formal *training* can be given to teach specific skills [C1] and *social rules* can be adopted to ensure a standardized process. Finally, a culture of open *communication* should be established to relieve the pressure caused by the change [C5].

4.5.6. Resource solutions

There were two reported resource solutions: *tooling* and *provide hardware resources* (Table 22).

Tooling. Tooling is necessary to achieve discipline [C1], make test results less ambiguous [C4], manage versionless documentation [C5] and execute database schema changes in conjunction with source code [C25c]. In addition, it was claimed in two sources that setting up the initial CD environment takes a lot of effort and if there was a standardized tooling available, it would make this effort smaller [C2, C26].

Provide hardware resources. Providing hardware resources can be done to solve time-consuming testing [C2, C11] and otherwise insufficient hardware resources [C4].

5. Discussion

In this section, we answer the research questions of the study and discuss the results. We also discuss the overall limitations of the study.

5.1. RQ1: What continuous delivery adoption problems have been reported in major bibliographic databases?

We found 40 distinct CD adoption problems that were synthesized into seven themes: build design, system design, integration, testing, release, human and organizational, and resource problems. Testing and integration problems were discussed the most (Fig. 5). Thus, it seems that less studied themes are system design, human and organizational, and resource problems, albeit that they were still studied in several cases. Build design and release problems were discussed in two cases only and are the least studied problems. In addition to problem quantity in the articles, we found that testing and system design problems are the most critical in a large number of cases (Fig. 11).

We believe that testing and integration problems are studied the most, because they relate directly to the CI practice and thus have been studied longer than other problems. CD, being a more

Table 22
Resource solutions reported in articles. Claimed solutions are marked with a star (*).

Solution	Solves	Description
Tooling	Lack of discipline [C1], ambiguous test result [C4], documentation [C5], database schema changes [C25c], effort [C26(*), C2 (*)]	Provide tooling to make the process easier to follow, to allow interpreting the test result and to document a changing software system.
Provide hardware resources	Time-consuming testing [C2, C11], insufficient hardware resources [C4]	Provide hardware resources for production-like test environments and for parallelization if tests are too time-consuming.

recent practice, has not been studied that much, and it could be that the other problems emerge only after moving from the CI practice to CD practice. In addition, technical aspects are also more frequently studied in software engineering in general, in comparison to the human and organizational issues.

No other secondary study has considered problems when adopting CD directly. Some of the attributes of the CI process model developed by Ståhl and Bosch [9] relate to the problems we found. For example, build duration relates to the time-consuming testing problem. Thus, based on our study, the elements of the model could be connected to the found problems and this could help the users of the model to discover problems in their CI process. After discovering the problems, the users could decide on necessary solutions, if they want to adopt CD.

Some of the adoption actions described by Eck et al. [10] are related to the problems we found. For example, one of the adoption actions was decreasing test result latency, which relates with the time-consuming testing problem. Although Eck et al. ranked the adoption actions based on the adoption maturity, the ranking cannot be compared to our categorization of initial and advanced cases. The ranking by Eck et al. considered adoption maturity, while our categorization considered technical maturity. It would have been difficult to interpret the adoption maturity from the articles. Nevertheless, the ranking created by Eck et al. allows relating the problems we found to the adoption maturities of the cases. For example, using the ranking, it can be said that cases with the broken build problem are less mature than cases solving the time-consuming testing problem.

Other related literature studies that studied problems did not study CD adoption problems but instead problems of CD [7] and rapid releases [6]. Thus, they identified problems that would emerge after adoption, not during it. Nevertheless, Rodriguez et al. [7] identified that the adoption itself is challenging and that additional QA effort is required during CD, which is similar to our finding in the resource problem theme. However, their study was a systematic mapping study and their intention was not to study the problems in depth, but instead discover what kind of research has been done in the area.

Some of the identified CD adoption problems are also CI adoption problems, but some are not. For example, build design and integration problems are clearly CI adoption problems. System design and testing problems are not as strictly CI adoption problems, as some of the problems consider deployments and acceptance testing which are not necessarily included in CI. Release problems are not related to the adoption of CI at all. It is even questionable are they really CD adoption problems or more specifically rapid release adoption problems, since CD does not imply releasing more often (difference between CD and rapid releases discussed in Section 2.4). Human and organizational and resource problems consider both CI and CD adoptions.

Although we achieved to identify different kinds of adoption problems and their criticality, we cannot make claims how widespread the problems are and why certain problems are more critical than others. These limitations could be addressed in future studies that surveyed a larger population or investigated individual cases in depth.

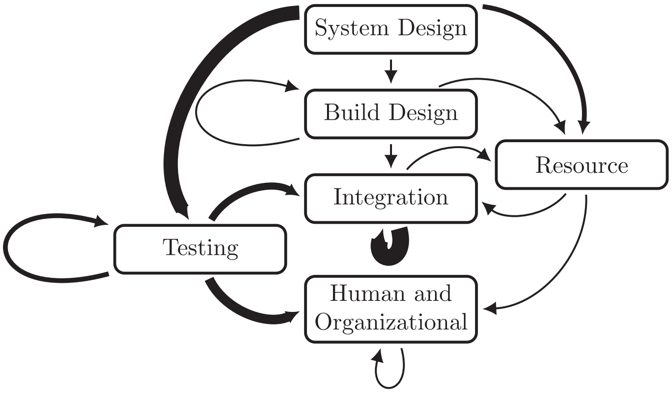


Fig. 12. Causal relationships between themes. Release theme did not have reported causal relationships. The widths of the arrows are proportional to the number of causes between themes and the number of cases that reported the causes.

5.2. RQ2: What causes for the continuous delivery adoption problems have been reported in major bibliographic databases?

Causes for the adoption problems were both internal and external of the themes (Fig. 12). System design problems did not have causes in other themes. Thus, system design problems can be seen as root causes for problems when adopting CD. In addition, human and organizational problems did not lead into problems in other themes. Therefore, one could claim that these problems seem to be only symptoms of other problems based on the evidence.

The design and testing themes had the largest effect on other themes. In addition, the integration theme had a strong internal causal loop. Thus, one should focus first on design problems, then testing problems, and finally integration problems as a whole. Otherwise one might waste effort on the symptoms of the problems.

Based on the contextual analysis (Fig. 10), more problems are reported by post 2010, large and commercial cases that are aiming for higher CD implementation maturity. We suspect that more problems emerge in those contexts and that CD as a practice is especially relevant in those contexts. However, the selected articles did not provide deep enough analysis on the connection between the contextual variables and faced adoption problems. Since the primary studies did not analyze the causal relationships between the contextual variables and the challenges, it is not possible to make such conclusions in this study either, merely based on the contextual classification of the cases. In addition, the study population was not appropriate for drawing statistical conclusions. This could be a good subject for future studies.

The reason for the lack of contextual analysis in previous studies might be that the effort to conduct rigorous studies about the causes of problems is quite high. This is because in the context of software development, problems are often caused by multiple interacting causes [16], and understanding them requires a lot of careful investigation.

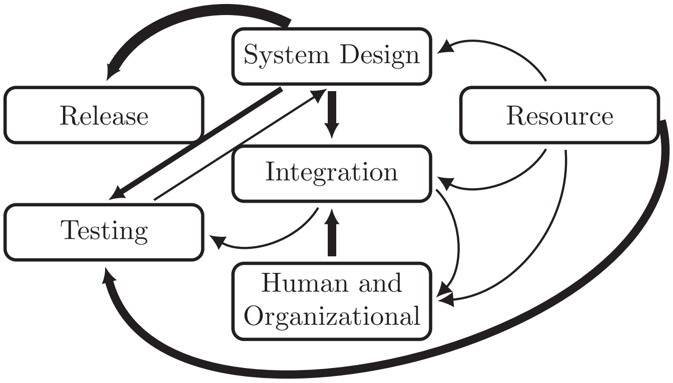


Fig. 13. Solutions between themes. Each theme had internal solutions. The widths of the arrows are proportional to the number of solutions between themes and the number of cases that reported the solutions.

The analyzed cases were from multiple kinds of development contexts (see Appendix B) and there were no substantial contextual differences regarding the problems and solutions, except for the obvious differences, e.g., that network latencies can be a problem only for distributed organizations. Thus, it seems that otherwise the problems and their solutions are rather general in nature.

We see that the amount of identified causal relationships does not yet cover the whole phenomenon of CD adoption. For 40 identified concepts of problems, we identified 28 causal relationships between the concepts, which seems to be less than expected. In contrast, when studying software project failures [16], the amount of identified causal relationships is much higher. We believe this was caused by the fact that academic articles are not necessarily the best material for causal analysis if the research focus of the articles is not to identify causal relationships. In future studies, causal analysis could be done by investigating the causes in individual case studies.

No other secondary study researched causes of the problems when adopting CD and thus no comparison to other studies can be done regarding this research question.

5.3. RQ3: What solutions for the continuous delivery adoption problems have been reported in major bibliographic databases?

Besides that each solution theme had internal solutions, many solutions in themes solved problems in other themes (Fig. 13). Testing, human and organizational and release solutions clearly were solving most of the problems internally while other solutions solved more problems in other themes. All other problem themes have multiple and verified solutions except the build and system design problem themes. Because the system design problems were common, had a large causal impact and lacked specific solutions, they could be determined as the largest problems when adopting CD.

The found solutions can be compared to the work by Ståhl and Bosch [9]. For example, test separation and system modularization attributes relate to the solution test segmentation. Thus, our collected solutions can be used to extend the model developed by Ståhl and Bosch, giving some of the attributes a positive quality.

It seems that generally there are no unsolved CD adoption problems. Thus, in principle, adopting CD should be possible in various contexts. However, solving the adoption problems might be to costly for some organizations, and thus CD adoption might turn out to be unfeasible if the costs override the benefits. Organizations who are planning to adopt CD can use this article as a checklist to predict what problems might emerge during the adoption and estimate the costs of preventing those problems. One should not blindly believe that adopting CD is beneficial for everyone; instead, a feasibility study should precede the adoption decision.

5.4. Limitations

Most of the selected articles were experience reports. This limits the strength of evidence whether the causal relationships are real, whether the most critical problems were indeed the most critical and whether the solutions actually solved the problems.

The data collection and the analysis of the results in the study required interpretation. The filtering strategies contained interpretative elements and thus results from them might vary if replicated. During data extraction, some problems might have been missed and some problems might be just interpretations of the authors. This applies to causes and solutions too. The contextual categorization might be biased, because not all articles provided enough information to execute the categorization with more rigor.

The studied sample of cases was from major bibliographic databases. There might be more successful and more problematic cases outside this sample. Publication bias inherently skews the sample towards a view where there are less problems than in reality.

Most of the articles focused on CI instead of CD, which can be seen to threat the validity of the study. One of the reasons for the scarcity of CD studies is that the concept of CD was introduced in 2010 [1] and some of the older articles using the term CI actually could be compared to other CD cases. It was difficult to determine whether a case was indeed practicing CI or CD just based on the articles.

The difference between CI and CD is not clearly defined in common use, and even academics have used the term CI while referring to the definition of CD [10]. However, it is commonly agreed that practicing CD includes practicing CI too. Thus, depending on the starting point of a CD adopter, also CI adoption problems might be relevant if they have not been addressed beforehand.

Just based on the articles, we cannot claim that a certain case did not have a certain problem if it was not reported. To actually answer question such as, “What were the problems in a case?” and “What problems did the case not have?”, the results of this study need to be operationalized as a research instrument in field studies.

6. Conclusions

Software engineering practitioners have tried to improve their delivery performance by adopting CD. Despite the existing instructions, during the adoption practitioners have faced numerous problems. In addition, causes and solutions for the problems have been reported. In this study, we asked the following research questions and provided answers for them through a systematic literature review:

RQ1. **What continuous delivery adoption problems have been reported in major bibliographic databases?** Problems exist in the themes of build design, system design, integration, testing, release, human and organizational and resource.

RQ2. **What causes for the continuous delivery adoption problems have been reported in major bibliographic databases?** Causes exist mostly in the themes of system design and testing, while integration problems have many internal causal relationships.

RQ3. **What solutions for the continuous delivery adoption problems have been reported in major bibliographic databases?** All themes have solutions on their own, but themes of system design, resource and human and organizational have the most effect on other themes.

System design problems are mentioned in many articles, cause multiple other problems but lack support for solving them. Thus, they are the largest problems when adopting CD.

Compared to previous secondary studies, ours has dramatically increased the understanding of problems, their causes and solutions when adopting CD. We identified a larger number of problems and describe the causal chains behind the adoption problems. Our results improve the understanding of the problems by investigating their interconnected causes and help practitioners by proposing solutions for the problems.

Software development organizations who are planning to adopt CD should pay attention to the results of this study. First, investigate in which theme your problems reside. Second, use the reported causal chains to help reason about whether the problems might be caused by problems in another theme. Finally, implement the adequate solutions either for the problems or their causes.

6.1. Future work

The problems, causes and solutions should be investigated in further field studies. Especially system design problems would be interesting to research further, because they seemed to have a large impact but not many solutions. Individual problems and solutions could be studied to deepen the understanding of the problems and give more detailed instructions how to apply the solutions. The build design and release problems could be studied more, although studying release problems requires a rather mature case with a frequent release cadence.

In addition, human and organizational problems could be compared to more general theories of organizational change, decision making and learning. Is there something specific with adopting CD or can the problems be generalized for other kinds of change too? Based on our study, the current collection of human and organizational problems are generic for other kinds of changes.

Acknowledgments

This work was supported by TEKES as part of the Need for Speed research program of DIMECC (Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

Appendix A. Selected papers (rows in italics identify duplicate cases)

Paper	Case	Authors	Year	Title	Source
P1	C1	Basarke Christian, Berger Christian, Rumpe Bernhard	2007	Software & systems engineering process and tools for the development of autonomous driving intelligence	Journal of Aerospace Computing, Information and Communication
P2	C2	Betz Robin M., Walker Ross C.	2013	Implementing continuous integration software in an established computational chemistry software package	Software Engineering for Computational Science and Engineering (SE-CSE), 2013 5th International Workshop on
P3	*C2*	Betz Robin M., Walker Ross C.	2014	Streamlining Development of a Multimillion-Line Computational Chemistry Code	Computing in Science Engineering
P4	C3(a,b)	Brooks Graham	2008	Team Pace – Keeping Build Times Down	Agile Conference
P5	C4	Cannizzo Fabrizio, Clutton Robbie, Ramesh Raghav	2008	Pushing the Boundaries of Testing and Continuous Integration	Agile Conference
P6	C5	Claps Gerry, Svensson Richard Berntsson, Aurum Aybüke	2014	On the journey to continuous deployment: technical and social challenges along the way	Information and Software Technology
P7	C6	Downs John, Hosking John, Plimmer Beryl	2010	Status Communication in Agile Software Teams: A Case Study	Proceedings of the 2010 Fifth International Conference on Software Engineering Advances
P8	*C6*	Downs John, Plimmer Beryl, Hosking John G.	2012	Ambient awareness of build status in collocated software teams	Software Engineering (ICSE), 2012 34th International Conference on
P9	C7	Feitelson Dror, Frachtenberg Eitan, Beck Kent	2013	Development and Deployment at Facebook	IEEE Internet Computing
P10	C8	Gruver Gary, Young Mike, Fulghum Pat	2012	A Practical Approach to Large-Scale Agile Development: How HP Transformed LaserJet FutureSmart Firmware	ISBN: 9780321821720
P11	C9(a,b)	Holck Jesper, Jørgensen Niels	2007	Continuous integration and quality assurance: A case study of two open source projects	Australasian Journal of Information Systems
P12	C10	Kim Seojin, Park Sungjin, Yun Jeonghyun, Lee Younghoo	2008	Automated Continuous Integration of Component-Based Software: An Industrial Experience	Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering
P13	C11	Lacoste Francis J.	2009	Killing the Gatekeeper: Introducing a Continuous Integration System	Agile Conference
P14	C12	Merson Paulo	2013	Ultimate Architecture Enforcement: Custom Checks Enforced at Code-commit Time	Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity
P15	C13	Miller Ade	2008	A Hundred Days of Continuous Integration	Agile Conference
P16	C14	Neely Steve, Stolt Steve	2013	Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy)	Agile Conference
P17	C15	Shen Tzu-Chiang, Soto Ruben, Mora Matias, Reveco Johny, Ibsen Jorge	2012	ALMA operation support software and infrastructure	Proceedings of SPIE - The International Society for Optical Engineering
P18	*C15*	Soto Ruben, González Víctor, Ibsen Jorge, Mora Matias, Sáez Norman, Shen Tzu-Chiang	2012	ALMA software regression tests: The evolution under an operational environment	Proceedings of SPIE - The International Society for Optical Engineering
P19	C16	Ståhl Daniel, Bosch Jan	2013	Experienced benefits of continuous integration in industry software product development: A case study	IASTED Multiconferences - Proceedings of the IASTED International Conference on Software Engineering, SE 2013
P20	C17(a–e)	Ståhl Daniel, Bosch Jan	2014	Automated Software Integration Flows in Industry: A Multiple-case Study	Companion Proceedings of the 36th International Conference on Software Engineering

(*continued on next page*)

(*continued*)

Paper	Case	Authors	Year	Title	Source
P21	C18	Ståhl Daniel, Bosch Jan	2014	Modeling Continuous Integration Practice Differences in Industry Software Development	Journal of Systems and Software
P22	C19	Stolberg Sean	2009	Enabling Agile Testing Through Continuous Integration	Agile Conference
P23	C20	Sturdevant Kathryn F.	2007	Cruisin' and Chillin': Testing the Java-Based Distributed Ground Data System "Chill" with CruiseControl System "Chill" with CruiseControl	Aerospace Conference, 2007 IEEE
P24	C21	Su Tao, Lyle John, Atzeni,rea, Faily Shamal, Virji Habib, Ntanos Christos, Botsikas Christos	2013	Continuous integration for web-based software infrastructures: Lessons learned on the webinos project	Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)
P25	C22	Süß Jörn Guy, Billingsley William	2012	Using Continuous Integration of Code and Content to Teach Software Engineering with Limited Resources	Proceedings of the 34th International Conference on Software Engineering
P26	C23	Yuksel H. Mehmet, Tuzun Eray, Gelirli Erdoğan, Biyikli Emrah, Baykal Buyurman	2009	Using continuous integration and automated test techniques for a robust C4ISR system	Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on
P27	C24	Zaytsev Yury V., Morrison Abigail	2012	Increasing quality and managing complexity in neuroinformatics software development with continuous integration	Frontiers in neuroinformatics
P28	C25(a–c)	Bellomo, S., Ernst, N., Nord, R., Kazman, R.	2014	Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail	Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on
P29	C26	Chen, L.	2015	Continuous Delivery: Huge Benefits, But Challenges Too	IEEE Software
P30	C27	Debbiche, A., Dienér, M., Berntsson Svensson, R.	2014	Challenges When Adopting Continuous Integration: A Case Study	The 15th International Conference of Product Focused Software Development and Process Improvement (Profes)

Appendix B. Cases

Table B.1
Cases, categories and themes of reported problems. **B** = Build Design, **S** = System Design, **I** = Integration, **T** = Testing, **Rel** = Release, **H** = Human and Organizational, **Res** = Resource Problems.

Case	Description	Time	# of Devs	Maturity	Context	B	S	I	T	Rel	H	Res
C1	DARPA Urban Challenge, self-driving car	2007	Medium	CD	Non-commercial	–	–	–	✓	–	✓	–
C2	Amber, chemistry simulation toolkit	2014	Medium	CI	Non-commercial	✓	✓	–	✓	–	–	✓
C3a	Java EE service	2007	Small	CI	Commercial	✓	✓	✓	✓	–	–	✓
C3b	Web application	2007	Small	CI	Commercial	–	–	–	–	–	–	–
C4	BT, telecommunications service	2007	Small	CD	Commercial	–	–	–	✓	–	–	✓
C5	Atlassian, web applications	2012	Medium	CD	Commercial	–	✓	✓	–	✓	✓	✓
C6	N/A	2012	Small	CI	Commercial	–	–	✓	✓	–	✓	–
C7	Facebook, web application	2012	Large	CD	Commercial	–	–	–	–	–	–	–
C8	HP, Futuresmart firmware	2012	Large	CD	Commercial	–	✓	✓	✓	–	–	✓
C9a	FreeBSD, operating system	2002	Medium	CI	Non-commercial	–	–	✓	✓	–	–	–
C9b	Firefox, web browser	2002	Medium	CI	Non-commercial	–	–	–	✓	–	–	–
C10	Samsung, Linux distribution for mobile devices	2008	Medium	CI	Commercial	–	–	–	–	–	✓	–
C11	Launchpad, web application	2009	Medium	CI	Non-commercial	–	–	✓	✓	–	✓	–
C12	TCU Brazil, Java applications	2013	Medium	CI	Commercial	–	–	–	–	–	✓	–
C13	Microsoft, Web Service Software Factory SDK	2007	Small	CI	Commercial	–	–	✓	✓	–	–	✓
C14	Rally Software, web application	2012	Medium	CD	Commercial	–	–	✓	✓	–	✓	–
C15	ALMA, scientific high-precision antenna array	2012	Medium	CI	Non-commercial	–	–	–	✓	–	–	–
C16	Ericsson, multiple products	2013	Medium	CI	Commercial	–	–	–	–	–	–	–
C17a	Ericsson product	2014	Large	CI	Commercial	–	–	✓	✓	–	–	–
C17b	Saab AB, military aircraft support system	2014	Small	CI	Commercial	–	–	–	–	–	–	–
C17c	Saab AB, military aircraft visualization system	2014	Small	CI	Commercial	–	–	–	–	–	–	–
C17d	Volvo Cars, electric vehicle on-board software	2014	Medium	CI	Commercial	–	–	–	–	–	–	–
C17e	Jeppesen, airline fleet and crew management	2014	Medium	CI	Commercial	–	✓	–	–	–	–	✓
C18	Ericsson, component of a network node	2014	Medium	CI	Commercial	–	–	–	–	–	–	–
C19	C# application	2008	Small	CI	Commercial	–	–	–	–	–	✓	✓
C20	NASA, MPCS Chill, ground data system	2006	Small	CI	Non-commercial	–	–	–	–	–	–	–
C21	Webinos, web-based software infrastructure	2013	Medium	CI	Non-commercial	–	✓	✓	✓	–	–	–
C22	Engineering course, Robocode	2011	Medium	CI	Non-commercial	–	✓	✓	✓	–	–	–
C23	Command and control system	2009	Medium	CI	Non-commercial	–	–	–	–	–	–	–
C24	NEST, neuronal network simulator	2012	Medium	CI	Non-commercial	–	–	✓	–	–	–	–
C25a	Federal business systems	2014	Small	CD	Commercial	–	✓	–	✓	–	–	–
C25b	Virtual learning environment	2014	Small	CD	Commercial	–	–	–	–	–	–	–
C25c	Sales portal	2014	Medium	CD	Commercial	–	✓	–	✓	–	–	–
C26	Paddy Power, multiple systems	2014	Small	CD	Commercial	–	✓	–	–	–	✓	✓
C27	Swedish telecommunications company	2014	Large	CI	Commercial	–	–	✓	✓	–	✓	–

References

[1] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st, Addison-Wesley Professional, 2010.

[2] M. Fowler, Continuous Delivery, 2013,

[3] D. Ståhl, J. Bosch, Automated software integration flows in industry: a multiple-case study, in: Companion Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 54–63. New York, NY, USA.

[4] A. Debbiche, M. Dienér, R. Berntsson Svensson, Challenges when adopting continuous integration: a case study, in: Product-Focused Software Process Improvement, in: Lecture Notes in Computer Science, 8892, Springer International Publishing, 2014, pp. 17–32.

[5] G.G. Claps, R.B. Svensson, A. Aurum, On the journey to continuous deployment: technical and social challenges along the way, Inf. Softw. Technol. 57 (0) (2015) 21–31.

[6] M.V. Mäntylä, B. Adams, F. Khomh, E. Engström, K. Petersen, On rapid releases and software testing: a case study and a semi-systematic literature review, Empirical Softw. Eng. 20 (5) (2015) 1384–1425, doi:10.1007/s10664-014-9338-4.

[7] P. Rodríguez, A. Haghighatkhah, L.E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J.M. Verner, M. Oivo, Continuous deployment of software intensive products and services: a systematic mapping study, J. Syst. Softw. (2016), doi:10.1016/j.jss.2015.12.015.

[8] D. Ståhl, J. Bosch, Experienced benefits of continuous integration in industry software product development: a case study, in: IASTED Multiconferences - Proceedings of the IASTED International Conference on Software Engineering, SE 2013, 2013, pp. 736–743.

[9] D. Ståhl, J. Bosch, Modeling continuous integration practice differences in industry software development, J. Syst. Softw. 87 (2014) 48–59.

[10] A. Eck, F. Uebernickel, W. Brenner, Fit for continuous integration: how organizations assimilate an agile practice, in: Twentieth Americas Conference on Information Systems, 2014. Savannah, Georgia, USA.

[11] M. Fowler, Continuous Integration, 2006.

[12] M. Meyer, Continuous integration and its tools, IEEE Softw. 31 (3) (2014) 14–16, doi:10.1109/MS.2014.58.

[13] T. Fitz, Continuous Deployment, 2009.

[14] H. Holmström Olsson, H. Alahyari, J. Bosch, Climbing the “Stairway to Heaven” - a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software, in: Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, 2012, pp. 392–399, doi:10.1109/SEAA.2012.54. Washington, DC, USA.

[15] B. Adams, S. McIntosh, Modern release engineering in a nutshell: why researchers should care, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 5, 2016, pp. 78–90.

[16] T.O. Lehtinen, M.V. Mäntylä, J. Vanhanen, J. Itkonen, C. Lassenius, Perceived causes of software project failures–an analysis of their relationships, Inf. Softw. Technol. 56 (6) (2014) 623–643.

[17] V. Garousi, M. Felderer, M.V. Mäntylä, The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM Press, 2016, pp. 1–6, doi:10.1145/2915970.2916008.

[18] B. Kitchenham, Guidelines for performing systematic literature reviews in software engineering, Technical Report, Keele University Technical Report, 2007.

[19] S. Jalali, C. Wohlin, Systematic literature studies: database searches vs. backward snowballing, in: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, ACM, 2012, pp. 29–38.

[20] V. García-Díaz, B. G-Bustelo, O. Sanjuán-Martínez, J. Lovelle, Towards an adaptive integration trigger, Adv. Intell. Soft Comput. 79 (2010) 459–462.

[21] A. Strauss, J. Corbin, Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory, SAGE Publications, 1998.

[22] ATLAS.ti, 2014.

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

[24] J. Cohen, A coefficient of agreement for nominal scales, Educ. Psychol. Meas. 20 (1) (1960) 37–46, doi:10.1177/001316446002000104.

[25] J.R. Landis, G.G. Koch, The measurement of observer agreement for categorical data, Biometrics 33 (1) (1977) 159–174, doi:10.2307/2529310.

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

[27] M.Q. Patton, Qualitative Research & Evaluation Methods, 3rd, SAGE Publications, 2002. Published: Hardcover.

[28] M. Csikszentmihalyi, Flow: the Psychology of Optimal Experience, 41, HarperPerennial New York, 1991.