Designing Software Architecture to Support Continuous Delivery and DevOps: A Systematic Literature Review

Robin Bolscher and Maya Daneva
University of Twente, Drienerlolaan 5, Enschede, The Netherlands

Keywords: Software Architecture, Continuous Delivery, Continuous Integration, DevOps, Deployability, Systematic Literature Review, Micro-services.

Abstract: This paper presents a systematic literature review of software architecture approaches that support the implementation of Continuous Delivery (CD) and DevOps. Its goal is to provide an understanding of the state-of-the-art on the topic, which is informative for both researchers and practitioners. We found 17 characteristics of a software architecture that are beneficial for CD and DevOps adoption and identified ten potential software architecture obstacles in adopting CD and DevOps in the case of an existing software system. Moreover, our review indicated that micro-services are a dominant architectural style in this context. Our literature review has some implications: for researchers, it provides a map of the recent research efforts on software architecture in the CD and DevOps domain. For practitioners, it describes a set of software architecture principles that possibly can guide the process of creating or adapting software systems to fit in the CD and DevOps context.

1 INTRODUCTION

The practice of releasing software early and often has been increasingly more adopted by software organizations (Fox et al., 2014) in order to stay competitive in the software market. Its popularity fueled the development of practices collectively labeled as Continuous Delivery (CD) Chen, 2015a). Over the past few years, the notion of CD organically evolved and was further built upon to create what is now known as DevOps (Bass et al., 2015). While embracing CD and DevOps in their organizations, many software development practitioners experienced that not all styles of software architectures are suitable for applying the CD principles and practices, particularly in case of large monolithic (Garousi et al., 2019) and highly coupled architectures (Sturtevant, 2017; Bucchiarone, 2018; Knoche and Hasselbring, 2018). Since both the CD and DevOps movements are growing more mature each year, it is important to have a clear understanding on the various approaches of software architecture that may or may not be suitable in this context. This need motivated us to do a systematic literature review (SLR) in order to create a state-of-the-art understanding on adapting existing and designing new software architectures tailored for CD and DevOps practices.

For clarity, before elaborating on the subject of this SLR, we present the definitions of the concepts that we will address: *Software architecture* of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both (Humble and Farley, 2010). *Continuous Delivery (CD)* is a software engineering discipline in which teams keep producing valuable software incrementally in short cycles and ensure that the software can be reliably released at any time (Chen, 2015; Humble and Farley, 2010). *DevOps* is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality (Bass et al., 2015). As one could see from the definitions, CD and DevOps are quite overlapping in their goals. Later in this paper, we will see that there is indeed a strong relation between the two and that DevOps practices rely heavily on the CD principles. Since the two concepts are so similar, the effect they have on software architecture is expected to be very similar as well. This is why these two concepts are both included in our SLR.

This paper is structured as follows. Section 2 describes the industrial relevance of the subject and

Bolscher, R. and Daneva, M.
Designing Software Architecture to Support Continuous Delivery and DevOps: A Systematic Literature Review.
DOI: 10.5220/0007837000270039
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 27-39
ISBN: 978-989-758-379-7

our motivation for this SLR. Section 3 discusses related work with particular focus on the potential benefits of DevOps practices. Section 4 presents the goal of this research effort. Section 5 describes the method (Kitchenham, 2007) used for our SLR. Section 6 presents the results to the research questions of our SLR and describes other relevant topics found in the literature. Section 7 is on the limitations to this research. Section 8 presents the discussion of the results. Section 9 concludes with summarized answers to the research questions. Section 10 is about future work.

2 MOTIVATION

DevOps is currently progressing from the "peak" of the Gartner Hype Cycle for Application Services towards the more valuable "plateau of productivity", according to a 2015 market research report by Cap Gemini on DevOps (Menzel, 2015). This source also states DevOps is evolving from a niche to a mainstream strategy employed by 25% of Global 2000 organizations. Furthermore, according to the 2017 State of DevOps Report by Puppet and the DevOps Research and Assessment Group (Kersten, 2017), the percentage of respondents claiming to work on DevOps grew from 16% in 2014 to 27% in 2017. This report (Kersten, 2017) also found that high performing DevOps IT organizations deploy software 46 times more frequently, have a 440 times shorter lead time for implementing changes, and 96 times faster mean time to recover (from downtime) and a 5 times lower change failure rate (in comparison to the low performing DevOps IT organizations). In general, one of the key conclusions of the report (Kersten, 2017) is that lean product management (including concepts such as CD and DevOps) drives higher organizational performance. This means that for IT organizations there is a lot to be gained through the adoption of CD and DevOps practices. Although a lot of research has been performed and practical experience gained over the last 5 years (Ghantous and Gill, 2017), to the best of our knowledge, little has been done so far to consolidate the output of this research in a systematic way and help develop a deep understanding of the CD and DevOps phenomena. In particular, we could find no literature source that consolidates the published knowledge regarding the relationship between the concept of software architecture and CD and DevOps. Understanding this relationship is important for shaping our understanding of the architecture principles and approaches that work in the context of CD and DevOps and those that do not. We felt motivated to respond to this need by carrying out a SLR aiming to provide an overview of the latest developments of software architecture approaches in the context of CD and DevOps.

3 RELATED WORK

DevOps is not a strictly defined method; its implementation varies a lot as shown in a recent qualitative study on DevOps in practice (Erich et al., 2017). Therefore, the desired and expected benefits of implementing DevOps vary just as much. Erich et al. (2017) indicated the varying benefits organizations set out to achieve by initiating DevOps: reduced lead and release time, improved problem solving, feedback gathering and overall product quality, increased velocity, and increased focus on new features. These authors reported however that not all these benefits were actually achieved by the organizations that implemented DevOps: in fact, the main benefits achieved in these authors' case study organizations were: higher deployment frequency, shorter lead time, improved automated testing, feedback gathering and problem solving, fewer escalations (caused by friction between development and operations departments), more public facing services and an increased velocity.

Furthermore, a 2017 SLR on DevOps (Ghantous and Gill, 2017) presents a set of 17 benefits that can potentially be achieved by implementing DevOps. These are all benefits found in literature up to 2017, however it does not necessarily mean that all these benefits are always achievable in practice.

Next, a case study on DevOps implementation in an IT company in New-Zealand (Senapathi et al., 2018) lists the realized benefits and their relations. It reports two main categories of benefits, namely, increased development team engagement and improved customer experience. Furthermore, Chen (2015b) presents from a practitioner's perspective the potential benefits of architecting for CD. The author describes five categories of observed benefits after moving 22 software applications to CD: accelerated time to market, improved ability to consistently build the right product, improved productivity and efficiency, improved product quality and improved customer satisfaction.

As these literature sources (Erich et al., 2017; Ghantous and Gill, 2017; Senapathi et al., 2018; Chen, 2015) suggest, there are many potential benefits reported so far, and many of those are in fact indicated by multiple authors. In what follows, we

aggregated and categorized the benefits reported by these sources in three categories: (1) benefits pertaining to culture, (2) to product quality, and (3) to development and operations processes. Some benefits might be considered in more than one category (albeit rephrased), however for the sake of simplicity this is kept to a minimum.

Culture: The main cultural benefit to be achieved by implementing DevOps is that teams are happier and more engaged. The benefits in this category include:

B1. Higher level of autonomy;
B2. Learning new technologies;
B3. Feeling valued;
B4. Improved collaboration;
B5. Knowledge sharing;
B6. Natural communication;
B7. Less of a blaming culture;
B8. Fewer escalations (caused by Dev vs. Ops friction).

Product Quality: Organizations implementing DevOps report to be able to create higher quality products faster and therefore create a better customer experience. The benefits in this category include:

B9. Shorter lead time;
B10. Improved code quality;
B11. Automated testing (quality assurance);
B12. Real time, automated monitoring;
B13. Continuous innovation;
B14. Frequent deployment;
B15. Better scalability;
B16. Less down time.

Development and Operations Processes: DevOps teams remove the friction between the previously separated development and operations departments and create understanding for each other's problems. Furthermore, processes are highly automated and traceable which improves the overall velocity of the software development and lowers the chance of bugs ending up in released software. The benefits in this category include:

B17. Continuous planning;
B18. Parallel deployment;
B19. Continuous integration;
B20. Improved cloud and database management (infrastructure-as-code);
B21. Easy code rollback;
B22. Improved feedback gathering;
B23. Secure pipeline;
B24. Automated deployments;
B25. Scalable, repeatable, traceable and automated processes.

4 RESEARCH GOAL

The goal of this SLR is to analyse the published scientific output on the topics at hand in order to create an overview of the issues and requirements for software architecture design in the context of CD and DevOps. These requirements could possibly be used to create a set of software architecture principles that can guide software developers and architects in the process of creating or adapting a software system to be used in a CD and DevOps context.

To this end, the central research question of this SLR is the following: ***What software architecture approaches support the implementation of Continuous Delivery and DevOps, according to published literature?***

This question has been decomposed in three sub-questions:

RQ1. What are the potential software architectural problems when adopting CD and DevOps practices on an existing software system?
RQ2. What characteristics of a software architecture are important for enabling CD and DevOps?
RQ3. What software architecture styles are suitable for being used in a CD and DevOps context?

5 RESEARCH PROCESS

Our SLR is conducted following the guidelines of Kitchenham (2007). These were complemented with the guideline of Kuhrmann et al. (2017). These guidelines define a process including three phases: (1) planning, (2) conducting, and (3) reporting. The first two phases have some (sub-) stages associated with it; planning consists out of the identification of the need for a review (see Section 2) and the development of the review protocol (see Section 5.1). The third phase, reporting, consists out of identification of research, selection of primary studies, study quality assessment, data extraction and monitoring, and data synthesis (see Section 6).

5.1 Review Protocol

In order to find relevant articles, two digital repositories of scientific publications were queried: Scopus and Web of Science. We chose them, because they are comprehensive and also because our university had subscriptions to both. The search string

("software architecture" AND ("continuous delivery" OR "continuous deployment" OR "devops" OR "dev-ops" OR "dev ops")) was used. The search was carried out on May 14, 2019. It was applied to the Title, Abstract and Keyword sections of the Scopus database, and to the Title, Abstract and Topic sections of Web of Science. The queries formatted and refined for each repository can be found in the Appendix items "Scopus search query" and "Web of Science Search Query".

This search string is composed as presented above since this research is focused on the intersection of software architecture and CD/DevOps, hence the query combines these concepts. Performing the search resulted in 39 papers from Scopus over a time span of 9 years (2010-2019). Executing the same query on Web of Science resulted in 15 papers over a time span of 5 years (2014-2018). Looking at the number of papers published per year it appears that 2015 through 2017 were the most active years. Therefore, we chose to only include papers published after Jan 1, 2015. This ensured that our SLR contains the most relevant and up-to-date literature.

We used the following inclusion and exclusion criteria for selecting papers out of the search results obtained in the previous step.

Inclusion Criteria:

I1. The paper discusses as its core topic either DevOps or CD in a software architecture domain;

I2. The paper takes a practical point of view on the problem domain (e.g. a case study or expert/practitioner experiences and opinions).

Exclusion Criteria:

E1. The paper is published before Jan 1, 2015;

E2. The paper presents no link to DevOps, CD or similar practices;

E3. The paper is purely theoretical;

E4. The paper is a duplicate of a paper that was already found either in Scopus or in Web of Science and has already been included;

E5. The paper is not written in English.

After applying exclusion criterion E1, Scopus provided 36 papers, and Web of Science −14. Since there are many publications on both software architecture and CD/DevOps, inclusion criterion I1 and exclusion criterion E2 were added to filter the materials that are not in any way applicable or related to both of these topics. It is the intersection of these subjects that we are interested in. Inclusion criterion I2 and exclusion criterion E3 were added to find published work presenting real-world experiences. Since this research is mainly conducted to help practitioners with the potential problems arising from applying CD/DevOps on their software architectures, it is important to explore the real-world practice by using case study-based research methods, expert interviews or practitioners' perceptions and personal experiences. Exclusion criterion E5 was added for obvious reasons. The initial selection of all 49 papers was done by applying the inclusion and exclusion criteria by reading the papers' title, abstract and metadata.

After the initial filtering, only 23 papers were still in scope of this SLR. There were many duplicates found between the results of Scopus and Web of Science. The papers that passed the initial filtering were read in detail and filtered by re-applying the inclusion/exclusion criteria. After the second selection round, there were 13 papers left, these finally represent the body of literature to be used in the final stage of Kitchenham's SLR method (2007): data extraction and monitoring, and data synthesis. The papers are: Stutevant, 2017; Chen, 2015b; Erder and Pureur, 2015; Woods, 2016; Shahin et al., 2016; Elberzhager et al., 2017; Villamizar et al., 2015; Schermann et al., 2018; Stahl and Bosch, 2018; Pahl et al., 2018; Berger et al., 2017; Chen et al., 2015; Bass, 2017.

6 RESULTS

Our selected papers and their mappings to our RQs are presented in Table 1. Note that two of the papers (Erder and Pureur, 2015; Woods, 2016) are not included in this table because they could not be properly mapped to any research question (i.e. they talk about architecture and CD/DevOps, but do not shed light on our RQs).

Table 1: Literature results mapped onto RQs.

Research question	Reference
RQ1. Issues	Shahin et al., 2016; Elberzhager et al., 2017;Villamizar et al., 2015; Schermann et al., 2018; Stahl and Bosch, 2018
RQ2. Characteristics	Stutevant, 2017; Chen, 2015b; Shahin et al., 2016; Pahl et al., 2018; Berger et al., 2017; Chen et al., 2015
RQ3. Styles	Shahin et al., 2016; Elberzhager et al., 2017; Villamizar et al., 2015; Schermann et al., 2018; Pahl et al., 2018; Berger et al., 2017; Chen et al.2015; Bass, 2017

As indicated in Table 1, there are papers that discuss potential software architectural issues of implementing CD/DevOps on existing systems in a meaningful way. Furthermore, 6 papers mention important software architectural characteristics, and 8 papers discuss some form of software architecture style, all in the context of CD/DevOps.

Before presenting the answers to our RQs, it is worthwhile mentioning that a number of authors of our set of 14 papers, emphasize the importance of the intersection of DevOps and software architecture. We observe in five out of the 14 papers make explicit statements on this. Chen et al. (2015) describe the importance of software architecture with regard to DevOps as follows: *"one cannot realize DevOps in a scalable way without building this into the architecture"*. These authors also state that architectural tactics must be implemented system-wide to support DevOps objectives (Chen et al., 2015). Furthermore, Pahl et al. argue that in order to address continuous service systems development and operation a particular software architectural style is needed (Pahl et al., 2018). Len Bass, author of the seminal book "DevOps: A Software Architect's Perspective" (Bass et al., 2015), states in his paper (Bass, 2017) that *"the architect is critical for success in adopting DevOps practices"*. Shahin et al. (2016) report on the growing realization that implementing CD/DevOps may necessitate software architectural modifications, and even go as far as saying that software architecture should take the lead when implementing CD. These statements further reinforced our believe that if we make a contribution to the body of knowledge on the topic of software architecture and CD/DevOps by means of this SLR, our work would possibly be of service to a wider audience than originally thought (at the moment when we initiated this work).

In what follows, we summarize the literature as it relates to our RQs.

6.1 Software Architecture Issues

This section pertains to RQ1. We observe that all five papers (Shahin et al., 2016; Elberzhager et al., 2017; Villamizar et al., 2015; Schermann et al., 2018; Stahl and Bosch, 2018) discussing architecture issues agree that it is not trivial to adopt CD/DevOps practices on an existing system.

Our examination of these papers yielded 10 software architecture issues which we present in Table 2. In fact, these are architectural obstacles in the adoption of CD/DevOps. Elberzhager et al. (2017) identify four key issues that are important to consider before moving towards implementation of DevOps, one of them is about the possible impact on the current software architecture (Elberzhager et al., 2017). What is more, Elberzhager et al. argue that *"in the case of an existing software product, a detailed analysis is needed as to the degree to which it can support the goals to be achieved with the DevOps approach"*.

Table 2: Identified issues.

ID	Issue	Reference
IS1	Scaling	Villamizar et al., 2015
IS2	Decomposition	Scherman et al., 2018
IS2	Methods and tools	Elberzhager et al., 2017; Stahl and Bosch, 2018
IS4	Highly coupled (monolithic) systems	Shahin et al., 2016
IS5	Ever-changing operational environments and tools	Shahin et al., 2016
IS6	Monolithic database	Shahin et al., 2016
IS7	Application level dependencies	Shahin et al., 2016
IS8	Testing	Shahin et al., 2016
IS9	Logging	Shahin et al., 2016
IS10	Monitoring	Shahin et al., 2016

When an application is not designed to scale (IS1), it is very difficult to apply DevOps practices, an important aspect of which is the presence of a highly automated infrastructure. This issue has a direct connection with highly coupled (monolithic) systems (IS4), since these systems are often not scalable by design. Scaling monolithic applications is a problem because they are composed of many internal services and if a service needs to be scaled due to increasing load, then the whole monolith has to be scaled up (Villamizar et al., 2015). Another problematic aspect of the monolithic codebase is its monolithic database (IS6). Many systems, which are not designed for decomposability, rely on a central database. This becomes a problem when the software architecture changes into a more loosely coupled, individually deployable, set of components. The database can form a choke-point for operations as it remains an undeployable (and unscalable) unit.

A first step to make an existing monolithic system more scalable is increasing its decomposition, which ultimately enables DevOps practitioners to scale individual components of the system instead of the whole. However, many systems are not decomposable as is (IS2). Even if a system is decomposed into multiple (separately deployable)

components, new issues arise such as tracing errors and finding root causes of production issues traveling through multiple system components (Schermann et al., 2018). Moreover, dependencies on application level (IS7) inhibit decomposition and decrease the deployability of a system (Shahin et al., 2016).

One of the cornerstones of CD/DevOps is tool support. However, finding and incorporating proper tool support is a challenge on its own (IS3). For example, continuous integration (CI) systems can become rather complex, such that they need their own development and operations team taking care of evolving and maintaining the CI workflow of the core products. Ståhl and Bosch (2017) even developed an architecture framework specifically for CI/CD systems, which indicates the potential complexity of the tools needed for CD/DevOps adoption. They also report that *"as a rule, continuous integration and delivery systems are highly customized and purpose-built software products".*

Shahin et al. (2016) point out that *"another challenge at architectural level was the influence of ever-changing environments and tools on architecture design to enable CD practice"*. These authors reports that practitioners may have had issues transferring and deploying systems in various heterogeneous operations environments (IS5).

Next, testing is another prominent obstacle that needs to be overcome when moving towards CD/DevOps (IS8). This issue consists of three sub-issues: (i) improving test quality, (ii) making code more testable, and (iii) test automation (Shahin et al., 2016). For example, automated testing is a core pillar of DevOps. Without it DevOps engineers cannot quickly determine the quality of the software that is to be deployed. This is because failure to do it consistently, would result in either (much) slower deployments, or decreased software quality, both which are undesired.

Finally, logging and monitoring are increasingly important when the CD/DevOps adoption becomes more mature. Highly automated pipelines, as a result of CD/DevOps, enables practitioners to deploy software to production faster than ever before, also including bugs. Some will slip through the (automated) testing process, therefore the authors of (Schermann, 2018) state that monitoring is a prerequisite for keeping practitioners aware of events in the production environments. "*CD increases the complexity of deployment process, which necessitates designing and implementing sophisticated logging and monitoring mechanisms"* (Shahin et al., 2016).

6.2 Beneficial Software Architecture Characteristics

This section reports the results related to RQ2. Table 3 summarizes our findings. We found 17 software architecture characteristics that our selected literature sources deemed beneficial. These are listed in the second column of Table 3. In the third column, we present the papers addressing each characteristic. The number of references in the rightmost column clearly indicates those software architecture characteristics have been treated most frequently in relation to CD/DevOps in scientific literature; these are: deployability (CH2), testability (CH11), automation (CH3), loosely coupled (CH6), modifiability (CH1).

It is not surprising that deployability (CH2) is the most prevalent software architecture characteristic that according to our selected literature would enable CD/DevOps adoption. Chen et al. (2015) argue that CD *"requires architectural support for deploying without requiring explicit coordination among teams"*. Chen (2015b) describes a list of Architecturally Significant Requirements (ASRs) which the author defines as *"requirements that have a measurable impact on a software system's architecture",* one of these ASRs is deployability. Adding to that, Chen states that one of the aspects of a deployable architecture is being able to deploy software without downtime and moving the software quickly between different environments (e.g. testing, production). Furthermore, Shahin et al. (2016) have identified five main architectural principles. The first one is concerned with small and independent deployment units such as services, components but also the database. Furthermore, Bass (2017) discusses that the ability to continuously deploy depends on the system architecture and the possible team dependencies arising from this architecture.

Modularity (CH15) is a way to make a software system more deployable (Shahin et al., 2016; Pahl et al., 2018), by reducing dependencies and isolating changes in the software. Another related characteristic is the use of stateless components (CH5). Berger et al. (2017) discovered that in order to improve the deployability and elastic scaling process it helps to make services/components stateless so that they can be stopped and restarted without causing issues.

Sturtevant (2015) states: *"Architecture will become the biggest bottleneck to your DevOps transformation. You need a balanced focus on agile process and agile architecture"* (CH1). Chen et al. (2015) describe a DevOps tactics tree that consists of tactics (a checklist of architectural concerns) that

would help enable the achievement of DevOps goals. One of the top level tactics of this tree is architecture modifiability (CH1). Chen (2015b) also discusses modifiability as one of the ASRs.

Table 3: Software architecture characteristics supporting CD/DevOps.

ID	Software architecture characteristics	Reference
CH1	Agility/Modifiability	Stutevant, 2017; Chen, 2015b; Chen et al.2015;
CH2	Deployability	Chen, 2015b; Chen et al.2015; Shahin et al., 2016; Bass, 2017
CH3	Automation	Berger et al., 2017; Chen et al.2015; Bass, 2017
CH4	Traceability	Berger et al., 2017; Bass, 2017
CH5	Stateless components	Berger et al., 2017
CH6	Loosely coupled	Stutevant, 2017; Pahl et al., 2018; Berger et al., 2017
CH7	Production versioning	Chen et al.2015;
CH8	Rollback	Chen et al.2015;
CH9	Availability	Chen et al.2015;
CH10	Performance	Chen et al.2015;
CH11	Testability	Chen, 2015b; Shahin et al., 2016
CH12	Security	Chen, 2015b
CH13	Loggability	Chen, 2015b; Shahin et al., 2016
CH14	Monitorability	Chen, 2015b
CH15	Modularity	Shahin et al., 2016; Pahl et al., 2018;
CH16	Virtualization	Pahl et al., 2018;
CH17	Less reusability	Shahin et al., 2016

Another important software architecture characteristic on Chen's ASR list (2015b) is testability (CH11). As the software moves through the CD pipeline it is subjected to tests at different stages to ensure the software is of known quality and ready for release. The author states that good testability needs to be implemented on the architectural level to make certain that developing tests is feasible and cost effective. Testability is also one of the five main architectural principles presented by Shahin et al. (2016). The authors distinguish between improving test quality, making code more testable and test automation, and in general argue that testability should be approached from an architectural point of view first. Finally, the DevOps tactics tree considers testability as one of the top level tactics (Chen et al., 2015).

Sturtevant et al. state that loosely coupled (CH6) architectures increases the performance in a DevOps environment by decreasing application and inter-team dependencies generally found in highly coupled architectures (Sturtevant, 2017). In the same vein, Pahl et al. (2018) state that one of the cloud architecture principles is loose coupling, this is especially important in the context of cloud resource virtualization and elastic scaling. Berger et al. (2017) note that loose coupling (in the form of a publish-subscribe pattern) is important to the software architecture in the context of CD/DevOps.

Automation (CH2) is one of the key principles of CD/DevOps and is paramount to a successful implementation. Chen et al. state that "everything must be automated" and they stress that the architecture should support this (Chen et al., 2015). Additionally Bass points out that DevOps practices rely heavily on tool support and automation, and that it is the task of the architect to guide and support the developers in setting up the automation processes with the appropriate tools (Bass, 2017). Finally, Berger et al. state that the deployment procedure should be automated to avoid manual interaction and increase overall deployment speed 9berger et al., 2017).

Once a system is decomposed into many services it becomes more difficult to see where in the system an error originated and how it travelled through several services. Traceability (CH4) of errors is therefore an important characteristic to consider in the architecture (Bass, 2017). There is another side to traceability, namely, tracing software sources throughout the integration and deployment process. Bergeret al., (2017) states that due to the many releases it is important to be able to trace a piece of software running in production back to a commit. This, in turn, creates certainty regarding those pieces of code that are actually running in production and helps for potential debugging or auditing purposes.

Chen et al. (2015) report on several other characteristics: production versioning (CH7), rollback (CH8), availability (CH9) and performance (CH10). Production versioning refers to the ability to have multiple versions of the same service in production simultaneously, which improves the deployability of the system by allowing for deployment strategies such as canary releases. The rollback characteristic is a useful safe guard because deploying software systems many times a day assumes that something would go wrong and then a roll back can restrict the impact. Next, performance

refers in fact to ‘scaling performance’, i.e. being able to provision and deploy new instances of services, which prevents the system from slowing down once the load gets higher. Chen et al. (2015) also mention availability (CH9), however from the text it does not become clear why this would be more applicable to CD/DevOps practices, it seems that this characteristic is more specific to the system being discussed in the paper.

A less apparent characteristic is security (CH12). Chen (2015b) stresses that during start-up an application might be more vulnerable compared to when it is fully started, since CD/DevOps increases the deployment frequency applications will start and stop more often. The author also describes other characteristics, loggability (CH13), and monitorability (CH14). These two characteristics both have the same goal: to give DevOps engineers more control and information over the complex production environment. When having a lot of services in production (which are replaced often by deploying) it is important to have proper (centralized) log aggregation in place (Shahin et al., 2016). This ensures that information, requests and errors can be followed when they travel through multiple services.

Pahl et al. (2018) add an important characteristic, namely virtualization (CH16), which refers specifically to infrastructure virtualization known as is one of the cornerstones of DevOps (the so-called infrastructure-as-code). In their research, virtualization (of infrastructure) enables elastic scaling and quicker (repeatable), more fault tolerant deployments, it also enables developers (with less operations experience) to work with the infrastructure.

The final characteristic, less reusability (CH17), goes against what developers have been doing for many years. With the popularity of DRY (meaning ‘don’t repeat yourself’) − i.e. a principle to reduce repetition in software − developers were told to reuse as much code as possible (e.g. by creating complex abstractions). However, Shahin et al. (2016) report that “focusing too much on reusability can be a huge bottleneck to continuously deploying software”. The main argument against reusability in CD/DevOps context, according to the authors, is that it hinders the deployability of autonomous teams by creating more dependencies between software components. It is also argued to be a threat to testability and overall development velocity.

6.3 Software Architecture Styles

This section is concerned with discussing literature found related to RQ3. In our SLR, there are 8 papers that discuss architectural styles in the context of CD/DevOps. There is a dominant architectural style present in these papers: micro-services. All 8 papers either discuss micro-services or present them as the “go-to” architectural style for CD/DevOps practices. We observe that only 2 out of the 8 papers also mention other architectural styles or patterns.

Micro-services are a set of small services that can be developed, tested, deployed, scaled, operated and upgraded independently, allowing organizations to gain agility, reduce complexity and scale their applications in the cloud in a more efficient way. Besides that, micro-services are very popular, they are being used and promoted by industry leaders such as Amazon, Netflix and LinkedIn (Villamizar, 2015). Shahin et al. describe micro-services as the first architectural style to be preferred for CD practice, by designing fine-grained applications as a set of small services (Shahin et al., 2016).

Three papers (Stahl and Bosch, 2018; Pahl et al., 2018; Berger et al., 2017) state explicitly some specific benefits of employing the micro-services architecture concept. Micro-services are said to be helpful in increasing modularity and isolating changes and as a consequence increasing deployment frequency (Bass, 2017). An experience report by Berger et al. (2017), where the authors implemented CD practices in a team developing software for self-driving cars, reports how a loosely coupled micro-service architecture helped them move towards CD. Chen et al. argue that micro-service architectures feature many of the CD/DevOps enabling characteristics (CH2, CH7, CH8) and are (in combination with DevOps) the “key to success” of large-scale platforms (Chen et al., 2015).

Three other papers (Shahin et al., 2016; Elberzhager et al., 2017; Schermann et al., 2018) explicitly state some downsides of the micro-services architecture. E.g. tracing errors and finding root causes of production issues traveling through multiple system components (Schermann et al., 2018), resulting in increasingly complex monitoring (IS10) and logging (IS9) (Shahin et al., 2016). Plus, at the inception stage of a project a micro-services architecture might be less productive due to the required effort for creating the separate services and the necessary changes in the organizational structure, eventually as the project matures the efficiency of the micro-services architecture surpasses that of the

monolithic architecture though (Elberzhager et al., 2017).

Other authors (Villamizar et al., 2015; Schermann et al. 2018; Pahl et al. 2018) treat the suitability of the concept of micro-services in a particular context. Pahl et al. (2018) state that the idea of micro-services has been discussed as a suitable candidate for flexible service-based system composition in the cloud in the context of deployment and management automation.

Furthermore, Schermann et al. (2018) look at micro-services from a continuous experimentation perspective which is based on CD. These authors state that *"continuous experimentation is especially enabled by architectures that foster independently deployable services, such as micro-services-based architectures"*.

Micro-services emerged as a lightweight subset of the Service-Oriented Architecture (SOA), it avoids the problems of monolithic applications by taking advantage of some of the SOA benefits (Villamizar et al., 2015). Pahl et al. (2018) note that loose coupling, modularity, layering, and composability are guiding principles of service-oriented architectures.

The last architectural style is vertical layering. It is mentioned by Shahin et al. (2016) and refers to removing team dependencies by splitting software components into vertical layers (instead of horizontal layers, e.g. presentation, business and persistence). It can be argued if this is an architectural style on its own, as it is also a characteristic of micro-services and SOAs in general.

6.4 Software Architecture Methods

This section discusses four interesting concepts that we found in the selected literature and that are not directly related to our RQs, but still are relevant to the problem domain addressed in our research.

The first one is the concept of Evolutionary Architecture which is designed to support incremental change to the architecture in CD context. Shahin et al. (2016) report the increased popularity of Evolutionary Architecture amongst the participants in these authors' research on CD, opposed to big upfront architecture.

The second is the concept of Continuous Architecture (CA), which refers to the method of managing the architecture originating from the need to encompass CD in the architecture process. A very clear way to explain the concept of CA is cited from the book on the subject by Erder and Pureur (2015): *"if our objective is to build a cathedral, an Agile developer will start shoveling, but an enterprise architect will look at a 5-year plan, the goal of Continuous Architecture is to bridge this gap"*. There are six CA principles defined in the book by Erder and Pureur:

1. Architect products – not just solutions for projects.
2. Focus on quality attributes – not on functional requirements.
3. Delay design decisions until they are absolutely necessary.
4. Architect for change – leverage "the power of the small".
5. Architect for build, test and deploy.
6. Model the organization of your teams after the design of the system.

Finally, Woods (2016) puts forward two other concepts instrumental to the architecture process: (i) release models and (ii) configuration management models. Release models describe the process of moving an application from the developer's machine to the production environment, while configuration management models help to get a grasp on the complex configuration spread out among the various micro-services.

7 LIMITATIONS

A limitation to this study is the time span for which papers are included (2015-2019). There is always a risk that a potentially relevant paper is excluded by enforcing this criterion. However, we have reasonable grounds to believe this risk is rather low. First, DevOps was only coined as a concept in 2009 when Patrick Dubois organized the first DevOps Days conference (https://legacy.devopsdays.org/events/2009-ghent/). In the next 5 years (2009-2014), research interest was minor: using the search queries found in the Appendix, there were only 3 papers published in the period 2010-2014. This is in stark contrast to the period 2015-2018, when 62 papers were published. From this we can conclude that by far the largest part of relevant literature has been included.

Another limitation to this study could be that only two scientific databases were used (Web of Science and Scopus). This might have reduced the variety of searchable literature. However, bibliographic studies (Harzing and Alakangas, 2016; Mongeon and Paul-Hus, 2016) on the coverage of Scopus and Web of Science suggest that it is one of the broadest and the most comprehensive searchable digital libraries. Therefore, we think that the chance of missing a study is relatively low.

As part of preparing this paper, we reviewed practitioners' articles from developers' community online magazines and industry-wide blogs that cover DevOps and related technologies: www.devops.com and www.devopsdigest.com. These two sites seemed relevant to our research in order to understand whether the practitioners' sources align with our findings. We searched these DevOps sites for papers by using "software architecture" as a search word. We then chose 20 papers from each site and looked in there for information related to our RQs. In this review, we however could not find a paper that provided information that was contradicting our findings. Nor information that adds to the lists presented in Tables 2 and 3. Of course, although this step was done in a structured way, it is not to mean that we compare it with a fully planned and executed systematic examination of grey literature (the practitioners' papers from community sites are in fact grey literature in the sense of Garousi et al. (2019)). We think that such examination that includes online community platforms in DevOps and CD is a worthwhile piece of work because only then we could develop a more complete understanding of what happens in practice.

8 DISCUSSION

Using Kitchenham's literature quality assessment guidelines (2007), the papers were ranked (see Table 4) on a scale from 1 to 5, where 1 means high quality and 5 means low quality. The result of this ranking is shown in Table 4, se the second column. It must be noted that the applicability of the quality assessment ranking schema (Kitchenham, 2007) could be questioned in the context of this research, as Kitchenham's ranking is based upon the research method used in the respective primary studies (i.e. the publications included in the set). We observe that very few of the papers collected evidence systematically by using a research method that was also explicitly described in much detail. Most of the papers included in our SLR report on the accumulation of experiences and anecdotic evidence by practitioners (e.g. Chen, 2015b). In turn, as our included papers did not have extensive presentation of research methods being employed, all papers have a relatively low quality ranking (see Table 4). This is important to take into account when reviewing the results of this SLR. We think there might be two reasons for the low quality: first, it can be the relatively young field of research (which is growing since 2014/2015); second, it can be the fact that the CD and DevOps are ideas created in industry, with industry taking the lead in publishing, compared to scholars.

Table 4 Literature quality assessment and countries of origin of the 13 selected papers.

Reference	Quality Assessment Rank	Origin
Stutevant, 2017	5	USA
Chen, 2015b	4	China
Erder and Pureur, 2015	5	USA
Woods, 2016	5	UK
Shahin et al., 2016	5	Australia
Elberzhager et al., 2017	2	Germany
Villamizar et al., 2015	2	Colombia
Schermann et al., 2018	5	Switzerland/ Austria
Stahl and Bosch, 2018	5	Sweden
Pahl et al., 2018	5	USA
Berger et al., 2017	5	Sweden
Chen et al., 2015	5	USA
Bass, 2017	5	USA

Another important aspect of the papers included in this SLR is the country or region of origin. As the third column of Table 4 shows, there is quite a variety in origin of the papers included in this SLR. This results in a representative body of literature that encompasses various organizational cultures, working styles and values. This allows us to think that our findings could possibly be generalizable (Wieringa and Daneva, 2015) across organizations in various countries.

A set of 10 potential software architectural issues has been identified in this paper. This set is not exhaustive and rather generic, however, it does create an image of the problems that practitioners might have to deal with when adopting CD/DevOps practices (on existing software systems). Some are rather obvious, such as decomposition and monolithic architectures, others are less straightforward, such as logging and monitoring in order to improve or retain the traceability capabilities of a system.

Furthermore, 17 software architectural characteristics that are considered beneficial for adopting CD/DevOps practices are presented. There is a clear Top-5 of most frequently discussed characteristics. It is apparent that there is some overlap to be found between the identified issues and the beneficial characteristics. These beneficial characteristics cannot directly be classified as solutions to the software architectural issues, e.g. testability (CH11) is not a solution to the problem domain of testing (IS8). However, the characteristics

indicate what features of a software architecture should be focused on in order to prevent or overcome the potential issues. The overlap between the issues and characteristics identified in the literature only strengthens their individual relevance to the problem domain.

We identified micro-services as the most dominant software architecture style that appears in literature with respect to CD/DevOps. All 8 papers that discuss a form of software architecture styles mention (or are solely centred around) micro-services. This was expected, due to the popularity of the style among practitioners and the fact that micro-services are designed to be independently developed and deployed. Many, if not all, of the beneficial software architectural characteristics are in some way addressed by the micro-services architecture style. Therefore, it seems that micro-services can be considered the answer to the main research question of this paper. We however think that this does not mean that the micro-services style itself does not bring any new problems, e.g. traceability, monitoring and logging are becoming increasingly complex when applying the micro-services architecture style. More research is needed in order to better understand the possible "side effects" of deploying micro-services in real world projects.

Furthermore while the selected literature in this work answered our RQs, these literature sources also touched upon a rather interesting topic: Evolutionary and Continuous Architecture. Both are considered as approaches of managing the architecture, which are designed to support incremental change to the architecture. This is in contrast to more conventional up-front architecture development methods, such as the well-known waterfall model Royce (1987) or the BDUF (Big Design Up Front) approach (these are not primarily software architecture development methods but do put emphasis on finalizing the architecture design before actually writing the software). Evolutionary and Continuous Architecture seem to bridge the gap between agile development methods (short term) and enterprise architecture (long term). This SLR could not go into depth regarding the topics of Evolutionary and Continuous Architecture, but it might be interesting for future research efforts to establish a link between software architecture in a CD/DevOps context and a form of Evolutionary/Continuous Architecture.

9 CONCLUSION

This section summarizes the answer to our RQs.

RQ1: What are potential software architectural problems when adopting CD and DevOps practices on an existing software system?

Since Jan 1, 2015, five papers were published that discuss software architectural problems when adopting CD/DevOps practices on an existing software system. From these five papers, 10 distinct problems were identified and reported in Table 2. The main architectural problem is traceable to the presence of highly-coupled monolithic systems that are hard to decompose for CD practices.

RQ2: What characteristics of a software architecture are important for enabling Continuous Delivery and DevOps?

From the six papers found in recent literature that discuss aspects or characteristics of software architectures that are important for enabling CD/DevOps, 17 (mostly distinct) characteristics were identified (see Table 3). Out of those, the Top-5 most frequently discussed characteristics are: Deployability (CH2), Testability (CH11), Automation (CH2), Loosely coupled (CH6), and Modifiability (CH1).

RQ3: What software architecture styles are suitable for being used in a Continuous Delivery and DevOps context?

Eight out of the 13 papers in our SLR discussed some form of software architecture style that would be suitable for use in a CD/DevOps context. Notable is the fact that all the papers that discussed some form of architecture style discussed micro-services. This demonstrates the dominance of this particular style in the research (and practitioners) domain. There were other styles mentioned, service-oriented architecture (of which micro-services is an implementation), and vertical layering (though it can be argued that this is not an architectural style on its own).

10 FUTURE WORK

This SLR has identified 10 distinct architectural problems when adopting CD and DevOps practices on an existing software system, the largest of these issues is that of highly coupled monolithic systems. Therefore, more research is necessary into efficient and fault-tolerant methods to migrate existing monolithic code bases towards micro-service architectures in order to improve the adoptability of CD/DevOps. The 17 characteristics that are important for enabling CD/DevOps, identified in this research, could be a contributing factor to that future research.

From another perspective, Continuous Architecture (CA), could fill a void between short

term software architecture and long term enterprise architecture. It would be interesting to investigate the role CA could play in enabling and improving CD/DevOps practices.

Finally, the problems and characteristics summarized in Table 2 and Table 3, respectively, could possibly be translated into a work of reference, a set of guidelines or a framework to help practitioners deal with software architecture related issues in the context of CD/DevOps. Developing such guidelines could be instrumental for consolidating practitioners' knowledge of the field.

REFERENCES

Fox, A., D.A. Patterson, and S. Joseph (2014) *Engineering software as a service: an agile approach using cloud computing*. 2014: Strawberry Canyon LLC.

Chen, L. (2015a) *Continuous delivery: Huge benefits, but challenges too*. IEEE Software, 32(2): p. 50-54.

Bass, L., I. Weber, L. Zhu (2015) *DevOps: A Software Architect's Perspective*. Addison-Wesley.

Sturtevant, D. (2017) *Modular Architectures Make You Agile in the Long Run.* IEEE Software, 35(1): p. 104-108.

Bucchiarone, A., et al. (2018) *From Monolithic to Microservices: An Experience Report from the Banking Domain.* IEEE Software, 35(3): p. 50-55.

Knoche, H., W. Hasselbring (2018) *Using Microservices for Legacy Software Modernization.* IEEE Software, 35(3): p. 44-49.

Bass, L., P. Clements, R. Kazman (2003) *Software architecture in practice*.Addison-Wesley.

Humble, J. and D. Farley (2010) *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education.

Kitchenham, B. (2007) *Guidelines for performing Systematic Literature Reviews in Software Engineering,* Keele University, UK.

Menzel, G. (2015) *DevOps - The Future of Application Lifecycle Automation*. [accessed July 11, 2018] https:// www.capgemini.com/wp-content/uploads/2017/07/ devops_pov_2015-12-18_final.pdf.

Kersten, N. (2017) *The 2017 State of DevOps Report*. Puppet + DORA, Portland, US.

Erich, F., C. Amrit, M. Daneva (2017) *A qualitative study of DevOps usage in practice.* Journal of Software: Evolution and Process, 29(6): p. e1885.

Ghantous, G.B., A. Gill (2017) *DevOps: Concepts, Practices, Tools, Benefits and Challenges.* In *PACIS 2017 Proceedings. 96.*

Senapathi, M., J. Buchan, H. Osman (2018). *DevOps Capabilities, Practices, and Challenges: Insights from a Case Study*. in *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. ACM.

Chen, L.P. (2015b) *Towards Architecting for Continuous Delivery*. 12th Working IEEE/IFIP Conference on Software Architecture, ed. L. Bass, P. Lago, and P. Kruchten. 131-134.

Erder, M., P. Pureur (2015) *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*. Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World. 2015. 1-303.

Woods, E. (2016) *Operational: The Forgotten Architectural View*. IEEE Software, 33(3): p. 20-23.

Shahin, M., M.A. Babar, and L. Zhu (2016) *The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives*.

Elberzhager, F., et al. (2017) *From Agile Development to DevOps: Going Towards Faster Releases at High Quality - Experiences from an Industrial Context*, in *Software Quality: Complexity and Challenges of Software Engineering in Emerging Technologies*, D. Winkler, S. Biffl, and J. Bergsmann, Eds. 33-44.

Villamizar, M., et al. (2015)*Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud*. 10th Computing Colombian Conference (10CCC).

Schermann, G., et al. (2018) *We're doing it live: A multi-method empirical study on continuous experimentation.* Information and Software Technology, 99 (7), 41-57.

Ståhl, D., J. Bosch, (2017) *Cinders: The continuous integration and delivery architecture framework.* Information & Software Technology, 2017. 83(3) 76-93.

Pahl, C., P. Jamshidi, O. Zimmermann (2018) *Architectural Principles for Cloud Software.* ACM Transactions on Internet Technology, 2018. 18(2).

Berger, C., et al. (2017) *Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices*. 2017 IEEE Int. Conf. on Software Architecture Workshops. 7-12.

Chen, H.M., et al. (2015), *Architectural Support for DevOps in a Neo-Metropolis BDaaS Platform*, in *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop*. 2015. p. 25-30.

Bass, L., *The Software Architect and DevOps* (2017) IEEE Software, 35(1), 8-10.

Harzing, A.-W., S. Alakangas (2016), *Google Scholar, Scopus and the Web of Science: a longitudinal and cross-disciplinary comparison.* Scientometrics, 106(2), 787-804.

Mongeon, P., A. Paul-Hus (2016) *The journal coverage of Web of Science and Scopus: a comparative analysis.* Scientometrics, 106(1), 213-228.

Royce, W.W. (1987) *Managing the development of large software systems: concepts and techniques*. in *Proceedings of the 9th international conference on*

Erich, F., Amrit, C., Daneva, M. (2014) A Mapping Study on Cooperation between Information System Development and Operations. PROFES'14, 277-280

Fritzsch, J., Bogner, J., Zimmermann, A., Wagner, S. (2018) From Monolith to Microservices: A Classification of Refactoring Approaches. DEVOPS 2018, 128-141

Kuhrmann, M., Méndez Fernández, D., Daneva, M. (2017) On the pragmatic design of literature studies in software engineering: an experience-based guideline. Empirical Software Engineering 22(6), 2852-2891

Wieringa, R.J., Daneva, M. (2015) Six strategies for generalizing software engineering theories. Sci. Comput. Program. 101: 136-152

Garousi, V., Felderer, M., Mäntylä, M.V. (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information & Software Technology 106: 101-121

APPENDIX

Scopus search query

TITLE-ABS-KEY ("software architecture" AND "continuous delivery" OR "continuous deployment" OR "devops" OR "dev-ops" OR "dev ops") AND (LIMIT-TO (PUBYEAR , 2018) OR LIMIT-TO (PUBYEAR , 2017) OR LIMIT-TO (PUBYEAR , 2016) OR LIMIT-TO (PUBYEAR , 2015))

Web of Science search query

TOPIC: ("software architecture" AND ("continuous delivery" OR "continuous deployment" OR "devops" OR "dev-ops" OR "dev ops")) Refined by: PUBLICATION YEARS: (2018 OR 2017 OR 2016 OR 2015) Timespan: Last 5 years. Indexes: SCI-EXPANDED, SSCI, A&HCI, CPCI-S, CPCI-SSH, ESCI