An empirical study on principles and practices of continuous delivery and deployment

Despite substantial recent research activity related to continuous delivery and deployment (CD), there has not yet been a systematic, empirical study on how the practices often associated with continuous deployment have found their way into the broader software industry. This raises the question to what extent our knowledge of the area is dominated by the peculiarities of a small number of industrial leaders, such as Facebook. To address this issue, we conducted a mixed-method empirical study, consisting of a pre-study on literature, qualitative interviews with 20 software developers or release engineers with heterogeneous backgrounds, and a Web-based quantitative survey that attracted 187 complete responses. A major trend in the results of our study is that architectural issues are currently one of the main barriers for CD adoption. Further, feature toggles as an implementation technique for partial rollouts lead to unwanted complexity, and require research on better abstractions and modelling techniques for runtime variability. Finally, we conclude that practitioners are in need for more principled approaches to release decision making, e.g., which features to conduct A/B tests on, or which metrics to evaluate.

An Empirical Study on Principles and Practices of Continuous Delivery and Deployment

Gerald Schermann*, Jürgen Cito*, Philipp Leitner*, Uwe Zdun†, Harald C. Gall*
*University of Zurich, Department of Informatics, Switzerland
†University of Vienna, Austria
*{schermann, cito, leitner, gall}@ifi.uzh.ch
†uwe.zdun@univie.ac.at

ABSTRACT

Despite substantial recent research activity related to continuous delivery and deployment (CD), there has not yet been a systematic, empirical study on how the practices often associated with continuous deployment have found their way into the broader software industry. This raises the question to what extent our knowledge of the area is dominated by the peculiarities of a small number of industrial leaders, such as Facebook. To address this issue, we conducted a mixed-method empirical study, consisting of a pre-study on literature, qualitative interviews with 20 software developers or release engineers with heterogeneous backgrounds, and a Web-based quantitative survey that attracted 187 complete responses. A major trend in the results of our study is that architectural issues are currently one of the main barriers for CD adoption. Further, feature toggles as an implementation technique for partial rollouts lead to unwanted complexity, and require research on better abstractions and modelling techniques for runtime variability. Finally, we conclude that practitioners are in need for more principled approaches to release decision making, e.g., which features to conduct A/B tests on, or which metrics to evaluate.

Keywords

empirical software engineering; release engineering; continuous delivery; continuous deployment

1. INTRODUCTION

In the wake of mainstream adoption of agile development practices, many software developing organizations are looking into ways to further speed up their release processes and to get their products to their customers faster. One instance of this is the current industry trend to “move fast and break things”, as made famous by Facebook [12] and in the meantime adopted by a number of other industry leaders [29]. Another example is continuous delivery [15], a release engineering practice that focuses on perpetually keeping software products in releasable state, supported by a high degree of automation.

Given that the adoption of continuous delivery and deployment (CD) has substantial impact on various aspects of software engineering (including, but not limited to, requirements engineering, architectural design, and testing), it seems imperative for the academic community to understand whether, and how, these ideas are actually implemented in industrial practice. Unfortunately, despite substantial recent research interest (e.g., [21, 27, 28]), our knowledge of CD practices outside of a number of well-known and outspoken industrial leaders remains spotty. This is concerning for two reasons. Firstly, it raises the question to what extent our view of these practices is coined by the peculiarities and needs of a few large Web-based innovation leaders, such as Facebook or Google. Secondly, it is difficult to establish what the real open research issues in the field are. This leads to the following research questions that guided this paper.

***RQ1**: What CD practices are already in use in the broader software industry?*

To address this question, we conducted a mixed-method empirical study in three steps. In a first step, we conducted a pre-study on literature to identify practices associated with CD in order to ground our research in the current state of the art. We then conducted a semi-structured interview study with 20 software developers and release engineers from 19 companies. We specifically focused on a mix of different team and company sizes, domains, and application models. Finally, we conducted a Web-based survey, attracting a total of 187 responses. Our study shows that the industrial adoption of CD practices is mixed. Some practices, most importantly continuous integration, health checking, “Developer on call” (this and other CD terminology is defined in Section 3.1), and, to a lesser extent, canary testing, are already widespread. Others, such as dark launches or A/B testing, are used much more seldomly, or are even largely unknown among practitioners.

***RQ2**: What are the underlying principles and practices that govern CD adoption in industry?*

A major recurring trend preventing the mainstream adoption of many CD practices are architectural concerns. Legacy applications are often not suitable for partial rollouts or higher automation, and require costly and difficult architectural redesign. The application model, most importantly whether the application is a Web-based SaaS application,

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

plays a role. Unsurprisingly, CD is most commonly used for Web-based applications, but variations of CD practices are also used for other application models. Customer expectations and domain are crucial. For instance, many companies simply do not have enough customers to sensefully conduct A/B tests on. Finally, we have seen that feature toggles as an implementation technique lead to unwanted complexity.

Based on the main observations of our study, we propose a number of promising directions for future academic research. Most importantly, given the importance of architecture for CD, we argue that further research is required on architectural styles that enable CD, for instance microservices. Further, given that feature toggles as an implementation technique for partial rollouts lead to unwanted complexity, research on better abstractions and modelling techniques for runtime variability are needed. Finally, we conclude that practitioners are in need of more principled approaches to release decision making (e.g., which features to conduct A/B tests on, or which metrics to evaluate). Ultimately, we envision this to lead to well-defined, structured CD processes implemented in code, which we refer to as Release-as-Code.

The rest of this paper is structured as follows. Section 2 gives more detail on our chosen research methodology, as well as on the demographics of our study participants and survey respondents. The main results of our research are summarized in Section 3, while more details on the main implications and derived future research directions are given in Section 4. Threats to the validity of our study are discussed in Section 5, and related previous work is covered in Section 6. Finally, we conclude the paper in Section 7.

2. RESEARCH METHODOLOGY

To answer our research questions, we conducted a mixed-method study [31] consisting of semi-structured, qualitative interviews followed by a quantitative survey. All interview materials and survey questions are part of the online appendix[1] of this paper. As a first step, prior to conducting qualitative interviews, we performed a pre-study to identify practices associated with CD and determine the scope of our interviews.

2.1 Pre-Study

The goal of the pre-study was to identify the practices companies are using in the field of continuous deployment and to serve as a basis for formulating questions for the qualitative part of our study. As a starting point, we studied [12, 15, 27], which we considered standard CD literature at the time we conducted our pre-study (the mapping study described in [28], which we also consider seminal for the field, was not yet available). From those sources, we extracted practices and categorized them into five categories: automation, rollout, quality assurance, issue detection and handling, and awareness. In order to also get an intial impression of the industrial view of the topic, and inspired by [4], we then used Hacker News[2] as an additional tool to revise and evaluate our categorization and findings. Articles were found using *hn.algolia.com*, a keyword-based Hacker News search engine. We searched for articles containing the keywords "continuous delivery" and "continuous deployment", which were posted between Jan 1 2011 and Nov 1 2015, and sorted them based on their popularity on Hacker News. For both keywords, we considered the first 80 articles, as we then reached saturation. Our main focus was on articles containing mainly experience reports, i.e., how companies make use of CD in the trenches. We removed those with dead links and those, which were mainly advertising specific tools. We ended up with 17 (continuous delivery) and 25 (continuous deployment) matching articles, which we then analyzed and compared to the practices derived from literature. The results showed that our set of practices was sound, and required only minor adaptations (e.g., renaming intercommunication to awareness). However, the results strengthened our confidence to use those derived practices as a basis to formalize our interview questions.

2.2 Qualitative Interview Study

Protocol. Based on our pre-study findings, we then conducted semi-structured interviews. To foster an exploratory character, we avoided asking direct questions (e.g., whether a given practice is used). Thus, we structured the interviews in five blocks: release process in general, roles and responsibilities, quality assurance, issue handling, and release evaluation. Each of those blocks started off with an open question. Except for the first block, topics were not covered in any particular order but instead followed the natural flow of the interview. The interviews were conducted by the first, the second, and the fourth author, either on-site in the areas of Zurich and Vienna, or remotely via Skype. All interviews where held in English or German, ranged between 35 and 60 minutes, and were recorded with the interviewee's approval. All selected quotes of interviews held in German were translated to English.

Participants. We recruited our interviewees from industry partners and our own personal networks. In total, we conducted 20 interviews with developers or release engineers (P1 to P20, one female) from companies across multiple domains and sizes, as illustrated on the left-hand side of Figure 1. Our interviewed companies ranged from single-person startups to global enterprises with more than 100,000 employees, located in Austria, Germany, Switzerland, Ireland, the Ukraine, and the United States. As the release process of mobile applications is strongly influenced by the peculiarities of app stores (e.g., the iTunes App Store or Google Play), we explicitly refrained from conducting interviews with companies developing mobile applications.

Analysis. The recorded interviews were transcribed by the first two authors. We coded the interviews on sentence level without any a-priori codes or categories. The first three authors then analyzed the qualitative data using open card sorting [33]. For this purpose, we created 683 cards in total from our interviewees' statements. We categorized cards into 9 themes that emerged over the course of card sorting. Each of those themes is further divided into multiple subcategories.

2.3 Quantitative Survey

Protocol. To validate and deepen the findings from our qualitative interviews on a larger sample size, we designed a Web-based survey consisting of, in total, 39 questions. The survey consisted of a combination of multiple-choice, single-choice, Likert-scale, and free-form questions. Depending on individual responses, we displayed different follow-up ques-

[1]http://www.ifi.uzh.ch/seal/people/schermann/projects/cd-study.html
[2]https://news.ycombinator.com/

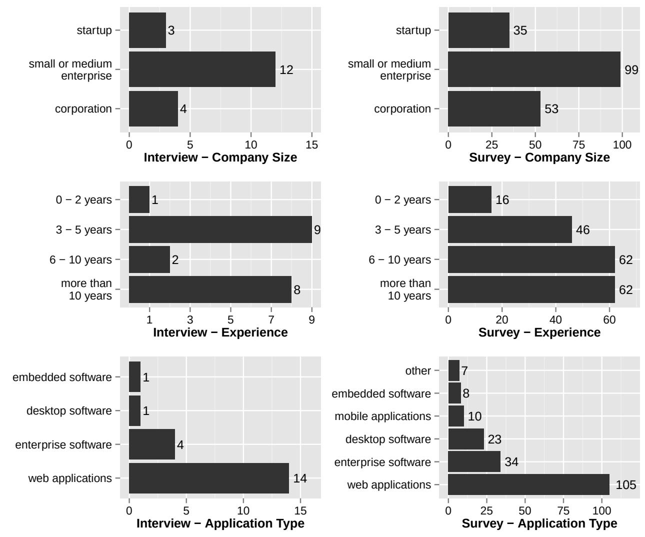

Figure 1: Demographics of interview study participants (left) and survey participants (right)

tions (branches) for the purpose of identifying underlying reasons. In total we had 7 branches in our survey, thus the number of questions a participant had to answer varied.

Participants. We distributed the survey within our personal networks, social media, via two DevOps related newsletters[3,4], and via a German-speaking IT news portal[5]. As monetary incentives have been found to have a positive effect on participation rates [32], we offered the option to enter a raffle for two Amazon 50$ gift vouchers on survey completion. In total, we collected 187 complete responses (completion rate of 28%). On average, it took the participants 12 minutes to fill the survey. The resulting participant demographics for the survey is summarized on the right-hand side of Figure 1.

Analysis. We analyzed the distributions of responses to Likert-scale, multiple-choice, and single-choice questions. In particular, we have correlated survey responses with the application model (Web-based or other) and the company size, as these two factors have emerged as important factors of influence in the interviews. Furthermore, we applied open coding on the answers to free-form questions. Those coded statements were then either attributed to the themes and categories which emerged from our card sorting, or led to new categories in cases where we were able to enhance our understanding.

3. RESULTS

We now discuss the main outcomes of our research, starting with the pre-study, followed by the main results of our qualitative and quantitative studies.

3.1 Pre-Study Results and Overview

From our pre-study, 9 practices emerged which are commonly discussed in a CD context. Following the example of [25], we arranged those practices in a "stairway to (CD) heaven" along the typical evolution path of companies moving towards more sophisticated, and often more automated, release engineering (Figure 2). The three phases relevant to this paper are *continuous integration*, *continuous deployment*, and *partial rollouts*. We now discuss these fundamental practices, in order to provide the reader with background information and relevant definitions, as we will use them in the rest of the paper. After that we will discuss the concrete implementation and prevalence of these practices based on our study outcomes. As there is no universally-accepted common definition of CD practices, the following definitions and descriptions represent the authors' own view as formed over the course of the pre-study.

Continuous Integration.

The core characteristic of continuous integration is that developers integrate code in a shared repository multiple times a day. To reduce the burden of long-living parallel development branches, companies have started to adopt the idea of **trunk-based development** [12], wherein all teams contribute to a single branch, usually called master, trunk, or mainline. Trunk-based development requires means to "switch on or off" individual code, if it is not ready for production. A common implementation technique for this are **feature toggles** [5]. In its simplest form, a feature toggle is a condition evaluating a flag (e.g., feature on/off) or an external parameter (e.g., userId) deciding which code block to execute. Besides enabling trunk-based development, feature toggles are also one potential implementation technique for various partial rollout practices [14]. Another hallmark of continuous integration is (full) **developer awareness**, sometimes referred to as transparency [28] or intercommunication [27]. Developer awareness is opposed to the siloization of release, status, and rollout information common to more traditional software engineering processes.

Continuous Deployment.

Fundamentally, continuous deployment assumes that the product perpetually remains in a shippable state. A (fully or at least partially automated) **deployment pipeline** [5, 15] comprises the core of a continuous deployment release process. A deployment pipeline consists of multiple defined phases a change has to pass until it reaches the production environment. Early phases handle compilation tasks and provide binaries for later phases focusing on either manual or automated tests in various environments. After deployment, **health checks** are central post-deployment activities to assess the deployed production code. Typically, health checks are implemented via monitoring [5] on infrastructure, application level, or business level. A cultural change strongly associated to continuous deployment and DevOps, which is a practice emphasizing a tighter collaboration between developers and operations, is the idea of **"developer on call"** [12]. "Developer on call" requires that software developers remain available for some time when their change is deployed to production. In case of problems, they know best about their change, and can help operations identify and fix problems faster.

Partial Rollouts.

Often seen as the epitome of CD, partial rollout practices are build on top of continuous deployment. Practically, partial rollouts are run-time quality assurance and requirements

[3]http://www.devopsweekly.com/
[4]http://sreweekly.com/
[5]http://heise.de

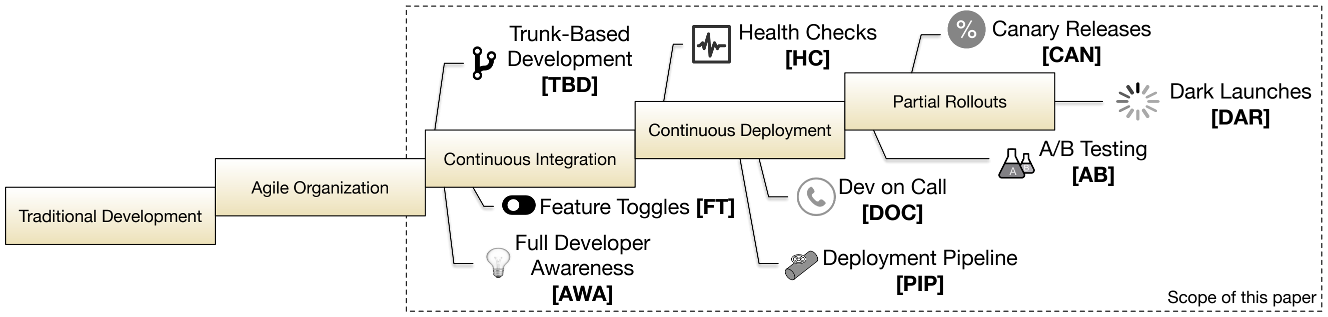

Figure 2: The "stairway to heaven" based on [25], and the main practices commonly associated to the phases in this evolution.

validation techniques. **Canary releases** [15] are releases of a new version to a subset of the user base (e.g., 10% of all users in a geographic region). This is done to test a change on a small sample of the user base first. **A/B testing** [20] comprises running two or more versions of an application in parallel, which differ in an isolated implementation detail. The goal is to statistically evaluate which of those versions performs better, based on technical (e.g., response time) and/or business metrics (e.g., items sold, conversion rate). Finally, **dark launches** [12] are used to test functionality in production, but without enabling it (yet) for any users. The idea is to assess whether a new or redesigned functionality is ready to handle production-scale traffic.

Results Overview.

Following the method used in [27], we provide a first understanding of the prevalence of these practices in the long tail of industrial practice by simply mapping, for each participant in the interview study, whether the participant's team uses (turquois), does not use (white), or partially uses (color graded turquois) the respective practice (Table 1). Participants are described in the form $PX_{Size}^{AppModel}$ in which X represents a particular participant followed by the company's size in subscript and the application model of the main product that this participant works on in superscript. Throughout the paper, we annotate our results by adding the interview participants that support the statements in parentheses. Mappings have been conducted by the first and third author, based on coded interview transcripts.

It is evident that adoption of CD practices does not strictly follow the "stairway to heaven" model. That is, for instance, while some semblance of continuous integration is a precondition to continuous deployment and partial rollouts, many companies have made a version of canary testing work without specifically implementing trunk-based development or feature toggles. Further, it is evident that some practices are substantially more prevalent in industry than others. Most of our interview partners have at least rudimentary health checks in place, as well as an at least partially automated deployment pipeline. Developer awareness (of the current build status, which users are currently served which version, etc.) is generally good, even if developers in many teams do not have access to *all* relevant information. "Developer on call" is a widely accepted practice among our interview participants. This strengthens our previous result that DevOps should by now be considered mainstream [9]. Further, and to our surprise, 13 of 20 interview participants are using, at

Participant	HC	PIP	AWA	DOC	CAN	FT	TBD	AB	DAR
$P1_{SME}^{Web}$	■	▨	▨	■	▨				
$P2_{SME}^{Enterpr}$		▨	▨						
$P3_{SME}^{Web}$	■	▨					■		
$P4_{SME}^{Web}$	■	■	▨	■	■				
$P5_{SME}^{Web}$	■	■	▨	■	▨				
$P6_{SME}^{Desktop}$	▨		▨		▨	■			
$P7_{Corp}^{Enterpr}$		■	■	■		▨	■		
$P8_{SME}^{Enterpr}$	■	▨	■	▨	▨				
$P9_{SME}^{Enterpr}$	■	■	▨	▨	▨	■	■		
$P10_{SME}^{Web}$	■	■	▨	▨					
$P11_{SME}^{Web}$	▨	■	■		■				
$P12_{Corp}^{Web}$	■	■	■	■	■				
$P13_{Startup}^{Web}$	■	■	■	■				▨	
$P14_{Corp}^{Web}$	■	■	■	■	■			■	
$P15_{Corp}^{Web}$	■	■	■	■	■				
$P16_{SME}^{Web}$	■	■	■	■	■	■			
$P17_{Startup}^{Web}$	■	■	▨	■			▨	■	
$P18_{Startup}^{Web}$	■	■	▨	■	▨	■			
$P19_{SME}^{Web}$	■	■	■	■	■	■	■	■	■
$P20_{SME}^{Embedded}$			▨			▨			

Table 1: Usage of CD practices by our interview participants. Acronyms (e.g., HC) follow Figure 2. Practices are ordered from the most prevalent (health checks) to the most uncommon (dark launches).

least for some changes, a variant of canary releases, even if it often takes the form of manually administered early-access or pilot phases. Feature toggles, trunk-based development, and A/B testing are known among our interview partners, but rarely used in practice. Finally, dark launches can be considered a niche practice in our study. All but one participant did not use it, and most participants were not even aware of this release practice, hence also had no intentions of using it in the future. Thus, we decided to refrain from putting questions about dark launches into our survey and set the focus for this paper on canary releases and A/B testing as practices for partial rollouts. Similarly, as we did not collect enough data to reason about trunk-based development, its implications for continuous deployments, and in general the implications of various branching models, we decided not to cover it in the result section of this work and leave it for future research.

We now discuss the results of the remaining seven practices that emerged from our study in more detail. As the adoption (or lack thereof) of automated and partial rollouts drives the adoption of many other CD practices, we start with a discussion of the industrial state of partial rollouts, then step down the stairway to heaven and discuss practices associated with continuous deployment, and finally cover practices related to continuous integration.

3.2 Partial Rollouts

The fundamental idea underlying partial rollouts is not uncommon in industrial practice. However, depending on the domain, technical sophistication of the project team, and other factors, the specific shape that partial rollouts take varies.

Canary Releases.

Interview Results. Our interviews have shown that among all practices associated with partial releases, canary releases are the ones that have most found their way into mainstream industrial practice. Users either apply canary releases for all changes, or, more commonly, use this practice for specific, often particularly, critical features. A common use case for canary testing was *scalability testing* in Web-based applications.

> *”[We use canary testing] especially in those cases when we have concerns how it would scale when all users get immediate access to this new feature.”* -$P4^{Web}_{SME}$

How these canary tests are implemented varies widely between participants. Many participants developing enterprise or desktop products (e.g., $P8^{Enterpr}_{SME}$, $P9^{Enterpr}_{SME}$) implement canary releases via manually-administered early-access or pilot phases with hand-picked friendly customers. For these participants, it may take weeks of canary testing before a change is rolled out to all customers. Others, particularly developers of Web-based applications, implement canary releases by gradually rolling out changes to some data centers or hosts. Finally, a small subset (e.g., $P19^{Web}_{SME}$) always roll out a change entirely, but then use feature toggles to serve only a subset of customers with the new version. A special case of canary releases via pilot phases is the *eat your own dog food* approach pioneered by companies such as Google or Facebook [12], in which companies (e.g., $P16^{Web}_{SME}$ in our study) use a very cutting-edge version of their own software internally before releasing externally.

Survey Results. Contrary to our interviews, our survey has shown that a majority of 63% of practitioners is not yet using canary releases (Table 2). However, and consistently with our interview results, the practice of canary releases – among those that actually make use of it – is not bound to companies developing Web-based applications.There is no statistically significant difference in our survey responses between canary release usage of developers of Web-based applications and others. However, while Web-based applications often use feature toggles and traffic routing to implement canary releases technically (both used by 45% of respondents that build a Web-based product and use canary testing), other application types (e.g., desktop applications) need to be canary tested by distributing specific binaries (47%) for technical reasons.

For the 63% of respondents that are not actually using canary releases, the largest obstacle is a software architecture that does not easily support partial rollouts. This was particularly evident for SMEs and corporations, and companies that develop Web-based products (64%, versus 48%

	all n=187	Web n=105	other n=82	start. n=35	SME n=99	corp. n=53
CAN: all features	18%	15%	22%	6%	22%	19%
CAN: some features	19%	21%	17%	17%	21%	17%
no CAN	63%	64%	61%	77%	57%	64%

Table 2: Canary release usage (single choice).

for others). It is likely that this is because most Web-based products in these domains are still deployed as monolithic 3-tier applications. For startups, software architecture is slightly less of a concern. However, startups often do not have a sufficiently large customer base to warrant partial rollouts. This is linked to a third, similar problem preventing the adoption of partial rollouts – some teams simply do not see any business value in conducting them. Interestingly, lack of expertise was seen only as a minor barrier to canary release adoption, given by 26% of respondents overall. A summary of the main reasons against adopting canary releases is shown in Table 3.

	all n=117	Web n=67	other n=50	start. n=27	SME n=56	corp. n=34
other	18%	1%	10%	7%	4%	6%
lack of expertise	26%	27%	24%	15%	34%	21%
no business sense	39%	39%	40%	41%	36%	44%
number customers	39%	46%	30%	56%	38%	29%
architecture	57%	64%	48%	44%	66%	53%

Table 3: Reasons against canary releases (multiple-choice).

Key Points. 37% do not roll out new features to every customer immediately. These canary releases are not limited to Web-based systems, but in other application types partial rollouts are often implemented manually. Monolithic architectures are the largest barrier for more widespread adoption of canary releases in Web-based applications.

A/B Testing.

Interview Results. A/B testing was rarely used among our interview participants. In total, three participants already conduct A/B tests, and one had concrete plans. The central use case for A/B tests was user facing frontends, mostly limited to the design of landing pages. $P14^{Web}_{Corp}$ is applying this practice not only to user interfaces, but also for the purpose of internal performance evaluation. Another participant emphasized that A/B testing is typically not a pure IT activity, but is strongly coupled with marketing and product development:

> *”If it's performance related, we will benchmark. If it's related to marketing, someone on the marketing side has to specify this.”* -$P17^{Web}_{Corp}$

An often-heard reason against using A/B testing in our interviews was that the customer base was too small to draw statistically valid conclusions from. However, the majority of participants also does not feel an urgent need for A/B testing, and consider other, more pragmatic, ways of assessing customer behavior, as sufficient:

> *"We only have around 130 customers, it is actually easier to just talk to everybody."* -$P18^{Web}_{Startup}$

Survey Results. Adoption of A/B testing among survey respondents was higher than indicated by our interview study, with 23% of respondents claiming to use A/B testing. Further, this practice is not only bound to companies developing Web-based applications, even though they still represent the majority with 63% of A/B test users. Consistent with our interviews, evaluating changes in the user interface is the most common use case (88%), but backend features are also A/B tested by 44% of the respondents that make use of A/B testing in the first place. A summary of the main reasons against A/B testing given by those not using the practice is depicted in Table 4.

	all n=144	**Web** n=78	**other** n=66	**start.** n=25	**SME** n=74	**corp.** n=45
other	6%	4%	8%	4%	1%	13%
don't know	6%	5%	6%	4%	7%	4%
lack of knowledge	15%	19%	11%	12%	15%	18%
policy / domain	21%	14%	29%	12%	22%	24%
number of users	28%	32%	23%	44%	27%	20%
investments	33%	35%	30%	44%	31%	29%
architecture	50%	53%	47%	40%	59%	40%

Table 4: Reasons against A/B testing (multiple-choice).

Unsurprisingly, and similarly to canary releases, for 77% of participants that are not making use of A/B testing, the biggest challenge is a software architecture that does not support running and comparing two or more versions in parallel. Confirming the results of canary releases, the architecture is mainly a problem for SMEs and corporations, while for startups a small user base is seen as a major obstacle. Once enough data points are collected to ensure statistical power, expertise is needed to analyze and draw correct conclusions. However, a lack of expertise was only mentioned by a minority of respondents (15%) as a problem. What is interesting and what we identified throughout our interviews as well, is that companies often do not have the features for which it would be worth conducting A/B tests. The return on investment, both financial and time, of creating and/or setting up appropriate tooling would be just too low. This was mentioned by 33% of our survey participants. While limitations because of internal policies are minor factor for startups (12%), for corporations this represents a strong barrier (24%). This underlines other findings (e.g., [21]) that besides technical challenges, organizational aspects are an important factor for CD adoption.

Key Points. 23% of survey respondents make use of A/B testing. In the majority of cases, A/B testing is applied on user frontends for Web-based applications. Similarly to canary releases, monolithic architectures are the most important barrier to adoption. However, organizational, business, user base, and domain considerations are additional factors that play into the decision of whether to adopt A/B testing.

3.3 Continuous Deployment

In this section, we report on the fundamental practices for successfully executing partial rollouts on production environments.

Health Checks.

Interview Results. Most of our interview participants have at least rudimentary health checks in place. The importance of monitoring applications arises once practices such as partial rollouts are in use. Health checks are not only used to determine if everything runs as expected, but also to support rollout decisions (e.g., increase traffic assigned to a canary release).

> *"The decision whether to continue rolling out is based on monitoring data. We look at log files, has something happened, did we get any customer feedback, if there is nothing for a couple of days, then we move on."* -$P16^{Web}_{SME}$

Metrics monitored by our interview partners primarily consist of well-known application (e.g., response times) and infrastructure (e.g., CPU utilization) metrics. This is consistent with our previous survey results [30]. Almost all participants mentioned that they do not only rely on such data, but also take customer feedback, for instance provided via bug reports, into account. Some companies (e.g., $P2^{Enterpr}_{SME}$) rely entirely on customer feedback, especially in cases when it concerns on-premises software, for which real-time monitoring is usually not feasible.

Most interviewees do not have strict rules or thresholds when defining what to monitor or how to interpret data. Instead, they conduct health assessments iteratively and primarily based on intuition. If something seems problematic, they take action based on experience (i.e., what has worked or was wrong in the past) rather than well-defined processes and empirical data. This is consistent with our experiences in earlier studies [9, 10]. If formal thresholds are used, they are often based on historical metrics gathered from previous releases.

> *"[Health assessment is] mostly based on the team experience. You may start with obvious ones [metrics and thresholds] and then over time, as you hit and bump into issues, you add more and more."* -$P14^{Web}_{Corp}$

Survey Results. Customer feedback (85%) and active monitoring (76%) are both widely used among survey respondents (see Table 5). For Web-based applications, monitoring and customer feedback are in balance, while for other application types, customer feedback (90%) is dominant (67% monitoring). This is not surprising, as monitoring Web-based applications is technically easier than for other application models, and supported by a large array of existing Application Performance Monitoring (APM) tools, such as New Relic[6].

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
don't know + other	4%	2%	6%	3%	5%	2%
monitoring	76%	83%	67%	89%	72%	75%
customer feedback	85%	81%	90%	80%	88%	83%

Table 5: How issues are usually detected (multiple-choice).

Based on our interviews, some companies have fixed schedules for rolling out releases (i.e., to a larger user base) driven

[6]http://newrelic.com/

by management decisions. Other companies are more flexible, they decide based on monitoring data and or by contacting customers whether to go forward with a release. Companies deploying a Web-based application (74%) are significantly more data-driven in their decisions than other companies (34%). Table 6 summarizes our findings across all company types. Survey respondents confirmed our finding that thresholds for assessing the state of a release are mainly formulated based on (developer) experience (72%), followed by the consideration of historical metrics from previous releases (54%).

	all n=70	**Web** n=38	**other** n=32	**start.** n=8	**SME** n=43	**corp.** n=19
don't know	7%	5%	9%	0%	9%	5%
monitoring data	56%	74%	34%	75%	51%	58%
customer feedback	59%	58%	59%	75%	60%	47%
mgmt. desicions	60%	58%	62%	38%	70%	47%

Table 6: How release decisions for partial rollouts usually are made (multiple-choice).

Key Points. At least rudimentary health checks are practiced by almost all companies. Besides active monitoring and logging, the majority of companies also take customer feedback (e.g., bug reports) into consideration for deciding on application status. Health checks are used for identifying issues in production and to provide a proper basis for rollout decisions. Data is mostly collected and interpreted iteratively without a principled approach, based on developer experience and intuition.

Deployment Pipeline.

Interview Results. All of our interviewed companies structure their release process into multiple phases. Continuous integration in a narrow sense is "solved" (see also [1]). All but one company is making use of CI, either on a nightly basis, or triggered after every push to the version control system. However, the degree of automation (i.e., manual or automated phase transitions), velocity (i.e., how fast single phases are passed), test scopes of single phases, and the number of phases along the pipeline, differ substantially.

Similar to [21] and [8], we identified both technical and organizational obstacles, which influence the pipeline's degree of automation as well as velocity. From the organizational perspective, internal policies (e.g., strict testing guidelines for $P4^{Web}_{SME}$), or customers which simply do not appreciate higher release frequencies (e.g., $P9^{Enterpr}_{SME}$) were stated several times. On the technical side, application architecture is again an essential driver for release frequency:

> *"It is difficult to release individual parts of the system since dependencies between new code and the system in the back are just too high"* -$P5^{Web}_{SME}$

In order to tackle this problem, $P5^{Web}_{SME}$ mentioned that they have started migrating from their monolithic application architecture to (micro-)services. In their current architecture, complex dependencies between various parts of the system prevent them from moving through their deployment pipeline faster. Besides a supporting architecture, the degree of automation is also bound to financial investments (e.g., tooling, setting up an appropriate infrastructure for automated quality checks):

> *"Depends on the size of the company. The larger you are, the more automation you want to have. The smaller you are, the more costly it is to build automation."* -$P19^{Web}_{SME}$

This reinforces a notion already described in [15]. Companies generally do not strive for immediate 100% automation just for the sake of it. Implementing higher automation is a slow process, and various challenges, of technical and organizational nature, need to be solved for climbing the "stairway to heaven".

An effect of highly automated pipelines is that not only new features reach production faster, but so do bugs, which can slip through automated quality checks along the pipeline. This changes the way how companies have to deal with issues:

> *"I think the faster you move, the more tolerant you have to be about small things going wrong, but the slower you move, the more tolerant you have to be with large change sets that can be unpredictable."* -$P18^{Web}_{Startup}$

Highly automated pipelines allow companies to fix those small issues fast, emerging practices such as automated health checks and "Developer on Call" help companies prevent severe damage. In addition, practices such as canary releases provide means for testing changes on small user bases first. However, in case that issues have found their way into production, some companies use different or modified pipelines to get patches faster deployed and to avoid situations as described by $P14^{Web}_{Corp}$:

> *"it is very frustrating for a developer ... the bug is found in production, the developer fixes it within a week, but it is not there until a month later or so."* -$P14^{Web}_{Corp}$

Fundamentally, we identified three ways how issues in production are handled within a delivery pipeline. (1) The same pipeline and the same steps are executed for hotfixes and feature commits. (2) The same pipeline is used for hotfixes, but some phases are skipped or only a subset of tests is executed. (3) There is an entirely separate pipeline for hotfixes. While the former is an indication that the delivery pipeline has reached a high degree of automation and established a high level of confidence, the second may be an indication that the pipeline per se simply takes too long. Completely separated pipelines have the benefit of avoiding interference between the version to be fixed and the progress made in development across the various phases and environments along the pipeline.

Survey Results. 68% of the respondents stated that they use the same pipeline for dealing with issues as for every other change. The remaining 30% (2% unknown) have either a modified or entirely separate pipeline. In these cases, executing a subset of tests and skipped single phases are dominant, while a completely separated pipeline is only used by a minority (see Table 7).

A majority of 74% of survey respondents agreed that, from a technical perspective (i.e., pipeline and tooling), they could release more frequently than they actually do. The main reason that keeps them from doing so is a missing business case. This is also valid for companies developing Web-based applications (50%). However, for non Web-based applications, internal policies are even more important (40%).

	all n=57	**Web** n=32	**other** n=25	**start.** n=9	**SME** n=31	**corp.** n=17
don't know	7%	3%	12%	0%	10%	6%
separate pipeline	23%	22%	24%	11%	26%	24%
subset of tests	51%	53%	48%	67%	42%	59%
reduced phases	53%	53%	52%	56%	55%	47%

Table 7: How the deployment pipeline for hotfixes differs from the usual pipeline (multiple-choice).

Considering various company sizes, a missing business case for faster releases is the top reason for startups (56%), while they are not limited by internal policies (4%). Table 8 shows the non-technical reasons in relation to more frequent releases.

	all n=138	**Web** n=80	**other** n=58	**start.** n=27	**SME** n=74	**corp.** n=37
don't know	12%	11%	14%	22%	12%	5%
other	13%	15%	10%	19%	9%	16%
customers	22%	18%	28%	15%	22%	27%
internal policies	30%	24%	40%	4%	39%	32%
no business case	42%	50%	31%	56%	38%	41%

Table 8: Non technical reasons that prevent companies from releasing more frequently (multiple-choice).

Key Points. The practice of deployment pipelines, i.e., structuring the release process into multiple, at least partially automated, consecutive phases is widespread. Barriers for higher flexibility and automation are internal policies and missing business cases, but also include architectural problems.

Developer Awareness.

Interview Results. Awareness in CD refers to activities that enable and promote transparency of the development progress throughout the "stairway to heaven". Most of our interview participants agreed that developers in their organization usually know or have access to the information in what stage a feature or change that they worked on is currently in. This information was usually provided in different ways. One option is tooling that tracks status or progress of features through tasks or tickets (e.g., Pivotal tracker). Another are online dashboards, or public monitors in the office, which display information such as build status, test results, or production performance metrics. Another way to promote awareness and transparency is through signals sent in the form of asynchronous communication that is integrated within team collaboration chat tools (e.g., Slack, HipChat) [22].

The majority of interviewees agree that the notion of "Developer on call" has become a widely accepted practice in their organization. Developers that contributed code to a certain release are on call for that release. In case of issues, developers know best about their changes and can help operations to identify the problem faster and contribute to the decision about subsequent actions. Additionally, $P16^{Web}_{SME}$ also specifically mentions a learning effect for developers due to "Developer on call":

> *"Developers need to feel the pain they cause for customers. The closer they are to operations the better, because of the massive learning effect."* -$P16^{Web}_{SME}$

This practice is strongly related to DevOps and emphasizes a shift in culture that is currently taking place. Traditional borders between development, quality assurance, and operations seem to vanish progressively. This plus of responsibility could lead developers to writing and testing their code more thoroughly, as one participant indicated:

> *"If you don't have enough tests and you deploy bad code it will fire back because you would be on call and you have to support it"* -$P14^{Web}_{Corp}$

Some participants mention that their companies avoid the additional burden of keeping developers on call on weekends by releasing only during office hours, for instance $P7^{Enterpr}_{Corp}$. However, for certain companies and domains, deployment weekends are a business necessity (e.g., $P9^{Enterpr}_{SME}$).

Survey Results. The survey confirmed our findings that "developer on call" has become mainstream. The majority of survey respondents stated that developers never hand off their responsibility for a change (see Table 9). When comparing company sizes, this practice is especially appealing to startups (74%), but even in corporations (45%) it is applied frequently. While in SMEs and corporations (23%) developers hand off their responsibility directly after development, this is almost never the case for startups (3%).

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
don't know + other	4%	2%	5%	3%	1%	8%
preproduction	9%	10%	9%	9%	8%	11%
staging	12%	15%	9%	11%	12%	13%
development	19%	12%	28%	3%	23%	23%
never	56%	61%	50%	74%	56%	45%

Table 9: Phase in the delivery pipeline after which developers typically hand off responsibility for their code (single-choice).

Key Points. Awareness and transparency in CD are achieved by means of tracking tools, publicly advertised dashboards, and chat integration. "Developer on call" has become mainstream. It is widely practiced across companies of all sizes, but particularly by startups. Companies appreciate that this additional responsibility results in increased overall awareness for contributed changes and strengthens collaboration within the company, but may induce additional developer burden, especially if releases are required during weekends.

3.4 Continuous Integration

The next sections cover the practices assigned to the continuous integration phase in the "stairway to heaven".

Feature Toggles.

Interview Results. Feature toggles were rarely used among our interview participants. Interestingly, some of our interview participants associated feature toggles with permission mechanisms, similarly to [14]. Feature toggles may be used as well for the purpose of regulating resource access (e.g., $P9^{Enterpr}_{SME}$). Another usage category we identified is enabling/disabling code that is not yet ready for production (e.g., $P19^{Web}_{SME}$, $P20^{Embedded}_{SME}$).

> *"We do that a lot. We'll push the code dark behind a feature flag. We might only enable it in certain environments, we might only enable it in CI and staging. And the day we decide to release it, we just 'turn it on' in production."* -$P19^{Web}_{SME}$

Only a minority was using feature toggles for implementing canary releases ($P16^{Web}_{SME}$, $P19^{Web}_{SME}$) and A/B testing ($P19^{Web}_{SME}$). From the group of participants not considering feature toggles, a major concern was added complexity. This is particularly relevant when multiple feature toggles are active at the same time.

> *"I'm not using feature toggles and I don't have plans to do so [...] configuration leads to complexity, every time you add complexity, you end up having additional complexity when you have to remove it at some point."* -$P13^{Web}_{Startup}$

This problem is aggravated by the fact that none of our interview participants has access to abstractions or tooling beyond the very basic to manage feature toggles.

Survey Results. In contrast to our interview study, our survey respondents are more commonly using feature toggles to implement partial rollouts (36%, see Table 10). Feature toggles are especially used by companies providing Web-based products (45%), while they are less frequently used for other application models (25%). Concerning company sizes, startups show the highest adoption rate. However, with $n = 8$, the sample size for this specific observation is small.

	all n=70	**Web** n=38	**other** n=32	**start.** n=8	**SME** n=43	**corp.** n=19
other	6%	8%	3%	12%	5%	5%
permissions	17%	18%	16%	38%	16%	11%
dont' know	20%	13%	28%	12%	21%	21%
binaries	29%	13%	47%	12%	33%	26%
proxying	30%	45%	12%	38%	23%	42%
feature Toggles	36%	45%	25%	50%	35%	32%

Table 10: Implementation techniques in use for partial releases (multiple-choice).

Key Points. While feature toggles are a common implementation technique for partial rollouts, they add complexity and need to be managed with care. Currently, no sufficient abstractions or tooling for managing this complexity are available.

4. IMPLICATIONS

We now discuss the main implications of our study for the academic community. We focus on the underlying problems and principles we have observed, and propose directions for future research.

Architectural support for CD. As discussed in Section 3.2, one substantial barrier for many companies to CD adoption is a (legacy) system architecture that makes advanced practices, such as canary or A/B testing, hard to implement in production. Similarly, application architectures drive the design of deployment pipelines (see Section 3.3) and thus set the pace with which companies bring their changes to production. Moreover, we have observed that applying feature toggles (see Section 3.4) to circumvent architectural limitations for implementing partial rollouts come at the price of increased complexity, which negatively affects maintainability and code comprehension. Hence, we argue that future research is required on how to architect a software system to enable CD. This could come, for instance, in form of a catalogue of architectural patterns which supports various CD practices. Microservices [3, 30] are a promising starting point, but the community is currently lacking formal research into the tradeoffs associated with the microservices architectural style, its suitability for various CD practices, and how to decompose an application into microservices in the first place.

Modelling of variability. Related to the previous implication, the results reported in Section 3.4 imply that practitioners currently struggle with the complexity induced by feature toggles. Hence, it can be argued that more research is needed on better formalisms for modelling the software variability induced by feature toggles, as well as for their practical implementation without polluting the application's source code with release engineering functionality. We suspect that concepts known from aspect-oriented software development [19] and (dynamic) product line engineering [13] could serve as useful abstractions in this domain. However, their usage did not emerge in our study even though these techniques have been available for years.

From intuition to principled release decision making. In Section 3.3, we have observed that release engineers are currently mostly going by intuition and experience when defining metrics and thresholds to evaluate the success, or lack thereof, of a (partial) rollout. Similarly, which features to conduct canary or A/B tests on, or which (or which fraction of) users to evaluate, is currently rarely based on a sound statistical or empirical basis. Hence, research should strive to identify (for various application types) the principal metrics that allow evaluating the success of a (partial) rollout, and identify best practices on how to select changes that require canary or A/B testing. Further, robust statistical methods need to be devised that suggest how long to run at which scope (e.g., number of users) to achieve the required level of confidence. A main challenge for this line of research will be that release engineers cannot generally be expected to be trained data analysts. This is particularly true for smaller companies, for which release decision making needs to remain cost-efficient and statistically sound on a small sample size.

Release-as-Code. Once a more principled approach for releasing is available, more research can be conducted on how to further automate releases based on CD, for instance through well-defined and (semi-)automated release scripts. We refer to this idea as "Release-as-Code", analogously to Infrastructure-as-Code [16]. Such Release-as-Code scripts are structured in multiple phases (e.g., canary release on 5% of traffic for 15 minutes on users of the free tier, after success based on a given metric 10% for an hour on all users, ...) with clearly specified gateways and repair actions. Release-as-Code will not only provide means for further automation and speeding up the release process, but also facilitate the documentation, transparency, and even formal verification of release processes. Besides a fundamental model, research should be conducted on a useful and usable domain-specific language to implement Release-as-Code.

The many hats of DevOps developers. Our results showed that not only DevOps, but also and more specifically "developer on call", is becoming mainstream across all application models and company sizes. We argue that it is required that the academic community revisits and validates through principled studies to what extent the roles and responsibilities traditionally associated with the term "software developer" in academic research are still a good representation of industrial practice.

5. THREATS TO VALIDITY

We have designed our research as a mixed-method study to reduce threats to the validity. However, as in any empirical study, there are still threats and limitations that the reader should keep in mind when interpreting our results.

External Validity. There is a possibility that our results are not generalizable beyond the subjects that were involved in the study. To mitigate this threat, we made sure to select interview participants that are approximately evenly distributed between organizations of varying sizes, backgrounds (years of experience and age), as well as the types of applications the participants deploy in their daily work. This threat is further addressed through the validation of our interview findings through a quantitative survey. We advertised our survey over various social media channels to attract a high number of respondents. However, participation in online surveys is necessarily voluntary. Hence, it is likely that the survey has attracted a respondent demography with substantial interest and familiarity with CD practices (self-selection bias). Furthermore, for both interviews and survey, we rely on self-reported (as opposed to observed) behavior and practices (self-reporting bias). Hence, participants may have provided idealized data about the CD maturity of their companies. Due to these biases, we suspect that our study, at least to a small extent, overestimates the industrial prevalence of advanced CD techniques.

Internal Validity. The selection of questions for the interview phase might have lead participants to answer towards our possibly biased notion of CD. We mitigated this threat by building a foundation of understanding on the topic that is based on both previous work and experience reports in online articles in a pre-study. Furthermore, it is possible that we introduced bias through the mis-interpretation or mis-translation of "raw" results (interview transcripts and survey results). To avoid observer bias, these results were analyzed and coded by at least three authors of the study.

6. RELATED WORK

There has recently been a multitude of research on the challenges companies face on their way to more continuous deployments. Leppanen et al. [21] and Olson et al. [25] conducted studies with multiple companies discussing technical and organizational challenges, and their state of CD adoption. Similarly, Chen [8], and Neely and Stolt [24] provide experience reports from a perspective of a single company. Besides technical challenges, Claps et al. [11] focused also on social challenges companies are faced with and present mitigation strategies. Bellomo et al. [6] investigated architectural decisions companies take to enable CD and introduced deployability and design tactics.

As Facebook is one of the drivers in the professional developer scene surrounding CD, the company is also commonly the subject of related studies. Feitelson et al. [12] describe practices Facebook adopted to release on a daily basis. In a recent work, Tang et al. [35] give insights how Facebook manages multiple versions running in parallel (e.g., A/B testing), make use of a sophisticated configuration-as-code approach, and monitor their applications at runtime. Bakshy and Frachtenberg [2] provide guidelines for correctly designing and analyzing benchmark experiments such as A/B testing. Considering A/B testing, Tamburrelli and Margara [34] rephrase A/B testing as a search-based software engineering problem targeting automation by relying on aspect-oriented programming and genetic algorithms. Tarvo et al. [36] built a tool for automated canary testing of cloud-based applications incorporating the automated collection and analysis of metrics using statistics.

In case that things go wrong with such partial rollouts, the question remains whether to deploy a hotfix ("roll forward") or to roll back to a known stable version. In a recent study, Kerzazi and Adams [17] took a detailed look at releases showing abnormal system behavior after being deployed to production. One of the findings was that source code is not the major artifact causing problems, they are often introduced by faulty configuration or database scripts. There has also been research on release frequency and its impacts. Khomh et al. [18] studied the impact of release frequency on software quality and observed that with shorter release cycles users do not experience more post-release bugs. Mäntylä et al. [23] analyzed the impact on software testing efforts when switching from a more traditional release model to a 6 weeks release train model in a case study for Mozilla Firefox.

Finally, there have also been some studies on the state of the art in DevOps. The most authoritive source on this is [26], albeit not an academic study per se. Other related works in this field include our own previous work [9] and the work conducted in the CloudWave project [7].

None of the research discussed so far has empirically evaluated how CD practices associated with CD are actually adopted in practice in a larger sample size. This was also mentioned by Adams and McIntosh [1] and Rodriguez et al. [28]. Besides giving an overview of release engineering practices and phases, Adams and McIntosh [1] provide a roadmap for future research, similar as Rodriguez et al. [28], who conducted a systematic literature review on continuous delivery and deployment research articles. Rahman et al. [27] did a step into the same direction as we do within this paper. Starting with the practices used by Facebook, they conducted a qualitative analysis of the CD practices performed by 19 software companies by analyzing company blogs and similar online texts. However, they did not conduct interviews or a formal survey beyond what is already available in blogs.

7. CONCLUSIONS

We report on a systematic, empirical study on what practices associated with CD have been adopted by software industry, and what the underlying principles and practices of adoption are. We have observed that some of the CD practices only play a minor role in industry at the moment (e.g., A/B testing, dark launches), while other practices have already found their way into mainstream adoption. Most companies have at least rudimentary health checks in place and at least partially automated deployment pipelines. 37% of our survey respondents make use of canary releases, even though some of them only in form of manually administered pilot phases. Architectural concerns, customer expectations, and domain are the most important barriers to CD adoption. We argue that particularly the first of these issues calls for future research on how to architect and decompose systems to enable CD practices, and how to model and implement them correctly. We have observed that feature toggles alone are often not enough, as many practitioners are daunted by the increase in complexity of managing them. Decision making in release processes is currently mostly an experience-driven "art", with little empirical or formal basis. Hence, more research on principled approaches is required, as well as tools and languages that actually implement these principled approaches in a way that is approachable and usable for non-expert release engineers. Finally, we have observed that "developer on call" is becoming a mainstream DevOps technique, requiring the academic community to revisit how it thinks about the role of software developers in industry.

8. ACKNOWLEDGMENTS

The authors would like to thank all interview and survey participants. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss National Science Foundation (SNF) under project "Whiteboard" (SNF Project no. 149450).

9. REFERENCES

[1] B. Adams and S. McIntosh. Modern Release Engineering in a Nutshell – Why Researchers should Care. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), Future of Software Engineering (FOSE) Track*, 2016.

[2] E. Bakshy and E. Frachtenberg. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pages 108–118, 2015.

[3] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. *IEEE Software*, 2016. To appear.

[4] T. Barik, B. Johnson, and E. Murphy-Hill. I Heart Hacker News: Expanding Qualitative Research Findings by Analyzing Social News Websites. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 882–885, New York, NY, USA, 2015. ACM.

[5] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, jun 2015.

[6] S. Bellomo, N. Ernst, R. Nord, and R. Kazman. Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 702–707, June 2014.

[7] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, and C. Woods. CloudWave: where Adaptive Cloud Management Meets DevOps. In *Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014)*, 2014.

[8] L. Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *Software, IEEE*, 32(2):50–54, Mar 2015.

[9] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA, 2015. ACM.

[10] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth. Runtime Metric Meets Developer - Building Better Cloud Applications Using Feedback. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*, New York, NY, USA, 2015. ACM.

[11] G. G. Claps, R. B. Svensson, and A. Aurum. On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way. *Information and Software Technology*, 57(0):21 – 31, 2015.

[12] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, April 2008.

[14] P. Hodgson. Feature Toggles. http://martinfowler.com/articles/feature-toggles.html, Jan. 2016.

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

[16] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. Testing Idempotence for Infrastructure as Code. In *Proceedings of the ACM/IFIP/USENIX Middleware Conference (MIDDLEWARE)*, pages 368–388, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] N. Kerzazi and B. Adams. Botched Releases: Do we Need to Roll Back? Empirical Study on a Commercial Web App. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[18] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do Faster Releases Improve Software Quality?: An Empirical Case Study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 179–188, Piscataway, NJ, USA, 2012. IEEE Press.

[19] G. Kiczales. Aspect-oriented Programming. *ACM Computing Surveys*, 28(4), Dec. 1996.

[20] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[21] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. Mantyla, and T. Mannisto. The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72, Mar 2015.

[22] B. Lin, A. Zagalsky, M. Storey, and A. Serebrenik. Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW) Companion*, pages 333–336, New York, NY, USA, 2016. ACM.

[23] M. V. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen. On Rapid Releases and Software Testing: A Case Study and a Semi-Systematic Literature Review. *Empirical Software Engineering*, 2014.

[24] S. Neely and S. Stolt. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In *Agile Conference (AGILE), 2013*, pages 121–128, Aug 2013.

[25] H. Olsson, H. Alahyari, and J. Bosch. Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 392–399, Sept 2012.

[26] Puppet Labs. State of DevOps Report. https://puppetlabs.com/2015-devops-report, 2015.

[27] A. Rahman, E. Helms, L. Williams, and C. Parnin. Synthesizing continuous deployment practices used in software development. In *Agile Conference (AGILE), 2015*, pages 1–10, Aug 2015.

[28] P. Rodríguez, A. Haghighatkhah, L. E. Lwakatare, S. Teppola, T. Suomalainen, J. Eskeli, T. Karvonen, P. Kuvaja, J. M. Verner, and M. Oivo. Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study. *Journal of Systems and Software*, 2016. To appear.

[29] J. Rubin and M. Rinard. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016. To appear.

[30] G. Schermann, J. Cito, and P. Leitner. All the Services Large and Micro: Revisiting Industrial Practice in Services Computing. In *Proceedings of the 11th International Workshop on Engineering Service Oriented Applications (WESOA'15)*, 2015.

[31] F. Shull, J. Singer, and D. I. Sjøberg. *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[32] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving Developer Participation Rates in Surveys. In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 89–92, May 2013.

[33] D. Spencer. *Card Sorting: Designing Usable Categories*. Rosenfeld Media, 2009.

[34] G. Tamburrelli and A. Margara. Towards Automated A/B Testing. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE)*, volume 8636 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2014.

[35] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 328–343, New York, NY, USA, 2015. ACM.

[36] A. Tarvo, P. F. Sweeney, N. Mitchell, V. Rajan, M. Arnold, and I. Baldini. CanaryAdvisor: A Statistical-based Tool for Canary Testing (Demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 418–422, New York, NY, USA, 2015. ACM.