Deployment strategies and practices in CI/CD Pipelines

Niko Benkler

Supervisor: Dr. Robert Heinrich
robert.heinrich@kit.edu

Abstract. In the last decade, software development experienced a huge transition. Since agile methodologies were introduces in the early 2000, software development became faster and faster. Today, another software development process is emerging: Continuous Software Engineering (CSE). CSE, especially Continuous Integration(CI), Continuous Delivery(CDE) and Continuous Deployment(CD) receive more and more attention in organizations such as Facebook, Paddy Power and Atlassian but also in small start-up companies. It enables them to e.g. deliver software more frequently, reduce time-to-market, obtain customer feedback faster, build the right product or to improve product quality. Therefore, this seminar paper presents the current state of the art concerning Continuous Practices, compares the traditional deployment strategies with the new CSE practices and proposes some tools, that can be used in a CI/CD Pipeline to support CSE. But also, it reveals the barriers an organization has to face while adopting Continuous Practices. CSE not only turns up with benefits, it also has negative side effects such as a time consuming and costly implementation or an increasing mental stress for development team members.

Keywords - Agile, continuous software development, continuous integration, continuous delivery, continuous deployment, DevOps, CI/CD Pipeline

Table of Contents

1 Introduction

Today, the software development process has to face many difficult demands. Fast-changing and unpredictable markets, changing customer requirements [ClSA15] and rapidly advancing information technologies [OlAB12] require a faster process of software development. To achieve this, several organizations adopt Continuous Practices in order to extend their agile practices [ClSA15]. Therefore, releasing software becomes even faster.
This seminar paper presents a possible evolution path, referenced as 'stairway to heaven' [OlAB12], which describes a transition from traditional development towards CD. As the core of CDE is a deployment pipeline [SCLZ+16], the paper also presents its usual phases and the possible tools, that support each phase. Nevertheless, studying several papers revealed, that CDE not only comes with benefits, but also with huge social and technical challenges [ClSA15]. The paper clarifies them and proposes some mitigation strategies. The remainder of the seminar paper is structured as follows: In section II, we define the terminology. Section III describes a possible transition from traditional deployment to CD based on the 'stairway to heaven' model [OlAB12] and [SBZZ17].This includes several deployment strategies used to adapt CSE practices and the comparison between traditional deployment and the CD process. That section is followed by an explanation of a possible pipeline and the available tools which can be used to support the tasks of each phase. Section V discusses the challenges that are caused by CDE and possible mitigation strategies. Finally, I present my conclusion in section VI.

2 Background

Here, I give an overview of the most important keywords. Those keywords are necessary to understand the content of the seminar paper. When studying the currently available information about Continuous Software Engineering (CSE), one could clearly see that there are no universally accepted definitions [SCLZ+16]. Terms like CDE and CD are sometimes used interchangeable, albeit the is a small but relevant difference [ShBZ17]. Therefore, the following subsections provide the definitions as they are used in this seminar paper.

2.1 Agile Methodologies

Agile software development is based on "iterative development, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams" [TaKB16]. It encourages adaptive planning, early delivery and continuous improvement. Besides, other characteristics such as flexibility, efficiency, speed, the ability to react to fast changing customer requirements and fluctuating market needs [OlAB12] make agile methodologies so attractive for organizations. As mentioned before, agile methodologies were introduced in the

early 2000 [ShBZ17]. So far, many development companies succeeded in implementing agile methodologies. However, agile software development 'only' allows frequent software releases but no continuous releases. The difference becomes clear, when it comes to the comparison of traditional deployment and CD. The original "Manifesto for agile software development" popularized by Kent Beck did not include any automation.

2.2 DevOps

"**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" [BaWZ15]. It is a coined word, that stands for Development (Dev) and Operations (Ops). The main task is bringing together programmers, testers and quality insurance engineers but also IT operations staff [ClSA15] to shorten feedback loop. According to [BaWZ15], the DevOps practices can be differentiated in 5 categories:

- Treat Ops as "first-class citizens". This means, the involvement of operations in the development process.
- Dev has to be more responsible for incident handling to shorten the time between error observation an the error fix.
- Include both, Dev and Ops into the deployment process in order to avoid errors caused by buggy deployment.
- Use continuous deployment.
- Use deployment scripts and other infrastructure code. This code is subjected to the same quality control practices than the application code. As a result, deployment error due to misconfiguration can be mitigated.

Especially the fourth category shows the close connection between DevOps practices and continuous practices.

2.3 Continuous software engineering

Continuous software engineering **(CES)** is a practice that promotes the development, deployment and the release of software products in very short cycles, typically hours or days. [ShBZ17] [TaKB16]. As a direct consequence, quick feedback allows i.a.: to determine new functionality to build, feature prioritization, information about the suitability of the current software architecture, to gather data for decision making in general. CES requires agile methodologies and DevOps practices [TaKB16]. According to [ShBZ17], CES involves 3 phases: Business Strategy and Planning, Development, Operations. This seminar paper focuses on the three development activities: continuous integration, continuous delivery and continuous deployment. Fig. 1 demonstrates the relationship between CI, CDE and CD.

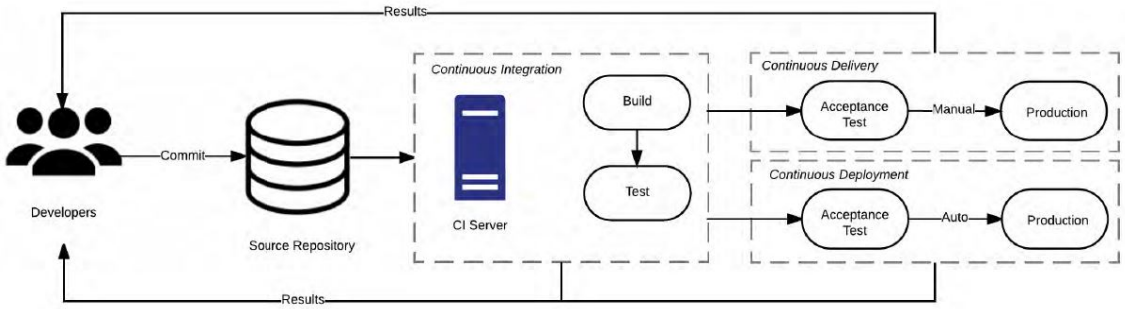

Fig. 1. The relationship between CI, CDE, CD [ShBZ17]

2.3.1 Continuous Integration Continuous Integration **(CI)** is a CSE practice in which developers integrate and merge their work into a shared repository very frequently, for example multiple times a day [ShBZ17] [SCLZ+16]. Also, automated builds and tests permit to discover integration, build and test errors as fast as possible. CI improves the effectiveness of a team as much as the software quality [ShBZ17]. In other words, CI ensures, that the software is always in a ready to deploy state [ClSA15].

2.3.2 Continuous Delivery A brief definition of Continuous Delivery **(CDE)** is given in [LLIP+16]: "Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time." Preconditions for CDE are CI and a mechanism (e.g a tool) to automatically deploy and deliver software to a production like environment [ShBZ17]. This practice has several benefits, such as a less risky release process [LLIP+16] due to automation or the ability to learn from real usage data [OlAB12]. As a consequence, the software developers can decide whether it is worth continuing the work on a certain feature or discard it. As shown in Fig.1, the release process is manually. This is concurrent with the definition of CDE, where it is mentioned that software 'can' be released automatically instead of 'is' released. Therefore, CDE practice is classified as a pull-based approach, "for which business decides what and when to deploy" [ShBZ17].

2.3.3 Continuous Deployment An extreme version of CDE is Continuous Deployment **(CD)**, where software 'is' actually released automatically. The goal of CD is to frequently deploy the software in a production environment. This happens for example several times a day (after each commit), one times a day (nightly build) or several times a week (weekend builds). Fig. 2 provides a quick overview of CDE's and consequently CD's benefits. CD is a push-based approach [ShBZ17], as no manual task takes place in order to deploy the software to customer site. The CD (and CDE) practice is attained by using a deployment

pipeline [LLIP+16], which is explained in Section IV. Whereas CD implies the usage of CDE practices, the inversion is not correct [ShBZ17]. Due to barriers such as "lack of automated (user) acceptance tests", "deployment as a business decision", "insufficient level of automated test coverage" [SBZZ17] or the unsuitability of software types like embedded systems [ClSA15], a forced transition from CDE to CD might not always be a good idea. This topic is discussed in Section 5

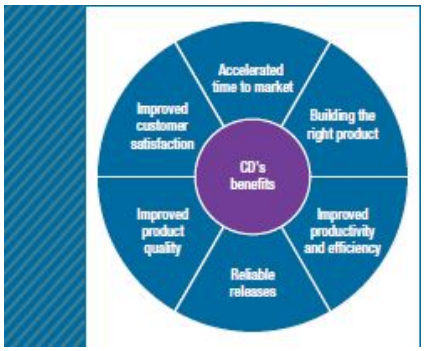

Fig. 2. CD's and CDE's Benefits [Chen15]

3 The evolution path towards Continuous Deployment

This section describes the transition from traditional development to Continuous Deployment and some main practices associated to these phases(Fig. 3). Many software companies are subjected to constant alteration concerning their software development in order to stay competitive. Therefore, they need to improve their way of building software.
It is important to say that the application model plays a big role for the success of the CD adoption. Albeit the traditional 'waterfall model' may be still adequate for some companies offering legacy software (e.g in the financial sector), CD gains more and more attraction for Web-based applications [SCLZ+16]. In the remainder of this section each phase of the "stairway to heaven" is discussed in detail. The barriers each transition causes, are discussed in section V.

3.1 The Stairway to Heaven

The main ideas within this section come from [ShBZ17], [SCLZ+16] and [OlAB12]. They refer to CD as a practice to keep the product "perpetually in a shippable state" using a "fully or at least partially automated deployment pipeline" [SCLZ+16]. This shows the interchangeability of CDE and CD as they are combining CDE and CD in one term. Besides, they put another practice on top of

CD: "Partial Rollouts" (Fig.3). [SCLZ+16] also points out that the adoption of CD does not always has to follow the "stairway to heaven" as many companies already implemented a few continuous practices.

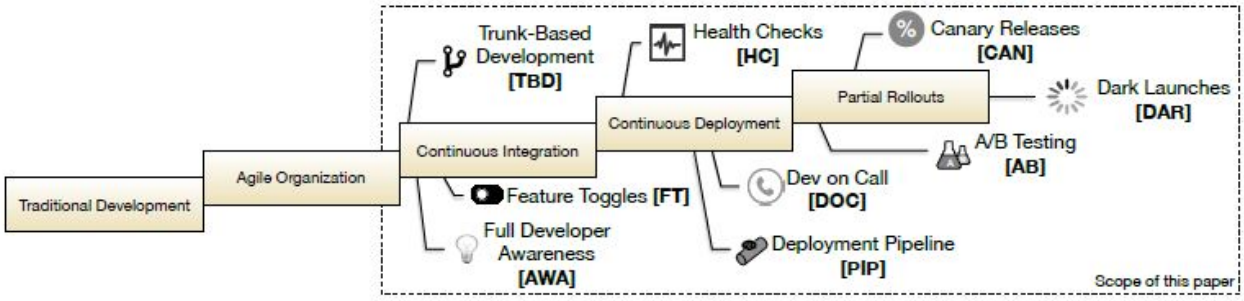

Fig. 3. The "stairway to heaven", and the main practices associated to the phase in this evolution step[SCLZ+16]

3.1.1 Phase 1: Traditional Development Traditional deployment is a software development approach, that is characterized by slow development cycles, the "waterfall model" or a bad customer integration [OlAB12]. The delivery and deployment process is typically at the end of the project. Usually, the developed software is formed as a monolithic 3-Layer architecture with all the included disadvantages. It has to be noticed, that it is not the aim of the seminar paper to judge about the suitability of this kind of software development and architecture. But for successfully implementing Continuous Practices, they are simply not suitable [ClSA15] [ShBZ17] [ShBZ16] [SBZZ17]. The next step in software evolution in agile organization.

3.1.2 Phase 2: Agile Organization A brief introduction in Agile Methodologies is given in 2. Agile Organization, of course, includes agile methodologies, but "product management and system verification still work according to the traditional development model" [OlAB12]. To successfully proceed to agile organization, the most important aspect is mindset. The resistance to change is a challenge for each evolution step. Therefore, managerial support is needed to convince each employee of the initiative. Having a few agile teams promoting their methods within the organization, has a positive effect on the whole company [OlAB12]. Besides, there is a strong need to improve information exchange within and across the teams. As a result, the development teams are organized in small and cross-functional teams. The members of cross-functional teams covers all aspects of software development and delivery [TaKB16]. This enables the organization having feature teams instead of component teams. Feature teams

work on complete features which supports iterative development. As a consequence, this improves the release frequency. It has to be noted, that the development teams do not yet continuously integrate their code to a organization-wide shared code base. So far, they develop a feature within their team and when it passes all required test, they merge it into the main project.
The next step of the "stairway to heaven" is Continuous Integration. The following subsection provides some basic concepts to achieve CI.

3.1.3 Phase 3: Continuous Integration The core of CI is the continuous integration of code into a shared repository across all teams developing a software product. Again, a cultural shift is necessary in order to realize CI practices [OlAB12]. Branching strategies like trunk-based development (TBD), the implementation of feature toggle (FT) or the concept of full developer awareness (AWA) facilitate practising CI. The continuous integration of code by many different teams can result in long-living parallel development branches . To reduce this complexity, [SCLZ+16] proposes trunk-based development, where all teams integrate into one single branch, called master branch or master trunk. What happens to features that are not ready to be deployed, but already integrated into master? The concept of Feature Toggles provides a good solution strategy. In very simple terms, Feature Toggles permit to switch on or off a certain code block. So whenever a feature is ready for production, you just turn it on. Another concept to enable and promote full transparency of the project (e.g build status, test coverage, latest client version or status and progress of features) is called (full) Developer Awareness(AWA). AWA is realized by Dashboards, public monitors in the office or chat tools informing everybody and creating transparency.
Fig. 1 demonstrates, that not only code integration, but also automated builds and test play a huge role in Continuous Practises, especially CI. Adopting "test-driven development" (TDD), "test planning" and "cross-team testing" improves the test phase during CI. TDD is a process in which requirements or user stories are turned into very specific tests. It is divided into 4 different steps: 1. Add a test based on a required feature, 2. Run all test and see which test fails, 3. Write or refactor the code, 4. Run the tests again. Repeat those steps in very short development cycles until all test cases are passing. This helps to continuously check the quality of the written code. "Test planning" is a concept to intensify the collaboration between Quality Assurance (QA) and developers according to the DevOps principles. Together, they make a comprehensive list of automated tests. This "liberates QAs from manually testing the majority of the software application for regression bugs" [ShBZ17]. In "cross-team testing", tests are performed by a group of developers from another team and vice versa.
Besides, defining new roles like a build sheriff is a useful practice to promote CI's success. The build sheriff constantly watches the CI process or indicates that build process failed [ShBZ17].

3.1.4 Phase 4: Continuous Delivery and Deployment As mentioned in the introduction of this section, CD is a practice to keep the product "per-

petually in a shippable state" using a "fully (CD) or at least partially(CDE) automated deployment pipeline" [SCLZ+16]. The core practice of CD, the deployment pipeline, is discussed in section IV. CD requires not just the development and operations team (DevOps), but also the product management as an interface to the customer. [OlAB12]. [ShBZ17] claims that work structures have to be clarified. There is a need of defining new roles and teams when adopting CD, such as a release coordinator or release manager. Fig. 4 demonstrates a potential team structure as it can be found in an software developing organization using CSE for their multiple projects.

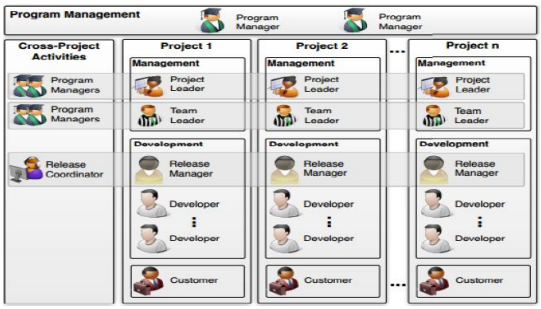

Fig. 4. Possible team structure in a CSE Team [TaKB16]

Further practices to successfully implement CD are "health checks" (HC) and the idea of "developer on call"(DOC). Monitoring on infrastructure (e.g CPU utilization) and application (e.g response time) level is used to extinguish 'health' problems of the deployed software. Those "health checks" can also be used to support rollout decisions (e.g increase traffic for a canary release) [SCLZ+16]. "Developer on call" is associated with the idea of DevOps: Developers shall be available and responsible for their code, all the more after deployment. In case of any error occured after a code change, they know best where to find and how to fix it.

3.1.5 Phase 5: Partial Rollouts [SCLZ+16] describes partial rollout practices as "the epitome" of CD. Releasing software is always the critical part in software engineering, as customers may receive buggy code. So what can be done to mitigate the consequences of deploying buggy software? [ShBZ17] and [SCLZ+16] propose the usage of "canary release", "A/B Testin"(AB) or "Dark Launches"(DAR). Those practices are considered as "run-time quality assurance and requirements validation techniques" [SCLZ+16].
Canary Release is a deployment method where a new software version is deployed only to a "small set of users". That mitigates the influence of possible buggy code and at the same time produces some real usage data. With that, a team decide if either the released version needs further improvement or can be

released completely. A variation of canary release is the "eat your own dog food" approach, where only internal organization members get to use the new version. A/B Testing is similar to canary release. It requires a bigger user base, as there are currently two versions running. By collecting statistics, the team is able to decide whether version A or B performs better.
Dark Launching is the process of deploying functionality to production in order to test it, but with limited or no access for any users. This enables the team to test new functionality in the production environment and consequently decide, whether the feature is ready to "handle production-scale traffic" [SCLZ+16]. To dark launch a feature, the team implements the old and the new feature with Feature Toggles and decides which one to show to the user.

3.2 A comparison between traditional development, agile development and the CD process

The following table compares the development and deployment of the traditional, agile and CD process.

	Traditional development	Agile development	Continuous Practices
Team size	Big Teams, a few dozen	Small teams, 2-8 people	Small teams, 2-8 people
Team competence	Divided in discipline	Cross-functional teams	Cross-functional teams and new roles
Planning	Rigorous planning phase at the beginning	Iterative planning in short cycles	Continuous planning in short cycles
Testing	Explicit testing at the end of the project	Testing at the end of each cycle	(Semi-)automated tests with every pipeline run
Architecture	3-Layer, monolith, strong dependencies between modules	3-Layer or Microservices, less dependencies	Desire: almost no dependencies, independent deployability of modules
Frequency of deployment	Once waterfall process finished (months, year)	Weekly	Daily
Relative risk of deployment	High: no automation, infrequent process, huge amount of code	Still High: no automation but frequent process	Low: automation, small amount of changes
Customer/developer feedback loop	Long due to infrequent deployments	Depends on the deployment frequency	Very short, customers constantly give feedback
Unnecessary features	High amount due to long feedback loop		prevent development of any wasted software

4 Deployment Pipeline and Tools

A Deployment Pipeline is considered to be the core of the CSE process. It enables organizations to automate code integration, build, test, configuration and deployment. Common stages of CD/CDE pipelines are "version control", "build", "continuous integration","artifact repository management", "unit/integration/acceptance/performance testing", "deployment", "configuration and provisioning", "log management and monitoring", "containerization" or "issue tracking" [SBZZ17] [TaKB16]. The vast amount of possible stages demonstrate, that there is no universally applicable pipeline. [Chen15] goes even further: Pipelines should be redesigned for each application to best suit the needs.
Nevertheless, the main idea of a pipeline in CDE is always the same: Moving from stage to stage is (at least semi-)automated. If any error occurs, the pipeline stops and notifies the development team. They are able to fix it and (generally) start the pipeline again by committing the changes.
Thus, the engineering process of a pipeline,the software's architecture, the degree of automation or the available tools generate some serious challenges. They are discussed in Section V.

4.1 Example: A deployment pipeline

The example pipeline is divided in 7 stages: 1. version control system, 2. code management and analysis, 3. build system, 4. continuous integration server, 5. testing, 6. configuration and provisioning, 7. continuous delivery/deployment server

Fig. 5. Deployment Pipeline based on [ShBZ17]

References

BaWZ15. Len Bass, Ingo Weber und Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional. 2015.

Chen15. Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2), 2015, S. 50–54.

ClSA15. Gerry Gerard Claps, Richard Berntsson Svensson und Aybüke Aurum. On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software technology*, Band 57, 2015, S. 21–31.

LLIP+16. Eero Laukkanen, Timo OA Lehtinen, Juha Itkonen, Maria Paasivaara und Casper Lassenius. Bottom-up adoption of continuous delivery in a stage-gate managed software organization. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, S. 45.

OlAB12. Helena Holmström Olsson, Hiva Alahyari und Jan Bosch. Climbing the" Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*. IEEE, 2012, S. 392–399.

SBZZ17. Mojtaba Shahin, Muhammad Ali Babar, Mansooreh Zahedi und Liming Zhu. Beyond Continuous Delivery: An Empirical Investigation of Continuous Deployment Challenges. In *Proc. 11th International Symposium on Empirical Software Engineering and Measurement. New York: ACM Press.[To appear]*, 2017.

SCLZ+16. Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun und Harald Gall. An empirical study on principles and practices of continuous delivery and deployment. Technischer Bericht, PeerJ Preprints, 2016.

ShBZ16. Mojtaba Shahin, Muhammad Ali Babar und Liming Zhu. The Intersection of Continuous Deployment and Architecting Process: Practitioners' Perspectives. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, S. 44.

ShBZ17. Mojtaba Shahin, Muhammad Ali Babar und Liming Zhu. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, Band 5, 2017, S. 3909–3943.

TaKB16. Sajjad Taheritanjani, Stephan Krusche und Bernd Brügge. A Comparison between Commercial and Open Source Reference Implementations for the Rugby Process Model. Technischer Bericht, A University Research Report, 2016.