Everything Continuous Deployment: For Beginners

Keivan Matinzadeh
keivanm@kth.se

May 2020

1 Introduction

One of the first things one sees and hears about when starting to learn about DevOps is the term *CI/CD*, which can stand for continuous integration/continuous deployment, continuous integration/continuous delivery, or both. As one can see, this can result in confusion for someone looking to learn more. If one wants to learn about continuous deployment, is it necessary to know about continuous integration as well? And why is continuous delivery sometimes used interchangeably with continuous deployment? How are they related?

This essay will explore continuous deployment with the above paragraph in mind. The intention is not to explain every software technique connected to continuous deployment in detail, but at least briefly mention the ones that are frequently used as to give the reader an understanding of what continuous deployment is and what is needed for it to work. Another goal is also to provide reasons for why one would want to implement continuous deployment.

2 Continuous Deployment

Traditionally, developers have been treating deployment and release as the same activity. The changes that have been deployed to production[1] have immediately been released and made accessible to users [2]. Software development was organised around releases. A change to the product would be released after writing code and testing it until everything worked. Updating software also used to be

[1]Deploying to production means deploying to the setting where software are put into operation for their intended uses by end users[1].

tedious until it was possible to update through the web which allowed for smaller but more frequent updates. This leads to what is today called *Continuous Deployment*; the aim is to release sets of changes multiple times per day. The releases are now smaller, and features are broken down into smaller blocks of code [3]. Features can be brought to costumers on demand and at will in much shorter cycles with the aim of achieving a continuous flow. Thus, the three major themes that characterise models of continuous deployment according to [4], are deployment, continuity and speed.

"Continuous deployment is a team-oriented approach sharing common goals but having decentralized decision-making." [5]

A big factor in Continuous Deployment, as stated in the quote above, is the decentralised decision-making. The developers are given significant responsibility and accountability when it comes to decisions related to releasing software.

2.1 Continuous Deployment and Continuous Delivery

Many articles on the web have been written about the difference between Continuous Deployment and Continuous Delivery. Thus, a logical conclusion to draw is that it is common for developers and people curious about the subject to confuse the two terms with each other.

According to [6], continuous delivery implies putting the release schedule not in the hands of IT, but in the hands of the business. Any build could potentially be released using a fully automated process which means the software is always production ready throughout its life cycle. But, every good build is not released to users, and the decisions for releasing a build are business related [7]. With continuous deployment, every change that passes the automated tests are deployed to production. *"Continuous deployment is the practice of releasing every good build to users - a more accurate name might have been "continuous release"."* [6].

In [7], continuous deployment is likened to continuous delivery with automated releases. Similarly, in [8], it is said that what differentiate continuous deployment from continuous delivery is a production environment (actual customers) and that the goal of continuous deployment is to automatically deploy every change into said production environment.

Lastly, in [4] the difference between the two methods are explained referring to [6], but it is also stated that much of the literature uses the two terms interchangeably.

2.2 Continuous Integration

Since one usually does not hear about continuous integration without hearing about continuous deployment and vice versa, it is important to briefly explain continuous integration.

Continuous integration is a software development practice where team members integrate their code frequently. Automated builds are used to verify each integration in order to quickly detect errors [9]. In practice, this means continuously merging the code of different developers to the mainline. Automated builds and tests are ran to verify that the changes do not break the program or produce the wrong output. This allows the developers to continuously, even several times a day, add new features to the project. As the new code is continuously integrated, errors and bugs are also detected more quickly.

Continuous integration is adopted by companies when they mature in their use of agile methods and system integration, then migrate to continuous deployment when customers, as a result of adopting continuous integration, express an interest in receiving enhancements and bug fixes more frequently [4].

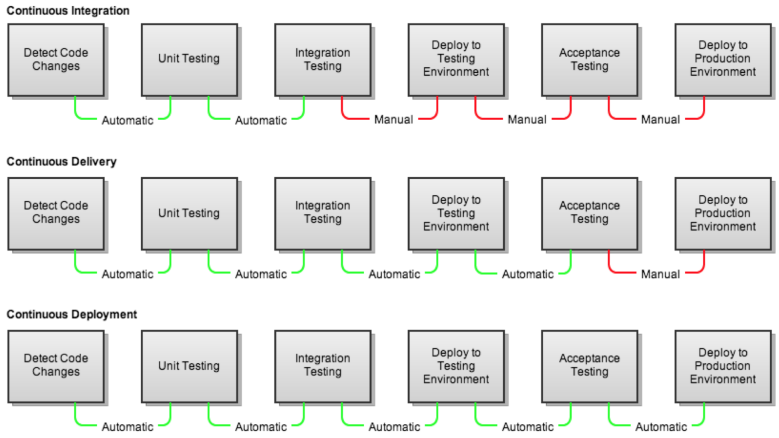

Figure 1: The difference between continuous integration, continuous delivery, and continuous deployment [10].

3 Common Practices

[11] found that all of the software companies they surveyed used at least these three software practices for continuous deployment, namely *automated testing*, *automated deployment* and *code reviewing*.

And similarly, According to [5], a continuous deployment process usually includes four different steps:

1. Testing
2. Code review
3. Release engineering
4. Deployment

This section will give a brief overview of the common practices as to give a rudimentary understanding of different processes involved in continuous deployment.

3.1 Automated Testing

Automated testing is the automation of testing usually done manually, conducted using what is referred to as test tools. Automated testing can be used in many areas of testing, such as automated test execution, automated test scripting and automated defect reporting [12].

Since automated testing is an integral part of continuous integration [9], and continuous integration is a pre-requisite for continuous deployment [4][13][14], successfully implementing automated testing play an important role in being able to successfully implement continuous deployment [8].

3.1.1 Automated Acceptance Testing

This technique is mentioned in [15] in relation to creating a continuous deployment pipeline. It is a complementary testing approach where the customer expresses requirements as input to the software paired with some expected result. The integration is at a higher level, between the business logic and the user interface, compared to unit testing which is a type of low level testing. The basic idea is to document requirements and desired outcomes in a format that can be tested automatically and repeatedly [16].

3.2 Code Reviewing

Code reviewing refers to the manual inspection of source code. In addition to defect related comments, code reviewing can provide team with knowledge transfer, team awareness and improved solutions to problems [17].

[5] found that when implementing continuous deployment, developers are fully responsible for the life cycle of the software thus making code reviewing more prevalent. The increased responsibility results in code reviewing being taken more seriously.

3.3 Release Engineering

There may be a separate release engineering team responsible for compiling, configuring and releasing the source code into production-ready products. If the software contain issues it is rejected and passed back to the developers.

Even though a number of organisations do not use a release engineering team e.g. Etsy and Netflix, [5] found from investigating Facebook and OANDA that having a separate team dedicated to release engineering was valuable.

3.4 Deployment

The name continuous deployment does imply that the deployment process is of high importance, and as stated before, the automatic deployment to production environments is what separates continuous deployment from continuous delivery 1. There is always a risk that the software released to costumers may be buggy. These defects may only get noticed when the software is running in the production environment [10]. Deploying software on a continuous basis can intensify this risk. Different deployment strategies are implemented to reduce these potential risks [8]:

- **Dark Launches.** New features are released during off peak hours, or installed but configured so that users do not see the effects [5].
- **Feature Flags.** The idea behind feature flags is to make part of your application into a series of toggles. Then you can disable features that are causing issues through some admin panel or API call [14].
- **Canary releases.** A software release is first released to a small group of users. The latest build is released to all the users when it has been determined that there are no problems with the new version [10].
- **Blue-green deployments.** Defective changes to the production environment (blue) can be switched to the latest production build (green). In the beginning, exposure is limited - a change may initially only be available to a small percent of the client base. When confidence increases the changes are exposed to more and more of the client base until the fraction reaches 100% [5].

- **A/B Testing.** A controlled form of testing where the user base is split into two groups and for which you expose different variations of a new feature [10].

3.4.1 Roll back

Roll back is a software development technique which "rolls back" the state of the production environment to the state before the latest deployment. Basically, if the state is S1 before deploying to production, and S2 after deploying to production, rolling back the state means going from state S2 back to state S1 [10].

4 Deployment Pipeline - Putting it All Together

The software practices mentioned before are combined and used to deploy fully functional and tested software to the hands of the users. A deployment pipeline is an automated manifestation of this process. This section will present a summarised version of the idea of a deployment pipeline as described by [15]. In its basic form, a deployment pipeline can be described by three different stages:

1. **Commit Stage.** This stage asserts that the system works at a technical level. Every check-in creates a new instance of the deployment pipeline. A new release candidate is created if the first stage passes. The purpose of this stage is to eliminate builds that are unfit for production and signal the team that the application is broken as quickly as possible. The following steps are typically included:
   - Compile the code.
   - Create *binaries*[2] for use by later stages.
   - Perform analysis of the code to check its health.
   - Prepare *artifacts*[3] for use by later stages.
2. **Testing - Automated Acceptance Testing Stage.** This stage assert that the system behaviourally meets the needs of its users and the specifications of the customer. The tests carried out in the commit stage will not catch all errors. Not only does this stage assert that the system delivers what the costumer is expecting, it also serves as a regression test suite, verifying that the new changes does not introduce any new bugs. Passing

[2] A binary is a type of binary file that contains machine code for the computer to execute[18]

[3] An artifact is a by-product produced during the software development process. It may consist of the project source code, dependencies or resource [19]

this stage means that the build is ready to be released. Further extending this stage means adding more tests, some of which are manual tests.

3. **Preparing to Release - Automating Deployment and Release stage.** Releases are always associated with a business risk. If a serious problem is introduced when releasing a new version but a back-out exists, new valuable features may be delayed. If no back-out exists the business may be left without critical resources.

   Mitigating these problems can be done by:

   - Have a release plan created and maintained by everybody involved in delivering the software.
   - Automating as much of the process as possible as to minimise the effects of people making mistakes.
   - Rehearse the procedure in production-like environments so the process can be debugged.
   - Have a strategy for migrating production and configuration data as part of the rollback and upgrade processes.

   In this stage the software is deployed to any environment, be it testing or production. It should be as easy as choosing a version, pressing a button and then release it. Backing out (rolling back) should be just as simple.

The pipeline that has just been described is a basic one. A great deal of information has been left out as to not steer too far away from the focus of this essay.

4.1 Jenkins Pipeline

All these processes and practices do not mean anything if it is not put into real use. There are many tools to use if one wants to implement continuous deployment. Jenkins is one of the most, if not the most popular tool out there.

Jenkins, originally named Hudson, is a popular, free, open source continuous integration server created by Kohsuke Kawaguchi in 2004. Jenkins gives the developer a more robust and faster way to integrate the build, test and deploy. Also, by allowing integration with a multitude of testing and deployment technologies, developers can continuously deliver their software [20].

Though it is possible to do simple chaining of jobs[4] to perform sequential tasks, there exist a feature specifically made for this called *Jenkins Pipeline*. The Jenkins pipeline carries out the process mentioned in 4; it is a user-defined

[4] Jobs refer to runnable tasks that are monitored by Jenkins [21]

model of a continuous delivery/deployment pipeline. The pipeline is defined in what is called a *Jenkinsfile*. This file can be commited to a project's source code repository, thus treating the pipeline as part of the application [22].

Below is an image of the code behind the Declarative Pipeline[5].

Jenkinsfile (Declarative Pipeline)

```
pipeline { (1)
    agent any (2)
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') { (3)
            steps { (4)
                sh 'make' (5)
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml' (6)
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

Figure 2: A simple Declarative Jenkins Pipeline [22].

A stage is a block that defines a subset of tasks. A step is a single task that tells Jenkins what to do at a particular point in time [22].

Since 2017 Jenkins also offers a plugin-in called *Blue Ocean* where one of the features is to present a visualisation of the pipeline [23].

[5] A more recent feature, the Declarative Pipeline provides richer syntactical features and is designed to make writing Pipeline code easier, compared to the other type of pipeline called Scripted Pipeline [22].

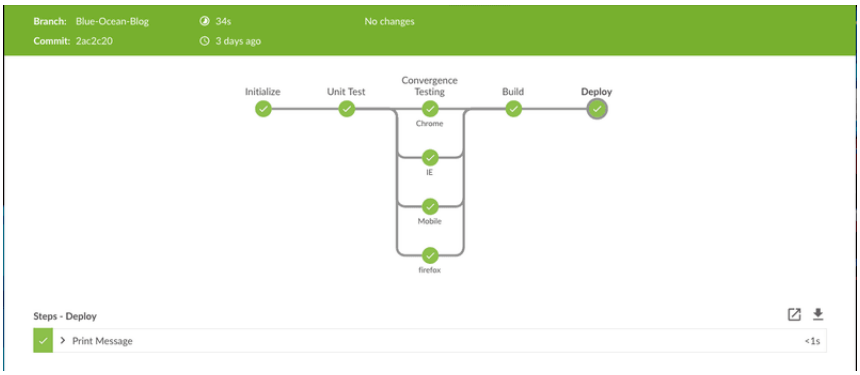

Figure 3: A visualisation of a pipeline with Blue Ocean[23].

5 Benefits

So what are the benefits of implementing continuous deployment? According to [4], the most immediate benefit is a shorter time-to-market through fast and frequent releases. Monthly and weekly long cycles are shortened to daily deliveries. The shorter release cycles let companies to constantly develop and improve based on costumer feedback. The faster feedback about features and bugs makes release planning slightly easier. They also found that these benefits, and in part due to a narrower test focus and extensive use of automation, seem to facilitate rapid innovation through experimentation. It is stated however, that the empirical evidence is limited mainly to the practitioners' perceptions, meaning these findings should be carefully interpreted.

[3] describes gaining similar benefits when adopting continuous deployment.

6 Challenges

Although continuous deployment seems like a good strategy to adopt, it does come with a few challenges.

[5] investigated Facebook and OANDA and found that versatile and skilled developers are required. Developers need to be generalists, good at system debugging, and able to be responsible and take accountability when making deployment decisions. When working on complex systems, being a generalist is especially important. A broader set of skills is required to reason with systems which the developer herself has not developed and have no intimate knowledge of. There is talented developers not well suited to this kind of role and may be better suited in more structured organisations. They also found that features or

products may get deployed too early by developers thinking that what they have deployed can be improved on later through quick iterations. End-users may be alienated by releasing too early, not giving the product a second chance. [5] state that the flexibility that the continuous deployment process offer should not be used carelessly.

[8] found that due to increased bugs, broken plug-in compatibility and frequent update notifications, not all customers were pleased with continuous releases. Also, policies and processes at the costumer organisation may not allow for continuous deployment to be truly implemented. They also found that it was difficult, if not impossible to simulate a production-like test environment, which in some cases may be because of a lack of access to and not a sufficient view of the customer's environment.

7 Discussion and Retrospective

The description of the continuous deployment pipeline is taken entirely from [15] definition. The reasoning was that it is a book that seems to be an industry standard with over 1000 citations, and often cited in studies about continuous practices. It also gave a very thorough explanation which could not be found anywhere else. The motivation for even having that part in the essay was to put together the processes mentioned into something more concrete, which was also the motivation behind mentioning Jenkins and how one can turn this concept into a working process.

Also, the scope may seem unclear. While the essay starts of with mentioning continuous deployment as well as continuous integration and continuous delivery, it is in fact titled "Continuous Deployment: ...". But trying to explain continuous deployment without mentioning continuous integration and delivery does not seem right, since continuous deployment is an extension of those other practices. When choosing studies, a point was made to make sure that the focus was continuous deployment, e.g. the common practices listed were found when the authors investigated companies that had implemented continuous deployment, not only continuous integration or delivery. Also, while continuous deployment is not a particularly tricky concept to explain, an interesting aspect to it are the processes and strategies used to make continuous deployment function correctly.

8 Conclusion

Modern technology have made it possible to break releases down into smaller parts, and at the same time speeding up the frequency of releases. Continuous deployment is a practice which if successfully implemented will help developers continuously deploy features to their projects. If one wants to implement continuous deployment successfully there are a number of strategies pertaining to testing, automation and deployment that is useful to know about.

What is called a continuous deployment pipeline is used to put these processes together in a continuous flow. From committing a project to deploying it. There exist a host of tools to make this idea a reality, such as Jenkins.

It seems beneficial to adopt continuous deployment and if not adopting it wholeheartedly, maybe adopting some practices commonly used, such as a deployment strategy like Blue-green deployments. The challenges that may arise will force a company to change their hiring processes and the infrastructure in their tech departments, but would a company succeed in doing this the reward would outweigh the cost in the long run.

This essay is already long as it is, but one could write 3000 more words on this subject and still feel like there is more to say. The goal was to achieve a middle ground between short superficial articles and long scientific publications, and introduce the reader to a type of subject where it seems others know nothing about or know everything about. Hopefully, whoever reads this essay leaves with an increased sense of clarity about this subject.

References

[1] *Production Environment*. URL: https://www.techopedia.com/definition/8989/production-environment.

[2] *Continuous Deployment*. Sept. 2019. URL: https://www.scaledagileframework.com/continuous-deployment/.

[3] Dan Quine. *Why Continuous Deployment matters to business*. Jan. 2012. URL: https://medium.com/continuous-delivery/why-continuous-deployment-matters-to-business-6a79b5602145.

[4] Pilar Rodríguez et al. "Continuous deployment of software intensive products and services: A systematic mapping study". In: *Journal of Systems and Software* 123 (2017), pp. 263–291. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.12.015. URL: http://www.sciencedirect.com/science/article/pii/S0164121215002812.

[5] Tony Savor et al. "Continuous deployment at Facebook and OANDA". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2016, pp. 21–30. URL: https://ieeexplore.ieee.org/document/7883285/.

[6] Jez Humble. *Continuous Delivery vs Continuous Deployment*. Aug. 2010. URL: https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/.

[7] Sten Pittet. *Continuous integration vs. continuous delivery vs. continuous deployment*. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment/.

[8] M. Shahin, M. Ali Babar, and L. Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943. URL: https://ieeexplore.ieee.org/abstract/document/7884954.

[9] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006. URL: https://martinfowler.com/articles/continuousIntegration.html.

[10] Ville Pulkkinen. "Continuous deployment of software". In: *Proc. of the Seminar*. Vol. 58312107. 2013, pp. 46–52. URL: https://helda.helsinki.fi//bitstream/handle/10138/42910/cbse13_proceedings.pdf?sequence=2#page=49.

[11] A. A. U. Rahman et al. "Synthesizing Continuous Deployment Practices Used in Software Development". In: *2015 Agile Conference*. 2015, pp. 1–10. URL: https://ieeexplore.ieee.org/abstract/document/7284592.

[12] Vahid Garousi and Mika V. Mäntylä. "When and what to automate in software testing? A multi-vocal literature review". In: *Information and Software Technology* 76 (2016), pp. 92–117. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.04.015. URL: http://www.sciencedirect.com/science/article/pii/S0950584916300702.

[13] Gururajan Raghavendran. *Learnings from the journey to continuous deployment*. Oct. 2019. URL: https://engineering.linkedin.com/blog/2019/learnings-from-the-journey-to-continuous-deployment.

[14] *Deploy ALL the Things*. Oct. 2011. URL: http://blog.lusis.org/blog/2011/10/18/deploy-all-the-things/.

[15] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010. URL: http://ptgmedia.pearsoncmg.com/images/9780321601919/samplepages/0321601912.pdf.

[16] Børge Haugset and Geir Kjetil Hanssen. "Automated acceptance testing: A literature review and an industrial case study". In: *Agile 2008 Conference*. IEEE. 2008, pp. 27–38. URL: https://www.researchgate.net/profile/Borge_Haugset/publication/4365340_Automated_Acceptance_Testing_A_Literature_Review_and_an_Industrial_Case_Study/links/02e7e5166aaf1c59a2000000.pdf.

[17] A. Bacchelli and C. Bird. "Expectations, outcomes, and challenges of modern code review". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 712–721. URL: https://ieeexplore.ieee.org/abstract/document/6606617.

[18] *what are binaries?* URL: https://softwareengineering.stackexchange.com/questions/121224/what-are-binaries.

[19] *What is a software artifact?* May 2019. URL: https://jfrog.com/knowledge-base/what-is-a-software-artifact/.

[20] Saurabh. *What is Jenkins? — Jenkins For Continuous Integration*. 2019. URL: https://www.edureka.co/blog/what-is-jenkins/.

[21] *Terminology*. URL: https://wiki.jenkins.io/display/JENKINS/Terminology.

[22] *Pipeline*. URL: https://www.jenkins.io/doc/book/pipeline/.

[23] Justin Bankes. *Blue Ocean Pipeline Automation*. Feb. 2017. URL: https://www.liatrio.com/blog/blue-ocean-pipeline-automation.