Continuous Integration, Continuous Delivery and Continuous Deployment.

What this means, how they relate to each other, and why are they so confused with each other?

The area of DevOps has grown significantly in recent years. The term derives from the combination of software development (Dev) and the operations surrounding information technology (Ops). DevOps has become the new buzzword within the software engineering industry but what is DevOps? DevOps is a collection of software related practices that aims to improve the overall outcome and delivery of a software product [1].

There are many practices within the area of DevOps but in this paper, I will take focus on only a few of them. Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CD) will be the focus of this paper as the terms are closely related and are at times, often confused with one another.

I am someone who has heard of the terms that will be discussed throughout this paper but have never knowingly been exposed to these practices in the workplace. Therefore, I chose to explore this topic by researching and writing this paper about it to inevitably understand what it means to practise these methods within the development of software.

In the following write up, I will explain what each of the three terms mentioned above mean and then will continue to go into detail about how they relate to each other.

CI: Continuous Integration

I will first begin with Continuous Integration, or simply referred to as CI in the software engineering industry. CI is a workflow strategy practiced by software development teams to ensure everyone's code can smoothly integrate together [2]. This is usually done over platforms such as GitHub and Bitbucket where separate developers work on separate branches in a repository. CI is implemented in this context by continually merging into the master branch as often as possible. However, before new code can be merged into the master branch, it must be tested to ensure no bugs or errors are being introduced onto the master [3]. This testing is done automatically, usually with the help of a CI server. This server will test the code and return a paper with the test's results indicating whether or not the code passed the tests. If all tests are passed, the new code will be automatically integrated onto the master branch [4]. If these tests return incorrect values or simply fail, the new code will not be merged and the developer (or someone on the development team) will be notified and the code can be promptly corrected.

As developers are continually pushing their code onto the master branch, there is no final release day madness of everyone pushing and merging all of their branches [3]. Along with this, as the most up to date and working codebase is always on that master branch and it has already all been tested and validated. The product could be released at any time. CI also makes way for smaller software releases meaning that if an issue or bug is discovered, debugging and finding the cause of the error is relatively simple and fast [3].

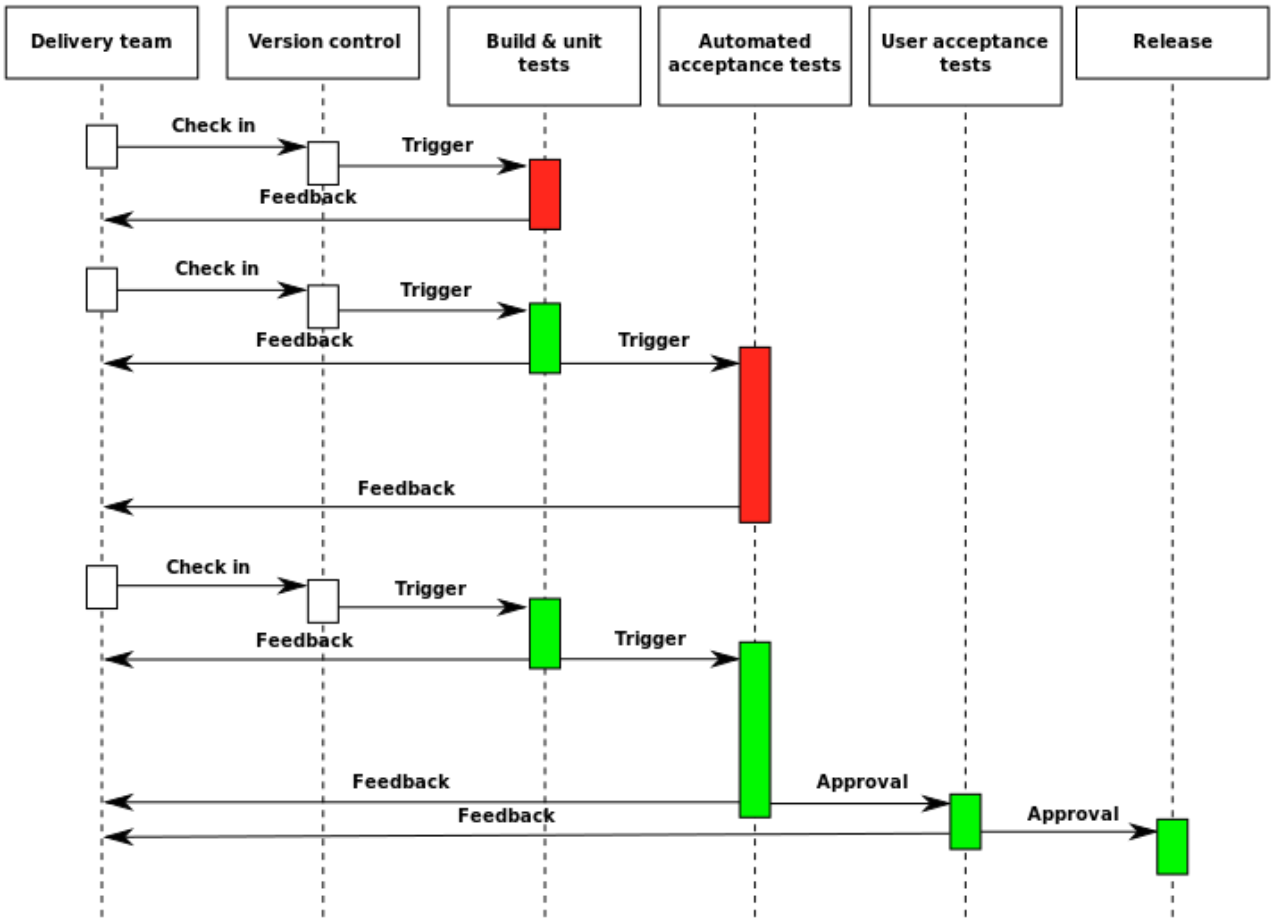
CD

The acronym CD covers two different meanings in DevOps; Continuous Delivery and Continuous Deployment. These both have Continuous Integration as a prerequisite and are each explained in detail now.

CD: Continuous Delivery

Continuous Delivery follows directly on from CI as it goes a step further. Similar to how CI allows for the automation of integration, CD allows for the automation of actually releasing a working, bug free build of the product. It also allows for immediate deployment into the production environment at any time at the click of a button. However, the clicking of that button is not automated yet [3]. That's next.

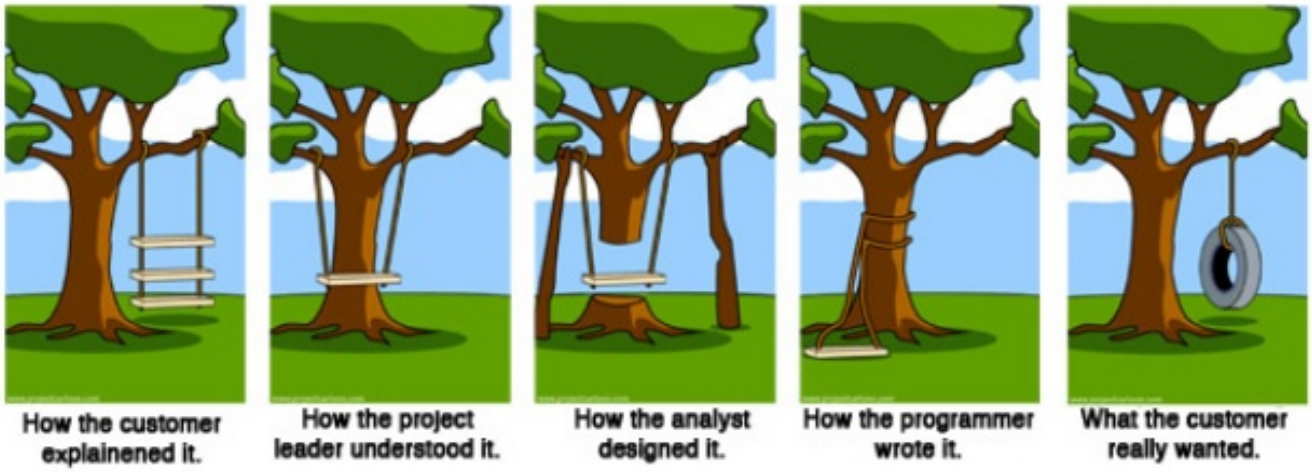
An important aspect of CD is the Pipeline. A Continuous Delivery Pipeline is what all newly developed code passes through before it can be ready for deployment [3, 4]. The Pipeline itself is where all automation, builds and tests are performed [5]. An example of such pipeline is shown in the diagram below.



[6]

Code can pass through this timeline at any time, but it is usually at the request of the business side of the product rather than the development or technical side. Having said this, an interval can be set on the codebase to ensure the code goes through this pipeline at any given interval [3].

An excellent benefit of introducing the practice of CD into a software development lifecycle is making quick and reliable releases to the customer. This has many advantages on its own such as giving the customer a working version of the product they paid for as soon as possible [3]. This allows for a fast feedback loop which will benefit both the development team creating the product as well as the customers who are buying the product. A great diagram that I was first introduced to during my first year of university is this one:



[8]

There are many variations of this diagram that introduce more people throughout a project's lifecycle but this one seemed the most appropriate and relevant to our topic. In short, the customer doesn't know what exactly they want or need from a group of developers. This is where a fast feedback loop can be of assistance. While this diagram may be true for the first iterations of the delivery of a product, the diagram doesn't have to be true for a final deployment of said product.

While the code does go through extensive automated testing procedures, no code is perfect. If an error or bug was missed during the CI or CD processes that was not present in the previous software release, the continuous delivery of the product ensures that each release is small and therefore if errors were found, they would be resolved clean and quickly [3]. This is also true for CI which is mentioned in the previous section.

A final note about introducing CD into a project is that the development style may need to change. As a feature of CD is to release the most recent, working product to the customer, features that are still in development or are just simply not ready to be shown to the customer must be hidden or not accessible from the user side [3, 4]. This is important as the customer wants it to feel like a polished product and showing these incomplete work items will break down that illusion.

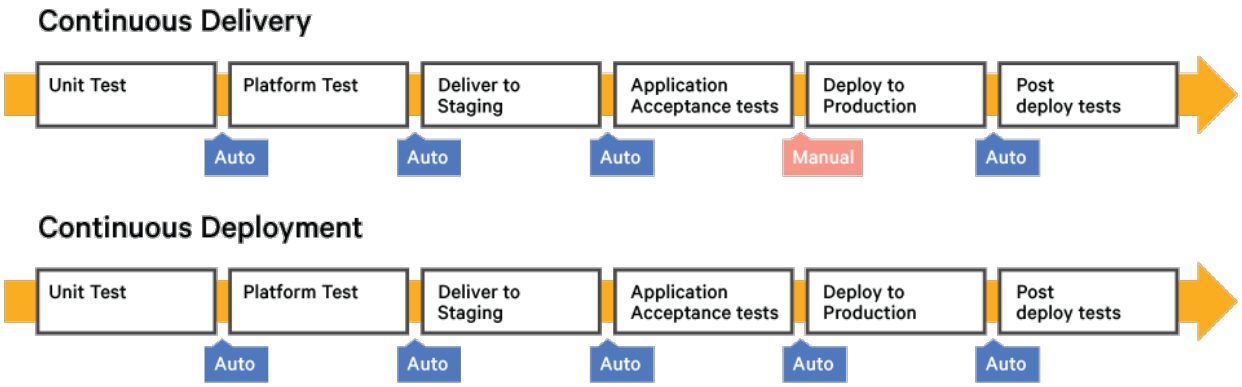
CD: Continuous Deployment

Once again, Continuous Deployment continues on from the previous automation process of continually delivering. That "button" that is mentioned in the previous section is automatically pressed in this definition of CD. Once new code is committed and delivered, it is almost guaranteed that the code is bug and error free. This means there will be little, if any, downtime of said product [2].

Similar with the previous definition of CD we talked about, Continuous Deployment will automatically deploy the software if all tests pass. Zero human interaction is seen from code commit to seeing the product go live [4]. This is great for the development team as they can see their work in a live environment minutes after they have committed that code. It is also another way to further accelerate the feedback loop between development and customers [3].

As mentioned previously, not all code is perfect. If an error does slip through, it will be caught and fixed as soon as possible with little to no downtime [3]. Using Continuous Deployment on a project means automatically creating the by-product of having the actual user as a tester. This user testing can be vital to a product succeeding as even though all the tests passed within the pipeline, it might be awkward for the user to actually use it. The user may also be able to see holes in the product where new features or requirements need to be added [9].

Also, as a final point of this section, if all of the discussion surrounding Continuous Delivery and Continuous Deployment sounded similar, you're not wrong. It is all very similar which is a main reason why people confuse the two terms so often. This following diagram visualises the exact different between the two practices. You can clearly see why people confuse them. It confused me writing and researching this paper, but this diagram really cleared things up.



[10]

How they relate to each other?

As you probably understood from the beginning of this paper, you can't really have one without the other (unless it's Continuous Integration). To have Continuous Delivery you need Continuous Integration and to have Continuous Deployment you need Continuous Delivery. Due to this, you can't simply just choose one and be done with it.

So, after all of this discussion, we now know what all these terms refer to and what they bring to a development lifecycle. But, which one do you choose?

An analysis must be carried out to determine to what extent a project should employ these strategies. Two of the main factors that go into deciding what a project needs are what the project is itself and what the business needs are. As a general rule of thumb, Continuous Delivery is best in almost all cases and then the decision as to whether Continuous Deployment is required is discussed within the development and business teams [7].

As an example, the introduction of Continuous Deployment is great for products that the development team has complete (or a large amount of) ownership over. This could be a web application or some other product that is web-based. The inverse of this would be practicing Continuous Deployment on a product that is required to be installed by an end user. This is just not efficient on the user's end as every update that is deployed would need to be reinstalled by the end user again [4]. Which basically defeats the purpose of Continuous Deployment as end users may not always be using the most up to date version of the product.

And to finish...

In conclusion, throughout this paper I have explored and researched what a few of the most well-known practices within DevOps currently are. To begin this paper, I briefly explained what DevOps meant as a whole and laid down the foundations of why I based my paper on this topic.

To begin the body of my paper, the inner practices of DevOps that are the focus throughout this paper are introduced. Explanations were also given as to what each of the practices are and why they are used. After all of the three practices were laid out on the table, it was time to get to the main event. While I had compared CI and CD when they were first introduced to an extent, the main comparison and subsequent suggestions came in at the end. This final act of the research contained suggestions and examples of when and where you should use certain practices and when you should not use them.

As stated in the introduction to this paper, I chose to write about this to better educate myself on these practices having heard of them but not actually knowing what they were. Thanks to the research I conducted I was able to articulate and explain in a way that spoke to me. I now believe I have a great foundational understanding of all three practices discussed throughout this paper, and that was my goal from the start.

As a final note, especially regarding a paper that talks about technical procedures, there is only a limited amount of information about a procedure that one can learn from simply reading and watching other people talk about CI and CD. I learn by example and I think I speak for many when I say the most valuable lesson one can learn is actually seeing a practice being, well, practiced.

References:

[1] Nolet, T 2018, *Exploring new frontiers in CI/CD and DevOps*, Hackernoon, viewed 4 April 2019, <https://hackernoon.com/exploring-new-frontiers-in-ci-cd-and-devops-420b0f9bde53>

[2] GitHub Training & Guides, *Professional Guides: Continuous Integration Continuous Delivery*, video, YouTube, 6 April 2017, viewed 4 April 2019, <https://www.youtube.com/watch?v=xSv_m3KhUO8>

[3] Pittet, S, *Continuous integration vs. continuous delivery vs. continuous deployment*, Atlassian, viewed 4 April 2019, <https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment>

[4] ThoughtWorks Products, *Continuous Delivery 101 (Part 1),* video, YouTube, 24 March 2017, viewed 4 April 2019, <https://www.youtube.com/watch?v=HnWuIjUw_Q8>

[5] Mukherjee, J, *Continuous delivery pipeline 101*, Atlassian, viewed 6 April 2019, <https://www.atlassian.com/continuous-delivery/pipeline>

[6] 2019, *Continuous Delivery*, Wikipedia, viewed 6 April 2019, <https://en.wikipedia.org/wiki/Continuous_delivery>

[7] IBM Cloud, *Continuous Delivery vs. Continuous Deployment*, video, YouTube, 1 August 2017, viewed 6 April 2019, <https://www.youtube.com/watch?v=hQ0recUXk9o>

[8] Bodenheimer, P, Conrad, B 2014, 'Software Development to Help You End Up with the Product You Really Want', The Idea Village, Slideshare, 25 April 2019, https://www.slideshare.net/IdeaVillage/software-development-33087796

[9] 2017, *Dev Leaders Compare Continuous Delivery vs. Continuous Deployment vs. Continuous Integration*, Stackify, viewed 27 April 2019, <https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/>

[10] Caum, C 2013, *Continuous Delivery Vs. Continuous Deployment: What's the Diff?,* Puppet, viewed 27 April 2019, <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>