Continuous Deployment: A look at Architecture

By:
Paul Batty

Supervisor:
Manfred Lau

June 2017

The dissertation is submitted to
Lancaster University
As partial fulfilment of the requirements for the degree of
Integrated Masters of Science in Computer Science

Name: Paul Batty

Student ID: 33423253

Dissertation Title: Continuous Deployment: A look at Architecture.

Module: SCC.421 Fourth Year Project (Computer Science)

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the Departments use of plagiarism detection systems in order to check the integrity of assessed work. I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date:

Signed:

Abstract

This paper looks into continuous deployment with a focus on the architecture of such a system from the ground up. Starting with where it came from, what it is and the technologies used. Then looking at what an ideal continuous deployment system would be all the way from developer to user. With a quick look at the cost-benefits and who would benefit from such a system. Then taking a look at what is missing, seeing that in general the tools and technology used is on the right track and destined for success. However, the embedded systems and continuous deployment seem to be out of sync therefore more tools need to be created to aid this area into being brought up to date with rest.

Keywords: Continuous deployment, Continuous integration, Automated builds, agile, VCS, workflow, Extreme programming

Introduction

Continuous deployment is no new idea as the field of computer science strives to automated all that it can. However, the idea and implementation of automating the development pipeline, going under the name of continuous integration and continuous deployment is an ever changing field. This has led to the rise of devops, also known as development operations dedicated to maintaining and building such systems with more of a focus on the collaboration within the different teams involved in the pipeline such as quality assurance, development and management.

During a recent project tasked with the aim to create the pipeline from the ground up there were many questions that had to be answered. Such as what is the difference between continuous integration and continuous deployment. How does each part of the system fit into the big picture. What is the end game of the system. This is covered in the first section of this paper.

Besides just understanding the reasoning and terms used behind the names, it was time to design and implement a system for the project. In addition to looking at the project that this idea spawned from, the paper will look at other attempts and experiences from both industry and academic to understand the ideal architecture. This is covered in the second and third chapters.

Now that the architecture has been decided upon, the paper will turn towards who should bother with such a system and the costs involved. Whether it is only for a solo developer or a team. This is covered in the fourth section.

From here the paper will look at how this field is progressing, including the patterns and the direction that this field is heading towards making sure that it is on the right track. This is covered in the final sections five and six.

During the creation of the project there seemed to be no coherent or common meanings behind some of the terms. There also was a lack of architectural designs and implementations with most arguing about the systems cost-benefits.

Therefore, the main aim of this paper is help define and understand the systems and terms used in this field. In addition to looking at the common themes used in practise to provide a guide for the creation of other automated systems, and those looking to understand this area more in depth.

1 Background

The following chapter will cover two sections. Firstly, looking at where the idea for continuous development came from and the how the field ended up where it is today. The second part looks at some of the tools, ideas and concepts that are needed when building or using such a system.

1.1 History of Continuous deployment

Continuous deployment is in a group of methodologies under the name of extreme programming (XP) which in turn is part of the Agile process (Wells D., 1999). The core principle of extreme programming is to be adaptive to change and quick feedback for everyone involved. Developers get feedback on the code, bugs and features. Clients get the features they need and Managers can make decisions about the direction of the project without bringing the whole system down. (Fowler M., 2006)

This movement started in March 1996 by Ken Back (Smith D., 2014) with continuous integration going further back to 1991 by Grady Booch (Vassallo C. (Zampetti F. Romano D.), 2016). The main change between that seen in Booch's design is that Booch placed one integration per day limit, whereas extreme programming favours much more.

The core concept behind Booch's initial design is to avoid problems when a new release is integrated into an old system. It could achieve this goal via automated unit tests. Each test would run through a single public method and make sure that it is performing as it should. For example if a method takes two numbers and return the sum of the numbers. A unit test would test that *1+1* will return *2*, trying edge cases such as using letters and so on. In total there would be a group of tests for every public function. (Berczuk S. (Appleton B.), 2002)

After the developer has made a change to the code base they would run the tests if they all passed then the code was OK to be check in and used in the next release. This was enhanced with the idea of test driven development, where the test are written first then the change.

This all started to kick off around 1997 with the continuous integration being placed inside of the extreme programming movement. This continued until 1999 through various books and publications by the movement, namely Kent Beck. (Vassallo C. (Zampetti F. Romano D.), 2016)

Up to this point continuous integration just consisted of developers writing unit tests and running them locally to make sure that everything passes. When all the tests pass the developer would then check the changes into the version control system (VCS). Other developers then working on the same code base will be able to get the latest code and know that it works.

This started to change around 2001 with the release of CruiseControl (Hanna T., 2016), because in the previous systems if a developer did not run the unit tests forgot or check in some files or had incorrectly formatted their code, it would work fine on their local set-up but nowhere else. Therefore rather than leaving it up to the developer it could be automated. This introduced the idea of build servers.

A build server will sit there and depending on the particular set up and workflow of the project, take the changes run the tests against them and then send out a report to the developer, or anyone who is interested. Now if the developer forgets something it will be caught before anyone else started working on top of the changes.

So far most of the work is performed by developers for developers, in order to assure that the current state of the code base is in an always working condition. This continues until 2008 when Patrick Debois and Andrew Shafer met up and discussed bridging the gap between development, system administrators and other roles within the agile infrastructure. For example the developer environment is different to the test environment which in turn is different to QA and production environments.

This then sparked the next stage in the movement, the creation of devops. This in turn created a whole host of new tools such as Jenkins (Hudson), Puppet and Chef just to name a few. These new tools made continuous integration easier than ever, and as they gained maturity started to see a lot of use in industry. (Rapaport R., 2014)

As these tools started to gain popularity and with the internet being widespread, there was a shift to not only be able to test, but as the code is in an always working condition push out to the user so they can always have the latest version, features and so on. This goes under the name of continuous deployment. This allows bugs to be fixed almost as quickly as they are found due to the reproduction of the users environment back over in the developers workstation.

Today, the transition over to continuous deployment is still being made, with more tools arriving. The idea of serverless severs and tools such as Docker in order to increase the reproducibility of the environments faster and with better accuracy.

In short the automation of the pipeline has been around science 1991 using a variety of tools to get to today. In addition to this the entire pipeline is beneficial to everyone and comes in three different levels, automated builds, continuous integration and continuous deployment. Each being a step up from the one before it.

1.2 Tools

The previous part of this section named various tools that are used in the pipeline, they fall into three large categories, version control systems, testing and infrastructure. This next part of the paper will look at each of these categories in turn to understand where they fit into the big picture and how they are used.

1.2.1 Version control systems

1.2.1.1 What are version control systems

A version control system (VCS) or version control goes under two other names, revision control and source control. A VCS has a simple premise, to manage the changes performed on a document for example, every time a document is edited, the changes are stored alongside the timestamp, user and other metadata. Then if at some point down the line a user wanted to see who made the changes or undo the changes as it broke the system, they should be able to easily.

In short it is a system to manage a documents version, over long periods of time even when the system is closed, restarted or moved. This lays the foundations for more complex operations.

1.2.1.2 Merging and pull requests

As the changes are stored comparisons of different versions can be made, allowing them to be merged together. If *user A* spots an error in a document that the owner has not fixed, they can show the changes to the owner, if the owner agrees and likes the changes they can merge them in. This applies the changes from *user A* into the main document. This is also known as a pull request.

If the owner makes a change to the document they do not need to send a pull request as they own the original, therefore they can merge right away. This

merging is also called checking in, as the person is checking in their changes to the VCS.

The VCS however, does not just record changes to just a single document but everything inside of a folder. Therefore it will track adding new files, deleting files and renaming files. Each VCS handles it slightly differently, this is irrelevant to this paper. The folder in the case of software development will be the project root.

If a VCS is tracking all files, then when checking in changes it would be a waste to do this per file, instead changes are grouped together under a commit. This commit is then checked in one go. Rather than a single file at a time, this allows relevant changes to be grouped together creating a nicer timeline / history for the project.

1.2.1.3 Branching

VCSs have one more main function to cover in respects to this paper, called branching. Similar to that in a tree VCSs will have a main or master branch called the trunk. A branch is similar to a workspace, and represents the changes on that workspace that got it to its current point.

When branching in the VCS it creates a copy of the current branch allowing it to take off in a different direction. Therefore development can go into two different directions without issues of people working on the same file. Then if needed can be merged back. This is better represented graphically as seen below in figure 1.1:

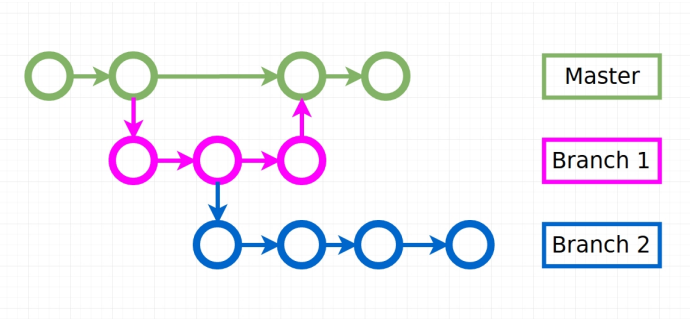

Figure 1.1: VCS Branching

Each circle on the figure represents a commit (a group of changes). The figure also shows that branch one splits off from master, using it as its base, then is merged back in. While branch two uses branch one as its base, however has not yet been

merged. There are different workflows around this feature covered more in depth in section 3.3.

1.2.1.4 VCS Systems

So far the paper has gone on about a folder that exists, this folder goes under the name of a repository. The next part will look at where the repository is located, such that multiple people or a single person can work on the project taking advantage of VCSs, and how they differ. There are two main way that this is achieved, both use a client server architecture.

The first has the server contain the repository, then the developer will create a local copy of the repository according to the branch they are on. They then work on the local copy editing files, however anything else such as creating a branch, merging or checking out files is performed by the server therefore a connection is required.

The second way is distributed and has the developer create a local repository that mimics the server version then everything can be performed locally. When a connection to the server is available they can push their changes onto the server repository. This can be seen figure 1.2:

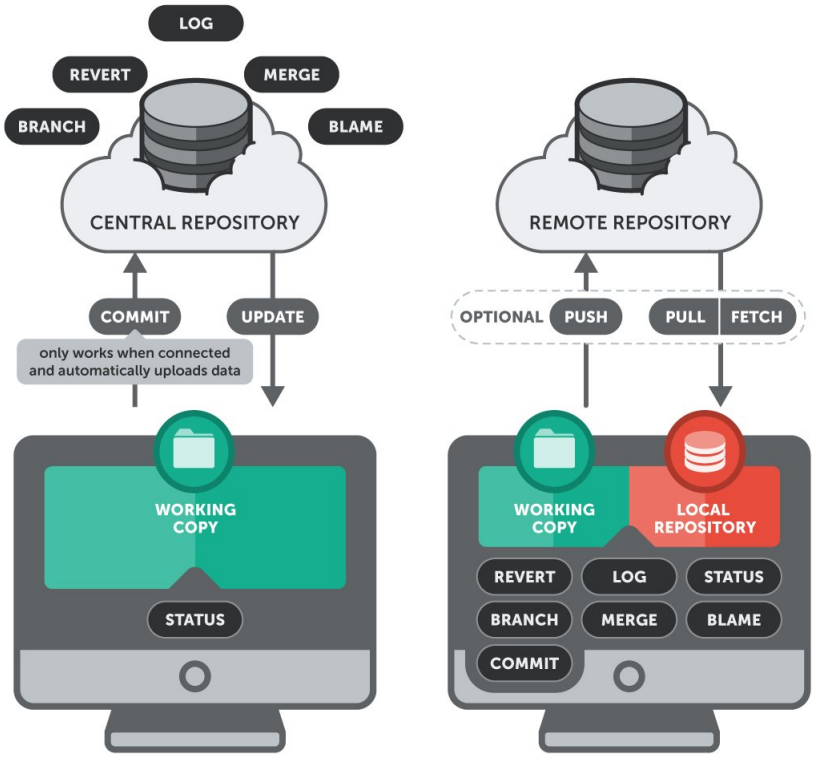

Figure 1.2: VCS systems modified from Fournova Software (2013)

There are a lot VCSs out there, however the most popular are GIT, TFS and Subversion (Stackoverflow (2017)). Git follows the distributed system whereas TFS and Subversion uses the repository server system.

There are a lot more intricate details to VCS however they will not be covered this paper.

1.2.2 Testing

Testing tools refers to the frameworks used to create the mentioned unit tests and other forms of tests. In line with this paper the frameworks will be based around code testing code so it can be automated, rather then that of a manual nature.

1.2.2.1 Unit Testing

Unit testing is generally the first form the tests that are run on a project, as they are fast, lightweight and give good coverage of the system. If the unit tests passed

then the developer should be safe in knowing that the program is in a baseline state of working order.

As mentioned earlier, unit tests, test all public functions to make sure that each one can handle invalid and valid input correctly. The individual tests are then packed into test suites allowing the developer to run only the tests relevant to the change. In addition to helping test organisation as there are normally as many if not more lines of code in tests as the project itself.

The most popular testing framework is the *xUnit* family, where *x* is the first few letters in the language of choice, for example, JUnit for Java, CUnit for C and CPPUnit for C++. The *xUnit* family are upheld as the standard for unit testing.

Before going any further, a quick tangent must be made into build systems. A build system takes the project and produces the final output that can then be placed where required. The build system in a basic set-up is a script that will call commands to build the project. This may involve coping files, deleting, renaming or checking that the environment has what it needs.

Therefore when building the project the build system can build the test suites as well. In a full extreme programming set up these tests will be ran automatically when the project is built. However, this does not prevent the tests from being ran without re-building, built separately or ran at all. This is covered again later in the paper, section 3.2.

1.2.2.2 Integration and Acceptance tests

There are other forms of testing that can be performed on a project, such as usability, regression, and security. However, the main focus will be on integration and acceptance testing, with the rest earning a brief mention later on (section 3.2).

Unit tests, test each method individually to make sure that it is performing up to scratch, however unseen side affect may occur when these parts are put together. For example, given a phone case, a test that could be performed might check that it can hold an object of a certain size, and that it does not fall out or that it will not break went dropped. But when the phone is placed inside the case it is the wrong size. This is integration testing, making sure each part while tested individually works as a whole.

These tests are the next step up from unit test, as in order to function they will need to simulate some of the running functionality, whereas unit tests will avoid

this at all costs. For example integration tests may require the use of a database. Whereas a unit test may fake a database just to see if the correct command is being sent.

Similarly to unit tests and will be a theme throughout, there are frameworks that are deigned to support and run integration tests, with varying levels of sophistication.

Following on from integration tests are acceptance tests. Acceptance tests are performed on a fully set up system, to decide whether the system is acceptable for use by the users. This will include making sure that the system has all of the functionality available, performance and completes each task successfully, from the users perceptive.

Acceptance tests will normally be process through the user interface (UI) that a normal user would go through. For example a web application the user would use their browser. Whereas a command line application the testing would go through the command line.

For web application there are frameworks such as selenium that allow the tests to interact through a mock browser. Others will need to have a browser on the system and available to run rather than mocking. For desktop application TestFX can be used to test JavaFX application and similar for other user interfaces.

Overall integration and acceptance tests are designed to make sure the system can be put together and when done so will work as expected. This makes sure that when the users interact with the system that it works and does not collapse because of a glaring issue such as the phone case being the wrong size.

1.2.3 Infrastructure

So far the paper has looked at where the code and other project files are stored, including some of the main type of tests that are ran on a project. However, continuous deployment aims to automate everything, therefore a pipeline is needed. This is where infrastructure tools step in. They act as the conveyor belt moving the project between the steps.

1.2.3.1 Automation tool

At the start of the pipeline there are standard tools such as Jenkins, Teamcity and Bamboo. Each provide a system to create steps that are then ran in order. To start running the steps the tools will provide several triggers.

Triggers come in four main types, manual which will require someone to press a button to start it. Sometimes this will require typing in arguments that are needed such as the version number.

Time based triggers come in two forms, when a certain time is met such as 6PM, or on a recurring frequency such as every two hours.

The third type ties into the version control system and can trigger on a commit or every five commits or when a commit is sent to a certain branch.

The final type is not so much a way to start the entire process but rather a way to keep it continuous, allowing the pipeline to be triggered whenever another step is completed.

Each step in the pipeline will often be calls to other programs and then storing the logs in a central place, keeping an eye on the entire process even across machines. The support for programs such as testing frameworks and other programming languages will vary from tool to tool. Most modern tools allow several pipeline to be running at once, it does this in the same way that it can track the pipeline across several machines.

Firstly, it provides a central web server install, this is often the part that the user will interact with and set up the system. This will often require the creation of user accounts and access settings.

Alongside the web server they will provide agents, this name will change depending on the tool. An agent is a program that will run the steps. When installed the agent will register itself to the central server allowing it to take on jobs. This allows there to be several agents on different or the same machines so several steps can be ran at once.

Some steps however may require a specific set up for example running a Linux build of the project will not work on Windows. Therefore the central server will factor this into job assignment to make sure that the agent can run the job successfully. This may lead to the situation where the first three steps are ran on one agent then the next three on another due to the different requirements.

1.2.3.2 Automated Deployment

Now that a automation tool is set-up and can be triggered, the end of the pipeline requires deploying the project onto the servers. In order to successfully deploy, the servers in question need to be set up correctly. This will require making sure libraries and programs are installed, correct operating system and configuration files are correct. Depending on the project this may require more or less. In addition to this the project may be located on one or twenty thousand servers. Therefore it will need to be able to set up and manage all the servers. The server set-up may depend on the project version, user or location.

This is where tools such as Chef, Puppet and Octopus come to use. There are a lot more available but, for this next part Chef will be used as an example due to its easier to understand terminology, however, they all work in the same way with different names.

First of all they run in a client server architecture where there is a central server and nodes that report to and take jobs from. As seen in the automation tools. Similarly the server has an interface that will allow management of the software.

Once the central server is installed and the chefs are placed on the servers where the project will be deployed to. The configuration on how to handle the software needed is set-up in ”recipes”. A recipe will be made for each of the programs used and needed by the project. As it is a recipe there can be variation for different versions or other reasons.

Once the recipes are made, they are grouped together in a ”cookbook”. The start of the cookbook will list the programs that need to be installed and in what order. Following the installation will be the recipes. When a deployment is issued the correct cookbook is passed to the chef(s) who then set up the server by running through the cookbook front to back.

This then allow the user to set-up cookbooks and recipes for each configuration needed, and with a single click can start up a new server or deploy over an old one easily.

This section has heavily focuses on the deployment of servers and other web application, for user downloaded programs such as that for desktop and mobile, rather then dealing with all the set-up binaries and other needed files should be packaged and sent to the user. More on this in section 3.5

1.2.3.3 Deployment

The previous section of the paper went over how deployments are achieved but not what it is deploying. This has been wrapped up as software, library or the project. This part aims to clear this up.

When talking about servers there are two parts, firstly, the hardware that the server is made out of and secondly, the part the projects uses. Generally when using a server rather than running directly on the hardware via the OS. A virtual machine (VM) is installed, with several instances. The VM is what people are normally referring to when they say server. This will be the case throughout this paper.

This brings about how to deploy. As a VM with nothing extra, it will act like normal server and allows the users to deploy how they like, this may involve using VM snapshots, VM resetting or re-installing the software. This comes down to how the software works and the standards around that.

However, more recently other tools have come to light such as Docker. Docker replaces the VMs, and so rather than creating a new Linux, windows or other environment instead acts as anther layer of abstraction. Allowing the developers to deploy to Docker, rather then the OS. Then providing the OS has Docker installed the software should run on it without any problems. This also allows the sharing of libraries and other software. In order to understand this the image in figure 1.3 shows the difference:

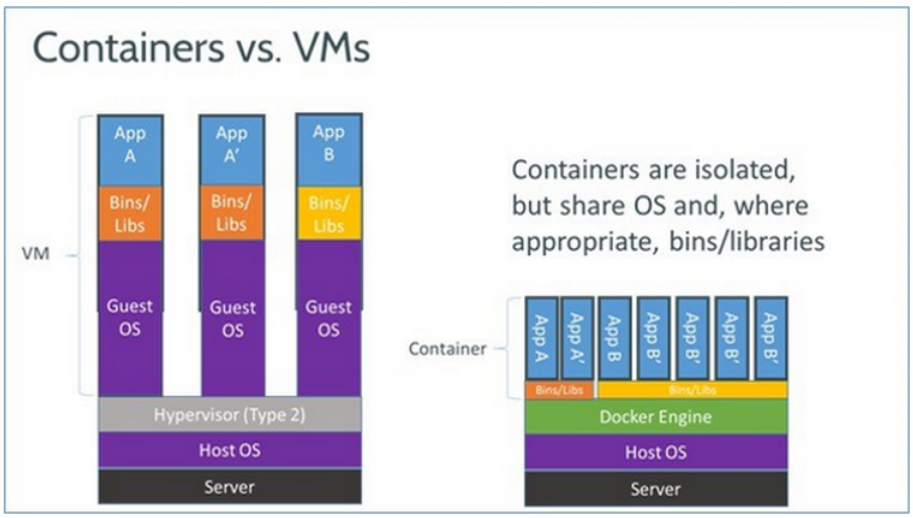

Figure 1.3: Dover vs VMs from Vaughan-Nichols S. (2014)

Not only this, but docker utilises a form of VCS, each container as they are called, is spawned from a Docker image. An image is the final product of the project and the library's that it needs. Theses images can then be placed into the Dockers VCS.

Then when it comes to deploy, just one image needs to be made, no matter the OS pushed into the VCS and pulled onto the servers. This not only eases deployment but also bug reproduction as the image is guaranteed to be the same no matter the machine. Allowing developers to reproduce the productions environment in a matter of minutes.

Another new technology that is emerging is server-less servers. This does not mean there are no servers, or that servers are not servers, servers are still used. However it switches the developer from thinking about servers to tasks.

Rather than building one project instead it is built out of small tasks, each task is then spun up when requested then, and subsequently removed when the task is complete. Meaning it only exists while the task is needed. This can be combined with Docker to spin up an image then remove it afterwards. In this way updates are pushing new images into the VCS.

1.3 Background Final words

This section has given a brief overview of where automated testing and continuous deployment has come from and where it sits at the time of writing this paper. It has gone over the various tools to a certain level, there are certainly more complex and deeper details then that covered here. However for the purposes of this paper it is not needed.

The software named throughout this section are some of the most well known in the industry, but there are certainly more each bringing their own advantages and disadvantages to the table.

From this point on the paper will shift over from what each part is to how it fits together in the best way possible.

2 Study requirements

From the now understood concepts and ideas presented, the paper will now shift onto the collection of data and compacting them into the final results.

Before stating what data is used and how it is collected some limitation will be placed on the project. Firstly, the data will be limited to that of continuous deployment and continuous integration. With regards to continuous integration, the data will be looked at as a continuous deployment point of view, as for the most part the difference is minuscule.

Secondly while the tools used are important the main focus will be on the architecture of the system rather than what they are using, as this is the main focus of the paper.

2.1 Goals

Before any data collection could take place several questions were outlined to form the foundations of this paper and to define a clear goal to head towards, they as follows:

- Is there a clear or ideal architecture that when creating a new or adapting another system should use or head towards.
- Where are the common traps and pitfalls found when creating a such a system, interlinked with goal one what can be changed to avoid them.
- Does the benefits or creating such a system, in both hours and effort pay off in the end.

2.2 Data Collection

The data collection process can be split into three distinct parts, the first one containing blogs and articles written by others who have implemented, used or worked with continuous deployment systems. This type of information will make up the majority of the data collected as it is intend to collect real world experience of such systems in order to find the flaws and how they were corrected.

The second type of of data will be in the form of other research carried out, due to the abjectly questionable nature about the architecture of a perfect system, the majority of papers are based around quantifying the cost-benefits of the systems

or how they work in certain areas rather than how they are put together. Therefore it will be a smaller percentage of the collected data.

The third type of data will be white papers and papers written by companies, and as such will have a bias to them, however, they will be used in conjunction with the rest in order to offset said bias as much as possible.

The rest of the paper from here onwards will be the results gathered from the data collected in order to answer the question above. The following first section will be aimed at the first two goals, with the second part focusing on the final question.

3 Architecture

Now that the paper has covered all the concepts and ideas needed to build a full continuous deployment pipeline, this next section aims to look at the different forms found in use and how they are put together.

3.1 Starting point

So far the paper has mentioned a pipeline, the pipeline will take the developers changes and pass them out to the user in working order. From a purely simplistic point of view the pipeline will look like that seen in figure 3.1.

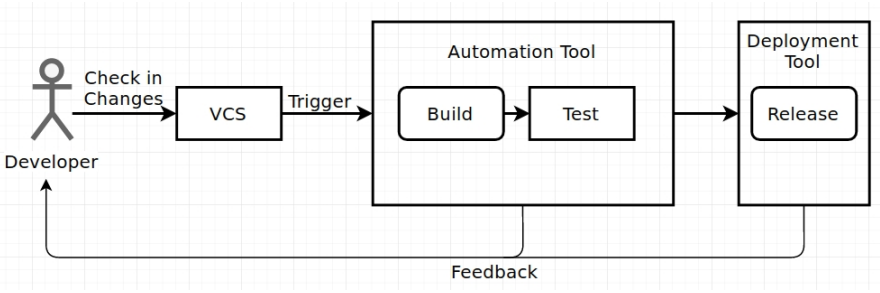

Figure 3.1: The pipeline made simple

The developer will check in the code to the VCS which in turn will trigger the automation tool. The automation tool will then build the project and test, before sending it out to be released, placing it in the users hands. If any of the stages fail the rest of the pipeline is not ran and feedback is sent to the developer so they can fix it, or anyone else interested in the status of the project.

The testing part of the pipeline refers to unit tests and the other forms mentioned in the earlier chapters. Even if the system does successfully pass the feedback will still be sent, this will help guard against false positives.

When talking about the architecture of such a system there are two sides, firstly the software pipeline as seen above. How each of the steps flow into each other and what is needed to pass between each of the steps. The second is the hardware layout, such as are the unit tests ran on the same server that it is built on, or maybe the entire pipeline in confined to a single server.

3.2 Software architecture

Figure 3.1 showed a basic pipeline for a continuous delivery pipeline, however there is not enough details to build a system from this. Below figure 3.2 shows the system developed for the project that sparked this paper:

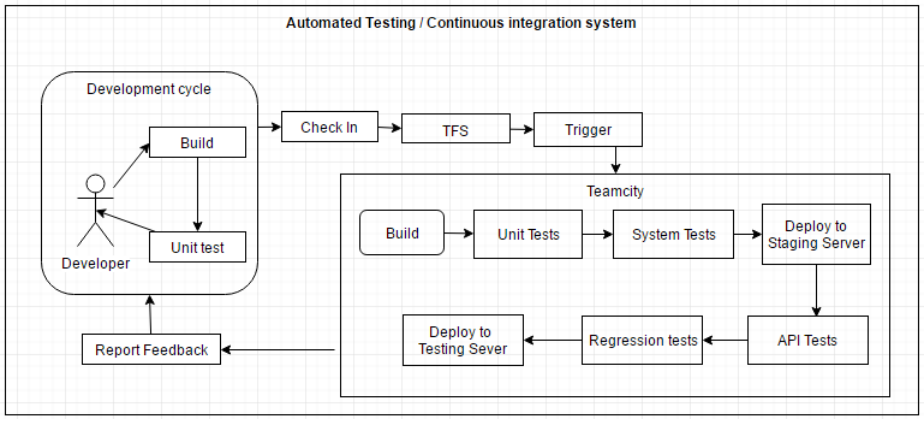

Figure 3.2: A full delivery pipeline

The figure shows two things, firstly it reveals the local development cycle. The cycle being that the developer will check the basics before checking in the code to the VCS in this case a TFS server. Not noted on the diagram the developer will also have access to a local running copy of the program allowing them to run other forms of tests if needed.

The second part, it expands on the build and test block previously seen, firstly, the build, then the unit tests followed by system tests. System tests not mention before are a form of integration tests. This then leads onto an interesting part where the system is deployed onto a staging instance, this allows the API and regression tests to be ran, as they require the system to be up and running. This is then deployed to testing for manual tests, where it can then be pushed to production. In continuous deployment theses steps would be automated.

The interesting part here is how four different deployments are needed, one for the developer, one for staging, one for testing and one for production. This is where Docker and other such tools fit into the picture.

This kind of pipeline is a common one, some may use different types of tests as it will suite their needs better, but the theme is still the same. One such system by Karadzhov S. (2014) adds an additional step after build to package the system

up, ready to deploy, and as mentioned they have swapped out the test types for that which suit their product.

Another interesting take on this by Mansoor U. (2016), has split their project into different components, this means rather than having one project to handle integration tests, both systems have to be built and then integrated together. Therefore they used the architecture in figure 3.3:

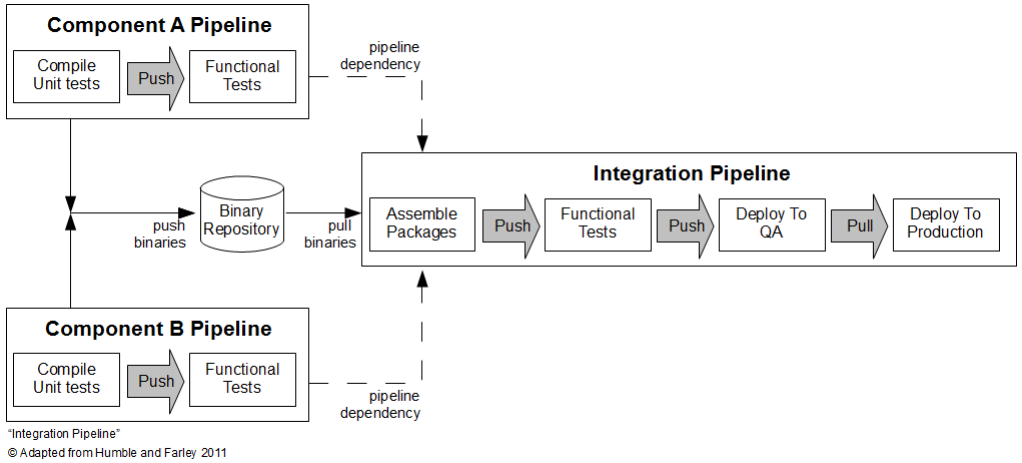

Figure 3.3: A dual channel delivery pipeline by Mansoor U. (2016)

This design has added an additional repository for binaries, then when a commit is placed in component A or B, it can progress through the entire pipeline by getting the latest version of the other components binaries from the repository.

Naik V. (2016) takes this a step further by not only splitting the pipeline into separate components but also places them in different repositories, making the first time they are integrated together being the integration tests as they all come from their own binary repositories.

This kind of pattern goes along with the development environments based on breaking up a monolithic architecture into smaller more manageable or modular units. This system makes it easy to add continuous deployment into a micro service or other types of modular architecture.

One thing not shown so far but some teams find invaluable is to run a static analysis on the code base to pre-emptively spot erroneous areas in the project before they happen. This can then be sent back with the rest of the report to the developers and other interested parties.

This basic pattern and flow of the pipeline does seem to be a common theme throughout all implementations. With the single monolith structure taking it from the start to end and the second combining multiple components into a single software package.

3.3 VCS workflow

As seen the pipeline is generally triggered via a commit into the VCS this makes the workflow with the branch structure and VCS a central component into how the rest of the systems are put together.

There are two main schools of thoughts when working with continuous deployment and VCS the first is more commonly seen in open source projects that are hosted on sites such as Github and Bitbucket, with the second when everyone has full access to the repository.

The first uses a pull request system, all the developers or contributors to the project will submit their changes through a pull request. When the request is made an automated system will start the process of running through the pipeline.

Here the system can automatically merge the change if it passes, however, in open source project it is more common to find the owners performing a code review and checks for harmful changes before merging the request manually. If however it is a private repository the code checked in can be automatically merged.

If the checks fail the request can remain open until an update is pushed to it where it will run the checks again. This is visualised in figure 3.4.

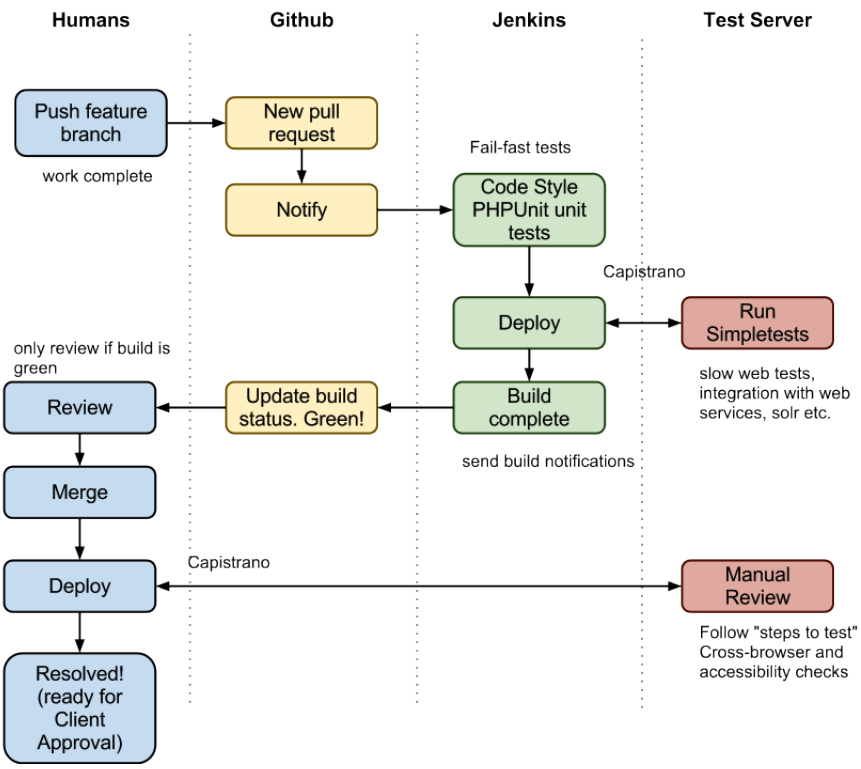

Figure 3.4: Pull request delivery pipeline by Pepper K. (2013)

The only thing to note about the diagram is the manual review as mentioned, this could be automated creating the full continuous delivery pipeline rather than continuous integration.

The second system is similar to the pull request however rather than preforming the change in a pull request the commit is placed inside the teams VCS workflow branch. When working in a agile and continuous delivery environment it is generally accepted that there should be a master branch that is the latest version of the software. This is also the branch where all the development goes and what the customer gets.

Following that system the main branch will have the pipeline attached, when a milestone is reached such as the end of a sprint in agile methodology. A new branch is created to represent the milestone at that point in time.

One reason for this workflow is it will carry well across multiple systems as TFS does not branch in the same way that GIT does. Branching in GIT is similar to changing the entire repository, for example if the current branch is master, then

when changing to *feature_one* branch it is like renaming the folder and swapping its contents. Think of it like a magic room that changes depending on what branch it is on. TFS on the other hand has branches located on a different path, so rather than a single magic room, its will create a new room for each branch.

This difference in design will affect how the automation tool will interact with the VCS, as if it is developed using GIT, the system can watch multiple branches at the same time as it is all located in the same place. Whereas with TFS a new configuration will have to be made for each branch as they are separate from each other.

With this in mind, if the workflow used is to create a new branch per version work on it until release then repeat, at the end of every release a new configuration will have to be made for the new branch. While this does not vary from the other workflow in terms the amount of branching, new configurations will not have to be made as they will also be branched off with the project as seen in the next part. But, as development is performed on a single branch there is nothing to change until the project does.

Before delving into the why a new configuration is not needed, the master branch of the project as defined must always be in working condition ready to be delivered the users. If commits are therefore placed directly in master and they make the build or tests fail, then the entire concept of continuous deployment has failed as master is not in a production ready state.

In order to combat this the preferred workflow is to use a feature branches, similarly to that of a pull request system, development is performed on another branch then when ready can be merged into the master. Similarly this allows tests and other action to be perform on the changes before they are merged into master. This assures that the master branch will be kept in a working condition.

This will also allow the merge to be reverted if there is an issue in the integration without losing the work performed as more commits can be made, tested and merged again.

Now that there is a workflow in place and the pipeline is ready to be attached to the branches, where does the code and other parts needed in the pipeline go. The tools such as Jenkins, Teamcity and CruiseControl all offer the ability to store scripts inside their systems through their user interface.

For example Teamcity comes with pre-scripted runners that will allow a click and

select experience to get the entire system set-up. While this is certainly a good selling point for non-experienced or quick one time set-ups over the long term this could be quite the opposite. In multiple ways, if the team decides they want to change from Teamcity to Jenkins now the entire system has to be built from the ground up. Otherwise it might end up trying to fit the way Teamcity handles things into Jenkins.

Other issues with software versions, setting up new branches and language compatibility, there are many more reasons as to why it is good in the short term on small project but for larger and more longer terms a better system is needed.

Rather than storing the scripts inside of the tool, they should be placed inside of the VCS then depending on the set up, the tool will just have to call a single or multiple external scripts, even if the scripts are a single line. In an ideal situation the scripts should be able to run on the target platforms for the project, such as using python or ruby rather than bash and batch, to save writing them twice.

This also has the benefits of allowing someone to walk up to a brand new machine get the repository and have the full system up and running within minutes as the VCS holds all the scripts needed.

3.4 Hardware architecture

Up to this point the paper has focused on the software side of things looking at the pipeline, VCS and scripts. This next section will look at where this is hosted and how where each part can be split.

Firstly, the set-up will depend on the size of the team and number of builds being performed at once, for example a single server will be fine for small teams of five whereas when dealing with hundreds the servers also need to scale to match.

At the minimum a single server is required, this will hold the VCS, automation tool and staging / testing instance. Finally it will have to include the deployment manager. In general this is a lot of work to place onto a single box and to expect quick feedback timings.

Going back to the start of this paper on how automation tools work, the general principle is the central server and lots of registered agents that can take jobs from the sever. This leads nicely into the following architecture seen in figure 3.5:

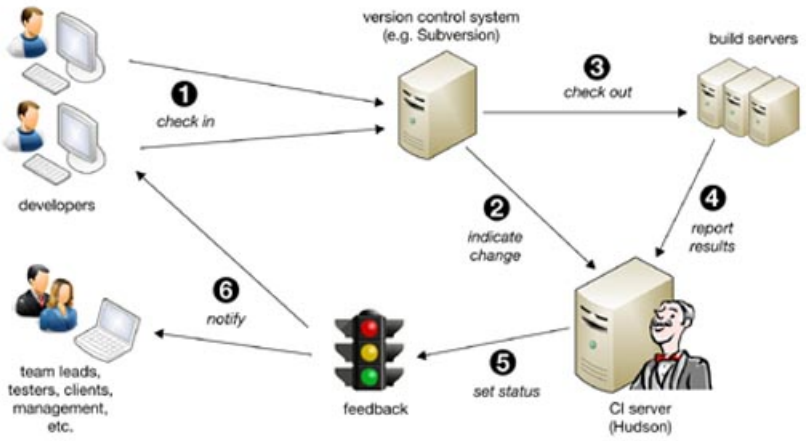

Figure 3.5: Server setup by Atlassian (2017)

The figure shows that rather than having a single sever it is split into three parts, the first containing the VCS, the second the CI / automation tool, and the third a number of machines to perform the builds and tests as necessary.

When the build severs are at max capacity new servers can be spun up registered to the automation tool ready to go. This creates an easily scalable solution that can also be downsized if needed.

The main issues with this set-up are that some of the build servers may become specialised. While there may be twenty build servers one may be used for build in Linux this creates a bottleneck in the system, ideally all servers should be able to accomplish all the tasks, looking towards Docker or similar programs again for the solution. In order to try and reduce dependency on the machines and the programs, creating an environment where the system can run on anything quickly.

For web application this is generally not a problem as the build servers can be set up in way that is appropriate and reflect the production environment, unless the application is designed to work across different servers such as Linux and Windows. This however is not the same for other types of development such as embedded systems.

Engblom J. (2017) has designed a work around, interestingly it uses the same principles as Docker, install once and run everywhere, however rather than dealing with software they have recommended building a software emulator to emulate the target hardware. This emulation can then be ran on any machine allowing the once specialise system to be generalised over multiple servers, removing the

bottleneck. This designed can be seen in figure 3.6:

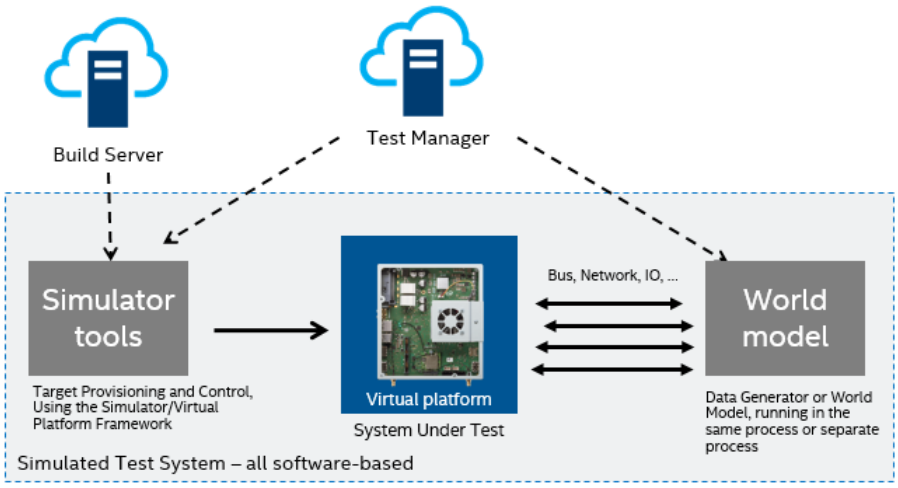

Figure 3.6: Embedded systems from Engblom J. (2017)

While this design works by abstracting the hardware platform into software, there is a large issue with the amount of effort required in order to create this emulation. This may be proprietary hardware or otherwise increasing the difficulty of such a task. There is also the issue with performance as it is now software rather then running directly onto the hardware.

In this part most of the solutions are focused on self-hosted and run solutions, however there is also the option to rent or buy the whole package rather than run them internally. For example, GihHub, BitBucket and GitLab provide a way to host GIT repositories, at a price, they even offer intergeneration with CI services such as Travis.

Cloudbees, BitbucketPipelines and others offer a cloud solution to take the repository and perform the operations on then removing the need to worry about servers and interaction between them. Ultimately the choice of system will come down to the project and budget at hand. With sensitive data, a self-hosted solution will work out better whereas an open source project will fare well with the cloud solutions.

While the cloud solutions remove the need for handling the severs and sever set up themselves, the architecture discussed in this section will still apply as there will still be a need for a VCS server and automation tool server, it is just a matter of not dealing with them directly, but rather through a third party.

3.5 Deployment

Up until this point the paper has covered all the parts within the pipeline apart from deployment, and how to deploy a system from the end of the pipeline. This next part aims to cover this.

3.5.1 Databases

Sometimes no matter what application it will have some form of database in the background, when the application updates, so does the database, changing the schema, tables or data types. In a continuous delivery pipeline this could happen multiple times in a day, whist trying to hold to the zero downtime expectations

Jong M. (Deursen A.) (2015) has designed an application that achieves just that on a small scale, the paper also lists several alternatives that try and accomplish this. This is an ongoing area of research and should be expanded on to get the process to be as smooth as possible without disrupting the running of the application.

3.5.2 Servers

Servers are unique as in general the company creating the product will also host or manage the server, this creates an easy situation where the entire pipeline is an internal process with minimal external needs, such as with the server company like Amazon web services (AWS).

With a standard set-up the new version should be deployed behind the scenes then once done so the live site should now redirect to the newer version rather than the old. This can create an easy fallback system if something goes completely wrong.

This can be done by setting up another Docker container, or however the application is designed to be deployed, it is key however to have this automated. With regards to the database, ideally the database should be handled separately from the rest of the application to minimise errors and when falling back to avoid data loss.

The deployment of servers are the type to be handled by the likes of Octopus, Chef and Puppet. That is not to say that they cannot be used for other type of deployment, more so that is where they are specialised.

3.5.3 Downloadable applications

As far as downloadable application or desktop applications, there are multiple options open, the main idea is to provide an area that can be checked and downloaded

via the software, by bundling the software up. On Linux type operating systems (OS) this is normally checking in the bundled software in to the package control repository, and updated via the system package manager such as apt-get or Pacman.

However, on a Windows like OS, even possible on Linux there are multiple options open to handle the updating of software Fitz T. (2009) covers this, the final choice will come down to the type of software that is being deployed, a web browser can update and install in the background without the users knowledge fine, however more critical software must not, and must be handled with great care.

3.5.4 Mobile applications

Similarly, to Linux like application there is normally a gatekeeper in the way of pushing update directly to the user and instead must go through the ”store”. Therefore the deployment stage will consist of packaging the software up and submitting it to the store.

The ability to automate this process is not so much down to the software but rather the gatekeeper, as they may have manual review and delayed times between submissions, while internally continuous deployment can be used the gatekeeper may make it impossible to do so all the way to the user.

3.6 Architecture final thoughts

Overall, there are some very clear structures and patterns to be found when creating a continuous deployment system. Starting with the pipeline, emerging two main styles depending on the type of application.

To the workflow around the pipeline ensuring that there is a master branch that is always in development and the practises around them. Either through the uses of a pull request or feature branch system.

Then onwards to the hardware and server set with the ideal three tier set up, allowing easy expansion and reduction of servers as needed. Ensuring that each of the servers are generalised as to be able to run any task. Even those on specialised hardware. Including the opportunity to use other hosted services.

Following on to the best practises around the deployment of the system for three different types of applications, web, downloadable and mobile.

While the final project may vary from depending on what part are used, the core

principles, design and workflow found here should be used as the backbone of creating a continuous delivery or continuous integration system.

4 Cost benefits

After looking at the entire pipeline this next section is dedicated to who should use such a system, as in when does it become beneficial to set-up all the overhead required to actually bring benefits.

4.1 Benefits

There have been a few benefits mentioned throughout the paper the main one being the quick feedback loop that can be provided however there are a lot more that have not yet been mentioned.

Firstly, as the pipeline is automated the project can be tested on all the different target systems with ease. This in turn will reduce the number of problems that can or might occur when deploying to a new or existing system. By doing so will allow developers to have a higher level of confidence in that the software will work on any system.

Secondly, it keeps the number of bugs low and the code quality to a set standard, as the entire system is automated the tests that are run ensure that there is a level of confidence in that there are no major bugs in the system. If there are, that they can be fixed quickly and put out to the users. The code quality while not a substitute for code reviews can still be automatically checked, making sure the style of the code is consistence.

Thirdly, with the quick feedback loop, there are benefits for everyone not only developers working on the software. As the automation tool is keeping logs and can send notifications to everyone who wants to keep in the know or gather information about the software can do so easily. Not just the information about the process but easy access to the binaries / packages to make additional deployments or set-ups.

Overall projects with continuous deployment or continuous integration end up with a higher base level of quality and bugs than those without, although it does not fix or prevent bugs from being created it allows developers and managers to see what does and does not and what is needed to make the system work, while preventing old bugs from being introduced to the system. On top of this the feedback is invaluable to everyone involved in the software pipeline, from developers to managers, creating an environment where everyone is more confident in the product as there are logs and process data available to back the claims ups.

4.2 Cost

While the benefits are great there are some costs that come with implementing continuous integration, delivery or other forms of these systems.

Firstly, the initial upfront costs are expensive, getting the severs or to rent a system from a cloud service. This is both time and money spent getting the systems set-up and running in a way that will suit the software package.

In the same vain if there are no tests the cost of writing the tests in the first place is a slow and steady process that will not happen overnight, as they have to be created and integrated into the new system. In addition to this the current tools may have to be adapted the new system, for example changing the VCS from TFS to GIT to provide better services. There are a lot of up font costs involved if the system is not simple or is not designed to allow this type of operation from the start, which excludes the trouble of specialised hardware and having to emulate them.

Secondly, when the system is up and running there is a maintenance cost attached to the tests, if the software updates and changes a core part of the system, so do the tests. There is also the matter of flaky tests that may or may not pass.

The idea of workflow is an important one as there maybe need to change the way developers commit, from every now and then to committing more frequently for smaller changes. These types of costs will heavily depend on the current situation of the project for example a project that already practises test driven development will have a lot of tests already written and so will find it easier to move over to continuous deployment as all they have to do is hook up the tests whereas a project with no tests will have to start from scratch.

Excluding the technical costs and focusing on the personal, users will have to be trained to both understand how, why and what continuous deployment is and how it can benefit them, else they will have a tenancy to shy away from it. This will also require training on the new workflow or hiring addition people to come in and maintain the systems also known as devops as mentioned towards the start of this paper. The main issue is while developers can run this system themselves, continuous delivery can effect everyone involved in the process, this not only means getting developers training but managerial, QA and sales.

4.3 Who

From the lists above it should be clear who can benefit from creating a system, the solo developer or the massive multilevel corporation.

A paper by Hilton M. (Tunnell T. Huang K. Marinov D. Dig D.) (2016) looked at the use of continuous integration in open source software finding that by using it, developers have a highier confidence in the software and release twice as often then those who do not. A similar study by Vasilescu B. (Yu Y. Wang H. Devanbu P. Filkov V.) (2015) showed that there is an increase in productivity and no drop in code quality or otherwise.

While there are plenty of cases where it can and should be used, it is easier to list off the places where continuous deployment may instead be worse. This comes in from three factors firstly the team size, secondly the project size and finally the project type.

Firstly, team size plays a role as the system is a type of managerial tool, as a solo developer there is still reason to use continuous deployment to manage and keep tabs on the status of the project. The system however does come into its own when more people get involved as everyone can be kept up to date on the ongoings of the project. If for instance working in an open source project at the start it may only be a solo developer but when others start contributing the project, the workflow will easily be able to scale up to and support the larger number of developers.

Secondly the project size, a small 50 line script compared to a million plus of lines in a web application come nowhere near close to comparing. There is almost going to be more set up and code in the automation tool for the first project then the second. Not only that but it is very a inefficient use of time and effort as the tools are not designed to work at this small of a scale as found by Svemar J. (2016).

Thirdly, keeping in mind the project size, a small automation script is hardly worth the trouble, similarly, if that script however is used by everyone in the project for everyone in the team, a small bug fix might tip the balance in favour of moving towards continuous delivery to make sure that everyone is using the latest version.

With these ideas in mind one does not have to go from nothing to continuous delivery, there are various middle grounds that might be better used or on the route to implementing continuous delivery into the project. These include automating the build, test automation and continuous integration. In addition to this, when

deciding what type to go for will heavily depend on the current situation of the project, the project type and team size.

4.4 Cost benefits final thoughts

Overall while continuous deployment and its variations bring many benefits the initial uphill climb and ongoing maintenance may not be for every project. Although it would always be a helping hand increasing the base level of confidence in the project.

5 Discussion

5.1 Results

This project started out to find the architecture in and around continuous deployment these where then redefined into three different questions as follows:

- Is there a clear or ideal architecture that when creating a new or adapting another system should use or head towards.
- Where are the common traps and pitfalls found when creating a such a system, interlinked with goal one what can be changed to avoid them.
- Does the benefits or creating such a system, in both hours and effort pay off in the end.

The first two questions are covered through all the chapters in this paper, both continuous deployment and its variations, seeing that they involve a lot of different technologies and process that deviate from some of the standard practises today. However, as explored in the previous two chapters looking at different ideas and designs being put into place there some are emerging patterns that seem to be gravitated towards.

Starting with the stages in the pipeline and how they connect to one and another, starting with the VCS in to the build and then tests. With minor variations for different project and modules using a more modular approach. This system applies to all forms of projects from web applications to embedded systems.

Then looking towards the workflow, there seems to be a set idea on using the feature based workflow with a single main branch where all development is carried out. This is achieved via pull request or branch based processes.

Moving onwards, the hardware and server set-up. The general principles allow the servers to accomplish every task with easy to scale up and down using the three tired architecture.

The final technical section the paper covered what the deployment strategies will depend on what is being deployed and what it is being deployed to, including the general procedures around that. The general idea is to make it work everywhere as quickly as possible with no painful experiences for the end user.

As for the second question, while they are not directly mentioned, they can be assumed from the context in which the user is trying to build a continuous deployment system. Such as trying to build a multi-tier server set-up for a small 50 line script.

Alternatively a system with a lot of flaky tests, or not enough tests of different types may struggle during the integration stages of the project. Most of these can be avoided by sticking to the core principles outlined in the paper. The cost of such a system is in the education and training of users on the new workflow and system to make sure that is used effectively.

For the last question, the paper took a quick look at the cost benefits and who should bother with such a system in the first place. Rather than looking at the places where it should be used identifying three main factors to take into account when deciding team size, project size and project type. For example a solo developer on a small script can survive without whereas a team of five hundred on a million plus lines of code probably cannot.

5.2 Future work

Judging from the current status of the technologies seen throughout this paper, this area seems to be going along the right tracks, making and automating the process as much as possible with quick feedback for all, in turn making the experience easy for users and developers alike.

There is however one area that is lacking, and could be explored further which is embedded and other low level systems, as destroying hardware every build to test can get expensive fast. Similarly creating a simulator for the technology is a whole different project by itself.

Web, desktop and mobile applications are all on the right course with easy to reproduce environments getting easier with Docker and other technologies however embedded and other lower level systems are starting to feel a little on the side lines, therefore this needs more looking into.

Such as how to reproduce the environments without attaching such a huge cost to them. Better integration of the tools into the current systems. In general trying to bring the rich ecosystem surrounding the other development areas in this field.

6 Conclusion

At the start of the paper, several goals where defined. The first to look at and understand the terminology, history and meaning behind the various forms of continuous integration, continuous deployment and automatic build and the surrounding culture of devops. This was achieved in the first chapter of this paper.

The second was to look at examples and other implementations of such system in the real world, taking a look at what went right, what they have in common and deriving the ideal systems from theses studies. This was achieved in the second and third chapters.

Thirdly, to look at the cost benefits and who should use such as system in the first place, while looking at the status of the current situation around Devops and where it is heading, making sure that it is on track. This took on the role looking at what is missing from the toolset and where things have been done right.

Overall, this project has been a success. The aims set out at the start have been achieved. There are a few minor issues that could be improved upon such as using more case studies to create a better foundation on which to draw from.

In the future, more time should be spent towards looking at creating easily reproducible and testing environments for embedded systems as they are non-existent when compared to the status of web, desktop and mobile development. In addition to this looking at auto-updating and integrating database multiple times a day.

But, this has been an enjoyable and successful project and having learnt a great deal about the various technology stacks such as Docker, Vagrant and some of the more unique style of applications such as serverless servers. In addition to a guide of which can be used when needed to implement a variation of such a system

References

Atlassian. (2017), On line publication, Understanding the Bamboo CI Server, https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server -289277285.html, Last Accessed 23rd May 2017

Vasilescu B., Yu Y,. Wang H., Devanbu P., Filkov V. (2015), Quality and Productivity Outcomes Relating to Continuous Integration in GitHub, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, August 30 - September 04, 2015, 805-816

Vassallo C., Zampetti F., Romano D. (2016), Continuous Delivery Practices in a Large Financial Organization,In Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution, Raleigh, NC, USA, October 02 - 07, 2016, 519-528.

Smith D. (2014), On line publication, Extreme Programming (XP), http://projectmanagementhistory.com/Extreme_Programming_(XP).html, Last Accessed 22th May 2017

Wells D. (1999), On line publication, Extreme programming: A gentle introduction, http://www.extremeprogramming.org/, Last Accessed 20th April 2017

Engblom J. (2017), On line publication, Continuous Delivery, Embedded Systems, and Simulation, https://software.intel.com/en-us/blogs/2017/03/13/continuous-delivery-embedde d-systems-and-simulation, Last Accessed 23rd May 2017

Fournova Software. (2013), On line publication, Switching from Subversion to Git, https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/ from-subversion-to-git, Last Accessed 25th April 2017

Fitz T. (2009), On line publication, Continuous Deployment for Downloadable Client Software, http://timothyfitz.com/2009/03/09/cd-for-client-software/, Last Accessed 23rd May 2017

Hanna T. (2016), On line publication, Comparing CI servers: Jenkins vs. CruiseControl vs. Travis, https://jaxenter.com/comparing-vi-servers-jenkins-vs-cruise-control-vs-travis-12

5426.html, Last Accessed 22th May 2017

Svemar J. (2016), Showstoppers for Continuous Delivery in Small Scale Projects, Softhouse Consulting resund AB, Faculty of Engineering (LTH), Lund University, 2016

Pepper K. (2013), On line publication, Automated Drupal Testing with Github Pull Requests, https://www.previousnext.com.au/blog/automated-drupal-testing-github-pull-requests, Last Accessed 9th May 2017

Fowler M. (2006), On line publication, Continuous Integration, https://martinfowler.com/articles/continuousIntegration.html, Last Accessed 22th May 2017

Hilton M., Tunnell T., Huang K., Marinov D., Dig D. (2016), Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, September 03 - 07, 2016, 426-437

Jong M., Deursen A. (2015), Continuous Deployment and Schema Evolution in SQL Databases, Delft University of Technology, Delft, The Netherlands

Rapaport R. (2014), On line publication, A Short History of DevOps, https://www.ca.com/us/rewrite/articles/devops/a-short-history-of-devops.html, Last Accessed 22th May 2017

Stackoverflow. (2017), On line publication, Developer survey results, http://stackoverflow.com/insights/survey/2017#work-version-control, Last Accessed 25th April 2017

Berczuk S., Appleton B. (2002), Software Configuration Management Patterns: Effective Teamwork, Practical Integration, Boston, ISBN 0201741172.

Karadzhov S. (2014), On line publication, Fundamentals of Continuous Integration with Jenkins and zend server, http://static.zend.com/topics/WP-Fundamentals-of-Continuous-Integration-with-Jenkins-and-Zend-Server-2014-03-31-EN.pdf, Last Accessed 4st May 2017

Vaughan-Nichols S. (2014), On line publication, What is Docker and why is it so darn popular?,

http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/, Last Accessed 1st May 2017

Mansoor U. (2016), On line publication, Continuous Delivery - Automating the Release Process, https://codeahoy.com/2016/06/18/continuous-delivery-automating-the-release-process/, Last Accessed 5th May 2017

Naik V., (2016), On line publication, Architecting for Continuous Delivery, https://www.thoughtworks.com/insights/blog/architecting-continuous-delivery, Last Accessed 5th May 2017