Continuous Deployment: A look at Architecture

By:

Paul Batty

Supervisor:

Manfred Lau

March 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

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 Schools 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:

To get the code and other documents visit the website at:
http://www.lancaster.ac.uk/ug/battyp/autotesting

Abstract

This paper...

Keywords:

Introduction

Continuous deployment in 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 a ever changing field. This has lead 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 of the different team 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 question 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 experience from both industry and academic to understand the ideal architecture. This is covered in the second and third chapters.

From here the paper will look at the how this architecture will work in practise, 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 four and five.

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, going over the terms used. The second part looking at some of the tools used.

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 principles 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. TODO:cite

This movement started in March 1996 by Ken Back TODO:cite with continuous integration going further back to 1991 by Grady Booch TODO:cite. The main change between that of Booch's design and extreme programming, is that Booch placed a one integration a day limit, whereas extreme programming favours much more. TODO:cite

The core idea behind Booch's idea 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.

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 place inside of the extreme programming movement. This continued until 1999 through various books and publications by the movement, namely Kent Beck.

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 test pass the developer would then check in the changes in to the version control system (VCS). Other developer 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, because in the previous system what if a developer did not run the unit tests, or forgot or check in some files, so it would work fine on their local set-up but nowhere else. Therefore rather then leaving it up to the developer it could be automated. This introduced the idea of build servers.

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

So far most of the work was performed by developers for developers, in order to assure that the current state of the code base was in a always working condition. This continues until 2008 when Patrick Debois and Andrew Shafer meet up and discuss 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 witch 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 then ever, and as they gained maturity started to see a lot of use in industry.

As these tools started to gain popularity and with the internet being widespread, there was a shift to not only able to test, but as as the code is in a always working condition push out to the customers 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 customers 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 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

In the previous part this paper named various tools that are used in the pipeline, they fall into three large categories, version control systems, testing and infrastructure. In this next part the paper will look at each of the 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 want 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 in 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 sees 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 there 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, however this is irrelevant to this paper.The folder will the in case of software development 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 history for the project.

1.2.1.3 Branching

VCS has one more main function to cover in respects to this paper, called branching. Similar to that in a tree VCS 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 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:
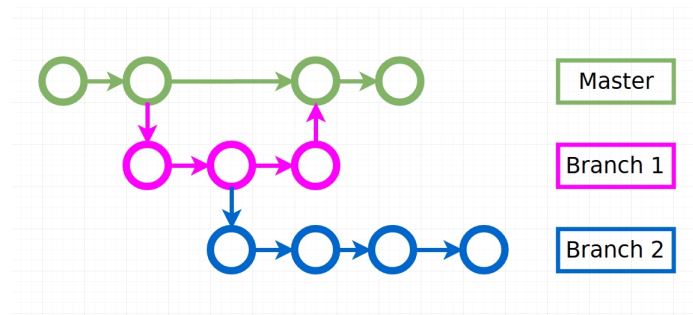

Figure 1.1: VCS Branching

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 merged back in. While branch two uses branch one as its base, however has not yet been merged. There are different work-flows around this feature covered in more depth in section TODO:link the section.

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 VCS, and how they differ. There are two main way that this is achieved, both use a client sever 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 mimic the server one then everything can be performed locally. When a connection to the server is available they can push their changes on the the server repository. This can be seen figure 1.2:
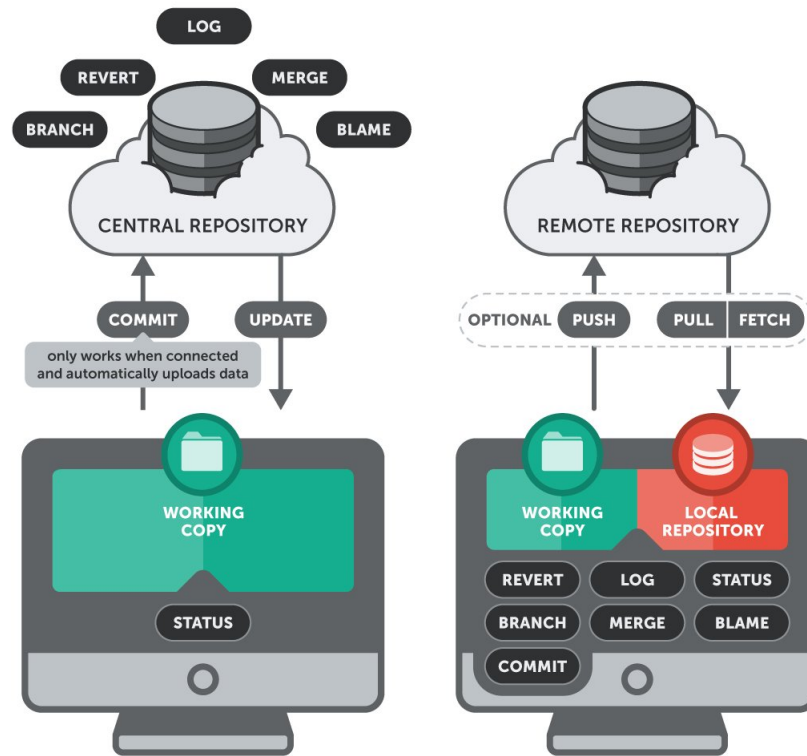
Figure 1.2: VCS systems modified from Fournova Software (2013)

There are a lot VCS out there, however the most popular is 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 reefers 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 save 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 organisational as there are normally as many if not more lines of code in test 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 and so on.

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. TODO: Chapter location

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. TODO: Chapter location

Unit tests tests 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, if you have a phone case, you test that it can hold an abject of a certain size, and that it does not fall out, it will not break went dropped and more. But when you place the phone inside the case its the wrong size. This is integration testing, making sure the the each part while tested individually works as a whole.

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

all costs. For example an 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 theme throughout there are frameworks that are deigned to support and run integration tests, with varying levels of sophistication.

Following on from integration tests there are acceptance tests. Acceptance tests are performed on a fully set up system, to decide that the system is acceptable for use by the customers. 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 for a command line application the testing would go trough 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 the 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 so on.

Overall integration and acceptance tests are designed to make sure the system can be put together and when done so will work as expected. The 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 the where the code and other project files are stored, some of the main type of tests that are ran on a project. However, continuous deployment aims automate everything, therefore a pipeline is needed. This is where infrastructure tools step in. They act the the conveyor belt moving the project between the steps.

1.2.3.1 Automation tool

At the start of the pipeline and the standard are tools such as Jenkins, Teamcity and Bamboo. Each provide a system to create steps that are then ran in order. 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 trigger, 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 continuous, allow the pipeline to be triggered whenever 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 and other programming language will vary from toll to tool. Most modern tools allow sever 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 window. 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 that 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 require deploying the project onto the servers. In order to successfully deploy

the server in question need to be set up correctly. This will require making sure libraries and programs are installed, correct operating systems and configure 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 server. Therefore it need to be able to set up all the server correctly. The server set-up as well 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 also 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 allow management of the software.

Once the central server is installed and the chefs are placed on the servers that 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 can set up the server by running through the cookbook.

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.

1.2.3.3 Deployment

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

When talking about server there are two parts, the first the hardware that the server is made out of and the part the projects uses. Generally when using a server rather than running directly on the hard via the first 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 bring about how to deploy. As it is 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 done 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 or windows or other environment instead acts as anther layer of abstraction. Allowing 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:
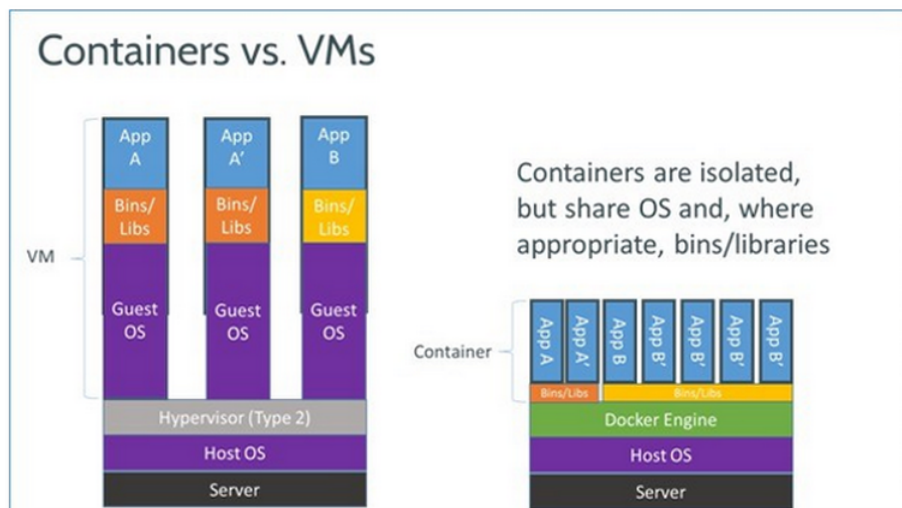

Figure 1.3: Dover vs VMs from Vaughan-Nichols S. (2014)

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

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

Rather than building one project instead its build out of small tasks, each task is then spun up when requested then, 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 at 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 cover 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 is the best way possible.

2 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 form found in use and how they are put together.

2.1 Starting point

So far the paper has mentioned about a pipeline, the pipeline being taking the developers changes and getting then out to the customer in a working order. From a purely simplistic point of view the pipeline will look like the that seen in figure 2.1.
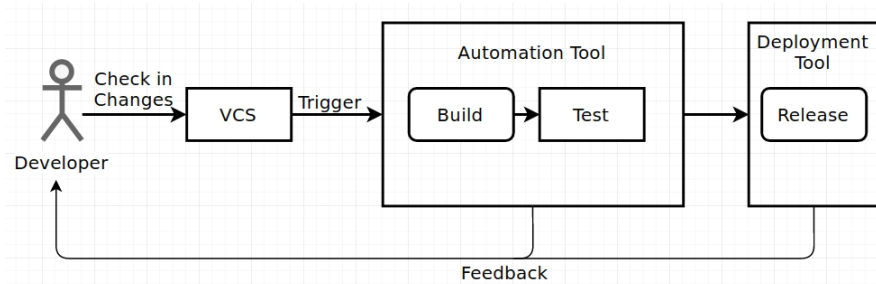

Figure 2.1: The pipeline made simple

The developer will check in the code to the VCS witch in turn will trigger the automation tool. The automation tool will then build the project and test, before sending it out to release. 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.

The testing part of the pipeline refers to unit tests and the other forms mentioned in the earlier chapters. Even on a full pass the feedback will be sent, this will help guard against false positives.

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

2.2 Software architecture

Figure 2.1 showed a basic pipeline for a continuous delivery pipeline, however there is not enough details to build a system from this. Below figure 2.2 shows the system developed from the project that sparked this paper:
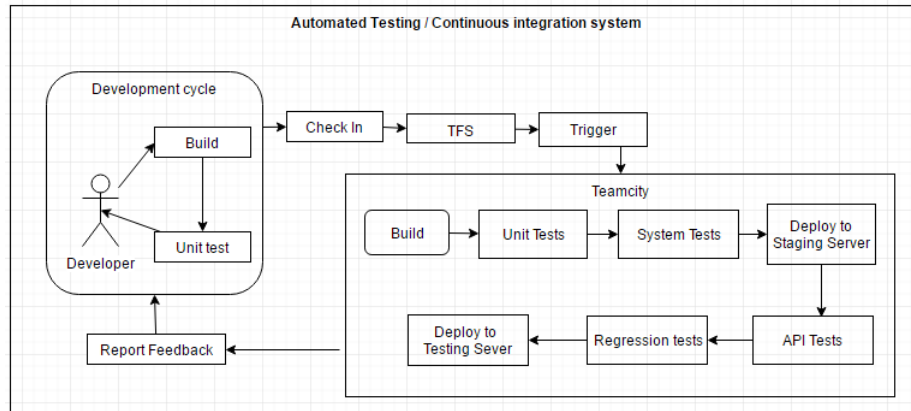

Figure 2.2: A full delivery pipeline

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 not mention before are tests a form of integration tests. This then leads on to a interesting part where the system is deployed on a staging instance, this allows the API and regression tests to be ran, as they require a the full system up and running. This is then deployed to testing for manual tests, where it can then be pushed to production.

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 system to to be build and then integrated together. Therefore they used the architecture in figure 2.3:
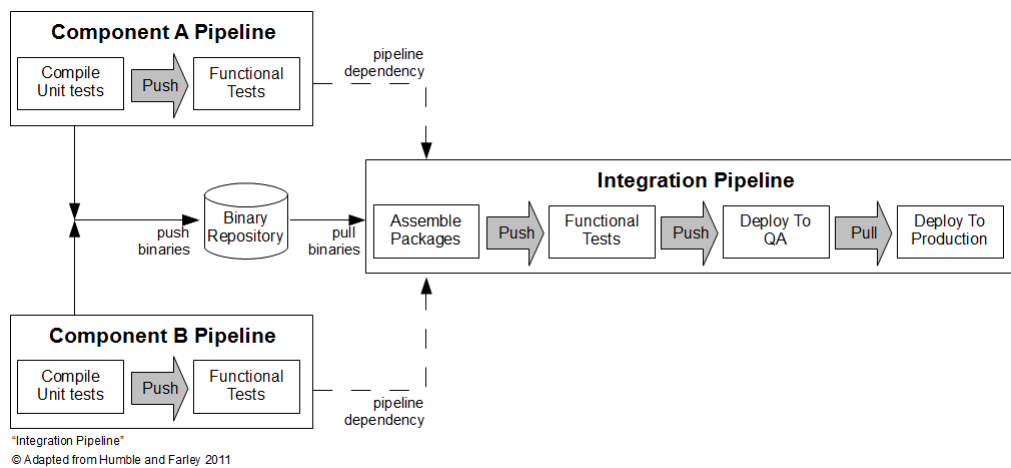

Figure 2.3: A dual channel delivery pipeline by Mansoor U. (2016)

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 monolith architecture of a project into smaller more manageable units. This system makes it easy to add continuous deployment into micro service an other types of modular architecture.

One thing not show 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 seems to be a common theme throughout all over implementations. With the single monolith structure taking it from the start to end and the second combining multiple components into a single software package.

2.3   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 fist uses a pull request system, all the developers or contributes to the project will submit their changes trough 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 to perform a code review and check 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 util an update is pushed to it where it will run the checks again. This is visualised in figure 2.4.
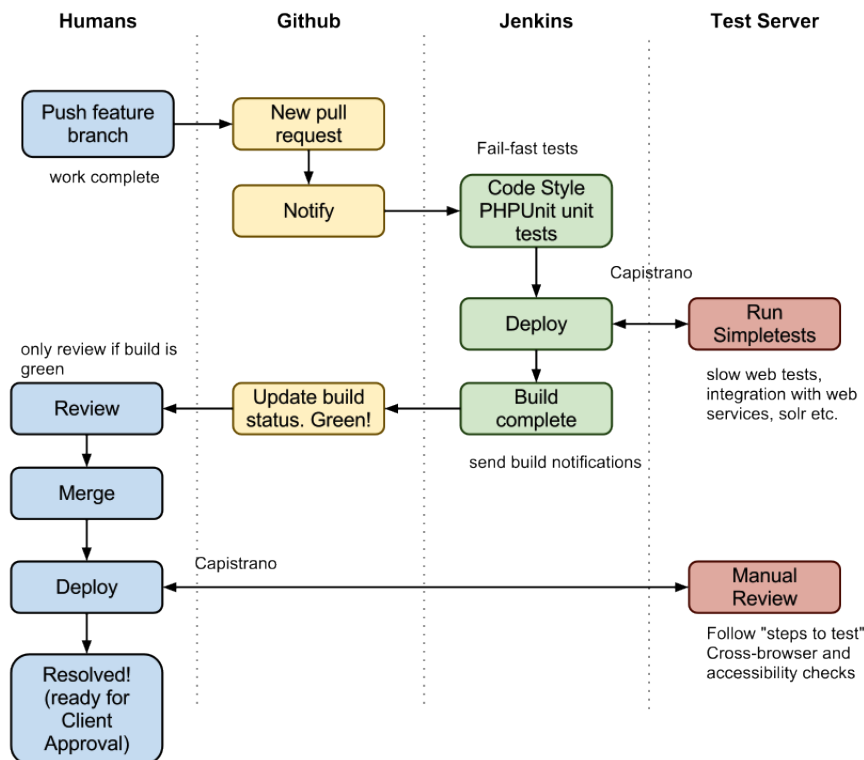
Figure 2.4: Pull request delivery pipeline by Pepper K. (2013)

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 general 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 its on. TFS on the other hand is 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 of branching, new configuration 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 the a new configuration is not needed, the master branch of the project as defied must always be in working condition ready to be delivered the users. If commit are therefore placed directly in master and they make the build or tests fail, then the entire concept of continuous deployment is failed as master is not in a production ready state.

In order to combat this the preferred workflow is to use a feature branches, similar to that of a pull request system, development is performed on another branch then when ready can be merged into the master. Similar this allows tests and other action to be perform on the changes fore they are merged into master. This assures that the master 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 commit can be made and tried to merge 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 again. Otherwise it might end up try to fit the way Teamcity handles things into Jenkins.

Other issues with software version setting up new branches and so on, 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, just have to call a single or multiple external scripts from the tool 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 file, to save writing them twice.

This also has the befts of 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.

2.4   Hardware architecture

Up to this point the paper has focused on the software side of thing 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.

2.5   Architecture final thoughts

3 Evaluation

eval

4 Conclusion

conc

References

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

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

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

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

5 Appendix