UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Automation of the Continuous Integration (CI) – Continuous Delivery/Deployment (CD) Software Development

A Degree Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Jordi Martí Janda

In partial fulfilment
of the requirements for the degree in
TECHNOLOGIES AND TELECOMUNICATION SERVICES ENGINEERING

Advisor: Jordi Perelló Muntan

Barcelona, October 2019

Abstract

In traditional software development, where teams of developers worked on the same project in isolation, often led to problems integrating the resulting code.
Due to this isolation, the project was not deliverable until the integration of all its parts, which was tedious and generated errors. The Continuous Integration (CI) emerged as a practice to solve the problems of traditional methodology, with the aim of improving the quality of the code. This thesis sets out what is it and how Continuous Integration is achieved, the principles that makes it as effective as possible and the processes that follow as a consequence, to thus introduce the context of its objective: the creation of a system that automates the start-up and set-up of an environment to be able to apply the methodology of continuous integration.

Resum

El desenvolupament de software tradicional, on els equips de desenvolupadors treballaven sobre el mateix projecte de forma aïllada, sovint comportava problemes a l'hora d'integrar el codi. Degut a aquest aïllament, el projecte no era presentable fins al moment de la integració de totes les parts, la qual era costosa i generava errors. La integració contínua (CI) va sorgir com una pràctica per a resoldre els problemes de la metodologia tradicional amb l'objectiu de millorar la qualitat del codi. Aquest treball de final de grau exposa que és i com s'aconsegueix la integració contínua, els principis que fan que aquesta sigui el més efectiva possible i els processos que se'n desprenen com a conseqüència, per a introduir així el context del seu objectiu: la creació d'un sistema que automatitzi la posada en marxa i configuració d'un entorn per a poder aplicar la metodologia d'integració contínua.

Resumen

El desarrollo de software tradicional, donde los equipos de desarrolladores trabajaban sobre el mismo proyecto de forma aislada, comportaba a menudo problemas al integrar el código resultante. Debido a este aislamiento, el proyecto no era presentable hasta la integración de todas sus partes, que era costosa y generaba errores. La integración continua (CI) surgió como una práctica para resolver los problemas de la metodología tradicional, con el objetivo de mejorar la calidad del código. Este proyecto de final de grado expone qué es y cómo se consigue la integración continua, los principios que la hacen lo más efectiva posible y los procesos que se desprenden como consecuencia, para introducir así el contexto de su objetivo: la creación de un sistema que automatice la puesta en marcha y configuración de un entorno para poder aplicar la metodología de integración continua.

Acknowledgements

I would like to express my honest gratitude to Jordi Perelló Muntán to accept the tutoring of my thesis being always willing to help and advice to achieve the best result for the project.

I am also gratefully to the people of Apiumhub for introducing me the subject of this thesis and being always disposed to give feedback about it.

Revision history and approval record

Revision	Date	Purpose
0	27/09/2019	Document creation
1	02/10/2019	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Jordi Martí Janda	jordimartijanda@gmail.com
Jordi Perelló Muntan	perello@ac.upc.edu

Written by: Jordi Martí Janda		Reviewed and approved by: Jordi Perelló Muntan	
Date	20/09/2019	Date	02/10/2019
Name	Jordi Martí Janda	Name	Jordi Perelló Muntan
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

1. Introduction

In Traditional software development, it was common for developers to work isolated for a long time, merging the resulting code after that. This practice had a bad consequence: it could take days or even weeks for software developers to integrate their code and also merge changes from the different version of each developer. These isolated developments produced, in addition to merge conflicts, code strategy divergence and duplicated effort. As a result, it was difficult to provide code updates quickly. The feedback about the developed changes was not immediate but after the code merge, being more likely to find bugs due to the code isolation. It is worth to mention that isolated code bugs combined together, could create more code problems.

1.1. Continuous Integration / Continuous Delivery (CI/CD)

According to Wikipedia [1], Continuous Integration (CI) in software engineering is *the practice of merging all developer's working copies to a shared mainline several times a day*. With this definition it can be seen that CI entails a cultural component, as developers have to learn to integrate code periodically. The main goal of CI is to reduce the time to feedback over the software integration process, allowing to locating and fixing bugs more easily and quickly, thus enhancing its quality while reducing the time to validate and publish new software updates. This means that, by following the CI culture, a team of developers can avoid the traditional problems of the merge and integration of the isolated parts of the code, getting a stable version of the developing software always ready. CI, on the other hand, entails an automation component. From the culture of CI, as a consequence, principles, practices and processes that automate and carry out the methodology are born.

Martin Fowler, a British software developer, author and international public speaker on software development, specialised in object-oriented analysis and design, patterns and agile software development methodologies including extreme programming, wrote in his blog [2] an article about the key principles that make CI effective. Those principles are:

- Maintain a Single Source Repository: when multiple people work on the same project, they should work on a shared single repository. It is important to keep track of all the files that involve the projects and the changes done by all the developers working on, so a source control manager[1] is indispensable. All artefacts required to build the project should be placed in the repository, due to the

[1] Source control manager: A component of software configuration management. Also named as version control.

convention that the system should be buildable from a local copy from the repository, without requiring additional dependencies. The mainline, i.e. the current state of the system, should be the place of the working version of the system.

- Automate the build: getting the sources turned into a running system can often be a complicated process involving compilation, moving files around, loading schemas into the databases, and so on. However, like most tasks in this part of software development, it can be automated, and as a result should be so. A single command should have the capability of building the system.

- Make your build self-testing: traditionally, a build means compiling[2], dependency download and all related stuff depending on the programming language. A program may run, but that does not mean that it does the right thing. A good way to catch bugs more quickly and efficiently is to include automated tests in the build process. The automated build, therefore, should also run tests to verify the code.

- Everyone commits to the mainline everyday: by committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making. Committing all changes at least once a day (once per feature built) is generally considered part of the definition of CI. In addition, performing a nightly build is generally recommended. These are lower bounds; the typical frequency is expected to be much higher. Frequent commits encourage developers to break down their work into small chunks of a few hours each. This helps track progress and provides a sense of progress.

- Every commit should build the mainline on the integration machine: using daily commits, a team gets frequent tested builds. This practice should ensure that there exists a healthy stable version of the mainline. In practice, however, things still go wrong. One reason is discipline, that is, people not doing an update and build before they commit. Another is environmental differences between developers' machines. To ensure the mainline healthy state, regular builds should happen on an integration machine, and only when this build succeeds, the commit

[2] To compile: In computer programming, the translation of source code into object code by a compiler

can be considered as successful. One of the important benefits of CI is to find bugs as soon as possible, and this rapid feedback is accomplished when every commit is built. The developers, responsible of the commits, need to monitor the mainline so they can fix it if it breaks. Here come into play the continuous integration servers.

These servers are used in the integration machine to build commits automatically. It is a way to automate this practice and they are widely extended in CI. Though the build can be done manually by the developer that commits the last changes to the mainline, it is a great option to accelerate the feedback related to the last build.

- Fix broken builds immediately: applying the best practices, CI converges to do continuous builds. If the build fails, it needs to be fixed immediately. The key point is to get a healthy and stable mainline.

- Keep the build fast: The rapid feedback about changes in the code is the point CI focuses the most on. Since CI demands frequent commits, this adds up to a lot of time. Keeping the build fast will reduce the time spent for each developer when they commit.

- Test in a clone of the production environment: The point of testing is to flush out, under controlled conditions, any problem that the system will have in production. A significant part of this is the environment within which the production system will run. If you test in a different environment, every difference results in a risk that what happens under test will not happen in production. As a result, the most convenient is to set up the test environment to be as exact a mimic of your production environment as possible

- Make it easy for anyone to get the latest executable: Anyone involved with a software project should be able to get the latest executable and be able to run it: for demonstrations, exploratory testing, or just to see what changed this week. This is strongly related with agile software development.

- Everyone can see what is happening: CI is all about communication, so it is the best practice to ensure that everyone can easily see the state of the system and the changes that have been made to it.

So far, the need for adopting CI in a project has been explained. The basic CI definition, that entails a cultural component, triggers, as a consequence, a set of principles that involve some practices and automation processes and tooling. This process does not explicitly include the deployment to the production-like environment.

Traditional software deployment involved IT operation teams in charge of the deployment process. This resulted as a wall between development and operations. IT teams responsible of the deployment often had other things they were working on in parallel. So, the deployment schedule was determined by when they were available, preventing them to meet the business needs. Adopting CI means getting a healthy mainline, with all the benefits from getting rapid feedback about the code to improve its quality, becoming ready to be deployed when the business needs it. This is strongly related with agile methodologies of releasing small pieces of useful functionalities.

In that context, Continuous Delivery and Continuous Deployment are two software slightly different approaches that automate the software deployment and let the team focus on building the product, being agnostic of the deployment process stage in the software development. The incremental and continuous builds offered by adopting CI are translated as continuous deployments with the CD approaches.

These two approaches are often interchanged, but they are not the same. Continuous Delivery automates the deployment of the software on a regular basis to a production-like environment. This mean that the latest stable version is deployed automatically to all test environments that the enterprise has available, replicating or mimicking the production environment as much as possible. The final objective is to ensure that the software can be deployed at any time, simply by clicking the deploy button. It is worth to mention that the deployment process is tested in the test environments each time a deployment is performed, so there is a strong confidence about the process. The continuous Deployment is almost the same but with one significant difference: the production environment deployment is also automatic when the tests in all production-like environments pass successfully. That is, no human interaction exists after the code commit that triggers the deployment process.

Continuous Deployment is an option to keep in mind in systems completely controlled by the business, such as web-based applications. However, it is not that attractive in systems that requires end-user installations, like mobiles or desktop applications. In my opinion, the continuous delivery is a more secure option that implies more flexibility in the

deployment process so from now on, in this thesis the CD[3] acronym will refer to as Continuous Delivery, and the development carried in this work will be based on this practice too.

The CD approach is modelled by the delivery Pipeline, where automated builds, tests and deployments are orchestrated as one release workflow. The delivery pipeline is a set of steps that code changes will go through until ready to be deployed to production. The steps that are grouped in pipeline stages are defined by the software constraints and the business needs.

1.2. Aim of this work

The main goal to accomplish is the design, modelling and implementation of an open source service that sets up a whole infrastructure allowing end-users to adopt the CI culture, following the key principles previously reviewed, also taking advantage of the CD approach minimising the human interaction in the set up and usage of the whole system.

The service will be divided in two main blocks; the first one is the environment creator, in charge of creating the necessary infrastructure that will make it possible to apply the CI principles:

1. Initialization of the integration machine, including all necessary tools and configurations: automation server to orchestrate all the steps involved in the delivery pipeline and necessary software to accomplish the processes.
2. Initialization of the test deployment machine, where the continuous deployments will be done with all necessary tools and configurations.
3. Linkage of the integration machine and the test deployment machine between them and with other services.

This infrastructure set up will be specified with the project seed file, including user predefined parameters such as project name, source control manager project link, and others such as credentials to access the servers where these configurations will apply.

The second service block is the implementation of a solution that will allow creating pipelines, thus reducing the effort when creating different ones.

[3] CD: Continuous Delivery

The final flow of the service will be as follows: from an existing project, located in the predefined source control manager by the user, the platform will prepare the deployment environments and stages to be executed every time a change is published in that project mainline. When a change is submitted, the state of the pipeline will be accessible in the CI server web interface, since every commit will trigger the automated build, tests and if succeed automated deployments.

In summary, this work can be specified as follows:

Goal: to allow developers experiencing the benefits from the CI culture with minimum configuration effort, automating processes as much as possible.

Project Blocks:

- Automation of the environment set-up and start-up
- Implementation of an easy way of pipeline creation with one practice case

Final Flow:

- Creation of the seed file with predefined values
- Execution of the developed platform
- Result of the first pipeline execution

1.3. Projects requirements and specifications

Project requirements:

- The service has to be designed to allow end-users adopting the CI culture with the Martin Fowler's CI principles. Some principles live under the user action, but the technical implementation must allow:
    1. Maintain a single source repository
    2. Automate the build
    3. Make the build self-testing
    4. Every commit should build the mainline on the integration machine
    5. Test in a clone of the production environment
    6. Make it easy for anyone to get the latest executable
    7. Allow everyone see what is happening

- The developed solution has to be adaptable to other technologies, different from those considered in this project.

- The platform must be designed allowing, as much as possible, its usage without the knowledge of the involved tools.

Project specifications:

- The initialization of the whole infrastructure has to be done in 2 steps:
    1. Creation of the configuration file: in this step the end-user has to generate the configuration file with the needed predefined variables as project name, source control manager link and secret cloud credentials.
    2. Execution of the service: the user interacts with the service to trigger the creation of the infrastructure and the first project deployment if the pipeline stages pass with success.

1.4. Projects Background

The project started from scratch. The origin is an idea about developing an open source tool based on existing custom solutions in big enterprises. When I was on an internship, I used a custom solution of this platform and I experienced the advantages of adopting these practices in order to increase the speed and quality of delivering an application to its final environment.

There are custom solutions in and for big companies. Some big companies create its own custom systems, and there are some paid subscription solutions. However, no open source tool exists to date capable of generating the whole platform in a simple where the user can choose the platforms to work with.

Likewise, this could be a great solution for a small team that do not have time and resources to automate the CI/CD processes. But, nevertheless, there are some private solutions that are free for small teams. It is worth to mention that these solutions are not self-hosted, so you do not have total control over them.

It is worth mentioning that this project has not been originated in the DAC of the UPC, namely, the context where it has been carried out. Conversely, its initial ideas have been proposed by the TFG candidate to the project supervisor (Jordi Perelló), who kindly accepted to supervise it.

1.5. Workplan

Tasks:

The main work packages to accomplish in this project are:

- Information Research
- Design
- Development
- Integration
- Documentation

A more detailed description of those work packages with their tasks can be found in Annex 1 of this document.

Milestones:

The principal milestones of the project are shown in Figure 1.

WP#	Task#	Short title	Milestone / deliverable	Date (week)
2	1	Ready design	Architecture designed to start the development	4
3	1	CD service implemented	Creation of servers on the cloud (automatically) and link them to a Jenkins 2 server master in order to be able to deploy applications on them.	8
3	2	CI service implemented	Creation of the necessary infrastructure to link existing projects from a version control to the Jenkins 2 master and create a Pipeline of CI/CD stages for them.	13
4	1	Platform integration	Integration of the two blocks together	15

Figure 1: Milestones Table

Gantt Diagram:

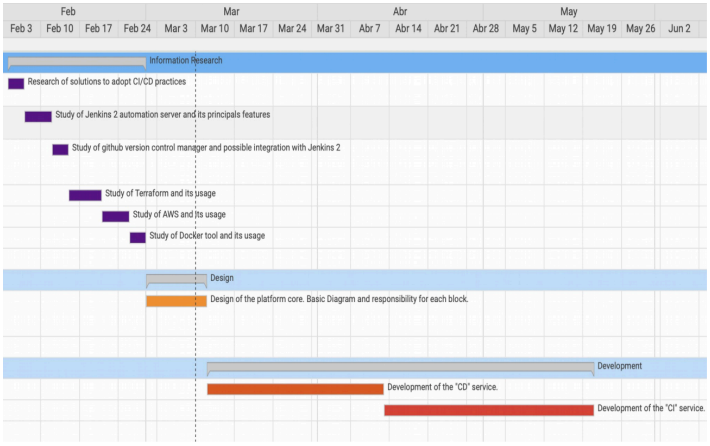

Figure 2: Information research, design and development work package displayed.

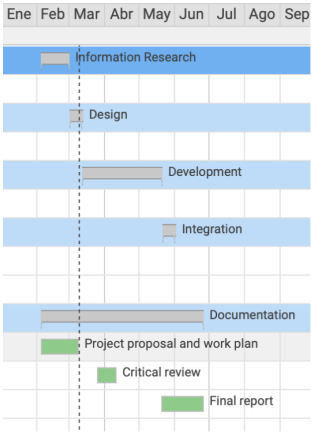

Figure 3: Entire Gannt diagram, only work package of documentation displayed.

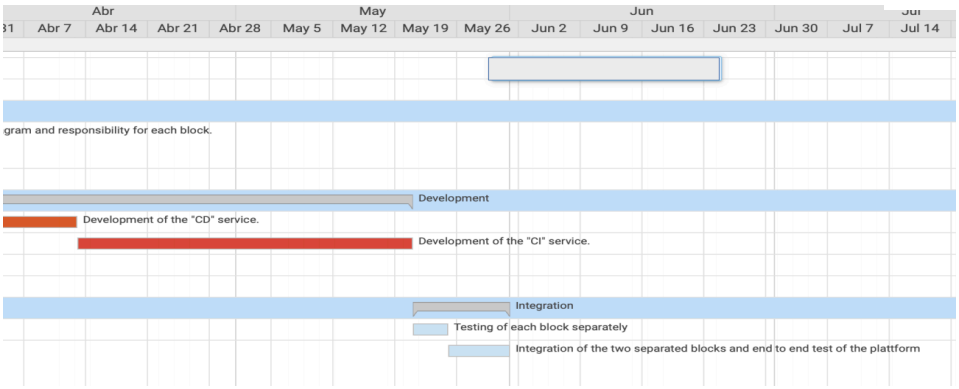

Figure 6: Integration work package displayed

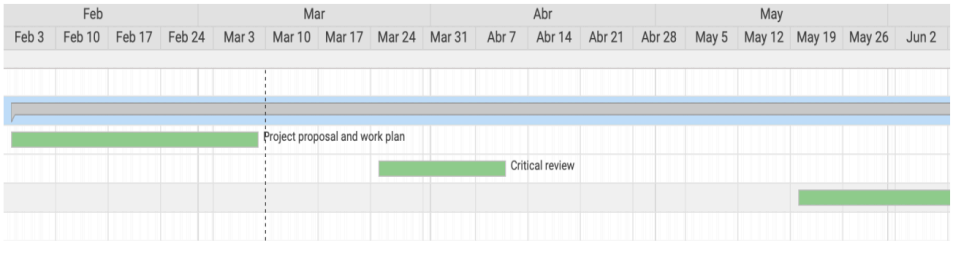

Figure 5: Documentation work package displayed. Part 1/2

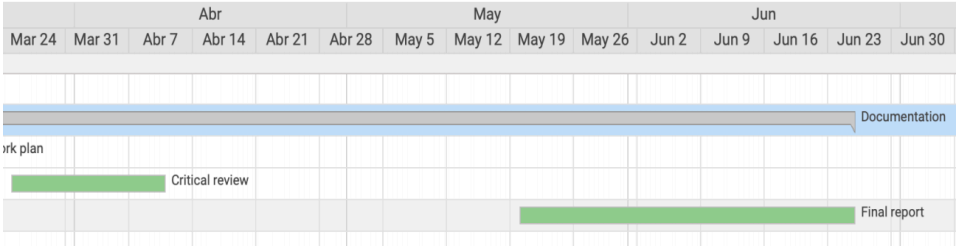

Figure 4: Documentation work package displayed. Part 2/2

Deviations and incidences:

As mentioned in the Critical Review document, "Ready design" and "CD service implemented" milestones were accomplished with some delay.

This workplan update was done before starting the development of the Jenkins shared library. When I started this development, I realized that it was not trivial at all to accomplish a universal solution, as most agnostic as possible to the user, allowing him/her to only care about code. The time I spend modelling and implementing the library was larger than I initially expected. For this reason, I decided to extend the project delivery from June to October 2019.

The other thing worth to mention is the fact that the Development and Integration work packages ended being a single one. Each time I developed a new task or functionality in the platform I integrated and tested it on the platform.

2. State of the art of the technology used or applied in this thesis

A set of tools follows from the project requirements and specifications. To be compliant with the Martin Fowler's principles, the main tools the project need were:

- A source control manager: to maintain a single source repository and get a method to ensure that every commit build on the integration machine.
- A CI server: to build automatically (and pass the project tests) every commit on the integration machine. The server will orchestrate the deployment in the production-like environments. Its function extends to show the state of the project pipelines, so that everyone can see what is happening.
- A virtualization tool to easily replicate the environment in the production-like environments, as well as in the development machine.
- An artefact storage system to make it easy for anyone to get the latest executable.

The creation of the infrastructure must be all done by the project services: the user only has to fill a configuration file and the system will generate all the necessary stuff. To accomplish this specification, a tool that performs the creation and setup of a cloud system instances is needed.

All these components will work on cloud service server instances. A server type and a technology to setup all the configurations in the servers will also be needed to accomplish in the best way the previous points.

With these specifications in mind, the tools used in this project will be presented in the next sections.

2.1. Source control manager

The technology behind the source control manager is git. Git is a widely extended open source version control manager. Its purpose is to track changes in any set of files, designed for coordinating work among programmers.

The usage of git is very simple: the end-user can download the source code, placed in a remote git repository (it is worth to mention that the tool can also be used locally, so that the user can initialize a project in the computer and get the benefits of the tool). Once downloaded, there is a broad range of different information that git can extract

about the downloaded project, or actions to do on the project code. The most used of these possibilities that can help in the good practice of the CI are:

- Showing the historic of the code changes. CI fosters a strong communication. So, related to the communication with the developer team, one can see the changes that the teammates have done. Each developer set of changes are submitted to the code through git commits. These commits are accompanied with a message that should summarize the changes, hence existing an explicit communication.
- One can revert the state of the project to the state of specific git commit. If some pushed commit to the mainline breaks the build, the developers can rapidly revert the last pushed commit to the state previous of the break and try to fix it before pushing it to the mainline again.
- When merging the code state with the mainline, git alerts and shows any integration conflict that may exist. When following the CI culture, the integration with other developers is key. Git makes this easier by telling where the integration conflicts exists thus making the integration more comfortable.

Git technology is a simple command line tool for Linux, Windows and MacOS systems. In addition to the tracker, we need some repository to store the code to make it accessible for all developers. Github is a hosting service for git repositories that will serve as a single shared code repository. Its widely extended usage and popularity has driven the integration of this hosting site with a lot of existing tools. For all these reasons, Github is the hosting platform the project will accept projects for.

2.2. CI server

Currently there are many CI server solutions to consider when creating a CI/CD system and a great part of them are open source projects. Its common usage is to define the pipeline stages and its steps in a configuration file with its domain specific language, whatever it is for that server. The CI server chosen for this project is Jenkins. Currently, Jenkins is one of the most extended CI servers and it has a lot of plugins for the interoperability with a broad range of technologies. The requirement to adapt to other source control managers is easy to accomplish with Jenkins, due to its large support for multiple of them. In Figure 7 a table is shown, comparing the most popular CI servers and source control managers they support. As can be seen, Jenkins is the only one supports all of them.

Name	AccuRev	BitKeeper	CA Harvest	ClearCase	CVS	Darcs	Git	GNU Bazaar	Integrity	Mercurial	Perforce	Plastic	PVCS	StarTeam	Subversion	Surround	Synergy	Team Concert	Team Foundation Server	Vault	Visual SourceSafe
Apache Gump	No	No	No	No	Yes	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No	No	No
AppVeyor	No	No	No	No	No	No	Yes	No	No	Yes	No	No	No	No	Yes[20]	No	No	No	No	No	No
Bamboo	Yes[21]	No	No	Yes	Yes	No	Yes	No	No	Yes	Yes	No	No	No	Yes	No	No	No	Yes[22]	No	No
Buddy	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
BuildBot	No	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	No	Yes	No	No	No	No	No	No
BuildMaster	Yes	No	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes
CABIE	No	No	No	No	Yes	No	No	No	No	No	Yes	No	No	No	Yes	No	No	No	No	No	No
CruiseControl	No	No	No	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	No	Yes	No	Yes
CruiseControl.NET	Yes	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Distelli	No	No	No	No	No	No	Yes	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No
Jenkins	Yes	Yes	Yes	Yes	Yes	Yes[23]	Yes	Yes	Yes[24]	Yes	Yes	Yes[25]	Yes	Yes	Yes	Yes[26]	Yes[27]	Yes[28]	Yes	Yes[29]	Yes
OpenMake Software Meister	Yes	No	Yes	Yes	Yes	No	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
QuickBuild (software)	Yes	No	No	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	No	No	No	Yes	No	Yes
Semaphore (software)	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No
TeamCity	Yes	No	No	Yes	Yes	No	Yes	Yes[30]	No	Yes	Yes	No	No	Yes	Yes	No	No	No	Yes	Yes	Yes
Team Foundation Server	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	Yes	No	No
Vexor	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No

Figure 7: Interoperability of the most used CI servers with the most used source control managers [3]

Another reason for choosing Jenkins is that it has native integration with external tools, such as IDE's[4] or notification services. Not only the state of the build can be seen through its web interface, but one can custom notifications to receive on a predefined basis. A summary of services Jenkins support is shown in Figure 8.

Name	Platform	License	Builders: Windows	Builders: Java	Builders: other	Notification	Integration, IDEs	Integration, other
Jenkins	Web container	Creative Commons and MIT	MSBuild, NAnt, Batch Script	Ant, Maven 2, Kundo	CMake, Gant, Gradle, Grails, Phing, Rake, Ruby, SCons, Python, shell script, command-line	Android, Email, Google Calendar, IRC, XMPP, RSS, Twitter, Slack, Catlight, CCMenu, CCTray	Eclipse, IntelliJ IDEA, NetBeans	Bugzilla, Google Code, Jira, Bitbucket, Redmine, FindBugs, Checkstyle, PMD and Mantis, Trac, HP ALM

Figure 8: Jenkins integration with other tools or services [3]

The way to create pipelines in Jenkins is through the Jenkinsfile. In this file, a pipeline is modelled by the Jenkinsfile domain specific language, based in groovy. This domain specific language has native support to call shell scripts in Linux and Windows systems, source control manager tools or notification support via email among others, with the advantage that one can program the flow as in a groovy script. Figure 9 shows an example of a pipeline described by Jenkinsfile.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

Figure 9: Jenkinsfile that define a pipeline with 3 stages. The step executed in each stage only prints in the console output of the web interface a message telling which stage in the pipeline is being executed.

Its integration with github allows throwing web hooks from the hosting site that trigger the start of the execution of the project pipeline. This will accomplish the principle of building each commit on the mainline.

[4] IDE: Integrated Development Environment

2.3. Virtualization tool

Replicate the production environment is desired to execute the tests in the same conditions as in which the software project will be executed on.

Docker is a tool that provides an abstraction layer of the Linux Containers[5] that automates the virtualization of an application in multiple operating systems. In a nutshell, this technology creates an isolated system into the host machine, sharing the kernel resources. It looks like a virtual machine but faster, with a better portability and more lightweight: it is a software virtualization instead of hardware virtualization. Figure 10 shows a comparison between applications running on a virtual machine vs. application running on containers.

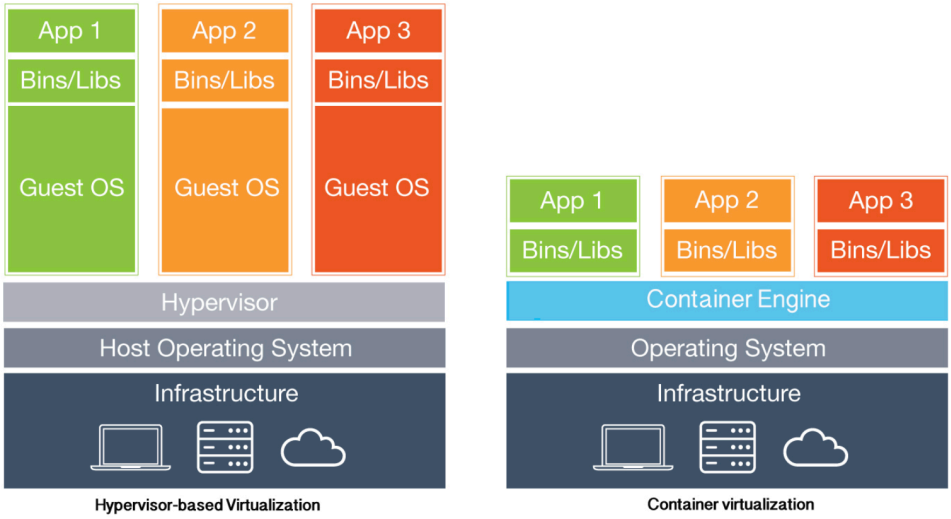

Figure 10: Virtual machine architecture vs. container architecture [4]

These isolated environments inside the host machine are called containers, and these containers are modelled by the docker images. A container is an initialized instance of an image. Each image contains the information about the system to create. The images are created from Dockerfiles, whose description defines the final container ecosystem.

[5] Linux Containers: virtualization technology in the operating system level for Linux.

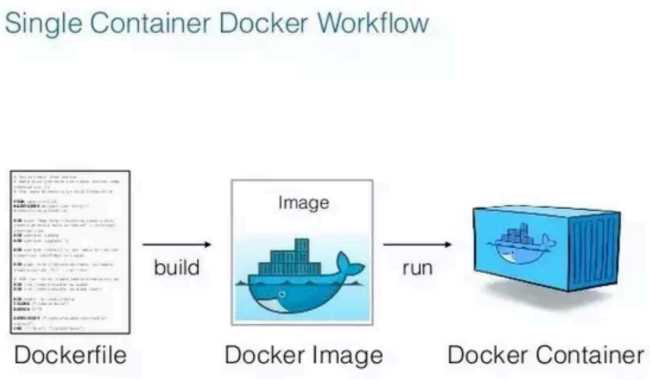

Figure 11: Single Docker Container Workflow [5]

The Dockerfile description allows a huge number of configurations, such as system type (Windows or Linux), environment variables, commands to execute in the start-up of the container, and a large etcetera. The last description in the Dockerfile is the command to execute when the container starts, and the lifecycle of this container ends when the command executed ends. To run our desired code inside a docker container, after describing the environment, we have to copy the build code in the container. The way to describe Dockerfiles is designed for this. At the end of the description, we must specify the main entry point of the application.

The replication of the production environment now is only a simple copy of the configuration file of the production environment (Dockerfile) with possible little changes. The resulting image will become the artefact to deploy on the test and production machines.

There exist multiple official images, because of its popularity. Imagine an example where some team works on a Java[6] project. They need the Java Development Kit (JDK) installed in the deployment machines and they have to set multiple environment variables, as a way of storing some information as credentials. There exists an official Java image ready to use. With this official image, the team can generate a new one creating a Dockerfile that uses this Java image: the description of the docker file must specify the image is built above the Java image, the variable environments they need to set, the build source code to copy and its location in the container filesystem, the start-up steps on the final container and, at the end, the execution of the program. This is only a high-level description. In practice, there are a lot of possible configurations to set-up. From this Dockerfile, developers can generate a new image with these needs.

[6] Java: statically typed programming language that runs on a virtual machine

Further, there exists multiple container orchestrators that can orchestrate them. An example of this is Kubernetes or Docker Swarm technologies, but these technologies are out of the scope of this project.

2.4. Artefacts storage

The artefacts storage role is to provide access to everybody to get the latest build completed successfully, which will be stored in that repository. In addition, the development team gets a way to deploy the version they need. In the situation when some bug is not detected by tests and the code gets deployed in production, the team need to roll back to the last stable version. This procedure is easily accomplished getting the last stable build and deploying it to the production. Once the virtualization technology chosen is Docker, and the deploy artefacts are docker images, the artefacts repository chosen for the project will be the Docker Registry. Docker registry is a hosting repository to push and pull images from. It can be easily used only running the Docker Registry image in the desired Docker Host. Each build will push the created image referenced by a version to the registry, and the deployment machine will pull these images from the registry to start the deployment.

2.5. Cloud Services handler

To start working with the CI/CD platform, there exists some initial automated configuration: the system to develop in this project is in charge of starting and configuring the integration machine and the deploy one and orchestrate its connection. In addition, I wanted to design the project system adaptable to other cloud technologies.

Terraform is the tool that can handle these requirements. It allows provisioning and managing any cloud, infrastructure or service. With this, the start-up and management of the cloud instances can be done programmatically with the Terraform domain specific language. To setup the machines, the main task is to create system scripts to configure all instance requirements. To do so, we need to choose the system type of the integration and test deploy instances. For the project purposes, the chosen cloud is AWS[7].

2.6. Server type

The election behind the system type used in this project was determined by my background. All these technologies are developed for both platforms, Linux and Windows.

[7] AWS: Amazon Web Services

The chosen server type is Linux because it is the system that I use daily and with whose shell (the system commands and scripts interpreter) I am more used to work with.

3. Methodology / project development

3.1. System Blocks

3.1.1. Infrastructure Block

The necessary infrastructure for this project is generated with Terraform. This tool manages the entire set-up process of the instances by reading its declarative configuration files.

These configuration files use of the following declarative blocks that are interpreted by the tool:

- provider: terraform declarative block to specify the cloud provider and the credentials to access in.
- resource: terraform declarative block to specify the resource type and name. Inside this block, the declaration of the OS[8] image[9] to boot, the instance type and the connection type to the machine are placed, among others. The declaration of the following connection and provider blocks is done inside this block as well.
- connection: the type of connection between the terraform host machine and the instance that has started. This is necessary, because once terraform started the machine, it must provision a set of files and execute some commands on it.
- provisioner: this block can declare files to copy to the target machine and commands to execute on it.

Now is the time to see how the project uses terraform to start-up the entire infrastructure. Remember that the needed one in this project is composed by:

- Integration machine, where the automated builds will be executed and orchestrated by the CI server. It must contain the technology to store the successful builds.
- Deployment test machine where the successful builds will be deployed.

[8] OS: Operating System
[9] OS image: An OS image is simply a file that contains the OS

The Terraform project, which will be interpreted by the tool to manage the start-up and configuration of the cloud instances, has the structure shown in Figure 12.

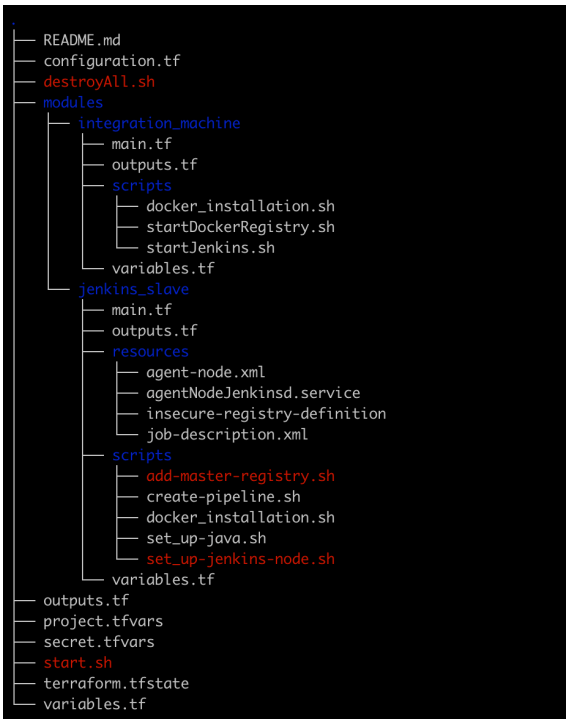

Figure 12: Terraform project tree

In the root of the project there are three Terraform declarative files (with *.tf* extension): configuration.tf, variables.tf and outputs.tf.

As its name indicates, the variables.tf file is used to declare input variables that will be used in the declaration blocks. The outputs.tf file variables will be displayed to the user upon the instance creation, e.g. the assigned public URL address to the instance.

The other Terraform file in the root is the configuration.tf file. This one is the main entry point of the project and its declaration blocks are read by the tool to create the entire ecosystem.

To start up the integration and test deployment machines, there are two modules created under the modules folder in the source root: integration_machine and jenkins_slave. Each module has its own declaration blocks in their main.tf file. The outputs.tf and variables.tf files there have a similar behaviour as the root ones. The variables.tf can be hardcoded or injected by the declaration of the module in the configuration.tf of the root. Figure 13 shows how some variables declared in the module variables.tf are injected when declared in the module block. The outputs.tf of the module can be accessed in the configuration.tf as well as in the output.tf of the root.

Following this module architecture, the configuration.tf file only declares the usage of these modules injecting some required variables.

```
module "integration_machine"{
  source = "./modules/integration_machine"
  access_key = var.access_key
  secret_key = var.secret_key
  https_password = var.https_password
  pem_key_name = var.pem_key_name
  pem_key_location = var.pem_key_location
  scripts_folder_source = "${var.project_root_path}/modules/integration_machine/scripts"
  jenkins_volume_source = var.jenkins_volume_source
}

module "jenkins_slave" {
  source = "./modules/jenkins_slave"
  access_key = var.access_key
  secret_key = var.secret_key
  jenkins_pass = var.jenkins_pass
  project_name = var.project_name
  git_repo = var.git_repo_url
  git_credentials = var.credentials_name
  master_url = module.integration_machine.instance_url
  pem_key_name = var.pem_key_name
  pem_key_location = var.pem_key_location
  scripts_folder_source = "${var.project_root_path}/modules/jenkins_slave/scripts"
  resources_folder_source = "${var.project_root_path}/modules/jenkins_slave/resources"
  jenkins_user = var.jenkins_user
}
```

Figure 13: Configuration.tf file content

The start.sh bash shell script is created to execute the Terraform tool. The execution of the start.sh script in the Terraform host machine will start the Terraform actions to trigger the infrastructure creation from the configuration.tf file.

```
tfg/sources/terraform-CICD  master
▶ bash start.sh
```

Figure 14: Linux shell instruction to start the terraform actions

The start.sh script content is shown in Figure 15.

```
#!/usr/bin/env bash
terraform init
terraform apply -var-file="secret.tfvars" -var-file="project.tfvars"
```

Figure 15: Content of start.sh script

This shell script invokes the Terraform apply command, passing as an argument the *secret.tfvars* and *project.tfvars* files. In these files, the secret user credentials and the project ones are placed and will be used as the seed files of the project. Once this command is executed, the configuration from the integration_machine and jenkins_slave modules are interpreted to create the two machine instances.

3.1.1.1. Modules

In this section the two modules declared in the configuration.tf, and that therefore are used to describe the infrastructure designed for the thesis purposes, are explained in detail:

1. Integration machine

The main description of this module is done in the main.tf that is placed in the integration machine module folder. The first block of the configuration file is the provider one.

```
//Integration machine configuration
provider "aws" {
  access_key = var.access_key
  secret_key = var.secret_key
  region = "eu-west-3"
}
```

Figure 16: Integration machine main.tf provider block

The resource block includes the ami[10], the instance type, and the private key that will be used to access to the instance. The ami was chosen from the AWS web interface where there are listed all available images for the instances. This one is the Ubuntu 18.04 image. The private key is stored in a .pem[11] file and is also generated and downloaded in the AWS web interface.

```
resource "aws_instance" "master" {
  ami = "ami-0ad37dbbe571ce2a1"
  instance_type = "t2.micro"
  key_name = var.pem_key_name

  tags = {
    Name = "Jenkins master"
  }

  connection {
    type = "ssh"
    user = "ubuntu"
    private_key = file(var.pem_key_location)  //private .pem file dowload from aws console
    host = aws_instance.master.public_ip
  }

  provisioner "remote-exec" {

    inline = [
      "sudo mkdir ${var.scripts_folder_destination}",
      "sudo chmod 777 ${var.scripts_folder_destination}",
      "sudo mkdir ${var.jenkins_volume_destination}",
      "sudo chmod 777 ${var.jenkins_volume_destination}",
    ]
  }

  provisioner "file" {
    source = "${var.jenkins_volume_source}/"
    destination = "${var.jenkins_volume_destination}/"
  }

  provisioner "file"  {
    source = "${var.scripts_folder_source}/docker_installation.sh"
    destination = "${var.scripts_folder_destination}/docker_installation.sh"
  }

  provisioner "file"  {
    content = data.template_file.startJenkins.rendered
    destination = "${var.scripts_folder_destination}/startJenkins.sh"
  }

  provisioner "file"  {
    source = "${var.scripts_folder_source}/startDockerRegistry.sh"
    destination = "${var.scripts_folder_destination}/startDockerRegistry.sh"
  }

  provisioner "remote-exec" {
    inline = [
      "sudo chmod +x ${var.scripts_folder_destination}",
      "sudo bash ${var.scripts_folder_destination}/docker_installation.sh",
      "sudo bash ${var.scripts_folder_destination}/startJenkins.sh",
      "sudo bash ${var.scripts_folder_destination}/startDockerRegistry.sh",
    ]
  }
}
```

Figure 17: Integration machine main.tf resource block

[10] ami: amazon machine image

[11] .pem: is a de facto file format for storing and sending cryptographic keys, certificates and other data

The provisioning is done as in the following steps:

- Creation of the destination scripts and the Jenkins volume (later explained on in this document) folders
- Provisioning of the scripts used in the setup of the machine and the Jenkins volume
- Execution of the scripts to set-up the machine:

As the script names in Figure 16 show, the configuration done in the machine is the following: installation of the Docker engine and start-up of Jenkins and Docker registry on the machine. Figures 18, 19 and 20 shows these scripts:

```
#!/usr/bin/env bash
sudo apt update
sudo apt install -y docker.io
sudo systemctl start docker
sudo systemctl enable docker
```

Figure 19: Docker installation script

```
#!/usr/bin/env bash
sudo docker run -d \
  -p 5000:5000 \
  --restart=always \
  --name registry \
  registry:2
```

Figure 18: Docker registry start-up script

```
#!/usr/bin/env bash
sudo docker run \
  --restart always \
  -d \
  -p 443:8443 \
  -p 50000:50000 \
  -v ${JENKINS_VOLUME}:/var/jenkins_home \
  jenkinsci/blueocean \
  --httpPort=-1 --httpsPort=8443 --httpsKeyStore=/var/jenkins_home/jenkins_keystore.jks --httpsKeyStorePassword=${HTTPS_PASSWORD}
```

Figure 20: Jenkins start-up script

With Docker and its multiple official images, the start-up of Jenkins and Docker registry only requires the execution of the Docker run command with some parameters. One of these parameters is the -v flag that maps some directory to the container file system.

With the filesystem mapping, a new Jenkins container can be started and configured with the purpose of copying the 'jenkins_home' folder that contains the entire information about the installed plugins, the configuration, security, etc. By copying the 'jenkins_home' of this preconfigured system, one can start an exact copy of Jenkins only by mounting the 'jenkins_home' folder to the '/var/jenkins_home' directory of the container with the -v flag. This is the reason because a jenkins volume folder is copied to the integration machine in the provisioning block.

2. Jenkins slave module

Once the integration machine module is created, the Jenkins slave one is interpreted to start-up the test deployment machine. The approach is the same as in the integration machine module:

```
//test deployment machine
provider "aws" {
  access_key = var.access_key
  secret_key = var.secret_key
  region = "eu-west-3"
}

resource "aws_instance" "slave" {
  ami = "ami-0ad37dbbe571ce2a1"
  instance_type = "t2.micro"
  key_name = var.pem_key_name

  tags = {
    Name = "${var.project_name}-test-deployment-machine"
  }

  connection {
    type = "ssh"
    user = "ubuntu"
    private_key = file(var.pem_key_location)  //private .pem file dowload from aws console
    host = aws_instance.slave.public_ip
  }
```

Figure 21: Provider and part of the resource blocks of the Jenkins slave module declaration

This block declaration is practically identical as that of the integration machine but changing the tag name of the instance.

The provisioning approach of this machine is the same as in the integration machine by creating the folders to place the content in, the declaration of the necessary files to copy to the machine and scripts execution.

```
provisioner "remote-exec" {

  inline = [
    "sudo mkdir ${var.scripts_folder_destination}",
    "sudo chmod 777 ${var.scripts_folder_destination}",
    "sudo mkdir ${var.resources_folder_destination}",
    "sudo chmod 777 ${var.resources_folder_destination}",
    "sudo mkdir ${var.jenkins_workspace}",
    "sudo chmod 777 ${var.jenkins_workspace}",
  ]

}

//File provisioning to the agent-node machine: scripts and necessary resources to cennect the machine as a jenkins agent node.

provisioner "file" {
  source = "${var.scripts_folder_source}/set_up-java.sh"
  destination = "${var.scripts_folder_destination}/set_up-java.sh"
}

provisioner "file"  {
  source = "${var.scripts_folder_source}/docker_installation.sh"
  destination = "${var.scripts_folder_destination}/docker_installation.sh"
}

provisioner "file" {
  source = "${var.resources_folder_source}/agent-node.xml"
  destination = "${var.resources_folder_destination}/agent-node.xml"
}

provisioner "file" {
  content = data.template_file.agentNodeJenkinsd.rendered
  destination = "${var.resources_folder_destination}/agentNodeJenkinsd.service"
}

provisioner "file" {
  content = data.template_file.set_up-jenkins-node.rendered
  destination = "${var.scripts_folder_destination}/set_up-jenkins-node.sh"
}

provisioner "file" {
  content = data.template_file.job-description.rendered
  destination = "${var.resources_folder_destination}/job-description.xml"
}

provisioner "file" {
  content = data.template_file.create-pipeline.rendered
  destination = "${var.scripts_folder_destination}/create-pipeline.sh"
}

provisioner "file" {
  content = data.template_file.insecure-registry-definition.rendered
  destination = "${var.resources_folder_destination}/insecure-registry-definition"
}

provisioner "file" {
  content = data.template_file.add-master-registry.rendered
  destination = "${var.scripts_folder_destination}/add-master-registry.sh"
}


//Java JDK installation to the agent machine:

provisioner "remote-exec" {
  inline = [
    "sudo chmod +x ${var.scripts_folder_destination}",
    "sudo bash ${var.scripts_folder_destination}/set_up-java.sh",
    "sudo bash ${var.scripts_folder_destination}/docker_installation.sh",
    "sudo bash ${var.scripts_folder_destination}/add-master-registry.sh",
    "sudo mv ${var.resources_folder_destination}/agentNodeJenkinsd.service /etc/systemd/system/agentNodeJenkinsd.service",
    "sudo systemctl start agentNodeJenkinsd.service",
    "sudo systemctl enable agentNodeJenkinsd.service",
    "sudo bash ${var.scripts_folder_destination}/create-pipeline.sh",
  ]
}
```

Figure 22: Jenkins slave module provisioning

The provisioning executes the copied set-up scripts that configure the deployment machine. First, there is an installation of Java and Docker. The Java installation is needed because the communication between the Jenkins CI server and this instance that will act as an execution node is done by executing a java program in the slave instance. The Docker installation is to deploy the Docker build images.

The Java installation script is shown in Figure 23. Docker installation is the same as in the integration machine module showed in the Figure 18.

```
#!/usr/bin/env bash
apt install unzip
apt install zip
export SDKMAN_DIR="/usr/local/sdkman" && curl -s "https://get.sdkman.io" | bash
source ${SDKMAN_DIR}/bin/sdkman-init.sh
sdk install java 8.0.222.hs-adpt
```

Figure 23: Java installation bash script

Docker by default does not accept connections to insecure registries. Since I wanted to push and pull images from the registry hosted in the integration machine, the following script adds this registry to Docker known insecure registries.

```
#!/usr/bin/env bash
apt install unzip
apt install zip
export SDKMAN_DIR="/usr/local/sdkman" && curl -s "https://get.sdkman.io" | bash
source ${SDKMAN_DIR}/bin/sdkman-init.sh
sdk install java 8.0.222.hs-adpt
```

Figure 24: Bash script to add insecure docker registry

By using the Linux systemd software suite, I register the script that connects the instance with Jenkins Master (hosted in the integration machine) as a daemon in the system.

```
"sudo mv ${var.resources_folder_destination}/agentNodeJenkinsd.service /etc/systemd/system/agentNodeJenkinsd.service",
"sudo systemctl start agentNodeJenkinsd.service",
"sudo systemctl enable agentNodeJenkinsd.service",
```

Figure 25: Systemd service file registration

The service file, named agentNodeJenkinsd.service, that handles the daemon lifecycle is shown in the figure 26:

```
[Unit]
Description=Jenkins agent to master connection service
After=network-online.target

[Service]
Restart=always
RestartSec=9
ExecStart=/usr/bin/env bash ${SCRIPTS_FOLDER}/set_up-jenkins-node.sh

[Install]
WantedBy=multi-user.target
```

Figure 26: Systemd service file to manage the connection with the jenkins CI server

The key parts of this file that will be used by systemd to manage this service are the *After*, *Restart* an *ExecStart* parameters. These parameters are interpreted by the systemd suite and the constraints they declare are:

- this service must be started after the network service is up, *After* constraint
- the service must be always restarted (if the service fails or the system is restarted), *Restart* constraint
- the script to execute in the start of the service is set in the *ExecStart* content

The script that starts the connection with the Jenkins CI server is shown in Figure 26.

```
#!/usr/bin/env bash
export PATH=$PATH:/usr/local/sdkman/candidates/java/current/bin
function createNodeFromParameters() {
template_agent=${TEMPLATE_AGENT}
sed -i "s#$\NAME#${AGENT_NAME}#g" $template_agent
sed -i "s#$\ENV#${ENV}#g" $template_agent

}


createNodeFromParameters
if [[ -z $(cat ${SCRIPTS_FOLDER}/agent.jar) ]]; then
    cd ${SCRIPTS_FOLDER} && wget https://${JENKINS_MASTER_URL}/jnlpJars/agent.jar --no-check-certificate
    chmod +x ${SCRIPTS_FOLDER}/agent.jar
fi
if [[ -z $(cat ${SCRIPTS_FOLDER}/jenkins-cli.jar) ]]; then
    cd ${SCRIPTS_FOLDER} && wget https://${JENKINS_MASTER_URL}/jnlpJars/jenkins-cli.jar --no-check-certificate
    chmod +x ${SCRIPTS_FOLDER}/jenkins-cli.jar

fi
nodeCreated=$(java -jar ${SCRIPTS_FOLDER}/jenkins-cli.jar -noCertificateCheck -auth ${USER}:${PASS} -s https://${JENKINS_MASTER_URL}  get-node ${AGENT_NAME}) || true
if [[ -z $nodeCreated ]]; then
    java -jar ${SCRIPTS_FOLDER}/jenkins-cli.jar -noCertificateCheck -auth ${USER}:${PASS} -s https://${JENKINS_MASTER_URL} create-node < ${TEMPLATE_AGENT}
fi
SECRET=$(curl -k -L -s -u ${USER}:${PASS} -X GET https://${JENKINS_MASTER_URL}/computer/${AGENT_NAME}/slave-agent.jnlp | sed "s/.*<application-desc main-class=\"hudson.remoting.jnlp.Main\"><argument>\([a-z0-9]*\).*/\1/")
java -jar ${SCRIPTS_FOLDER}/agent.jar -noCertificateCheck -jnlpUrl https://${JENKINS_MASTER_URL}/computer/${AGENT_NAME}/slave-agent.jnlp -secret $SECRET
```

Figure 27: Bash script that initializes the communication with the Jenkins CI server.

This connection between the Jenkins CI server (Jenkins Master) and the test machine can be done in two ways:

- The Jenkins server adds the deployment machine. The connection is started by the Jenkins server
- The instance connects themselves as a node executor in the Jenkins server. The connection is started by the instance.

In this project, the developed approach is the second one, are the instances that starts the communications. The reasons for this selection are as follows:

- The Jenkins server is agnostic of the connection and there is no need direct interaction with the server.
- The deployment machine can be behind a firewall. If the deployment machine would be behind a firewall, the Jenkins server could not start the communication with it.

Finally, once the deployment machine is linked to the Jenkins CI server, the pipeline for the project is created and executed for the first time.

```
#!/usr/bin/env bash
export PATH=$PATH:/usr/local/sdkman/candidates/java/current/bin
#create job in jenkins master
java -jar ${SCRIPTS_FOLDER}/jenkins-cli.jar -noCertificateCheck \
-auth ${JENKINS_USER}:${JENKINS_PASSWORD} \
-s https://${JENKINS_MASTER_URL} \
create-job ${PROJECT_NAME}-pipeline < ${RESOURCES_FOLDER}/job-description.xml
#build the first execution of the job
java -jar ${SCRIPTS_FOLDER}/jenkins-cli.jar -noCertificateCheck \
-auth ${JENKINS_USER}:${JENKINS_PASSWORD} \
-s https://${JENKINS_MASTER_URL} \
build ${PROJECT_NAME}-pipeline
```

Figure 28: Bash script for the project pipeline creation and first execution.

3.1.2. Pipelines Block

As described in the section 2.2 the way to create pipelines in Jenkins is through Jenkinsfiles. This Jenkinsfile must be placed in the root of the project and its content describes them, i.e. the stages and its steps to execute with all its needed parameters. An example of a pipeline is show in Figure 9.

To make the platform as user agnostic as possible, it comes with predefined pipelines to execute Java projects. It means that such projects only by following the gradle[12] structure will be accepted in the Jenkins server by only adding a Jenkinsfile definition in the root of them. The goal of this development block is a simple declaration of the pipeline type and not the whole stages and other configurations.

To accomplish the previous goal and taking advantage of native way in Jenkins to create shared libraries, in this project there has been developed a library that contains pipelines for the previous technologies. This library will allow to define the pipeline by only importing the library and then with the declaration the type of pipeline to adopt in the project.

The Jenkins shared libraries have to be developed as a groovy project. According to the shared libraries documentation [6] the project structure to create them is as follows:

```
(root)
+- src                     # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

Figure 29: Jenkins shared libraries structure

[12] gradle: open-source build-automation system

The *src* directory should look like standard Java source directory structure. This directory is added to the classpath[13] when executing Pipelines.

The vars directory hosts script files that are exposed as a variable in Pipelines. The name of the file is the name of the variable in the Pipeline. So, a file called vars/log.groovy with a function like def info(message), it will be accessible like log.info "hello world" in the Pipeline.

This library developed in this project models the pipelines as OOP objects. These pipelines declare stages, also modelled as objects, to accomplish stage steps on the specified node, i.e. the instance where the stage will be executed.

The way to create a pipeline in the library is as follow: one has to create a new pipeline class and add the desired stages. These stages could exist as a stage classes because are used in some existing pipelines or, in the opposite case, must be created. The creation of a new stage it was also designed to be as simple as possible: its implementation consist on the creation of a new custom stage class extending an abstract *BaseStage* class and implementing the *stageSteps()* abstract method.

The project structure is shows in the Figure 30.

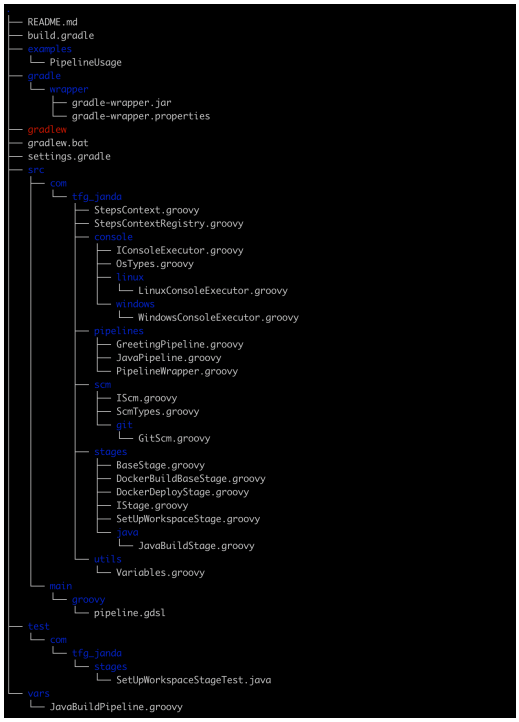

Figure 30: Project developed shared libraries project structure

[13] classpath: classpath is a parameter in the Java Virtual Machine or the Java compiler that specifies the location of user-defined classes and packages.

As it can be seen in the library structure, there is one ready pipeline to use: JavaPipeline. The class that describes it is show in Figure 30.

```
package com.tfg_janda.pipelines


import com.tfg_janda.stages.SetUpWorkspaceStage
import com.tfg_janda.stages.DockerDeployStage
import com.tfg_janda.stages.java.JavaBuildStage
import com.tfg_janda.utils.Variables

class JavaPipeline {

    private PipelineWrapper _pipeline
    private String _gitRepo
    private String _gitBranch
    private String _gitCredentials
    private String _testNode
    private String _dockerRegistry
    private String _projectName
    private String _buildNode = Variables.BUILD_NODE

    JavaPipeline(script, String node, String gitRepo, String gitBranch, String gitCredentials, String dockerRegistry, String projectName) {
        _pipeline = new PipelineWrapper(
                script
        )
        _gitRepo = gitRepo
        _gitBranch = gitBranch
        _gitCredentials = gitCredentials
        _dockerRegistry = dockerRegistry
        _projectName = projectName
        _testNode = node
    }

    def run() {
        _pipeline
                .setUp( os: 'linux',  scm: 'git')
                .addStageToNode(new SetUpWorkspaceStage( stageName: 'set-up', _gitRepo, _gitBranch,  _gitCredentials), _buildNode)
                .addStageToNode(new JavaBuildStage( stageName: 'build',_dockerRegistry,_projectName), _testNode)
                .addStageToNode(new DockerDeployStage( name: 'deploy staging', _dockerRegistry, _projectName), _testNode)
                .exec()
    }
}
```

Figure 31: Java Pipeline class

As seen, the pipeline is composed by three stages:

- Workspace set-up
- Build
- Deploy

These stage classes are shown in the Figures 31, 32 and 33:

```
package com.tfg_janda.stages

import com.tfg_janda.StepsContextRegistry

class SetUpWorkspaceStage extends BaseStage{

    private String _gitRepo
    private String _gitBranch
    private String _gitCredentials

    SetUpWorkspaceStage(String stageName, String gitRepo, String gitBranch, String gitCredentials) {
        super(stageName)
        _gitRepo = gitRepo
        _gitBranch = gitBranch
        _gitCredentials = gitCredentials
    }

    @Override
    def stageSteps() {

        def context = StepsContextRegistry.getContext()
        def console = context.getConsoleExecutor()
        def scm = context.getSourceControlManager()

        //Remove all items from working directory
        console.wipeOutWorkSpace()

        //Download the git project and checkout branch
        scm.cloneAndCheckout(_gitRepo, _gitBranch, _gitCredentials)
    }
}
```

Figure 32: SetUpWorkspaceStage class

```groovy
package com.tfg_janda.stages.java

import com.tfg_janda.stages.DockerBuildBaseStage

class JavaBuildStage extends DockerBuildBaseStage {

    JavaBuildStage(String stageName, String dockerRegistry, String projectName) {
        super(stageName, dockerRegistry, projectName)
    }


    def  dockerImage(){
        return "FROM openjdk:8 as builder\n" +
                "\n" +
                "COPY . /home/gradle/src\n" +
                "WORKDIR /home/gradle/src\n" +
                "RUN ./gradlew build\n" +
                "\n" +
                "FROM openjdk:8-jre-slim\n" +
                "EXPOSE 8080\n" +
                "COPY --from=builder /home/gradle/src/build/libs/artifact.jar /app/\n" +
                "WORKDIR /app\n" +
                "ENTRYPOINT [\"java\",\"-jar\",\"artifact.jar\"]"
    }
}
```

Figure 33: JavaBuildStage class

```groovy
package com.tfg_janda.stages

import com.tfg_janda.StepsContextRegistry
import com.tfg_janda.utils.Variables

class DockerDeployStage extends BaseStage{

    String _dockerRegistry
    String _projectName

    DockerDeployStage(String name, String dockerRegistry, projectName) {
        super(name)
        _dockerRegistry = dockerRegistry
        _projectName = projectName
    }

    @Override
    def stageSteps() {
        def console = StepsContextRegistry.getContext().getConsoleExecutor()
        def tag = StepsContextRegistry.getContext().getExecutionCommmonValue(Variables.GIT_TAG)
        console.exec('docker run -d -p 8080:8080 '+_dockerRegistry+'/'+_projectName +'-artifact:'+tag)
    }
}
```

Figure 34: DockerDeployStage class

The previous images shows the process of pipeline creation: the stage classes extends the *BaseStage* abstract class and overrides the *stageSteps()* method.

For the build class is quite different because there is another abstract class, the DockerBuildBaseStage. This is an abstract class that extends the BaseStageClass and implements the *stageSteps()* method in order to create a docker image for the code. Since each technology must declare different Dockerfile for the image creation, an abstract method *dockerImage()* must be implemented in the different stage classes to define the image for a different technology. In the case of the JavaDockerBuildStage that extends the DockerBuildBaseStage class, the dockerImage*()* define a Dockerfile for a java image.

The entire project code can be found in the github repository https://github.com/JandaTheMan/jenkins-shared-library.

4. Results

All the development in this thesis have as a result two blocks: the infrastructure block and the pipeline block. These two blocks work together to accomplish the goal of the project: the start-up of the necessary infrastructure to work following the CI principles and execution of the first pipeline execution based on the information of the seed file.

First of all, to start the usage, the user has to download the project in its computer. The download link is as follows https://github.com/JandaTheMan/terraform-CICD. After that, the installation of the terraform tool in the user computer it is also mandatory. The installation page can be found in [7].

Once the computer have terraform installed and the project downloaded, there exist two files in the root of the downloaded project where the user have to fill the necessary information: secret.tfvars and project.tfvars. Figures 35 and 36 show them:

```
access_key = ""
secret_key = ""
jenkins_pass = "admin"
https_password = ""
```

Figure 35: Secret.tfvars seed file

```
#project for which the pipeline will be created
git_repo_url = ""
#name of the project
project_name = ""
#name of the created .pem key in AWS
pem_key_name = ""
#location of the key in the system
pem_key_location = ""
#location of the predefined jenkins source in the system
jenkins_volume_source = ""
#location of the project installation path
project_root_path = ""
#user
jenkins_user = "admin"
```

Figure 36: Project.tfvars seed file

The necessary variables the user has to provide are as follows:

- access_key and secret_key: AWS credentials created in the account to access the cloud provider account resources. In the Annex 2 of this thesis there are explained the steps to follow to create these keys.
- https_password: explained in the Annex 3 of the document.

- git_repo_url: URL[14] of the project where for which the pipeline will be created
- pem_key_name: the .pem file name of the one generated in the AWS web interface
- pem_key_location: the location of the .pem in the machine where the project will be executed
- jenkins_volume_souce: location of the jenkins volume in the machine where the project will be executed
- project_root_path: location of the terraform project in the machine where the project will be executed

The jenkins user and the jenkins password, since the jenkins volume folder is provided by the author of the thesis, are set with predefined values that can be changed by the final user. The predefined values as seen in Figures 35 and 36 are admin admin.

The project provided in the git_repo_url must contain a Jenkinsfile. For Java projects the Jenkinsfile to include must have the content shown in the Figure 37.

```
library identifier: 'shared-liraries-POC@master', retriever: modernSCM(
    [$class: 'GitSCMSource',
     remote: 'git@github.com:JandaTheMan/jenkins-shared-library.git'
     ])

 JavaBuildPipeline(
 "$GIT_URL",
 "$GIT_BRANCH",
 "$GIT_CREDENTIALS",
 "$DOCKER_REGISTRY",
 "$PROJECT_NAME_IN_LOWER_CASE",
 "$DEPLOY_NODE"
 )
```

Figure 37: Jenkinsfile for Java projects

The parameters to include are:

- $GIT_URL: the same as git_repo_url, the URL of the project where for which the pipeline will be created.
- $GIT_BRANCH: the git branch that the project will pass the pipeline for.
- $GIT_CREDENTIALS. The name of the credentials to access the github repo if the repo is private. If this case, the credentials must be created in the Jenkins server.

[14] URL: the address of a World Wide Web page

- $DOCKER_REGISTRY: the integration machine public URL followed by the 5000 port, e.g. amazon_provided_public_url.com:5000. The registry installation accept connections to the port 5000 as shown in Figure 18.
- $PROJECT_NAME_IN_LOWER_CASE: the name of the project in lower case
- $DEPLOY_NODE: the project_name provide in the project.tfvars seed file followed by '-test'.

Once these parameters are provided, the user can execute the main program.

The start of the program is done by the execution of the start.sh script located in the root of the project. Once executed the command shown in the figure 38, there will be displayed the next dialog in the terminal of the computer:

```
      + delete_on_termination = (known after apply)
      + device_name           = (known after apply)
      + encrypted             = (known after apply)
      + iops                  = (known after apply)
      + kms_key_id            = (known after apply)
      + snapshot_id           = (known after apply)
      + volume_id             = (known after apply)
      + volume_size           = (known after apply)
      + volume_type           = (known after apply)
    }

  + ephemeral_block_device {
      + device_name  = (known after apply)
      + no_device    = (known after apply)
      + virtual_name = (known after apply)
    }

  + network_interface {
      + delete_on_termination = (known after apply)
      + device_index          = (known after apply)
      + network_interface_id  = (known after apply)
    }

  + root_block_device {
      + delete_on_termination = (known after apply)
      + encrypted             = (known after apply)
      + iops                  = (known after apply)
      + kms_key_id            = (known after apply)
      + volume_id             = (known after apply)
      + volume_size           = (known after apply)
      + volume_type           = (known after apply)
    }
}

Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

Figure 38: Terraform dialog after first execution

The complete dialog shows the properties of the instances terraform is about to start. The summary of the actions to execute is shown in the Plan section. After the user entre yes, terraform starts the provisioning.

After the creation of the infrastructure, terraform will provide an output in the console where the *start.sh* was executed: the URL of the instances that it has just created, the integration machine and the jenkins slave where the 'staging' deployments will take place. This output can be seen in the figure 39.

```
Outputs:

integration_machine_url = ec2-35-180-198-188.eu-west-3.compute.amazonaws.com
test_deploy_machine_url = ec2-35-180-61-168.eu-west-3.compute.amazonaws.com
```

Figure 39: Terraform execution output

If this is the first execution done by the user and there still did no exists a Jenkins Master, the first pipeline execution will fail. This is because the user can not set the docker registry variable in the jenkins file because the URL of the integration machine is not known yet.

After this first execution, the user just knows the integration machine URL, and adding it to the Jenkinsfile followed by ':5000' in the $DOCKER_REGISTRY field, he is able to start the first execution manually.

The state of the builds can be accessing to integration_machine_url over https protocol. Figure 40 shows all the pipelines in the Jenkins server.

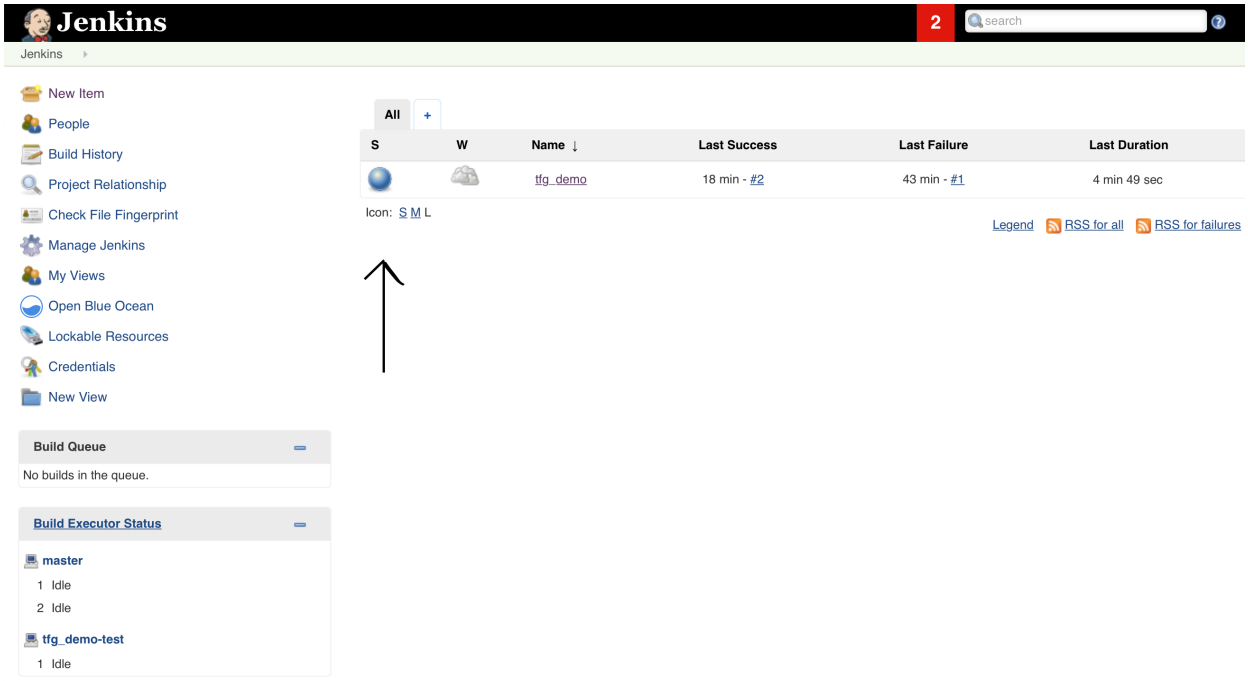

Figure 40: Pipelines in the Jenkins server

The pipeline state of the created pipeline shows that the build failed.

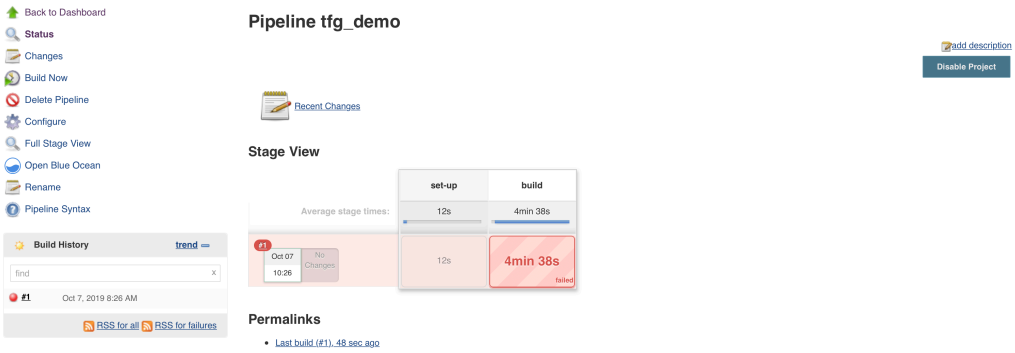

Figure 41: First pipeline execution

The log shown in Figure 42 of the build shows the reason of the fail: the docker registry is no accessible.

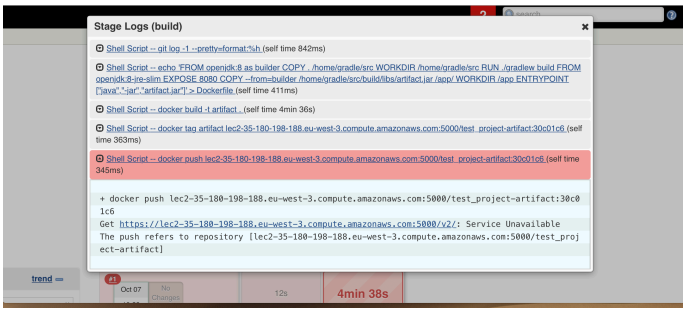

Figure 42: Build fail information

After changing the docker registry host in the pipeline and executing a new build the pipeline pass with success as shown in Figure 42.

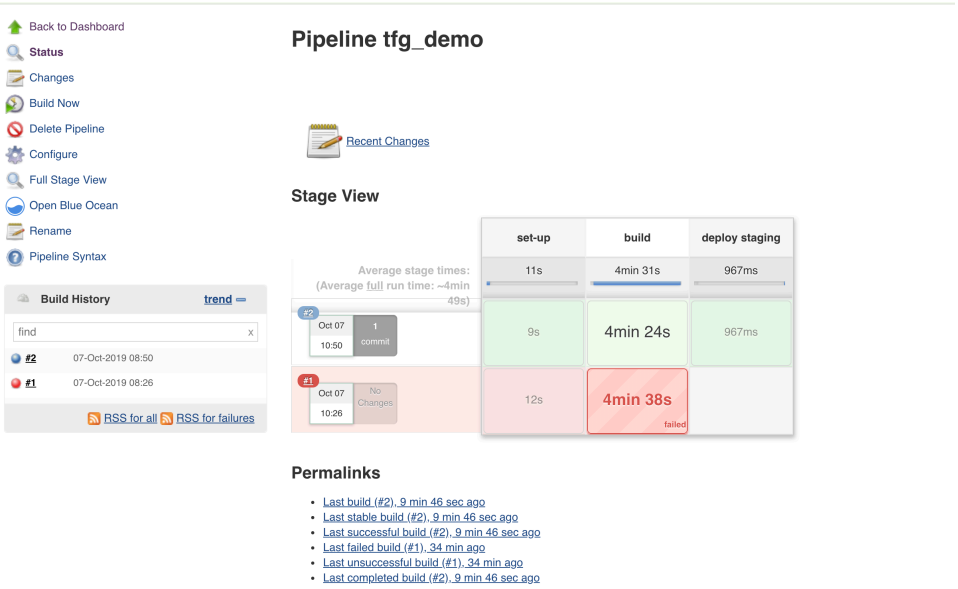

Figure 43; Success pipeline

Then, when accessing to the project endpoint, we can see the following output:

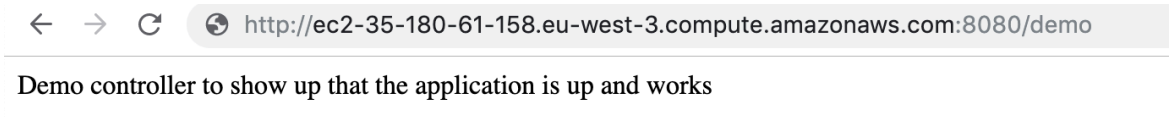

Figure 44: Demonstration project endpoint

The demonstration used in this example can be found in:
https://github.com/JandaTheMan/demo

To configure automatic commit for each build, the user has to set-up the github project and add web hooks to the Jenkins server. The steps to follow are:

1. Sign in, then select the related repository you own.
2. Click on "Settings" on the right panel.
3. Then click on "Webhooks & Services" on the left panel.
4. Click on the "Add WebHook" Button.
5. Paste the copied URL in the URL form field.
6. Select "application/json" as the content type.
7. Select "Let me select individual events" and check "Issues".
8. Leave the "Active" checkbox checked.
9. Click on "Add webhook" to save the webhook.

From now, all the changes done in the project will be triggered and if the pipeline pass the new deployed project will be accessible in the test deployment machine (jenkins_slave) URL.

5. Budget

The totally of hours dedicated to this thesis are 495, the corresponding one to 18 ECTS credits with a time dedication of 27.5h for each one. The cost of the development of the project is thus:

$$495\ hours * 12\frac{€}{hour} = 5940\ €$$

The cloud service used to develop the solution in AWS, but the usage for the development was with the free trial account that did not generate any cost.

The software for the development purposes was IntellIJ Idea that have an annual cost of 649 € but I used the university student licence provided by UPC.

6. Conclusions and future development:

The system to implement in this thesis was designed and developed to follow the CI principles listed in the introduction. Excepting these ones that entail the human component and cannot be part of a CI ecosystem, the project complies with the principles as follows:

- Maintain a single repository: one of the requirements of the developed solution is to add the Github source control manager link to the project seed. This principle has been now converted into a system's constraint: its usage requires a source control manager to work with.
- Automate the build: the developed solution starts a Jenkins CI server that is used to automate the builds. These builds are modelled as a build stage in the Jenkins project's pipeline. These pipelines are expressed as Jenkins files and, in addition, a library has been developed to model these pipelines, in a way to reduce the effort when creating multiple pipelines for different projects with repeated stages.
- Make your build self-testing: the responsibility of writing test is of the developers but the project, as with the builds, uses the Jenkins technology in order to create pipelines. The developed pipelines for java projects in this thesis come with coverage for unit tests.
- Every commit should build the mainline on the integration machine: every time a commit is done to the mainline, a pipeline starts a new execution.
- Test in a clone of the production environment: the execution of the developed solution triggers the start-up of an instance to deploy the 'deploy staging' stage on, i.e. the test machine. The use of Docker ensures that the environment will be the same in the test machine as in the production one.
- Make it easy for anyone to get the latest executable: every successful build stores the resulting artefact (the Docker image) in a Docker registry, accessible for the whole team by only pulling this image.
- Everyone can see what is happening: the Jenkins CI server chosen to develop the solution comes with a web UI where the team can see the state of the pipelines.

The instances of AWS cloud used to start the Jenkins CI server and the test machine are interchangeable for other possible solutions. Since the instance configuration is done through the provisioning and execution of Linux shell scripts and some resources on them, the modification of the provider, resource type and connection are enough to change the infrastructure to work on.

The infrastructure block provides an easy way to add projects: the user only has to replicate the 'jenkins_slave' module with new parameters to start another project.

The Jenkins library developed for this project brings the next benefits:

- On the one hand, the developer is agnostic of the steps involved in the execution of the pipeline: the project comes with predefined pipelines for Java projects and the users are agnostic of the pipeline definition.
- It allows the creation of multiple projects without the need of copying the content of the same pipeline definition file (the Jenkinsfile) among all the projects, the use of the library allows use the pipeline by only declaring the needed one with its input parameters.
- It allows modelling and developing pipelines as OOP objects with a more ordered and reusable way of pipeline creation.

The final solution usage is quite simple, it consists of:

- Downloading of the project code hosted in Github.
- Some extra configuration
- Fill of the seed project file with the needed predefined parameters: secret.tfvars and project.tfvars.
- The execution of the start.sh script of the tool.

After the execution, the whole infrastructure will start-up. The first execution of the pipeline for the specified project will be triggered.

The CD approach is almost accomplished as well. The deployment is done in the test environment: the pipelines for java projects created in this thesis only have the 'staging' environment.

The production could be done by only adding an extra stage in the pipeline replicating the test stage but in the production machine after requiring an interaction from the user to start. Since the environments are replicated with docker, the deployment is done by the same instructions in all the environments (only changing the execution node).

In addition, the two blocks developed in this thesis can be used separately. The infrastructure block can be used to set-up all the infrastructure but a different library o Jenkinsfile definition can be used. On the opposite case, the library can be used by teams that have a stable infrastructure to get a clearer and reusable ways to declare their pipelines.

The main idea of the project was the design and development of a system that generates the necessary infrastructure to accomplish the principles of the CI getting the CD approach benefits. The final solution provides the necessary stuff to accomplish it.

Objectives not met:

The principal unmet objective is the total automatization with a unique tool to handle everything. If the tool users wanted to add another project, they had to copy manually the 'jenkins_slave' module changing the input variables. This objective is not met by a lack of time during the development.

Another objective not met is the total automatization of the user experience. The first execution is not totally automatic: the integration server is not started before that. This means that the first execution of the pipeline triggered in the start-up of the slave instance will fail because in the Jenkinsfile of the project there will not be still set the docker registry.

The principle of starting pipeline executions for each commit is another objective that to be accomplished requires of user interaction. It needs to modify the configuration of the github project to set web hooks to the Jenkins server.

All these additional steps makes the usage of the platform more complex than only fill the seed file and execute the program.

The future development would be the implementation of the previous unmet objectives.

Bibliography:

[1] "Continuous Integration". *Wikipedia* [Online] Available https://martinfowler.com/articles/continuousIntegration.html

[2] Martin Fowler. "Continuous Integration". *Martin Fowler blog* [Online] Available: https://martinfowler.com/articles/continuousIntegration.html

[3] "Comparison of Continuous Integration Software". *Wikipedia* [Online] Available: https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software

[4] "Docker Containers vs. Virtual Machines". *Aquasec*. [Online] Available: https://www.aquasec.com/wiki/display/containers/Docker+Containers+vs.+Virtual+Machines

[5] "Using docker containers to improve reproducibility in software and web engineering". *SlideShare*. [Online] Available: https://www.slideshare.net/vincenzoferme/using-docker-containers-to-improve-reproducibility-in-software-and-web-engineering

[6] "Jenkins Shared Libraries". *Jenkins*. [Online] Available: https://jenkins.io/doc/book/pipeline/shared-libraries/

[7] "Terraform Downloads. *Terraform*. [Online] Available: https://www.terraform.io/downloads.html

Appendices (optional):

Annex 1: Description of the Work Packages and its tasks:

Project: Information research	WP ref: 1	
Major constituent: Research	Sheet 1 of 5	
Short description: Research of existing tools used to adopt CI/CD methodologies and the state of the art of solutions used in the industry.	Planned start date: 4/02/2019 Planned end date: 28/02/2019	
	Start event: End event:	
Internal task T1: Research of solutions to adopt CI/CD practices. Internal task T2: Study of Jenkins 2 automation server and its principal features. Internal task T3: Study of github version control manager and possible integration with jenkins 2. Internal task T4: Study of Terraform tool and its usage. Inetrnal task T4: Study of Amazon Web Services and its usage. Internal task T6: Study of Docker tool and its usage.	Deliverables:	Dates:

Project: Design	WP ref: 2	
Major constituent: Software	Sheet 2 of 5	
Short description: Design of the platform's architecture to make the project adaptable to other version control managers and cloud technologies.	Planned start date:01/03/2019 Planned end date: 10/03/2019	
	Start event: End even	
Internal task T1: Design of the platform core. Basic diagram and responsibility for each block.	Deliverables:	Dates:

Project: Development	WP ref: 3	
Major constituent: Software	Sheet 2 of 5	
Short description: Development of the whole infrastructure	Planned start date:11/03/2019 Planned end date: 20/05/2019	
	Start event: End even	
Internal task T1: Development of the "CD service". This block will be in charge of creating servers on the cloud and link them to a Jenkins 2 server master in order to be able to deploy applications on them. Internal task T2: Development of the "CI service". This block will be in charge of creating the infrastructure necessary to link existing projects from a version control to the Jenkins 2 master and create a Pipeline of CI/CD stages for them.	Deliverables:	Dates:

Project: Integration	WP ref: 4	
Major constituent: Software	Sheet 4 of 5	
Short description: Integration of the developed blocks and testing of the platform.	Planned start date:21/05/2019 Planned end date:31/05/2019	
	Start event: End event:	
Internal task T1: Testing of each block("CI service" and "CD service") separately. Internal task T2: Integration of the two separated blocks and end to end test of the platform.	Deliverables:	Dates:

Project : Documentation	WP ref: 5	
Major constituent: Documentation	Sheet 5 of 5	
Short description: Elaboration of the project deliverables.	Planned start date:4/02/2019 Planned end date: 25/06/2019	
	Start event: End event:	
Internal task T1: Project proposal and work plan document elaboration. Internal task T2: Critical review document elaboration. Internal task T3: Final report document elaboration.	Deliverables: Project proposal and work plan Critical review Final report	Dates: 10/03/2019 12/04/2019 25/06/2019

Annex 2: Steps to follow to create access and secret key in AWS:

The Access Key and the Secret Access Key are not your standard username and password but are special tokens that allow our services to communicate with your AWS account by making secure REST or Query protocol requests to the AWS service API.

To find your Access Key and Secret Access Key:

1. Log in to your AWS Management Console.
2. Click on your username at the top right of the page.
3. Click on the Security Credentials link from the drop-down menu.
4. Find the Access Credentials section, and copy the latest Access Key ID.
5. Click on the Show link in the same row and copy the Secret Access Key.

Annex 3: Steps to follow to create access and secret key in AWS:

In order to provide http over TLS[15] the next actions must be performed on the jenkins volume folder:

The next command must be executed on the machine:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore jenkins_keystore.jks -storepass mypassword -keysize 2048
```

Then the generated jenkins_keystore.jks must be copied in the jenkins volume folder root. After that, the password to set in https_password the 'mypassword' set in the previous command.

[15] TLS: Transport Layer Security

Glossary

CI: Continuous Integration

To compile: In computer programming, the translation of source code into object code by a compiler

CD: Continuous Delivery

Linux Containers: virtualization technology in the operating system level for Linux.

Java: statically typed programming language that runs on a virtual machine

AWS: Amazon Web Services

OS: Operating System

OS image: An OS image is simply a file that contains the OS

ami: amazon machine image

.pem: is a de facto file format for storing and sending cryptographic keys, certificates and other data

gradle: open-source build-automation system

classpath: classpath is a parameter in the Java Virtual Machine or the Java compiler that specifies the location of user-defined classes and packages.

URL: the address of a World Wide Web page

TLS: Transport Layer Security