

# DevOps : Continuous Integration and Continuous Deployment applied

A Degree Thesis  
Submitted to the Faculty of the  
Escola Tècnica d'Enginyeria de Telecomunicació de  
Barcelona

**Universitat Politècnica de Catalunya**

by  
Héctor Pascual Haba

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

**Advisor: Israel Martín**

Barcelona, June 2020

## Abstract

DevOps is a trending concept in the SW industry introduced and popularized during the last decade, this thesis goes deep into the concept, the culture and fields related to it which are present in almost each project and company nowadays. The research model followed in the thesis basically consists in getting in contact with the DevOps culture by designing, developing and implementing a Python tool with some Continuous Integration and Continuous Deployment features. Basically this tool consists on a back-end REST API capable of creating and automating builds on any connected slave node to the tool, similar to traditional DevOps tools logic such as Jenkins or TeamCity, but in a lightweight, portable and OS independent solution, also a front-end is developed in order to make the tool easier and simpler to use. In order to demonstrate the power and the capabilities of the tool we will containerize a build environment with Docker and automate its build process with the tool as well as deploying the binaries resulting from the build process.

## Resum

DevOps és un concepte de tendència a la indústria del SW, introduït i popularitzat durant la darrera decada, aquesta tesi aprofundeix en el concepte, la cultura i aspectes relacionats, que son present a la majoria de projectes i empreses avui en dia. El model d'investigació a seguir durant la tesi es basa en entrar en contacte directe amb la cultura de DevOps dissenyant, desenvolupant i implementant una eina en Python amb característiques tant d'Integració Continua com de Desplegament Continu. En resum, l'eina consisteix en un backend REST API que permet la creació i automatització de construccions de projectes en qualsevol node esclau connectat a l'eina, de manera semblant a eines tradicional de DevOps com són Jenkins o Teamcity pero d'una manera lleugera, portable i independent del sistema operatiu. També s'ha desenvolupat un frontend per facilitar l'ús de l'eina. Finalment, per a demostrar el funcionament i la capacitat de l'eina crearem un entorn de construcció amb docker del qual automatitzarem el procés de construcció amb l'eina a més de desplegar els binaris resultants del procés.

## Resumen

DevOps es un concepto de tendencia en la industria del SW introducido y popularizado durante la última década, esta tesis profundiza en el concepto, la cultura y los campos relacionados que están presentes en casi cada proyecto y empresa en la actualidad. El modelo de investigación seguido en la tesis consiste básicamente en ponerse en contacto con la cultura DevOps mediante el diseño, desarrollo e implementación de una herramienta Python con algunas características de integración continua y despliegue continuo. Básicamente, esta herramienta consiste en una API REST de back-end capaz de crear y automatizar compilaciones en cualquier nodo esclavo conectado a la herramienta, similar a la lógica de herramientas DevOps tradicionales como Jenkins o TeamCity, pero en una solución ligera, portátil e independiente del sistema operativo, también un front-end se ha desarrollado para hacer que la herramienta sea más fácil y simple de usar. Para demostrar el poder y las capacidades de la herramienta, contenerizaremos un entorno de compilación con Docker y automatizaremos su proceso de compilación con la herramienta, así como desplegaremos los binarios resultantes del proceso de compilación.

## Acknowledgments

I would first like to thank my thesis advisor Israel Martin of the ETSETB at Politechnics University of Catalonia, due to his instant e-mail answers when I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work, but guided me in the right the direction whenever he thought I needed it.

I would also like to thank my supervisor Oscar Alonso from TTTech Auto Iberia, the company I was when this project was carried out, due to his corrections and document validations as well as technical support. Also DevOps department colleagues which were always available for technical issues and give me a hand when I needed it. It is always good to have a technical support when you are entering and discovering a new field on research.

Finally, I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you all who have helped in one way or another to make this project possible.

## Revision history and approval record

Revision	Date	Purpose
0	20/04/2020	Document Creation
1	22/05/2020	Minor corrections proposed by tutor
2	06/06/2020	Minor corrections proposed by tutor
3	29/06/2020	Final version corrections

Table 1: Revision history and approval record

Name	e-mail
Héctor Pascual	rcpascualhector@gmail.com
Israel Martín	israel.martin@upc.edu
Óscar Alonso	oscar.alonso@tttech-auto.com

Table 2: Document distribution list

	Written by :	Reviewed and approved by :
<b>Name</b>	Hector Pascual	Oscar Alonso and Israel Martín
<b>Position</b>	Project Author	Project Supervisor and Advisor

Table 3: Approval Record

## Contents

<b>Abstract</b>	<b>1</b>
<b>Resum</b>	<b>2</b>
<b>Resumen</b>	<b>3</b>
<b>Acknowledgments</b>	<b>4</b>
<b>Table of Contents</b>	<b>6</b>
<b>List of Figures</b>	<b>8</b>
<b>List of Tables</b>	<b>9</b>
<b>1 Introduction</b>	<b>10</b>
1.1 Statement of purpose . . . . .	10
1.2 Requirements and specifications . . . . .	10
1.3 Project Background . . . . .	11
1.4 Work plan . . . . .	12
1.5 Deviations from the initial plan and incidences . . . . .	16
<b>2 State of the art of the technology used or applied in this thesis</b>	<b>17</b>
<b>3 Methodology/Project Development</b>	<b>21</b>
3.1 Design . . . . .	21
3.2 Implementation, dependencies and toolchain . . . . .	22
3.2.1 APSCHEDULER . . . . .	22
3.2.2 SQLITE . . . . .	23
3.2.3 SQLALCHEMY . . . . .	23
3.2.4 PARAMIKO . . . . .	24
3.2.5 LOGGING . . . . .	25
3.2.6 GUNICORN . . . . .	25
3.2.7 FLASK . . . . .	26
3.2.8 REQUESTS . . . . .	27
3.3 Implementation, project architecture . . . . .	27
3.3.1 Job . . . . .	28
3.3.2 Build . . . . .	28
3.3.3 Node . . . . .	28
3.3.4 Pipenv . . . . .	29
3.4 Testing . . . . .	32
<b>4 Results</b>	<b>34</b>
4.1 Bringing up the tool . . . . .	36

5	Budget	37
6	Conclusions and future development	38
	Bibliography	39
	Glossary	40



## List of Figures

1	Work Package breakdown, self-elaboration. . . . .	12
2	GANTT diagram, self-elaboration. . . . .	16
3	DevOps loop, author: Kharnagy . . . . .	19
4	Toolchain used, self-elaboration . . . . .	22
5	Simple back-end jobs structure, self-elaboration . . . . .	27
6	Action flow example, self-elaboration . . . . .	31
7	HTTP GET example performed with Postman, self-elaboration . . . . .	33
8	HTTP POST example to be performed with Postman, self-elaboration . . . . .	33
9	/ . . . . .	35
10	/jobs . . . . .	35
11	/create_job . . . . .	36
12	/create_build/<int:build_id> . . . . .	36

## List of Tables

1	Revision history and approval record . . . . .	5
2	Document distribution list . . . . .	5
3	Approval Record . . . . .	5
4	WP 1.1 . . . . .	12
5	WP 1.2 . . . . .	13
6	WP 1.3 . . . . .	13
7	WP 1.4 . . . . .	13
8	WP 2.1 . . . . .	13
9	WP 2.2 . . . . .	13
10	WP 2.3 . . . . .	13
11	WP 2.4 . . . . .	14
12	WP 2.5 . . . . .	14
13	WP 2.6 . . . . .	14
14	WP 3.1 . . . . .	14
15	WP 3.2 . . . . .	14
19	WP 4.3 . . . . .	15
16	WP 3.3 . . . . .	15
17	WP 4.1 . . . . .	15
18	WP 4.2 . . . . .	15
20	Differences with initial planning . . . . .	16
21	Software Deployment Frequency on diverse companies, source : The Phoenix Project . . . . .	18
22	DevOps toolchain . . . . .	19
23	Log levels used in the tool, source : Python docs . . . . .	25
24	Back-end REST API documentation, source : Python docs . . . . .	34
25	Front-end routes documentation, source : Python docs . . . . .	35
26	Project budget . . . . .	37

# 1 Introduction

The project is carried out at TTTech Auto Iberia an automotive company which works with embedded projects and big build environments that can be automatized in different ways, during my stay in the company I have got in touch with DevOps related concepts such as automatizing infrastructure deployment, containerizing build environments, designing Continuous Integration pipelines, between other. I have taken all this knowledge learnt and decided to focus my thesis on creating an own CI and CD tool capable of doing most of the common CI/CD features such as automatizing builds, taking as input pipeline files, storing artifacts resulting from builds, capability of scheduling jobs through cron expressions and so on.

## 1.1 Statement of purpose

Along this project I will go deep into the DevOps culture and the trending concepts related to it, which are present in almost each project and company nowadays by designing, developing and implementing a Python tool with some CI and CD features.

The project main goals are:

- Get involved with trending DevOps technology.
- Get a deep knowledge on DevOps culture and its related concepts.
- Develop a CI / CD (Continuous Integration / Continuous Deployment ) tool with features based on other CI tools.
- Containerize a build environment and automate its build process as an example.

## 1.2 Requirements and specifications

The requirements for the project are strictly related with the tool developed and with the coverage of all DevOps concepts that are connected to the tool implementation.

- The project should cover in deep all the DevOps related concepts mentioned in the sections.
- The project should contain reflections on different ways to perform automation of processes.
- The development of the tool should be submitted to constant commits to a VCS (git for this project).
- The tool should be able to automatize any process that can be done manually via shell commands.

- The tool should be able to schedule jobs periodically through cron expressions.
- The tool should be able to describe the processes to automatize through pipelines.
- The tool should be able to store binary artifacts after a successful build.

Referring to the specifications:

- The project will contain technical examples when an automation process is described (i.e: Wrapping up a build environment with Docker and deploying it with Ansible to multiple physical hosts.)
- The tool will be able to be hosted on any OS system connected to a network.
- The tool will be accessible through a REST API developed on Python.
- The tool will be also accessible through a Front-end which interacts directly with the REST API back-end in order to ease its use.
- The Python version used on development will be at least Python 3.6 due to the constant usage of formatted string literals included in version 3.6 (see PEP-498).

### 1.3 Project Background

This project is conceived as an abstraction wrap up and continuation of all the work I have been doing in the company during last year and a half and also a deep analysis and practical implementation of concepts related to DevOps culture that I had to learn in order to proceed. With abstracted I mean that no confidential data will be present on any document. The only way I will be using all the work done internally in the company is as base knowledge and start point to develop this project.

### 1.4 Work plan

The structure of the project tasks follows the next order :

**Major constituent → Work Package → Sub Tasks** (if any)

Conceiving a work package as a single well-defined structure (not general or abstract as Major Constituents are) containing a notable amount of work which could include sub-tasks, as it is shown in the following diagram and in below tables with work packages definitions :

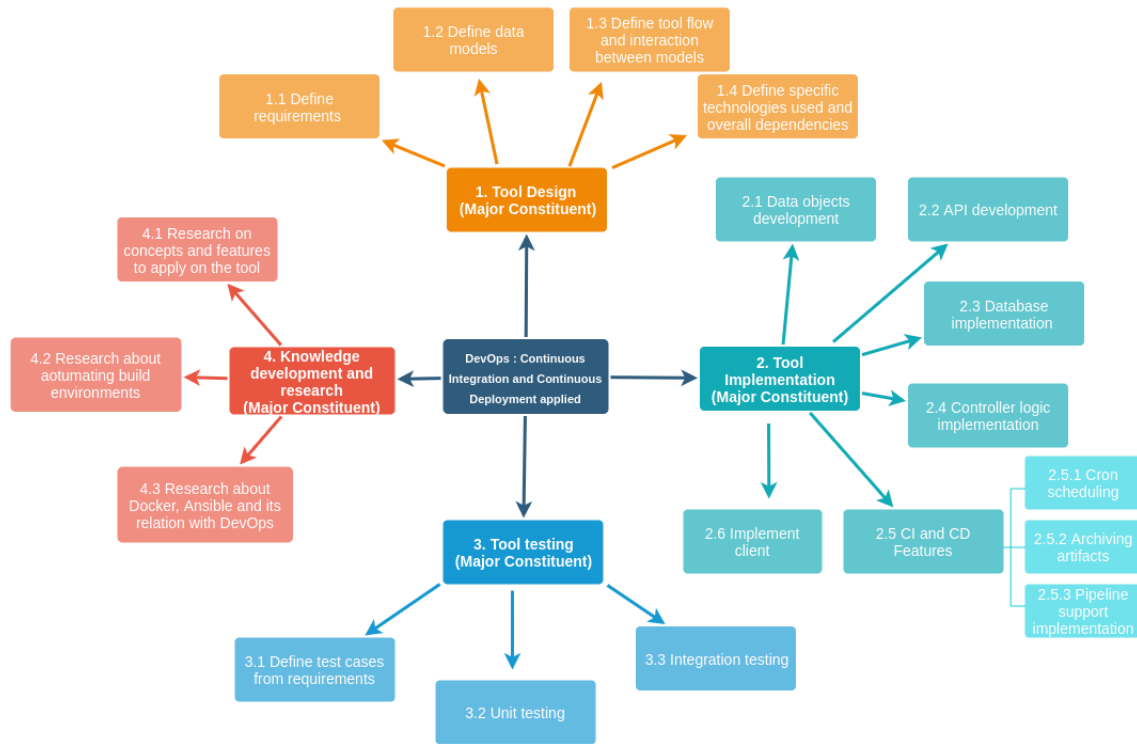


Figure 1: Work Package breakdown, self-elaboration.

<b>WP Name:</b> Define requirements	<b>WP ref:</b> 1.1
<b>Major constituent:</b> Tool Design	-
<b>Short description:</b> Define specific requirements based on DevOps real world culture and concepts	<b>Planned start date:</b> 16/Feb/2020
	<b>Planned end date:</b> 19/Feb/2020

Table 4: WP 1.1

<b>WP Name:</b> Define data models	<b>WP ref:</b> 1.2
<b>Major constituent:</b> Tool Design	-
<b>Short description:</b> Think about data models that will define the tool and its DB representation.	<b>Planned start date:</b> 24/Feb/2020
	<b>Planned end date:</b> 26/Feb/2020

Table 5: WP 1.2

<b>WP Name:</b> Define tool flow and interaction	<b>WP ref:</b> 1.3
<b>Major constituent:</b> Tool Design	-
<b>Short description:</b> Define the application flow of the application and how the models will interact between them.	<b>Planned start date:</b> 26/Feb/2020
	<b>Planned end date:</b> 27/Feb/2020

Table 6: WP 1.3

<b>WP Name:</b> Define specific technologies and dependencies	<b>WP ref:</b> 1.4
<b>Major constituent:</b> Tool Design	-
<b>Short description:</b> Define programming languages involved, general dependencies and technologies that will be used.	<b>Planned start date:</b> 25/Feb/2020
	<b>Planned end date:</b> 25/Feb/2020

Table 7: WP 1.4

<b>WP Name:</b> Data objects development	<b>WP ref:</b> 2.1
<b>Major constituent:</b> Tool Implementation	-
<b>Short description:</b> Represent the data models in Python class and the database models as well.	<b>Planned start date:</b> 02/Mar/2020
	<b>Planned end date:</b> 05/Mar/2020

Table 8: WP 2.1

<b>WP Name:</b> API Developments	<b>WP ref:</b> 2.2
<b>Major constituent:</b> Tool Implementation	-
<b>Short description:</b> Development of the API in order to make the models accessible from a client.	<b>Planned start date:</b> 09/Mar/2020
	<b>Planned end date:</b> 24/Mar/2020

Table 9: WP 2.2

<b>WP Name:</b> Database Implementation	<b>WP ref:</b> 2.3
<b>Major constituent:</b> Tool Implementation	-
<b>Short description:</b> Implement the logic for accessing to the SQL database using Python.	<b>Planned start date:</b> 03/Mar/2020
	<b>Planned end date:</b> 10/Mar/2020

Table 10: WP 2.3

<b>WP Name:</b> Controller logic implementation	<b>WP ref:</b> 2.4
<b>Major constituent:</b> Tool Implementation	-
<b>Short description:</b> Develop the logic for using the data models and giving capabilities for user interaction through the API.	<b>Planned start date:</b> 08/Mar/2020
	<b>Planned end date:</b> 29/Mar/2020

Table 11: WP 2.4

<b>WP Name:</b> CI and CD Features	<b>WP ref:</b> 2.5
<b>Major constituent:</b> Tool Design	-
<b>Short description:</b> Basing on the research done in parallel in WP 4.1 implement features.	<b>Planned start date:</b> 30/Mar/2020
	<b>Planned end date:</b> 22/Apr/2020
<b>2.5.1 Cron scheduling:</b> Implementing capability to schedule jobs periodically with cron expressions	
<b>2.5.2 Archiving artifacts:</b> Implementing capability to store binaries and outputs from build executions	
<b>2.5.3 Pipeline support:</b> Implementing capability to define builds through YAML pipelines.	

Table 12: WP 2.5

<b>WP Name:</b> Implement Client	<b>WP ref:</b> 2.6
<b>Major constituent:</b> Tool Implementation	-
<b>Short description:</b> Implement a client with GUI in order to interact with the API.	<b>Planned start date:</b> 24/Apr/2020
	<b>Planned end date:</b> 24/May/2020

Table 13: WP 2.6

<b>WP Name:</b> Define test cases from requirements	<b>WP ref:</b> 3.1
<b>Major constituent:</b> Tool Testing	-
<b>Short description:</b> Define test cases based on the requirements defined on WP 1.1	<b>Planned start date:</b> 21/May/2020
	<b>Planned end date:</b> 25/May/2020

Table 14: WP 3.1

<b>WP Name:</b> Unit tests	<b>WP ref:</b> 3.2
<b>Major constituent:</b> Tool Testing	-
<b>Short description:</b> Write unit tests for each module and function that is critical.	<b>Planned start date:</b> 25/May/2020
	<b>Planned end date:</b> 27/May/2020

Table 15: WP 3.2

<b>WP Name:</b> Research about Docker, Ansible and its relation with DevOps	<b>WP ref: 4.3</b>
<b>Major constituent:</b> Knowledge development and research	-
<b>Short description:</b> Research and write ways in which these technologies can be used and applied.	<b>Planned start date:</b> 14/May/2020
	<b>Planned end date:</b> 02/Jun/2020

Table 19: WP 4.3

<b>WP Name:</b> Integration Testing	<b>WP ref: 3.3</b>
<b>Major constituent:</b> Tool Testing	-
<b>Short description:</b> Write integration tests checking overall tool functionality.	<b>Planned start date:</b> 27/May/2020
	<b>Planned end date:</b> 29/May/2020

Table 16: WP 3.3

<b>WP Name:</b> Research on concepts and features to apply on the tool	<b>WP ref: 4.1</b>
<b>Major constituent:</b> Knowledge development and research	-
<b>Short description:</b> Read books and articles trending on the topic to conclude features that might fit on the tool.	<b>Planned start date:</b> 10/Mar/2020
	<b>Planned end date:</b> 04/Jun/2020

Table 17: WP 4.1

<b>WP Name:</b> Research about automating build environments	<b>WP ref: 4.2</b>
<b>Major constituent:</b> Knowledge development and research	-
<b>Short description:</b> Strictly related to 4.3, get conclusions on this topic and write thoughts.	<b>Planned start date:</b> 11/May/2020
	<b>Planned end date:</b> 01/Jun/2020

Table 18: WP 4.2



## 1.5 Deviations from the initial plan and incidences

Below you can see the current GANTT which slightly differs from the initial planning :

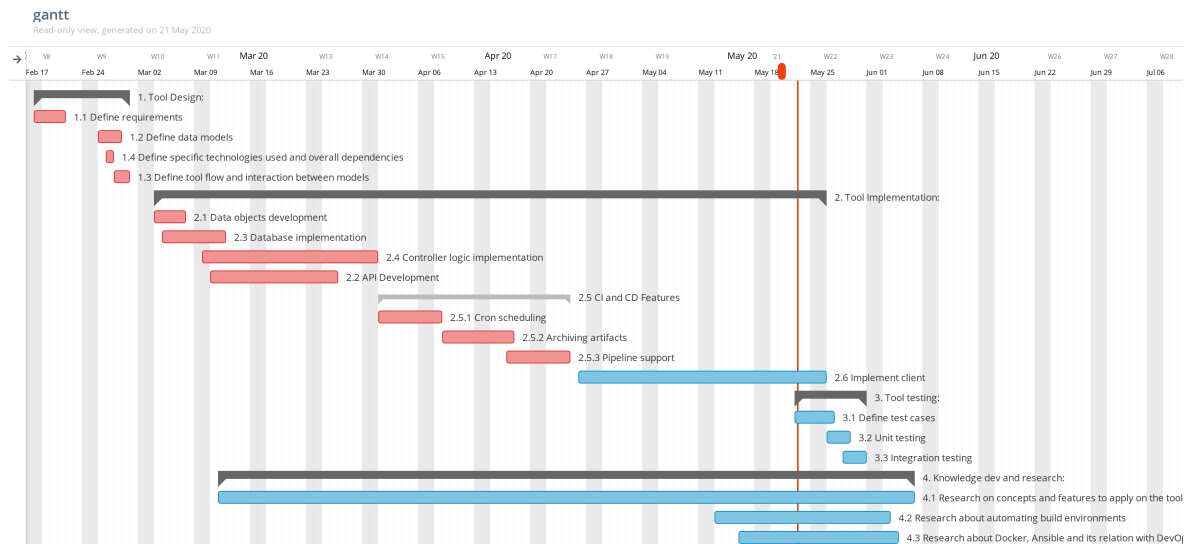


Figure 2: GANTT diagram, self-elaboration.

The differences with the initial GANNT are the following :

WP	Reason
<b>WP 2.6</b>	Client implementation duration has been extended by 4 days, developing front-end has been a challenge for me, as a Software Developer my skills are more related to back-end technologies, then I needed more time to complete the task.
<b>WP 3.4</b>	Automated testing task has been deleted, for the sake of simplicity while developing I decided to not automate the tests (with an external tool), but simply run them after every commit with a simple bash script that run the tests and then perform the commit if tests has passed. This could be considered in some way automated testing but still has a manual part.
<b>WP 4.1</b>	In April the priority for this work package upgraded and I started dedicating more hours per week due to the reason that I was spending less time to researching on concepts and developing the knowledge on DevOps than developing the tool.

Table 20: Differences with initial planning

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

Before getting into DevOps state on nowadays SW industry I will talk about an even more important concept, which is the basis of everything related to DevOps, this concept is automation, conceived as "the technology by which a process or procedure is performed with minimal human assistance", automation is the key for speeding and improving any process related to software, it allows all of the common tasks involved in the creation and deployment of software to be performed by developers, testers, and operations personnel, at the push of a button. Automation tools reduce labor, energy, and materials used to improve quality and accuracy of outcomes.

In the field of software we can distinguish between different types of automation :

- Server Installation : Consists on setting up and configuring servers through automation.
- Infrastructure Automation : Basically, this consists in what is conceived as IaC, an IT paradigm oriented to provision and deploy infrastructure with code, just as the rest of your software. Tools such as Ansible have this purpose as a goal.
- Test and Build Automation : This is the field the thesis mainly focuses on, test and build automation tools make possible to keep developing without having our work laptop frozen due to RAM or CPU run out when compiling a big project, this kind of tools make possible to connect slave nodes in which the builds will take place so the developers don't have to run the build processes locally on their computers. Also is a good practice for having centralized all the builds of a project as well as its resulting binary artifacts, and opens the door to DevOps world. There are also different types of build and tests automation which will be seen along the thesis :
  - On-demand automation : In this case the build is triggered by the user.
  - Scheduled automation : This kind of automation gives rise to the known nightly builds, the builds are scheduled often through Cron Expressions, this makes possible running builds when no-one is working on the project in order to build all the changes committed by the developers along the day.
  - Triggered automation : This is strictly related to VCS such as Git or Svn, this kind of automation trigger builds depending on the workflow followed, for example if the project follows a feature branching git model, builds or tests can be triggered when pull requests to master are merged, or even with every commit performed by the developers.

Let's introduce the DevOps concept itself with an example, imagine your first days as a developer in a company on a field you have already experience in, you get your laptop with the proper development environment setup. After you write some new code you perform a commit and these changes are queued to be built and tested against a server which contains the build environment setup, if tests passes now you have a certain degree of confidence that the new code is correct and you can decide (depending on the complexity and other factors) whether to open a Pull Request, which will be submitted to a code review, in order to bring these changes to production environment.

The duration of all this processes can vary depending on the project size, it is so common that can be done more than 5 times a day per developer (see in the following table typical deployment frequencies) so the steps followed must be efficient and well structured. The wrap up of all this processes is conceived as Pipeline or DevOps pipeline, a pipeline consists of a set of tools, flows, and automated processes, enabling teams to to build and deploy software efficiently, and includes Continuous Integration and Continuous Deployment.

Company	Deployment Frequency
Amazon	23,000 per day
Google	5,500 per day
Netflix	500 per day
Facebook	1 per day
Twitter	3 per week
Typical Company	1 every 9 months

Table 21: Software Deployment Frequency on diverse companies, source : The Phoenix Project

DevOps is not just another software development methodology, is a way of thinking and operating that enable teams to deploy software in efficient and lasting ways, it is part of the culture that shapes how and why we work, a philosophy of close collaboration between traditionally distinct disciplines, the Development team and the Operations team.

A merge of Development and Operations teams conceiving the DevOps team, becomes a requirement on most of nowadays projects and companies on the software field, in order to avoid deployment failures produced by the increase of complexity:

A key factor that has led to the high level of deployment failures we see today is a corresponding increase in software and infrastructure complexity over time ... Overcoming all this complexity requires a tremendous amount of coordination and communication.

(Collaborative DevOps with Rational and Tivoli, 2011 IBM)

Moving into more specific DevOps related concepts, Pipelines, Continuous Integration and Continuous Deployment will be defined more in detail. The following image describes the steps of a general DevOps loop cycle or pipeline:

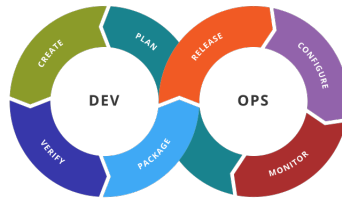


Figure 3: DevOps loop, author: Kharnagy

All the steps in the image above are simple and easy to understand, but are also complex when it refers to proceed with a project, each of them implies the usage of certain tools, going step by step some will be mentioned :

		Step	Tool
<b>D</b>		Plan	Jira, VersionOne
		<b>CI</b>	
<b>E</b>	<b>CI</b>	Create	PyCharm, Visual Studio Code
		Verify	JUnit, Selenium
<b>O</b>	<b>CD</b>	Package	Artifactory, Docker Hub
		Release	Docker, AWS
<b>P</b>	<b>CD</b>	Configure	Ansible, Chef
		Monitor	Datadog
<b>S</b>			

Table 22: DevOps toolchain

Continuous Integration is conceived as a software development practice where project members integrate their work often, where the frequency can go up to multiple integrations per day. With CI, code changes are integrated into a central repository several times a day. As a result, merging the different code changes from each developer becomes easier and less time-consuming. Bugs will also be encountered early and this will make it easier to resolve them. Referring to above pipeline model Continuous Integration would include the whole Create step, which can be splitted into code and build and also would include the Verify step.

Continuous Deployment is the process of deploying changes into production environment by minimizing the risk of failure, having passed a set of tests. You increase the frequency of releasing new features. Consequently, it enhances the customer feedback loop, hence creating the opportunity for better customer involvement. Regarding the pipeline model defined above, it would include the Package, Release and Configure steps.

Focusing on the tool that has been developed along this thesis, it is mainly written in Python, which I had to learn in order to proceed, chosen due to its huge amount of frameworks that eases the development and its learning curve. The minimum Python version required for running the tool is Python v3.6 as specified in the requirements.

Web frameworks was the main focus for developing the tool, as a back-end REST API was one of the requirements, a web framework is a code library that makes web development faster and simpler by providing common patterns for building reliable and maintainable web applications. A REST API simply consists on building a service by specifying a set of operations on top of HTTP Protocol by using its methods (GET, POST, PUT and DELETE). The web framework chosen in order to proceed with the tool development is Flask, the reasons will be detailed in Methodology/Project Development section.

Also database related concepts had to be studied in order to carry out the tool development, as the tool itself requires the capability to store jobs and builds, I decided to focus the research only on relational type databases, avoiding NoSQL databases such as MongoDB for sack of simplicity at the moment of linking and creating relations between tables, and also because the database structure was clear and well defined from the beginning of the project. Finally, sqlite was the RDBMS chosen over other SQL database engines since sqlite is not a client-server database, it is embedded into the end application.

In order to schedule jobs periodically inside the tool, which is a requirement, I decided to use cron expressions, as I am familiar with Unix systems, so I had to find a way to schedule code execution on Python using cron expressions. Eventually, one of the libraries which allows this kind of code scheduling on Python is called Advanced Python Scheduler.

Regarding the front-end, the requirement was to ease the usage of the tool, then something simple and lightweight was necessary. A single page app was created with Flask, the content of which changes dynamically, by performing queries to the back-end REST API, and by using a web template engine called Jinja and JQuery.

## 3 Methodology/Project Development

### 3.1 Design

Along this section the methodologies and the practices used in the tool development and the project research will be described in detail.

The first weeks of the project it was under a design phase in which I created the whole specification for the tool development, the requirements exposed at the beginning of the thesis, and also the toolchain which would be used. Basically I wanted to create a lightweight, portable, easy to deploy and OS independent tool with CI and CD features.

The tool is an automation server that allows the user to schedule builds on-demand and get the build outputs, consisting on a back-end designed under an MVC pattern and a simple front-end in order to speed up the usage of the tool.

The CI features the tool has are the following :

- **Job Scheduling** : The tool is capable of scheduling periodically builds using cron expressions.
- **Pipeline support** : The commands that will be run in a build can be contained in a YAML file ordered by stages in order to have a clear order and sense of the what will be happening in the build.

These features are grouped into Continuous Integration because they ease the process of testing a project and increases the build frequency in order to find bugs in less time, apart from automating the whole process.

Regarding the CD features of the tool :

- **Artifacts archiving** : The tool is capable of storing binaries or any kind of output that a build produces, selectable by the user, for example after the build of a C project you can store all \*.o and \*.ar files.

This Continuous Deployment is a must in any DevOps tool, as it closes the loop, after the build is automated and the outputs are stored, we can deploy or deliver this output to the client, flash it to some hardware, monitor parameters. Hence, joining the CI and the CD features the DevOps loop is closed efficiently with the usage of the tool.

### 3.2 Implementation, dependencies and toolchain

In this section we will see in detail the tool by reviewing the use cases for each dependencies and also an architecture overview that will be related with the previous dependencies explanation in order to fully understand the tool purpose and usage.

The tool has been developed with git as VCS in order to keep a clean change history, apart from the local index (where I stored all the changes), a github repository has been used as platform in some cases to push the changes (when I needed to share the work to another computer or work station). There is one repository for the frontend : <https://github.com/HectorPascual/ci-cd-client> and another one for the backend : <https://github.com/HectorPascual/ci-cd-tool>.

The tool has been implemented in Python using libraries and frameworks that have eased the process of development, in the following diagram you will see the toolchain used :

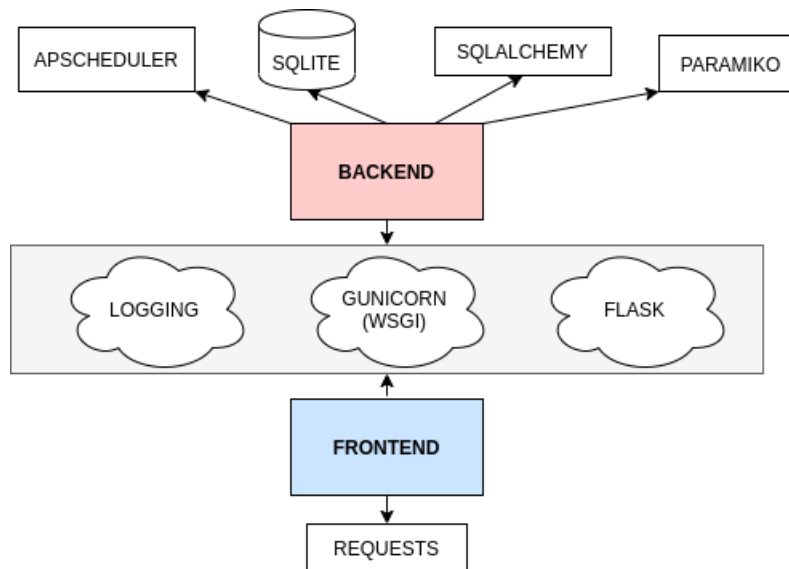


Figure 4: Toolchain used, self-elaboration

The dependencies in the diagram will be briefly explained and justified for the use case :

#### 3.2.1 APSCHEDULER

Is a Python library that lets you schedule code pieces defined as jobs, in concrete I am using an instance of the object BackgroundScheduler in order to run pieces of code in background while running the application.

The following piece of code schedules a build, the piece of code scheduled is contained in `create_build` function, parsed as first argument, then we use `kwargs` in order to pass the arguments regarding the `create_build` function, with `trigger` parameter we specify the type of trigger, in the case of our tool we want to use cron expressions, then we specify the cron parameters (minute, hour, day, month and day of week) finally an `id` that will identify the job and then we start the scheduler:

```
1 scheduler.add_job(func=create_build, kwargs=kwargs, trigger="cron",
2                   minute=minute, hour=hour, day=day_month, month=month,
3                   day_of_week=day_week, id=cron_key)
4 scheduler.start()
```

### 3.2.2 SQLITE

Is a very lightweight and easy to include type of database and I decided to use it on the tool, the database is automatically created if there's none on the base directory of the tool.

### 3.2.3 SQLALCHEMY

SQLAlchemy is a framework that eases Python management of SQL databases, on our main app file we create a database object using SQLAlchemy constructor, this will allow us to perform operations and define schemas for the database.

```
1 from flask_sqlalchemy import SQLAlchemy
2 # ...
3 # app object initialization
4 # ...
5
6 # specify db path
7 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:/// ' + os.path.join(
8     os.path.dirname(os.path.dirname(__file__)), 'app.db')
9 # db object initialization
10 db = SQLAlchemy(app)
```

We use the `db` object mainly for defining the schemas (models) for the data types that will be stored in the database, for instance, a job is represented the following way in the database :

```
1 import datetime
2 from src.app import db
3
4
5 class Job(db.Model):
6     id = db.Column(db.Integer, primary_key=True)
7     title = db.Column(db.Text)
8     description = db.Column(db.Text)
9     created_date = db.Column(db.DateTime, default=datetime.datetime.utcnow)
10
11     builds = db.relationship('Build', backref='job', lazy = False)
12     # ...
```

Basically the models consists of a primary key, which is the `id`, the properties and then we are taking profit of SQL relationships between tables, a Job can contain multiple builds.



Note that we are not taking profit of lazy loading capability of SQLAlchemy, we are using *Eager* loading which specifies that you always want to retrieve the relationship, not only when the object is being accessed.

### 3.2.4 PARAMIKO

Paramiko is the SSH Python library by reference, it contains an SSH client and server implementation and allows you to perform all SSH operations through Python code. The use case for the tool is basically creating SSH connections to nodes where the builds can be executed.

Not only I am taking profit of the SSH connection for executing commands but also sharing files between the slave node and the master (which is the one that hosts the tool) in order to retrieve binary files and results of certain builds, by using SFTP protocol over SSH.

In the tool architecture, which will be seen in the next subsection, the concept of Runner appears, which is an object that allows a node to run commands and perform SSH operations. Its definition using Paramiko is as follows :

```

1 class RunnerSSH():
2     def __init__(self, node):
3         self.node = node
4         self.ssh_client = paramiko.SSHClient()
5         self.ftp_client = None
6
7     def connect(self):
8         try:
9             self.ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy)
10            self.ssh_client.connect(self.node.ip_addr, self.node.port, self.node.
user, self.node.password)
11            logging.info("[SSH] Connection established successfully")
12        except Exception as e:
13            logging.warning(f'[SSH] There was an error while establishing
connection {e}')
14
15    def run_commands(self, commands):
16        output = ""
17        cmd_list = commands.split(';')
18        status = "passed"
19        for cmd in cmd_list:
20            logging.info(f"Executing shell command : {cmd}")
21            _, stdout, stderr = self.ssh_client.exec_command(f"cd {self.node.
workspace};{cmd}")
22            output += stdout.read().decode("utf-8") + stderr.read().decode("utf
-8") + '\n'
23            if stderr.channel.recv_exit_status() != 0:
24                status = "failed"
25        return output, status
26
27    def get_files(self, artifacts, local_path, workspace):
28        # Create local path if doesn't exist
29        Path(local_path).mkdir(parents=True, exist_ok=True)
30
31        # Init sftp session if not existing yet
32        if not self.ftp_client:
33            self.ftp_client = self.ssh_client.open_sftp()
34        # Logic for obtaining files ...
35        # ...

```

The SSH Runner class allows connecting to a node, running commands on the node as well as setting a failed build status in case any command returns an error code, and getting files on a certain workspace using wildcards and specifying the file extension.

### 3.2.5 LOGGING

Python's logging module is included in the standard library, and is such a powerful tool for emitting different kind of messages through configuring different log handlers and a way of routing log messages to these handlers on scripts and applications.

The use case for the tool is basically having different levels of logging, so the output of the app is more traceable and readable. Logging module also allows to configure the format of the messages and select which is the default level of the mentioned below. The app will ignore any message with less level than the selected one.

Level	Numeric value
ERROR	40
WARNING	30
INFO	20
DEBUG	10

Table 23: Log levels used in the tool, source : Python docs

### 3.2.6 GUNICORN

The tool can be directly launched with Flask, but according to Flask documentation : "This launches simple builtin server, which is good enough for testing but probably not what you want to use in production". Even though the builtin server is good for demos and testing, I decided to go deeper into this topic and use a real WSGI.

As I am a UNIX environments user I chose Gunicorn for my productive environment, but any other WSGI such as `mod_wsgi` from apache or `uWSGI` can be used.

Gunicorn is an HTTP Server that meets WSGI specification and allows serving Flask applications with multiple workers in order to increase the performance of our application.

In the tool deployment I am using the most basic workers type, a synchronous worker, that handles a single request at a time. But this does not mean poor requests handling per second, as it is bound to the CPU, with good hardware it can handle even hundreds or thousands requests per second with between 4 and 12 workers.

For instance, in order to run the backend we use the following bash cmd :

```
1 $ gunicorn -b localhost:5000 -w 4 src.app:app
```

Where `-b` flag is used to bind a server socket in this case we are deploying to localhost in port 5000.

### 3.2.7 FLASK

Flask is the basis for both the front-end and the back-end, it allows to prototype a REST API well defined relatively fast and the usage of blueprints eases the creation of a modular application.

Basically the whole tool spins around the app object, defined in the main app module :

```
1 from flask import Flask
2 from src.api import api_blueprint
3
4 app.register_blueprint(api_blueprint)
5 app = Flask(__name__)
```

The blueprint allow to define routes in other modules. For instance, the module which contains all the routes in the back-end looks as follows :

```
1 from flask import Blueprint, Response, request
2 import logging
3
4 logger = logging.getLogger('root')
5
6 api_blueprint = Blueprint('api', __name__)
7
8 @api_blueprint.route('/jobs', methods=('GET', 'POST'))
9 def jobs():
10 # ...
11
12 @api_blueprint.route('/jobs/<int:job_id>', methods=('GET', 'DELETE'))
13 def job(job_id):
14 # ...
15
16 @api_blueprint.route('/jobs/<int:job_id>/builds', methods=('GET', 'POST'))
17 def builds(job_id):
18 # ...
19
20 @api_blueprint.route('/nodes', methods=('GET', 'POST'))
21 def nodes():
22 # ...
```

As it is shown above, the blueprint allows to define routes in a different module, the API methods will be seen in detail in the following subsection.

### 3.2.8 REQUESTS

Requests is an elegant and simple HTTP Python library, the use case for the tool is widely found on the front-end code, in order to perform REST API operations against the backend.

For instance, in the next piece of code from the front-end code, the handler for a build route can be seen, basically it performs an operation against the backend and gets the required data, finally renders the template with the data fulfilled.

```

1 @app.route('/jobs/<int:job_id>/builds/<int:build_id>')
2 def build(job_id, build_id):
3     build = requests.get(f'http://localhost:5000/jobs/{job_id}/builds/{build_id}')
4     node_id = build['node_id']
5     node = requests.get(f'http://localhost:5000/nodes/{node_id}').json()
6     return render_template("index.html", node=node, build=build)

```

### 3.3 Implementation, project architecture

Basically the tool consists on a front-end and a back-end which runs independently with different sockets bound, this decision was taken for being able to bring up the client locally in any machine and connect it to a centralised back-end that can be hosted on a dedicated server or virtual machine.

The back-end consists on a REST API capable of automating builds with the following structure defined :

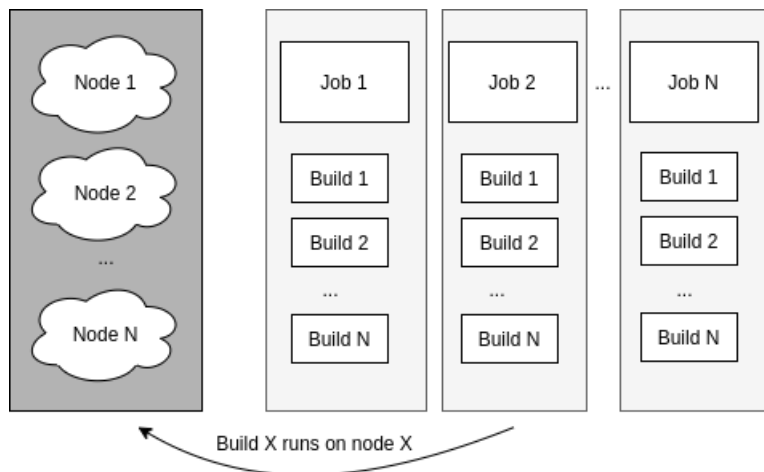


Figure 5: Simple back-end jobs structure, self-elaboration

Above diagram leads to the definition of the concepts mentioned :

### 3.3.1 Job

A job in the tool is conceived as a folder or way to group builds of a same project with a similar purpose. For instance, we could have a job which contains nightly builds and run tests on a project. I have given jobs the only purpose to serve as containers of different kind of builds, so a same job could store compiling builds and test builds.

### 3.3.2 Build

A build stores the configuration and description of a set of commands that might accomplish a purpose on a project (i.e : run tests, build project ...), in the tool there are two kinds of builds, the regular ones and the cron builds, this last type allows to schedule the build to be repeated based in a Cron Expression. Every build has a status which can be passed or failed depending on the return code returned by the commands run.

As explained, builds are based on a set of commands, but there are two ways in the tool to specify this commands, the first one consists on passing all the commands in the following format :

*command1;command2;...;commandN*

The second way of passing commands consists of a Pipeline, a YAML file the structure of which is defined by stages, a stage is a group of commands which can have a name that gives a generic reference of what are these commands doing, so the overall set of commands is much easier to read and understand. A pipeline example looks as follows :

```

1 ---
2 - stage:
3   name: Checkout
4   commands :
5     - git clone https://github.com/HectorPascual/ci_cd_tool.git
6     - echo 'hello'
7
8 - stage:
9   name: Build
10  commands:
11    - pwd
12    - ls
13    - time
14    - echo bye
15
16 - archive: 'ci-cd-tool/*.py'
17 - archive: '*.key'
18 ---
  
```

### 3.3.3 Node

A node is where the builds takes place, basically a machine, the tool is capable of running builds on two types of nodes, the localhost node and SSH nodes. Localhost node is basically the machine where the tool is hosted, and the SSH nodes

is any machine that has SSH capabilities (internet connection and an SSH protocol implementation), for testing purposes with the tool I used virtual machines.

The back-end software architecture is based on an MVC pattern, see the backend structure :

```

1 |--- app.db
2 |--- artifacts/
3     +--- job_2/
4         |--- build_39/
5         +--- build_40/
6 |--- Pipfile
7 |--- Pipfile.lock
8 |--- README.md
9 |--- src/
10     |--- app.py
11     |--- runner.py
12     |--- api/
13         |--- api_routes.py
14         +--- __init__.py
15     |--- controller/
16         |--- build_controller.py
17         |--- cron_controller.py
18         |--- __init__.py
19         |--- job_controller.py
20         +--- node_controller.py
21     |--- __init__.py
22     +--- schemas/
23         |--- build.py
24         |--- cron_build.py
25         |--- __init__.py
26         |--- job.py
27         +--- node.py
28 +--- tests/
29     +--- test.yaml

```

First of all, we find the sqlite database where the data of the tool is stored, then an artifacts folder where all the artifacts archived from a build are stored with the job structure in order to clarify the origin of each artifact, then we find the Pipfile and the Pipfile.lock, which will be described in detail :

### 3.3.4 Pipenv

In order to run the tool in a fast and easy way on any environment and not worrying about dependency installation, we are using Pipenv, which automatically creates and manages a virtual environment, by removing or adding packages from the Pipfile as you install/uninstall packages inside the virtual environment. It also generates a Pipfile.lock, which is used to produce deterministic builds by storing hash codes and relevant data about the dependencies versions.

Installing all the dependencies of the tool with pipenv is as fast as typing the following command in the project directory :

```
1 $ pipenv install
```

For instance, in order to enter the virtual environment and bring up the back-end the procedure would be the following :

```
1 $ pipenv shell
2 $ gunicorn -b localhost:5000 -w 4 src.app:app
```

Following the tree, we get to the `src/` folder where all the source code is contained, the `app.py` which is the main file where the `db` object and the `app` object are initialized and the `runner.py` (already referenced before), where all regarding node SSH connections and node `localhost` are described. Also 3 directories are present, `api/` which is where all the routes are defined using Flask blueprints (would fit the View part on an MVC pattern), the `controller/` and finally the `schemas` (which would fit the model part on an MVC pattern).

Controller files contains the logic for accessing the database and handling the data based on the models defined in the schemas. For instance, `build_controller.py` contains the following functions :

```
1 def get_builds(job_id, build_id=None):
2     # ...
3
4 def create_build(job_id, commands, node_id, description, artifacts=''):
5     # ...
6
7 def delete_build(job_id, build_id):
8     # ...
9
10 def parse_yaml(cmd_file): # Pipeline parser (utility function)
11     # ...
```

Schemas files contains the database columns definition for every type of structure, also functions for parsing the object to string and to a Python dictionary, for instance Job schema looks as follows :

```
1 import datetime
2 from src.app import db
3
4 class Job(db.Model):
5     id = db.Column(db.Integer, primary_key=True)
6     title = db.Column(db.Text)
7     description = db.Column(db.Text)
8     created_date = db.Column(db.DateTime, default=datetime.datetime.utcnow)
9
10     builds = db.relationship('Build', backref='job', lazy = False)
11
12     def to_dict(self):
13         return {
14             'id' : self.id,
15             'title' : self.title,
16             'description': self.description,
17             'created_date': self.created_date.strftime("%m/%d/%Y, %H:%M:%S")
18         }
19
20     def __repr__(self):
21         return f"<id {self.id}, description : {self.description}>"
```

The front-end consists on a single page flask application, which renders HTML code dynamically using Jinja2 and request python lib for performing HTTP operations against the back-end.

```
1 |--- app.py
2 |--- Pipfile
```

```

3 |--- Pipfile.lock
4 |--- static
5     |--- css
6         +--- index.css
7     +--- js
8         +--- index.js
9 +--- templates
10    +--- index.html
  
```

Front-end repository structure is quite simple, we have (as before) the app.py where this time we also have the routes (no use of blueprints) for sack of simplicity, the Pipenv related files for the virtual environment and the dependencies, and then a static folder, where all the static files that will be served by the application are stored (for instance, the css code but also images or icons). The templates folder contains the main HTML file which not only contains HTML code but also is plenty of Jinja2 syntax in order to render the template.

In order to understand how the front-end renders the data an example regarding nodes is going to be exposed. First, the route is defined in the app.py module :

```

1 @app.route('/nodes')
2 def nodes():
3     nodes = requests.get('http://localhost:5000/nodes').json()
4     return render_template("index.html", nodes=nodes)
  
```

As it can be seen, the render\_template function is used to server a static page with dynamic content (keyword arguments) by providing the name of the template and the variables you want to pass to the template engine as keyword arguments.

Then in the template we can handle the nodes variable using Jinja syntax, for instance :

```

1 {% if nodes %}
2     {% for node in nodes %}
3         <a id="node_link{{ node.id }}">
4             <div class="item">
5                 {{ node.id }} {{ node.workspace }} {{ node.ip_addr }} {{ node.user }}
6             </div>
7         </a>
8     {% endfor %}
9 {% endif %}
  
```

The flow of an action would be, a user clicks on the front-end on a button, then a GET method is performed against /nodes route on the front-end, inside the route handler the data is queried to the back-end and the template is rendered by passing the nodes variable, once in the template with the usage of Jinja we display a list of nodes. Graphically, the flow simplified looks as follows :



Figure 6: Action flow example, self-elaboration



### 3.4 Testing

Testing is a fundamental part on any application development, so basic that many companies nowadays are switching their development strategies to the usage of TDD (Test Driven Development) which consists on creating tests from a developer's perspective, this is the case of Thoughtworks, they got many articles on their website sharing thoughts about practices on TDD <sup>1</sup>. This methodology focuses specifically on unit tests based on the requirements, then the code is written by the developer to pass those test cases.

The development of this tool can't be considered fully that it has been under a TDD methodology, but I have shared some of the principles of it, such as think test cases from the requirements at the design part and test each case passes successfully once the development finished, the difference with TDD though, is that the even the cases were planned at design part, they were defined after the development part was over.

Testing can be split into 2 big groups, functional testing, which ignores internal parts and focuses only on the output, and non-functional testing which focuses on testing non-functional requirements such as Load Testing, Stress Testing, Security, Volume. Between these 2, groups we can distinguish a big variety of testing procedures.

The tests performed on the tool are basically integration and unit tests (both types are functional tests). The difference between those two types of tests is simple and fundamental, unit tests are designed for testing small and specific parts of the tool, while integration testing is done to demonstrate that different pieces of the tool work together.

The main focus for the tests was the back-end REST API, tested using Postman, which is an API client designed for testing and speed up the building of API's. With Postman you can easily store your HTTP requests ordered by collections, and compare the responses with the responses you expect, as well as checking correct behavior and testing some specific behaviors of your application under wrong requests or rare uncommon cases. In the following images, it can be seen an HTTP GET example and an HTTP POST example performed with Postman.

---

<sup>1</sup>Thoughtworks article about TDD <https://www.thoughtworks.com/insights/blog/test-driven-development-best-thing-has-happened-software-design>.

```

4511 },
4512 {
4513   "id": 452,
4514   "description": "this is a test build",
4515   "commands": "git clone https://github.com/HectorPascual/ci_cd_tool.git;echo 'hello';pwd;ls;time;echo bye",
4516   "output": "fatal: destination path 'ci_cd_tool' already exists and is not an empty directory.\n\nhello\n\n/home/
management\n\nCANoeV9.zip\n\nCBD1800868_D02_Tricore.7z\n\nCBD1900487_D00_Tricore.zip\n\nnci_cd_tool\n\nnci-cd-tool\n\nDesktop\
zip\n\nMusic\n\nPictures\n\nPublic\n\nRelease.key\n\nsnap\n\nTC297T_triboard_MultiCore.cmm\n\nTemplates\n\nTRACE32_R_2018_02_00009
zip\n\nVideos\n\nVirtualBox VMs\n\n\nnreal\t0m0,000s\n\nuser\t0m0,000s\n\nsys\t0m0,000s\n\nbye\n\n",
4517   "status": "failed",
4518   "job_id": 2,
4519   "node_id": 1,
4520   "created_date": "05/07/2020, 10:29:00"
4521 },
4522 {
4523   "id": 452,
4524   "description": "this is a test build",
4525   "commands": "git clone https://github.com/HectorPascual/ci_cd_tool.git;echo 'hello';pwd;ls;time;echo bye",
4526   "output": "fatal: destination path 'ci_cd_tool' already exists and is not an empty directory.\n\nhello\n\n/home/
management\n\nCANoeV9.zip\n\nCBD1800868_D02_Tricore.7z\n\nCBD1900487_D00_Tricore.zip\n\nnci_cd_tool\n\nnci-cd-tool\n\nDesktop\
zip\n\nMusic\n\nPictures\n\nPublic\n\nRelease.key\n\nsnap\n\nTC297T_triboard_MultiCore.cmm\n\nTemplates\n\nTRACE32_R_2018_02_00009
zip\n\nVideos\n\nVirtualBox VMs\n\n\nnreal\t0m0,000s\n\nuser\t0m0,000s\n\nsys\t0m0,000s\n\nbye\n\n",
4527   "status": "failed",
4528   "job_id": 2,
4529   "node_id": 1,
4530   "created_date": "05/07/2020, 10:29:00"
4531 },
  
```

Figure 7: HTTP GET example performed with Postman, self-elaboration

KEY	VALUE
<input checked="" type="checkbox"/> description	this is a test build
<input checked="" type="checkbox"/> commands	pwd; ls
<input checked="" type="checkbox"/> node	2
<input type="checkbox"/> cron_exp	****
<input type="checkbox"/> cron_key	cron5
<input type="checkbox"/> commands_file	Select Files

Figure 8: HTTP POST example to be performed with Postman, self-elaboration

## 4 Results

The results of this thesis basically consist on a REST API (the back-end) capable of automating jobs with CI and CD features and a basic client whose purpose is to ease the usage of the back-end. In the following table all the API entries are shown :

Route	Method	Description
/jobs	GET	Returns a list of all the jobs
/jobs	POST	Creates a job <b>Params:</b> Title, Description
/jobs/<int:job_id>	GET	Returns a job with all info displayed
/jobs/<int:job_id>	DELETE	Deletes a job
/jobs/<int:job_id>/builds	GET	Returns a list with all the builds contained on a job
/jobs/<int:job_id>/builds	POST	Creates a build <b>Params:</b> node, description, commands, cron_exp, cron_key, commands_file
/jobs/<int:job_id>/builds/<int:build_id>	GET	Returns a build with all info displayed
/jobs/<int:job_id>/builds/<int:build_id>	DELETE	Deletes a build
/nodes	GET	Returns a list of all the nodes
/nodes	POST	Creates a node <b>Params:</b> workspace, ip_addr, port, user, password
/nodes/<int:node_id>	GET	Returns a node with all info displayed
/nodes/<int:node_id>	DELETE	Deletes a node
/cron_builds	GET	Returns a list with all the Cron Builds stored in the db
/cron_build/<cron_key>	GET	Returns a Cron Build with all info displayed
/cron_build/<cron_key>	DELETE	Deletes a Cron Build

Table 24: Back-end REST API documentation, source : Python docs

Regarding the front-end, routes are similar to the back-end but with slight differences, such as that there are specific routes for displaying the forms for job creation, build creation, etc.

Route	Method	Description
/	GET	Displays the index page, showing the menu and nav bar
/jobs	GET	Display the list of jobs
/jobs	POST	Internal route called from creat_job in order to perform an operation against the back-end
/jobs/<int:job_id>	GET	Displays the job info and all the builds contained in the job
/jobs/<int:job_id>	POST	Internal route called from creatin order to perform an operation against the back-end
/jobs/<int:job_id>/builds/<int:build_id>	GET	Returns a build with all info displayed
/nodes	GET	Returns a list of all the nodes
/create_job	GET	Displays the job creation form
/create_build/<int:build_id>	GET	Displays the build creation form

Table 25: Front-end routes documentation, source : Python docs

Some screenshots regarding above routes are attached in order to improve the comprehension of the routes :



Figure 9: /



Figure 10: /jobs

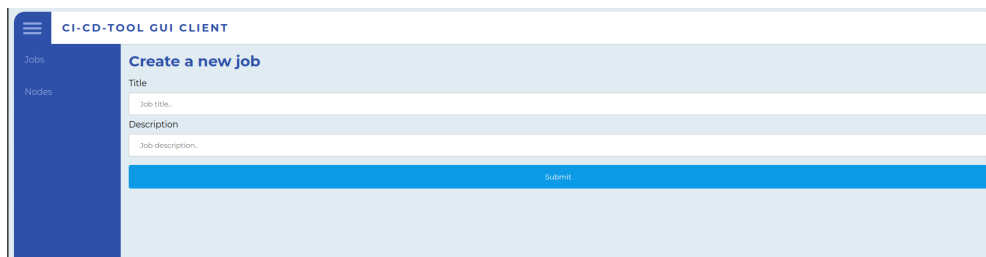


Figure 11: `/create_job`

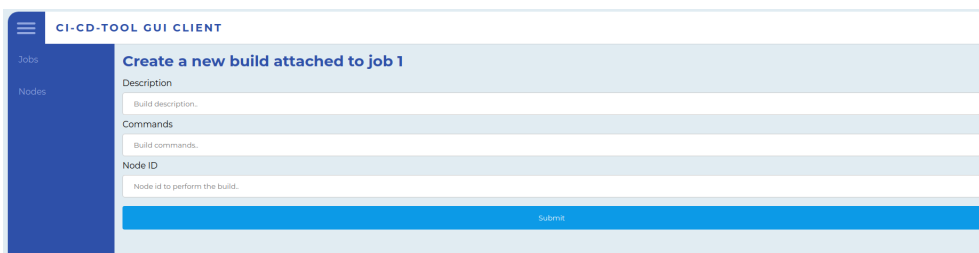


Figure 12: `/create_build/<int:build_id>`

## 4.1 Bringing up the tool

In order to bring up the tool we have to start the back-end and the front-end WSGI bound to the ports desired, this is as easy as placing on the repository directory and execute gunicorn the following way using pipenv as explained before in detail:

```

1 $ cd ci-cd-tool
2 $ pipenv shell
3 $ gunicorn -b localhost:5000 -w 4 src.app:app
4 $ cd ../ci-cd-tool-client
5 $ pipenv shell
6 $ gunicorn -b localhost:8000 -w 4 app:app

```

That way, we could access the client (front-end) by entering `http://localhost:8000` URL on our browser, and we could perform calls to the REST API (back-end) to the `localhost:5000` address.

## 5 Budget

This project consisted mainly in two big tasks, research and development, hence the budget is adjusted to the cost of these two activities, regarding licenses and tools used for the development I have only used Open Source and free community licenses.

The resulting costs based on above explanations are set by the amount of hours worked, 542 hours of dedication in total, setting a price per hour of 9 €/h which is the minimum established by the ETSETB for a degree student under a work agreement between a company and the university for 2019/2020 school season, the budget of this project is :

Hours (h)	Price per hour (€/h)	Total (€)
542	9	4878

Table 26: Project budget

## 6 Conclusions and future development

DevOps is guiding companies through a transformation in the way of proceeding with software development and software delivery processes, this is the main reason why I wanted to get in touch first hand with this field in order to learn as much as possible about the culture. Regarding the general topic of the thesis, DevOps itself, it became very trending in the last decade and has a bright future granted along the lines of innovation and automation, due to the possibility of efficiently treating every project according to its own characteristics.

Designing a DevOps tool with CI and CD features has allowed me to directly get involved with a big part of the relevant concepts as well as strengthening my knowledge and thoughts on the topic and answering the questions I had before getting deeper on the research, by comprehending the DevOps loop putting it in practice with the tool development.

This thesis aims to contribute on automating project tasks in what refers to CI and CD processes. Referring to the future development of the tool, as it will be used for some projects, different instances of the repository will be created with specific features required by each project, even though it is designed for being as much general as possible. Also, some general features are thought to be implemented in a future such as direct git and other tools integration, by using a DSL (Domain Specific Language), which allows performing common operations such as working on the file-system or cloning repositories, etc, by using DSL commands on the pipelines. The goal of the tool is not to replace any other automation tool but to allow the parallel usage along with other services and reduce complexity, as this tool is easy to deploy and very lightweight.

On a personal note, the development of this thesis and my previous experience on the field has led me to focus my professional and labor career on DevOps, as it is a field where you can always learn new techniques of automation, new tools to work with, and there are also lot of events held annually regarding the topic in addition to the investment that big companies (such as Redhat or Amazon) and big open source projects (such as Docker or Kubernetes) are doing in order to stimulate, encourage and standardize the DevOps culture.

Researching and reading authors thoughts on the topic along these months have been very satisfying, I hope the reading of the thesis is also didactic for everyone who has not been introduced yet to DevOps culture, so you can get the basics on the topic and hopefully develop interest on the field, as I said before, there is a bright future on this field and companies are starting to compete for hiring DevOps engineers.

## Bibliography

- Brenn (Feb. 2019). *Noobs Guide: Continuous Integration Continuous Delivery*. Medium. URL: <https://medium.com/@brenn.a.hill/noobs-guide-continuous-integration-continuous-delivery-continuous-deployment-d26ac4f2beeb> (visited on 06/29/2020).
- Daniels, Jennifer Davis Ryn (2016). *Effective DevOps*. O'REILLY. ISBN: 978-1-491-92630-7.
- DevOps: What It Is and Why It Matters* (2017). Toptal Insights Blog. URL: <https://www.toptal.com/insights/innovation/what-is-devops>.
- Farley, Jez Humble David (2010). *Continuous Delivery - Reliable Software Releases Through Build, Test And Deployment Automation*. Addison-Wesley. ISBN: 978-0-321-60191-9.
- Geerling, Jeff (2018). *Ansible for DevOps*. Leanpub. ISBN: 978-0-9863934-0-2.
- Gene Kim, Kevin Behr George Spafford (2013). *The Phoenix Project*. IT Revolution Press. ISBN: 978-0-988-26250-8.
- Gene Kim Jez Humble, Patrick Debois and John Willis (2016). *The DevOps handbook*. IT Revolution Press. ISBN: 978-1-942788-08-9.
- Hüttermann, Michael (2012). *DevOps for Developers*. Apress. ISBN: 978-1-4302-4570-4.
- Kevin Behr Gene Kim, George Spafford (2005). *The Visible Ops Handbook*. Information Technology Process inst. ISBN: 978-0-9755-6861-3.
- Michel Goossens, Frank Mittelbach Alexander Samarin (2011). *Collaborative DevOps with Rational and Tivoli*. IBM Corporation.
- Moe, Myint Myint (2019). *Comparative Study of Test-Driven Development (TDD), Behavior-Driven Development (BDD) and Acceptance Test-Driven Development (ATDD)*. University of Computer Studies, Hpa-An, Kayin State, Myanmar.
- MongoDB vs SQLite: What are the Differences?* (Dec. 2019). CodeClouds. URL: <https://www.codeclouds.com/blog/the-differences-between-mongodb-and-sqlite/> (visited on 06/29/2020).
- Ohara, Dave (2012). *Continuous delivery and the world of devops*. GigaOM Pro.
- PEP 8 - Style Guide for Python Code* (2013). Python.org. URL: <https://www.python.org/dev/peps/pep-0008/>.
- Types of Software Testing - GeeksforGeeks* (Aug. 2017). GeeksforGeeks. URL: <https://www.geeksforgeeks.org/types-software-testing/>.
- Types of Software Testing: 100 Examples of Different Testing Types* (Sept. 2019). Guru99.com. URL: <https://www.guru99.com/types-of-software-testing.html>.
- What Is DevOps?* (Jan. 2019). the agile admin. URL: <https://theagileadmin.com/what-is-devops/>.
- Wright, Graham (July 2018). *Continuous Integration (CI)*. Medium. URL: [https://medium.com/@gwright\\_60924/continuous-integration-ci-e81032bb8502](https://medium.com/@gwright_60924/continuous-integration-ci-e81032bb8502) (visited on 06/29/2020).



## Glossary

- API** Acronym for Application Programming Interface. 11
- CD** Acronym for continuous deployment, a software practice which consists in making automatic continuous deployments of a project. 10
- CI** Acronym for continuous integration, a software practice which consists in making automatic continuous integration of a project. 10
- DSL** Acronym for Domain Specific Language, a language specialized only for an application domain. 38
- IaC** Acronym for Infrastructure as Code, process of managing infrastructure through definition files. 17
- RDBMS** Acronym for Relational Database Management System. 20
- SQL** Acronym for Structured Query Language, a domain specific language for managing relational database systems. 23
- SSH** Acronym for Secure Shell, a protocol for accessing machines remotely. 24
- SW** Acronym for Software. 1
- TDD** Acronym for test driven development, consists on creating tests from a developer's perspective and write code in order to make tests work. 32
- VCS** Acronym for Version Control System. 17
- WSGI** Acronym for Web Server Gateway Interface, a calling convention for web servers to forward requests to web applications. 25