

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN**



**MÁSTER UNIVERSITARIO EN
INGENIERÍA DE TELECOMUNICACIÓN**

TRABAJO FIN DE MASTER

**Design and Implementation of a ChatOps Environment using
the Framework Rasa**

IGNACIO CERVANTES VILLALÓN

2020

TRABAJO DE FIN DE MASTER

Título: Diseño e Implementación de un Entorno ChatOps mediante el Framework Rasa

Título (inglés): Design and Implementation of a ChatOps Environment using the Framework Rasa

Autor: Ignacio Cervantes Villalón

Tutor: Carlos Ángel Iglesias Fernández

Departamento: Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente: —

Vocal: —

Secretario: —

Suplente: —

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes



TRABAJO DE FIN DE MASTER

Design and Implementation of a ChatOps Environment using
the Framework Rasa

Enero 2021

Resumen

En la actualidad, es habitual interactuar con una máquina a través de una aplicación de chat para solicitar servicios como asistencia técnica o información sobre un pedido. La mensajería por Internet se ha convertido en parte de nuestro día a día, especialmente en la actualidad donde la interacción personal está limitada. La sociedad se está acostumbrando a comunicarse a través de dispositivos electrónicos tal y como hacemos en persona.

A la vez que avanza la inteligencia artificial y ofrece respuestas cada vez más humanas, interactuar con una máquina se reconoce como una herramienta útil e incluso deseable en algunos casos. Comunicarse con una inteligencia artificial tal y como haríamos con una persona ofrece nuevas posibilidades, y acerca aplicaciones complejas a los usuarios. En este contexto, los llamados *Chatbots*, que consisten en una inteligencia artificial capaz de simular una conversación humana, pueden ofrecer una interfaz amigable para automatizar tareas complejas.

Por otra parte, en los últimos años las compañías de tecnologías de la información se enfrentan a una revolución. Metodologías como *DevOps* han introducido innovaciones en el desarrollo y operación de estas empresas, brindando a los desarrolladores de nuevas formas de trabajar. Incluso si estas innovaciones aumentan la automatización y fiabilidad, requieren especialización por parte del usuario para poder utilizarlas.

El objetivo de este TFM es desarrollar un caso de estudio de un chatbot integrado en un entorno DevOps, lo que se conoce como *ChatOps*. Para ello, se propone un caso práctico haciendo uso del chatbot de código abierto Rasa. Mediante técnicas de *Machine learning* para entrenar sus modelos de conversación, el objetivo será permitir la gestión del entorno DevOps a través de la conversación con el chatbot. Para conseguirlo, se va a desplegar un entorno DevOps instalado en contenedores y conectado con el chatbot.

El desarrollo se va a llevar a cabo mediante Rasa para implementar el chatbot, Python 3 como el principal lenguaje de programación, Jenkins para gestionar el entorno DevOps y Docker para desplegar el entorno de contenedores.

Palabras clave: ChatOps, Chatbot, Rasa, Machine Learning, DevOps, Python, Jenkins, Docker.

Abstract

Nowadays, it is common to interact through instant messaging with a machine to request services such as technical attention or information about an online order. Chatting online is an essential part of our day, especially in times like today when personal interaction is limited. We are getting used to communicating through devices like computers or smartphones just like we do in person.

As *Artificial Intelligence* advances and offers human-like responses, interacting with a machine is recognized as a useful tool and even desirable in some cases. Talking to an AI bot as if it is a person can offer new ways to work, and closes the gap between complex applications and users. In this context, the so-called Chatbots, consistent in an AI capable of simulating a human-like conversation, can offer a friendly user interface to automatize complex tasks and make them simpler.

Moreover, *Information Technology* companies are confronting a revolution in recent years. Methodologies like *DevOps* introduced innovations in development and operations, focusing on giving the developers the independence and freedom to work in more flexible ways. Even if these changes bring automation and reliability to the previous workflows, they also come with complex tools and processes that require specialization.

The objective of this thesis is to develop a case study of a chatbot integrated into a DevOps environment, which is commonly known as ChatOps. In order to do this, we propose using the open-source chatbot framework Rasa. By using machine learning techniques to train its conversation models, the objective is to provide a way of operating a DevOps environment through conversation with the chatbot. To achieve this, a state-of-the-art DevOps environment will be deployed. This environment will consist in a containerized environment connected with the chatbot, which will sent orders based on the user messages.

The development will be done using Rasa as the chatbot framework to make the use case, Python 3 as the main scripting language, Jenkins to manage the DevOps environment and Docker to build a containerized environment.

Keywords: ChatOps, Chatbot, Rasa, Machine Learning, DevOps, Python, Jenkins, Docker

Agradecimientos

Quiero agradecer a todos los que me han acompañado en la realización de este Máster. Empezando por los profesores y el personal de la Escuela, que desde el primer día me han hecho ir ilusionado a estudiar. También quiero agradecer a mis compañeros de clase, ahora amigos, que hicieron más llevaderas las jornadas de prácticas y biblioteca.

Muchas gracias a mi tutor Carlos Ángel Iglesias por orientarme y motivarme en la realización de este proyecto. Tu apoyo ha sido fundamental para seguir adelante y poder estar escribiendo estas líneas.

A mi familia, por apoyarme y creer en mí desde la distancia, aunque todavía no haya sido capaz de hacerles entender en qué consiste este proyecto (especialmente a Rocko).

Muchas gracias a mis amigos y a Marta por haber estado siempre ahí y haber servido de apoyo en estos tiempos extraños. Quiero agradecer especialmente a Fernando, a Jose Luís y a Abraham, que han sido tan distractivos como fundamentales estos meses.

Gracias también al lector por tomarse el tiempo de leer este Trabajo de Fin de Máster, que me ha hecho crecer como ingeniero.

Contents

Resumen	VII
Abstract	IX
Agradecimientos	XI
Contents	XIII
List of Figures	XVII
1 Introduction	1
1.1 Context	1
1.2 Project goals	2
1.3 Structure of this document	2
2 State of Art	5
2.1 Introduction	6
2.2 Development cycle	6
2.2.1 Development methodologies	7
2.2.1.1 Waterfall model	7
2.2.1.2 V-model	8
2.2.1.3 Scrum	9
2.2.2 DevOps	10
2.3 Cognitive computing	13

2.3.1	Cognitive computing properties	13
2.3.2	Use Case of Cognitive Systems	14
2.4	Chatbots	15
2.4.1	Chatbot Design Techniques	16
2.4.2	Chatbot examples	17
2.4.2.1	Hubot	18
2.4.2.2	Errbot	18
2.4.2.3	Rasa	19
2.5	ChatOps	19
2.5.1	ChatOps examples	20
2.5.1.1	IoT environment	20
2.5.1.2	A classroom chatops environment	22
2.6	Use case: A ChatOps development environment	23
2.7	Conclusions	24
3	Enabling Technologies	27
3.1	Introduction	28
3.2	Automation tools	28
3.2.1	Telegram	29
3.2.2	Rasa	30
3.2.3	Gitlab	33
3.2.3.1	Git	34
3.2.4	Docker	35
3.2.5	Jenkins	36
3.3	Conclusions	38
4	Architecture and Methodology	39

4.1	Introduction	40
4.2	User interactive side	42
4.2.1	Telegram	42
4.2.2	RASA Open Source	45
4.2.2.1	ChatOps Agent	46
4.2.2.2	RASA NLU	48
4.2.2.3	Rasa Core	53
4.2.2.4	Action server	60
4.3	Backend side	65
4.3.1	Jenkins	67
4.3.2	Deploy agent	71
4.3.3	Development server	81
4.4	Conclusions	83
5	Case study	85
5.1	Scenario overview	86
5.2	Development server deploy	87
5.3	Package installation	90
5.4	Conclusions	93
6	Conclusions	95
6.1	Introduction	96
6.2	Conclusions	96
6.3	Achieved Goals	97
6.4	Problems faced	98
6.5	Future work	98
A	Project impact	101

A.1	Context	102
A.2	Social Impact	102
A.3	Economic Impact	102
A.4	Environmental Impact	103
A.5	Ethical Impact	103
B	Project budget	105
B.1	Hardware Expenses	106
B.2	Software Expenses	106
B.3	Payroll Expenses	106
B.4	Indirect Expenses	106
B.5	Total expenses	107
	Bibliography	108

List of Figures

2.1	Waterfall model	8
2.2	V-model stages	9
2.3	Generic DevOps cycle diagram [62]	11
2.4	Person using Microsoft Hololens [57]	15
2.5	Conversation between a person and ELIZA [76]	16
2.6	Generic components of a chatbot [1]	17
2.7	Hubot logo [19]	18
2.8	Errbot logo [21]	18
2.9	IoT system architecture [50]	22
2.10	Architecture of LTKA-Bot [62]	23
2.11	High level design of the use case	24
3.1	Telegram logo [80]	29
3.2	Rasa logo [44]	30
3.3	Incoming message processing [37]	32
3.4	Gitlab logo [20]	33
3.5	Typical local Git workflow [56]	34
3.6	Docker logo [31]	35
3.7	Jenkins logo [48]	37
4.1	Architecture of the project	40
4.2	Telegram web user interface	43

4.3	Telegram webhook message flow [85]	44
4.4	Interaction between Rasa components.	45
4.5	Detailed Rasa architecture [40]	46
4.6	Ngrok console user interface	48
4.7	Representation of vector space used to classify the intents [66]	52
4.8	Steps taken by Rasa to respond a message	54
4.9	Interactive learning user interface	56
4.10	Representation of two iterations of TED policy [84]	59
4.11	Sequence diagram for the Python Jenkins build job process.	64
4.12	Backend section of the architecture	66
4.13	Jenkins web user interface.	68
4.14	Configuration flow from Rasa Action server towards Development Server.	83
5.1	Search made to found <i>ChatopsAgent</i> bot	88
5.2	Chat box used to send messages.	88
5.3	Message used to launch the development server.	89
5.4	Intent recognition from the user message.	89
5.5	Jenkins interface after executing <i>deploy_server</i>	90
5.6	Message replied by Rasa	90
5.7	Checking server status	90
5.8	Response from Rasa after installing <i>pytest</i>	91
5.9	Package list after installing <i>pytest</i>	92
5.10	Installed packages checked in development server.	93

Introduction

1.1 Context

Nowadays, it is common to interact through instant messaging with a machine to request services such as technical attention or information about a brand. Chatting online is an essential part of our day to communicate with our friends, family or work partners, especially in times like today when personal communication is restricted. We are getting used to communicating through devices like computers or smartphones just like we do in person.

As *Artificial Intelligence* (AI) advances and offers human-like responses, interacting with a machine is recognized as a useful tool and even desirable in some cases [83]. Talking to an AI chatbot as if it is a person can offer new ways to work, and closes the gap between complex applications and users.

Related to this, *Information Technology* (IT) companies are confronting a revolution in recent years. Methodologies like *DevOps* [8] changed the rules of development and operations, focusing on giving the developers the independence and freedom to work in more flexible ways. Even if these changes bring automation and reliability to the previous workflows, they also come with complex tools and processes that require specialization [52].

In this scenario, approaching complex environments with easy to use tools is desirable, as it may reduce the points of failure and improve productivity. This is the goal of *ChatOps* [3], a new paradigm that incurs in interacting with complicated environments via an AI through chatting. Among the advantages of this approach, allowing the users to manage an environment from a chat accessible everywhere is revolutionary.

In this project, we try to offer a wide vision of how a ChatOps environment is composed by developing a use case based on a typical IT company workflow. This will incur in building an operative DevOps environment managed by a *chatbot* that interacts with the user to perform operations over the environment.

1.2 Project goals

This project aims to build a state-of-the-art ChatOps environment using some of the most used technologies in the current IT business scenario. To do so, this project will be formed by a Natural Language Understanding (NLU) [73] chatbot capable of extracting information from the user messages, an automation server that centralizes the management of services and a container platform with a development server. Through the integration of these modules, we will build a functional solution that depicts a current ChatOps environment, where the chatbot offers an interactive interface with the backend components.

The main goals to achieve this project are the following:

- Design an operative ChatOps environment.
- Design and train a NLU assistant robot accessible from a public application.
- Deploy a corporate-like DevOps backend.
- Implement an end-to-end operative solution managed through conversation with the assistant.

1.3 Structure of this document

In this section we provide a brief resume of the chapters included in this project. The structure is the following:

Chapter 1 provides an introduction to the context of this project. It exposes the current status of ChatOps and sets the goals to achieve in the project.

Chapter 2 focuses on describing some terms that are treated in this project and exposes the state of the art of some paradigms developed afterward.

Chapter 3 describes the main technologies used in this thesis. It will focus on explaining the features of these tools and the reason to choose them to develop the project.

Chapter 4 depicts the project itself by providing the architecture of the project. It gives an insight into the configuration of each component and how the interaction between them. Furthermore, we will provide an explanation of the developments made for each one of the elements of the project.

Chapter 5 describes some use cases of specific interactions between the user and the ChatOps environment.

Chapter 6 gathers the conclusions found throughout the fulfillment of this project and gives some future lines of work to extend the functionalities of the environment.

State of Art

This chapter offers information about some of the concepts that are treated in this project. To elaborate on these concepts, several solutions that use chatbot-integrated development environments are going to be described and explained, focusing on the architecture. Finally, a high-level design of the use case is depicted.

2.1 Introduction

Nowadays, software development is more than ever an iterative process. In an ever-changing environment, developing state-of-the-art applications that run in different systems such as smartphones, *Internet of Things* (IoT) [74] devices or the traditional computers require a fast adaption and continuous updates to keep complex services up. It is more important than ever keeping pace with the changes in the market and the appearance of bugs in projects that may have hundreds of developers.

In this context, the methodology of work is essential to get the results fast and as error-free as possible. In recent years, many new techniques in project planning and development became popular due to the results and scalability of their solutions.

In this chapter, we are going to explain how software development evolved from the beginning to the newer ways of development. First of all, we will take a look at the traditional approach of software development, taking focus on its flaws and problems.

Secondly, we are going to analyze the modern approach of development and maintenance of software, known as DevOps. To picture this, we are going to take a deeper look into the different parts of it and its advantages.

In third place, we are going to focus on the main theme of this project: the implementation of a chat robot into a DevOps environment. To do so, we are going to give a brief resume on cognitive systems and chat robot technology, explain the implementation of some solutions already deployed and functional, and how the bot helps to manage the environment.

Finally, we will propose a use case that will be developed in the execution of this project.

2.2 Development cycle

As complex as software development is, the *Software Development Life Cycle* (SDLC) [55] always has been discussed and studied. It is vital to follow practices and models that are sustainable and allows the developer to not lose precious developing time and resources.

In this section, we are going to explain some of the most used methodologies by their advantages and flaws, from the more simple models to the state-of-the-art ones. Based on this, we are going to deepen in the basics of the most modern models that we are going to use.

The objective of this section is to provide a wider vision about the evolution of the SDLC techniques, show some examples and how choosing the right methodology for a project can affect the final application.

2.2.1 Development methodologies

When developing software, planning turns out to be as important as having the right resources. Since software development is a complex process with many steps, in the last decades a significant effort has been invested in finding good methodologies to achieve the most efficient development.

As software development and engineering evolved over the years, many new methodologies appeared and offered new ways of planning. We can differentiate over two different categories:

1. *Classical software engineering methodologies*: These methodologies can be described as monolithic and heavy plan-driven. Classic techniques require a heavy upfront requirements definition, documentation and planning, as they are based on phases that require to be completed before passing to another.
2. *Agile methodologies*: Often called light-weight or agile, these methodologies follow the 12 agile principles described by Cockburn [24]. These methodologies adapt quickly to changes in requirements and are based on constantly reviewing and delivering results.

To compare these two main approaches of development models, we will describe some of the most recognized. This way, we can see the differences between them and see how the methods evolved from the older ones to the most modern techniques.

For the classic SDLC techniques, we will explain the *Waterfall model* and its evolution, *V-model*. For agile methodologies, we will introduce *Scrum*. Then, we will show how modern approaches led to the DevOps paradigm.

2.2.1.1 Waterfall model

The Waterfall model [63] is a classical methodology that consists of several linear and sequential phases, where each phase depends on the produced deliverables of the previous task.

This model dates from 1956 when Herbert D. Benington described the usage of these

phases in software engineering, but it was formally described for the first time in 1970 by Winston W. Royce [65].

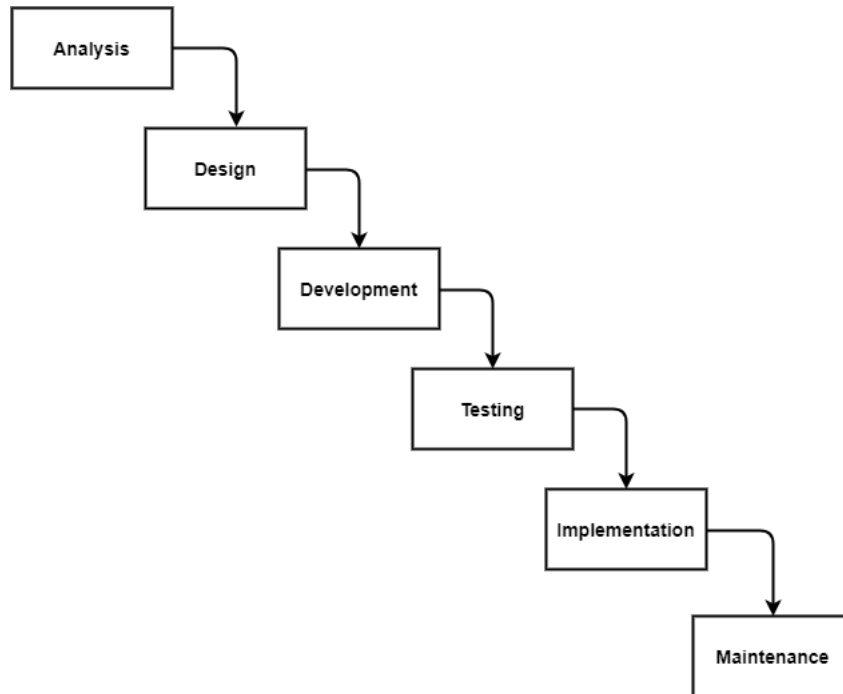


Figure 2.1: Waterfall model

The Waterfall model needs to define clearly all the requirements before passing to the next phase of design, which makes this model very straight and easy to implement and not very resource-requiring. Also, as documentation is made before passing to the next stage, the quality of the development is ensured.

On the other hand, the linearity of this model makes it not very adaptable to changes in requirements, forcing them to be applied in another development workflow. Also, as testing is made only in a phase after the development phase, many errors and bugs in the development may be detected late, which makes them harder to manage.

2.2.1.2 V-model

The V-model (from Validation & Verification model) [49] is a traditional model that consists of an extension of the waterfall model. Even if their characteristics and stages are similar and both are sequential, the v-model is heavily focused on testing and every stage has its testing part.

The model's first stages are similar to the ones in the waterfall model before reaching

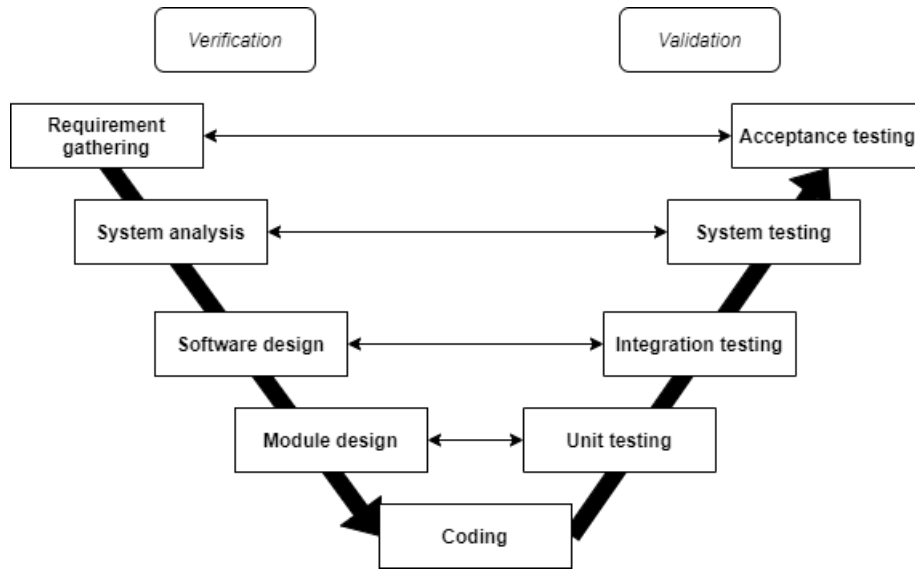


Figure 2.2: V-model stages

the coding step. After coding, there are several testing stages instead of the unique testing phase that the waterfall model has. Each of these testing phases is related to another phase before coding, so if there are any problems detected in one of the testing stages there is a counterpart to solve it instead of doing the whole process again. This way, the development is dependant on testing and makes it more adaptable to errors and requirement changes.

The counterpart of this model is that is more complex than waterfall, and even if it is more adaptable it is still very rigid. Also, like waterfall, this model has periodical reviews and documenting after each stage, so holding more phases makes necessary to have more resources and people to deliver the results fast [7].

2.2.1.3 Scrum

Scrum [77] is an agile framework based on roles and steps to manage and control the software development process. It can be designed as a combination of an iterative model and an incremental one, as the builds of the software are successive and incremental in content. It is one of the most popular SDLC technique based on agile, but its characteristics make it a model not suitable for every occasion [72].

Scrum's workflow consist of the collaboration between various agents involved in the development:

- The *Product Owner*, that manages the results and takes the decisions about the product.

- The *Scrum Manager*, who leads the development team and eliminates the impediments on the development
- The *Scrum team*, consisting of a multi-functional team formed by developers, testers and experts in various fields required in the development.

These agents work in time units called *sprints*. A sprint lasts from 1 to 3 weeks, and the tasks for the sprint are noted on the sprint backlog and can't be modified. At the end of the day, a daily scrum takes place to talk about the progress for that day. At the end of the sprint, the objective is to have a potentially deliverable product. That product is shown to the product owner in a sprint review, where the problems faced in the sprint to get a retrospective vision.

Scrum highlights are the communication between the team and the planning schedules, which gives developers the freedom to discover different ways to get to the solution. Also, as it is based on daily meetings and iterative releases, if requirements change they can be adapted in the next sprint.

However, Scrum is a complex methodology that requires specialization. The main roles must have people formed in agile methodologies and Scrum to be effective, and the development team should have notions to follow the whole process. Also, it is most functional in bigger teams, and not so worthy in smaller teams where simpler SDLC techniques could be easier to implement [68].

2.2.2 DevOps

In the last section, we talked about some of the different SDLCs and how they manage the different phases of development. We noticed that the traditional approach of software development was linear and has separated functions with separated teams performing each task, differentiating development and testing clearly.

Furthermore, we also commented on the *agile* SDLC by talking about Scrum, which implied a more homogeneous development team with shorter development cycles, integrating into each cycle continuous testing and decision-taking with the upper layers of the team that lead to faster releases.

Nowadays, the software development tendency is to use this second approach. Development is shorter in time and has continuous incremental releases. To achieve this, testing and deployment must be fast enough to keep up with the development. These necessities lead to the paradigm shift that is DevOps [54].

DevOps can be described as “a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality” [8]. These practices are associated with including new technologies, automation and faster testing in the current development cycles.

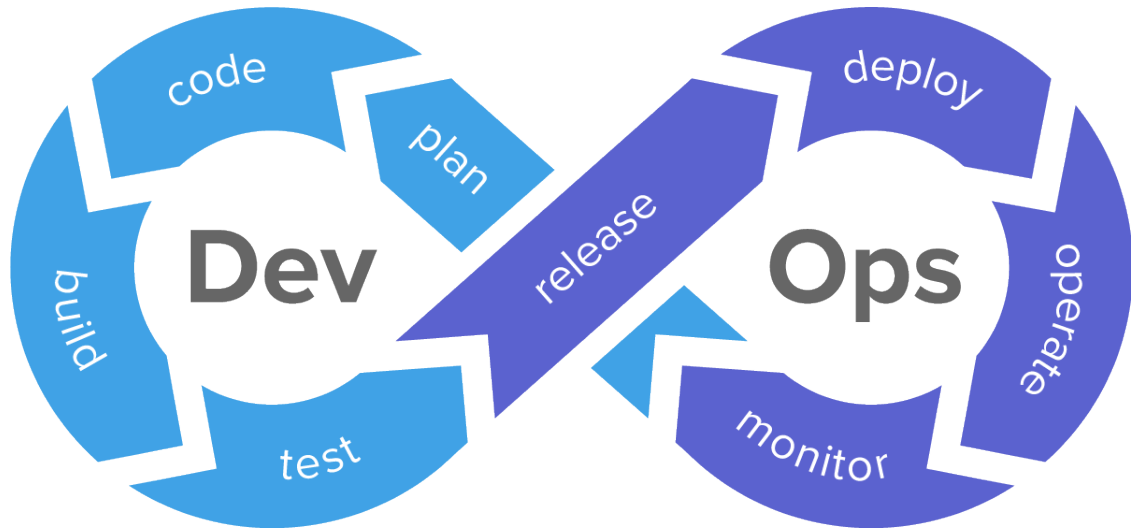


Figure 2.3: Generic DevOps cycle diagram [62]

As DevOps can be seen as a culture shift towards performance [14], integrating the new methodologies into the current development workflow can be challenging. Some of the principal challenges that a DevOps project faces are the following:

- Breaking complex architectures into smaller independent pieces.
- Maintaining an environment that provides visibility to what is deployed, with its versions and dependencies.
- Keeping a pre-staging development environment and production environment.
- Link the traditional figures of development and operations.

The objective behind fulfilling these challenges is to offer **continuous integration and continuous delivery (CI/CD)**. This is achieved by automation and pipelines to test, deploy and monitor each new release automatically.

To achieve CI/CD, the usage of tools is mandatory. These tools meet different goals, such as allowing teams to work coordinated, enabling continuous delivery through automation of processes and maintain reliability in the software. However, the tools are not the main core of DevOps but a way for the team to facilitate the development efficiency and sustainability.

As the tools available have different characteristics and functions, we will differentiate them by their usage [52]. The tools categories are the following:

- *Tools for knowledge sharing:* As DevOps focuses on different departments sharing knowledge, tools that make this knowledge reachable are essential. These tools focus on letting developers communicate and work together in common resources without having to manage conflicts. Some of these tools are Trello [2], which allows users planning projects or chats like Rocket Chat [10].
- *Tools for source code management:* Also known as *Version Control System* (VCS), these tools intend to allow collaboration among developers when writing code. To do so, these tools allow each developer to develop their own code and facilitate pulling together the code by solving conflicts. They also manage the difference in versions to keep the code sustainable. The most important examples are Git and platforms that wrap it, like Gitlab [20] or GitHub [35].
- *Tools for the build process:* The goal of these tools is to enable the *Continuous integration/Continuous delivery* concepts by giving information and automation to the developer. Some of the functions that implement is performing unit-test on the code, check the code quality or give the user a graphical environment to perform the build of the projects. Examples of these tools are SonarQube [47] or Maven [17].
- *Tools for continuous integration:* These tools orchestrate actions that allow implementing deployment pipelines, that consist of stepped executions to perform a full deployment of a system. Generally, a user interface is given that simplifies the process and gives logging about how the process went. The most important tools of this kind are Jenkins or Gitlab CI [34].
- *Tools for deployment automation:* These tools focus on providing *continuous delivery* to the DevOps environment. They allow the developers to execute frequent and reliable deployments by giving them platforms to deploy and give additional features such as high availability or redundancy. In this category, we count with platforms such as Docker [31], which allows users to make containerized deployments, or Openstack [45].
- *Tools for monitoring:* To keep track of operation, monitoring tools provide information such as stress metrics, log management or give alerts about performance. Some examples of these tools are Nagios [15] or Prometheus [4].

2.3 Cognitive computing

As artificial intelligence and new ways of programming grow, the principal companies have found new advantages in their usage. IBM, Microsoft or Facebook are making a great investment in these technologies to make their companies more relevant, and take a leap to cognitive applications. *Cognitive computing* is a term that has been in the sector for years with different definitions. As stated by the Cognitive Computing Consortium [12], a cross-disciplinary group of experts, cognitive computing may be defined as:

Cognitive computing makes a new class of problems computable. It addresses complex situations that are characterized by ambiguity and uncertainty; in other words, it handles human kinds of problems. In these dynamic, information-rich, and shifting situations, data tends to change frequently, and it is often conflicting. The goals of users evolve as they learn more and redefine their objectives. To respond to the fluid nature of users' understanding of their problems, the cognitive computing system offers a synthesis not just of information sources but of influences, contexts, and insights. To do this, systems often need to weigh conflicting evidence and suggest an answer that is best rather than right.

Unlike current computing paradigms, the cognitive computing model infers some data that is out of scope for basic computing. This way, cognitive computing makes context computable, taking account and using some state-of-the-art technologies like machine learning, natural language processing or feature engineering. In other words, as the Cognitive Computing Consortium states, cognitive computing applications suggest an answer that is best rather than right.

To sum up, cognitive computing aims to endow computer systems with the faculties of knowing, thinking and feeling [22] to close the gap between people and the informatics environment. For this reason, it is the basis of technologies like virtual assistants or chatbots. However, it is thought that new applications of cognitive computing will appear in other fields of knowledge.

2.3.1 Cognitive computing properties

As the Cognitive Computing Consortium states, there are some features that a cognitive system has to fulfill:

- **Adaptive:** They must be capable of learning as information and goals evolve. They must resolve ambiguity and tolerate unpredictability. Also, they must be engineered

to feed on dynamic data in real-time, or near real-time.

- **Interactive:** A cognitive system must be easy to interact with, so the users can define their needs and requirements in a simple way. They may also be capable of interact with other actors and devices as other users, Cloud services or online applications.
- **Iterative and stateful:** They must be capable to aid the user in defining a problem by asking questions or finding additional source input if a problem statement is ambiguous or incomplete. They must take into consideration previous interactions in a process and return information that is suitable for the specific application at that point in time
- **Contextual:** They must be capable of understanding and extract contextual elements of the information, such as meaning, syntax or time. Also, they may be able to obtain information from various sources, structured or not, as well as sensory inputs if the system supports them.

2.3.2 Use Case of Cognitive Systems

As cognitive systems advance, more and more applications that uses them appear and their relevance rises. Some of the most relevant use cases are:

- **Virtual Private Assistant (VPA):** VPAs [51] are *Artificial Intelligence* applications that assist the user utilizing their devices. They act as an “human-like” interface to perform some tasks, launched by the user by giving orders to the assistant via voice or text.

Most VPAs are present in several devices and offer services such as access to device functions as making a phone call or writing a message, appointment management or domotic control, and they are linked to personal accounts to take into consideration user’s profile and preferences. Some of the most popular are Apple’s Siri, Google Assistant, Microsoft’s Cortana or Amazon Alexa.

- **Augmented reality:** *Augmented Reality* (AR) [71] technology is a variation of virtual environments where the user is able to see the real world with virtual objects composited in it. Therefore, AR saysupplements reality, rather than replacing it [6].

AR applications take information from cameras to put into the captured images 3D models or layers of information that are reactive to the environment and are capable of adapt to it. These applications can run on various devices, being the most typical

smartphones or *Head Mounted Devices (HMD)* such as Microsoft Hololens [58]. Some of the applications of this technology are engineering, 3D design or video games, and are usually consumed by smartphone apps.



Figure 2.4: Person using Microsoft Hololens [57]

- **Face recognition:** Face recognition allows systems to make use of images captured by a camera to identify a person's face. These systems recognize patrons in faces through AI to identify unequivocally a person. This technology is widely used in biometric security authentication systems, such as Apple's Face ID [27], or in social networks like Facebook to tag users automatically by identifying their faces in photographs uploaded by their contacts.

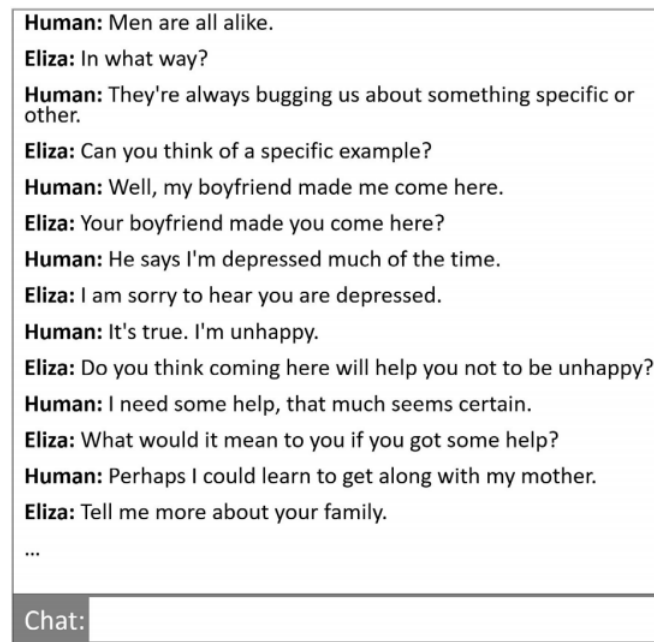
2.4 Chatbots

Since the beginning of domestic computers, there was always an interest in giving the user a friendly interface to use the computational resources. Investing in a clear user interface or designing an attractive and simple website makes the user want to use these resources. As the number of people that use computers rose, it turned out to be a necessity that may decide if an application success or fails. In this context, giving the user a natural language interface to interact with is desirable.

As speech is one of the main forms of communication between humans, the researchers aimed to be inspired by it. Speech interaction is gaining interest in the past few years, as can be seen from developments from *Google*, *Apple* or *IBM*.

Chatterbots (or, from now, *chatbots*) [13] are applications designed to imitate human-like conversation. From the simplest models to modern AI-powered, the goal is to offer a more friendly way to the user to interact with a digital system.

The first recognized chatbot was *ELIZA* [9], a program created by Joseph Weizenbaum at MIT in 1966, that communicated with humans based on hand-crafted scripts. The conversation scope was limited as it was designed using pattern recognition, but enough to keep a realistic conversation, as can be seen in the Fig. 2.5:



Human: Men are all alike.
Eliza: In what way?
Human: They're always bugging us about something specific or other.
Eliza: Can you think of a specific example?
Human: Well, my boyfriend made me come here.
Eliza: Your boyfriend made you come here?
Human: He says I'm depressed much of the time.
Eliza: I am sorry to hear you are depressed.
Human: It's true. I'm unhappy.
Eliza: Do you think coming here will help you not to be unhappy?
Human: I need some help, that much seems certain.
Eliza: What would it mean to you if you got some help?
Human: Perhaps I could learn to get along with my mother.
Eliza: Tell me more about your family.
...
Chat:

Figure 2.5: Conversation between a person and ELIZA [76]

Another milestone in chatbot history was Parry, a chatbot developed by Colby designed to have a paranoid behavior in conversation. Parry was the first chatbot that passed the Turing test, which means that it showed indistinguishable behavior from a human [69]. It was also ruled-based as ELIZA, but Parry had a better language understanding capabilities and models that allowed it to mimic human emotions. For example, it was capable of showing hostility if the anger level in conversation was high.

2.4.1 Chatbot Design Techniques

To be able to give suitable responses to user inquiries, some programming skills and techniques must be used to program a chatbot. In this section, we are going to briefly explain them, as well as giving insight to the different parts of a chatbot to understand its functions.

As chatbots evolve into more realistic conversational partners, the complexity in programming them also multiplies. For this reason, designing a chatbot require identifying its main parts. A chatbot can be classified into three main parts [1]:

- **Responder:** This part acts as the interface between the bot logic and the user. Its goal is to pass the data from the user to the next step and controlling the inputs and outputs of the system.
- **Classifier:** This layer connects the responder and the graphmaster, and is able to filter and normalize the inputs from the user to pass it to the graphmaster. It also processes the output of the graphmaster if needed.
- **Graphmaster:** This part is in charge of the logic of the chatbot. It keeps the storage and performs the operations needed to determine the response (i.e. pattern matching)

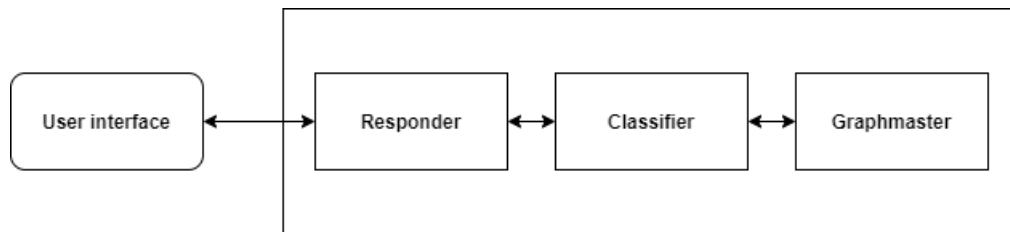


Figure 2.6: Generic components of a chatbot [1]

To design a chatbot, some programming techniques and skills are needed. Some of the most important are:

- **Parsing:** Analysing the input text received from the user and manipulate it (i.e. morphemes recognition) with NLP functions.
- **Pattern matching:** Detecting patterns in the user messages to catch the meaning or the sentiment of them, and respond in consonance.
- **Markov Chain:** It is a stochastic model widely used in chatbots to give responses taking account of previous responses, which means that the next state probability is conditioned by the previous states.

2.4.2 Chatbot examples

As it is a technology in bloom and new applications proliferate, there are many chatbots available. As automation is a topic present in all kinds of environments, there are many companies developing chatbots to rule over complex systems.

To get a global view of the chatbot industry, in this section we will list some of the most relevant options. The following chatbots, however, are focused on helping teams to work together and manage automation flows instead of chatbots designed primarily to interact with the consumer, as it is the scope of this project.

2.4.2.1 Hubot



Figure 2.7: Hubot logo [19]

Hubot [19] is a chatbot framework developed by GitHub. First developed for internal use only, Hubot was rebuilt as an open-source project and available online, so developers can include it in their personal projects.

Hubot is a *node.js* application. It has some built-in functions such as posting images, interacting with different languages or integration with popular apps, but these functions can be expanded by the user. The user can write their own scripts in *CoffeeScript* to expand the functionalities of the chatbot. As it can be deployed on a high variety of chat applications, it is one of the most popular options for deploying a chatbot.

2.4.2.2 Errbot



Figure 2.8: Errbot logo [21]

Errbot [21] is an open-source chatbot framework based on Python [16]. Among its advantages are the capacity to deploy new scripts without to re-deploy the chatbot service and the capacity to script functions using Python, one of the most used scripting languages.

2.4.2.3 Rasa

Rasa [44] is an open-source *Artificial Intelligence* (AI) chatbot framework based on *Natural Language Processing* (NLP). Rasa offers a complete framework to develop and deploy chatbots, with a Natural Language Understanding (NLU) module, dialogue policies and an agent to operate the environment.

Compared with the chatbots mentioned before, Rasa functionality is based on the context of the conversation. Unlike first pattern-recognition chatbots, where the engine searched for specific word sequences to give a response but did not take in consideration the surrounding information of the conversation, contextual assistants can handle a conversation in a more human-like way, making the correct questions to get the information needed to fulfill more complex needs. Rasa will be further explained in Sect. 3.2.2

2.5 ChatOps

In the Sect. 2.2.2, we explained how DevOps introduced changes into the development culture towards a modern and resilient environment where automation is key. We listed the different tools that allowed the developers to communicate between them, make sustainable code, test and deploy efficiently.

However, even if there are loads of tools available, mastering a DevOps environment is a challenge. Additionally to the required knowledge in development and deployment, each tool requires specific knowledge to use it and are complex to operate. To manage a DevOps environment, users must have knowledge over various fields that not every developer has.

To ease the entry barrier, one of the solutions is to give users a more friendly interface to interact with the environment instead of a console or an administrator site. The solution we are going to cover is adding a conversational artificial intelligence capable of managing the environment by chatting with the user, named **ChatOps**.

ChatOps (from *Chat and Operations*) is a paradigm that consists of interacting by talking to a conversational robot (or *ChatBot*) to operate with the environment. That chatbot is deployed in a chat application and connected with typical DevOps tools such as deployment or logging tools to allow users to perform tasks such as deploy or shut down a process by chatting.

Providing a friendly user interface gives the environment more flexibility and allows it to be used by many different user roles. Moreover, it comes with other benefits:

- As the operation is made through conversation with the bot, it simplifies the operation by giving users access to powerful processes without having to deal with complex tools.
- Chatbot operations have to be programmed beforehand, which allows the developers to choose which operations are supported. This way, interacting with the environment is safer as users can't use the tools in an unintended way.
- Gives transparency to operation, as conversations with the bot can be tracked and logged.
- It is versatile, as the chatbot can be reached by different platforms. A user may control a deployment or check the environment by chatting with the bot with a smartphone, without the need of connecting through a PC.

These advantages allow software development companies to improve the efficiency and reliability in the operation, increasing productivity rates [23]. Furthermore, the improvement in transparency and communication allows the team to interact more by using a unified interface and to work in the same direction with ease.

2.5.1 ChatOps examples

In the last section, we explained the characteristics and advantages of a ChatOps environment, and how it can improve the efficiency of a DevOps environment by improving transparency and communication.

In this section, we are going to examine some ChatOps examples. This will allow us to depict how a working environment functionalities, and how the chatbot simplifies the operation of the previous implementations.

2.5.1.1 IoT environment

The following environment is a service built around IoT devices [50]. These devices, majorly sensors and cameras, require connection with a backend server to store and process the data, so downtime must be as low as possible to keep the integrity of the recordings.

Before applying the ChatOps paradigm to the environment, the deployment was done by hand with a script to update the server package over a single server. As the environment has several servers that the cameras connect to, it was common for the cameras to timeout to the server they were connected to as it was deploying, and trying to reconnect to another updating server, causing long breaks in the data history.

Some of the problems deployments had been that the newly upgraded servers joined the production environment immediately, so there was no staging to test that new versions worked without any issues. If any problems happened, the server would have to be stopped and rolled back to the previous version, which inquired in more downtime and more lost data.

To improve the workflow of the environment, the objective was to automate the deployments to lessen the time and errors while having control of them. The final solution introduced the following tools:

- Moved the backend of the environment to a cloud service. This way, it was possible to have a pre-production stage to test the new versions before implementing them in the production environment. The new releases would be deployed as new instances, and a load balancer would point to them making them instantly available.
- Introduced an automation tool for managing and configuring new instances. The chosen tool is *Ansible* [64], an open-source software developed by Red Hat.

Ansible uses *YAML* scripts to describe the configuration that is going to be applied to new instances, allowing to automatize deployments using simple directives.

- To manage and govern the deployment, a ChatBot was included. The chosen bot was *Hubot*. Hubot works listening to messages in a chat and performs actions when a message matches a *listener*, a previously defined pattern. Also, Hubot supports middleware to intercept messages before acting, allowing for example different user authorization to interact with the bot.

In this environment, Hubot was integrating with the instant messaging application *Flowdock* [79].

With these tools, the deployment of the environment consists of two phases. The first one is to create a mirror of the instances currently running by Ansible scripts. With these instances created, the new server package is deployed with an SSH utility and tested to check that the new version works the intended way.

The second phase is pictured in Fig. 2.9. If it passes the tests, the new environment is taken into use by redirecting the load balancer to the new instances. The old environment would be deleted after two hours after taking this step to give time for a possible rollback if the version is faulty.

In this environment, the new ChatOps environment allowed the team to increase the number of updates and the reliability of them. The inclusion of the chatbot gave the

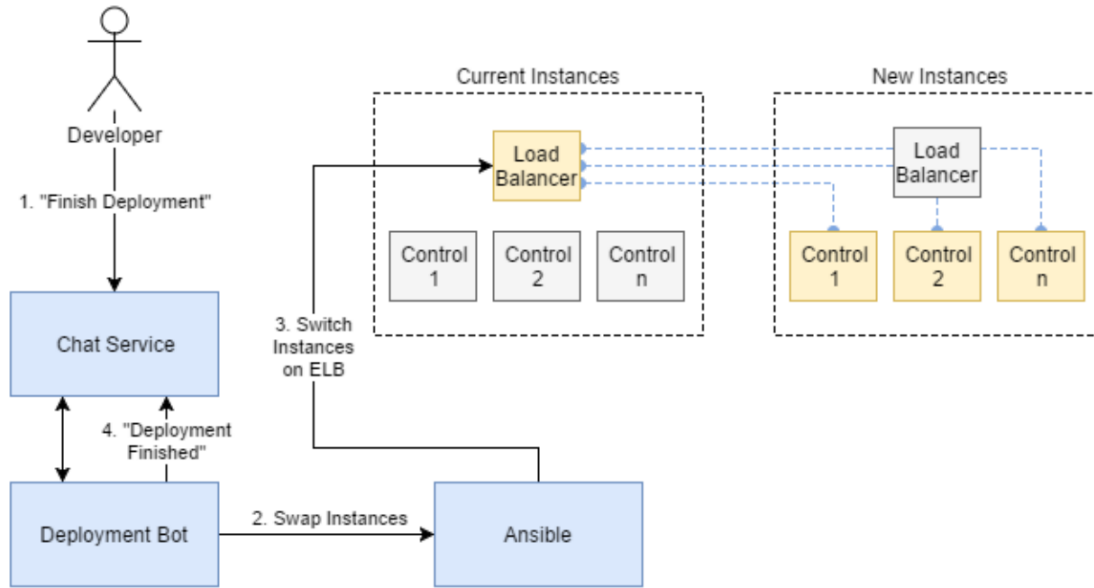


Figure 2.9: IoT system architecture [50]

environment several improvements, such as allowing team members to work from different locations and interact with the environment on the go and gave a common deployment interface that got all the team informed of changes.

2.5.1.2 A classroom chatops environment

Chatops methodology can be applied to different environments and not only in software development, as the automation and simplicity in operations can be useful in different situations.

This is the case of LTKA-Bot [60], a chatbot developed in the *Institut Teknologi of Bandung*. This chatbot is designed to assist the work of the teachers and educators in the college, but not to replace them for teaching. As the authors claim, *LTKABot is not meant to replace real teaching assistants by doing free-form conversation with or answering subject matter questions from the students, but it does provide services related to course activities. These services cover many aspects, ranging from group or task management for students to automatic document generation for reporting and accreditation purposes.*

LTKA-Bot covers 4 kinds of functionalities: planning, instruction-related activities, assessment-related activities and administrative tasks. These activities are available for different kinds of roles such as student, lecturer or administrator and offer them different functionalities depending on them, such as delivering docs or submitting a test as a student or review them as a lecturer. The system architecture is showed in Fig. 2.10:

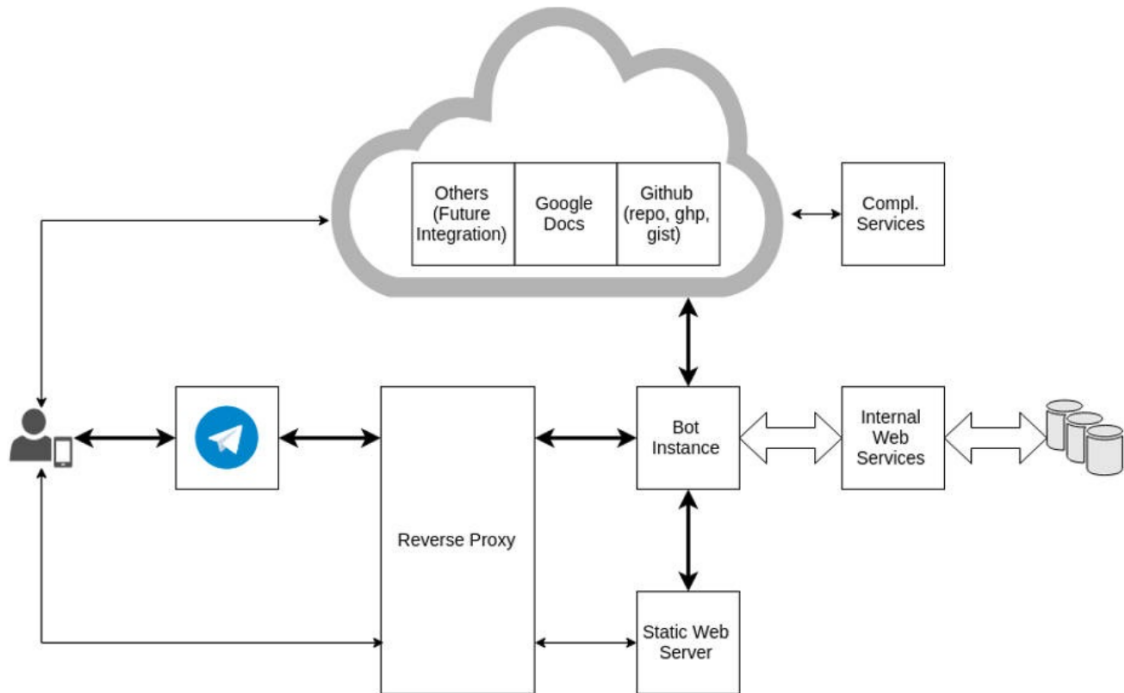


Figure 2.10: Architecture of LTKA-Bot [62]

The chatbot, based on *Hubot*, is deployed behind a reverse proxy and reached by the chat application *Telegram* [80], available on a wide range of platforms. The bot access to internal web services to persist data, and connects to a static webserver to host images and documents. Also, the bot connects to other services such as Github or Google Docs to get some of the data.

LTKA-Bot is currently in use and under active development, giving the teachers the capacity to perform typical tasks in university schools by using a friendly interface like a chat application, and the teachers to automate tasks that usually takes a dedicated platform.

2.6 Use case: A ChatOps development environment

As we have seen in this chapter, ChatOps is a paradigm that can be applied in different kinds of environments to benefit from the automatism and reliability of having an interface between users and the environment. The examples given in the previous section focused on how the chatbot allowed the environments to be more accessible and reliable by giving users a way to operate with the tools from different places and run the commands needed without the complexity of these tools.

In this section, we will describe the use case that will be developed in the document. The objective of this is to show how a ChatOps environment is developed from scratch, what it can perform and the problems faced before reaching the final solution.

The proposed use case pictures a usual use case in development. The user uses a development machine and connects to it to develop software. This machine is prepared with the needed development tools and versions required to perform the job. The machine is maintained by the company and users can access it by their usual computer. As it is a machine needed to perform its job, the maintenance of this machine is critical, and error in updates or deployments may cause the workflow to stop until it's fixed.

To keep the maintenance as simple as possible, the company decides to update the current environment and implement a ChatOps approach. The idea is to allow the user to check the status of the environment, run the machine if it is not and manage the installed packages by chatting with a bot. The high-level architecture can be seen in Fig. 2.11:

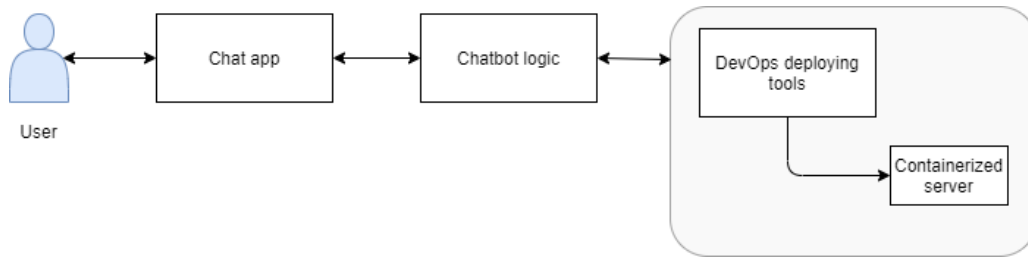


Figure 2.11: High level design of the use case

The user would chat with the chatbot by a chat application. The chatbot would recognize patterns and intents from the user to perform several actions. Then, with the information got from the conversation, the bot would launch automation flows using DevOps tools like the ones explained in the Sect. 2.2.2. These tools, connected to the development server, would perform the desired operations on it and give information about the changes to the chatbot. Then, it will return the user that information via the chat app.

2.7 Conclusions

In this chapter, we introduced some of the terms and concepts that will be used in the development of the project. We pictured the traditional development cycle and paradigms by studying some of the most popular models. After that, we introduced DevOps by detailing its characteristics, tools and advantages over the previous models.

Then, we introduced cognitive computing systems, one of the bases of chatbots. We

described their properties and use cases. Based on this, we detailed some of the key concepts of chatbots design and continued explaining some of the most popular chatbots and their features.

Moreover, we introduced the ChatOps paradigm and the peculiarities of this model. We described some ChatOps developments to study their components, how do they work and the problems that they solve. Finally, we proposed the environment that will be developed in this project to represent a real-world necessity.

Enabling Technologies

This chapter offers a brief review of the main technologies that have made possible this project. First, a brief summary of the software selected is shown. The following sections explain the fundamentals of these packages and others that are used in the execution of this project.

3.1 Introduction

To develop this project, many technologies are involved that need to be described in order to comprehend the architecture. In this chapter, we are going to analyze and explain some of them to offer a wider vision of how the project is achieved.

First of all, we are going to list the types of applications and services needed to get to the solution proposed in the last chapter. This will allow us to decide which tools adapt better to the needs of this project.

Then, information about the applications chosen and how do they work will be served. We will focus on their behavior and their main characteristics to understand how they can help to achieve the objectives of the project.

Finally, we are going to give conclusions about the decision taken and how they will affect the following chapters.

3.2 Automation tools

As we stated in the Chap. 2, modern development environments based on DevOps need tools in order to be deployed. The goal of these tools is to give the user power through automation to perform complex tasks by themselves, from coding to deploying the final solution.

In the previous Sect. 2.2.2, we listed some of the categories where we can group DevOps tools by their characteristics. With these categories, the tools can be chosen using the criteria of what the project needs or the features that the environment will implement.

As the objective of this project is to implement the environment depicted in the Sect. 2.6, some services and tools will be required to fulfill the requirements:

- To interact with a ChatOps environment, the first software the user reach is the chat application. To chose one that is available on different platforms will make our environment more reachable and portable. Because of this, the chat application chosen is *Telegram*.
- The chatbot framework is the key part of our solution. It has to be capable of connecting with the environment to perform the actions demanded by the user and to communicate with the user by chatting to get the information needed.

As there are many solutions in the market it is important to select one that fits the needs of the environment. For this project, we have chosen *Rasa* because it is based on *Natural Language Processing* (NLP), allowing the bot to interact with the user with human-like conversational behavior. Also, as it is open-source and allows the user to write their own scripts and code, it will fit the requirements needed for this project.

- As a ChatOps environment is a cross-technology environment, the project will have different kinds of coding and the development will take time, keeping it safe and managing the versions is important, so source code management will need to be covered. The most used VCS is Git for its simplicity, so in this project we will use *GitLab* as the platform to manage code.
- Also, as the chatbot has to be capable of deploying and managing the environment, an easy-to-deploy platform is relevant to achieve this. The environment will consist of different micro-services, so a light-weighted solution based on containers would fit. The chosen platform is *Docker*, as it is one of the most extended container-based platforms.
- To govern these containers, a continuous integration tool must be put in between the chatbot and the environment. This tool has to be able to operate with these containers to deploy them or to connect with them to manage the services. As there are many solutions to implement automatic pipelines to deploy, we will choose *Jenkins* for its simplicity and for being one of the most popular solutions.

To get a better understanding of how the environment works, the following sections will describe the main features of these tools. This will allow us to comprehend the decisions taken at choosing the tools.

3.2.1 Telegram



Figure 3.1: Telegram logo [80]

Telegram [80] is a free messaging cloud service based on open source. Initially launched for iOS [28] in 2013, nowadays there are available apps for Android [39], iOS, Windows [59], macOS [29] and Linux, and accessible via the website [82].

Among its main features, Telegram offers end-to-end encryption that gives the user security in the communications. It also allows creating broadcast channels without user limits, and groups up to 200.000 users. Unlike other chatting services, Telegram allows identifying users by nickname, so their privacy is preserved.

The messages are synced up with a private cloud, so the information is backed up and the chat history can be accessed from different applications. This allows the user to chat from different platforms indistinctly.

One of the key features is the bot integration [81] in Telegram. There is an available bot API that allows us to deploy a chatbot in Telegram, and integrate it in a chat to interact with it. This way, telegram can be an interesting ChatOps tool as it can be accessed everywhere, and allow the team to interact with the environment from everywhere.

As Rasa supports Telegram by default, it will be the chat application chosen to interact with the chatbot. This decision will also make the environment more accessible by giving the user a trusted chat application available in a variety of operative systems.

3.2.2 Rasa



Figure 3.2: Rasa logo [44]

Rasa [44] is a conversational AI framework for building contextual assistants. Unlike IBM Watson or Google Dialogflow, Rasa is an Open Source alternative, which allows developers to deploy it on a diverse set of platforms and tune it to the user needs. Rasa offers a paid “Enterprise plan” with analytics and support from the company, but the user can get full functionality with the free version.

As we introduced in the Sect. 2.4.2.3, Rasa conversational functionality is based on the context of the conversation. Unlike first pattern-recognition chatbots, where the engine

searched for specific word sequences to give a response but did not take into consideration the surrounding information of the conversation, contextual assistants can handle a conversation in a more human-like way, making the correct questions to get the information needed to fulfill more complex needs.

Rasa architecture is modular, allowing to easily integrate with other systems. As it is coded in Python and makes use of standard libraries such as *scikit-learn* or *spaCy*, its parts can be reused in other projects and implementations. Also, Rasa allows the user to override the built-in libraries and scripts used and use their own, giving assistants more flexibility in the development than their competitors.

The main components of Rasa are the following:

- **Rasa Agent:** The agent acts as an interface of the rest of the modules. When a message arrives at the system, it will reach the agent which will send it to the NLU module to begin the processing. It also implements methods to train the models used by the rest of the components. Moreover, the agent is also capable of receiving and replying to other chat systems supported by Rasa.
- **Rasa NLU:** *Natural Language Processing* module used in the interpreter. It is the part where the messages sent from the user are processed. It can be trained in any language by modifying the components used to train the model. Operations like *tokenisation*, *Parts Of Speech* (POS) annotation or *featurizing* are executed by these components in order to process the entry. These operations are performed through a predefined pipeline that the user can configure to its need.

Rasa NLU extracts the intent that matches the message received from the user. It also extracts the entities if there are any. The message recognition made by Rasa NLU is the first step performed when a message arrives, and the detected intents will be used to decide the next steps performed by the chatbot.

- **Rasa Core:** The core component is the dialogue engine used for building AI assistants. It is built based on machine learning models trained with conversations to decide the actions to perform, instead of classic programming if/else statements.

It is based on *stories*, which are representations of a conversation between the user and the assistant. The stories relate the intents with specific actions that Rasa can perform. These stories will be used to train the conversational model and the dialogue policies used to decide the action to take after an intent is detected. Like other assistants, Rasa can also connect with external services and events.

Aside from using stories to train the model, the administrator can also train it interactively. This is achieved by conversating with the chatbot through the command line or by using Rasa X [42], a toolkit with a user interface. These conversations consist of the developer sending a message to the bot, which will try to detect an intent and asks the user for confirmation. When the conversation ends, Rasa will build training data.

The whole message processing is shown in Fig. 3.3:

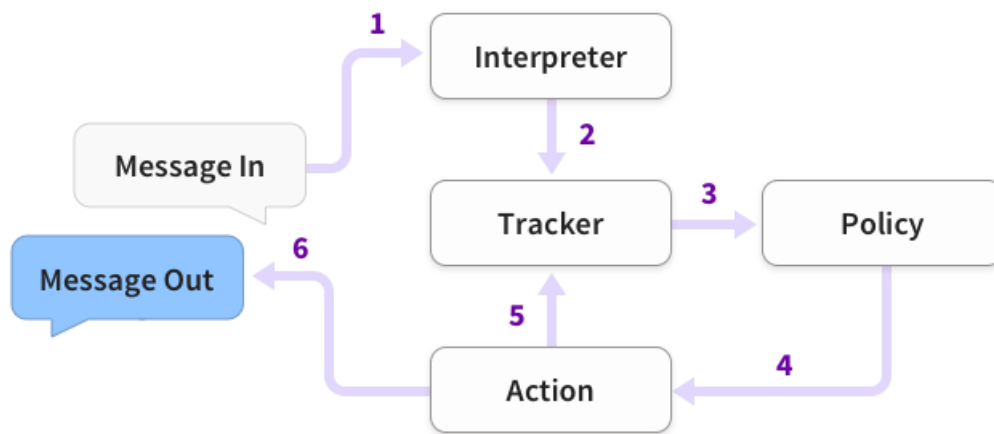


Figure 3.3: Incoming message processing [37]

The steps followed in processing the message are:

- The message is received and passed to the Interpreter, or NLU module. It performs several transformations and operations to convert it into a dictionary with the original text, the intent and entities found in it.
- The object that keeps track of the conversation state is the Tracker, which receives the info that a new message has arrived. This tracker will also keep a log of the detected intents and actions performed by the bot.
- The trained policy receives the current state of the tracker, and in consequence chooses decides which action to take next.
- The chosen action is logged by the tracker, and then sent to the user. This action can be either a message or executing an operation.

The development made to train the different Rasa components is explained in 4.2.2. There, the components and how they interact will be further explained.

3.2.3 Gitlab



Figure 3.4: Gitlab logo [20]

Gitlab [20] is an open-source DevOps lifecycle tool based on Git protocol. It gives the developers not only the habitual *Version Control System* (VCS) features, but also gives the teams the capability to automate *Continuous integration/Continuous delivery* (CI/CD) workflows within the platform.

GitLab offers many capacities to deploy a DevOps environment by itself:

- Works as a VCS based on Git, allowing several developers to work over the same code, keeping versions and facilitating merging each code into a release version.
- Give insight and metrics about the releases and software delivery lifecycle. For example, it can give track of how many issues were created, resolved by whom and the average time of resolution.
- Offers planning management to teams, with tools like kanban boards to help teams organize and align the tasks.
- Allows configuring pipelines to work on the code. For example, GitLab can automate the builds of the code, the integration or perform automated testing. This is configured through pipelines that the user can configure to execute as needed.
- Provides security testing integrated into the platform. By itself, GitLab can perform security automated auditory with Static Application Security Testing (SAST) or Dynamic Application Security Testing (DAST).

As in this project we will use it as a GIT code repository to get certain files used during deployments, we are going to introduce Git to comprehend its benefits.

3.2.3.1 Git

Git [11] is a free and open-source distributed *Version Control System* created by Linus Torvalds in 2005. It is based on tracking the state of files, primarily source code, to coordinate cooperative code development.

Git is based on taking snapshots of the files in a state determined by the user, called *commits*. These commits have information related to them, such as the user that made it, the time or a describing message. Also, each commit has its own unique identifier that identifies it in the whole project.

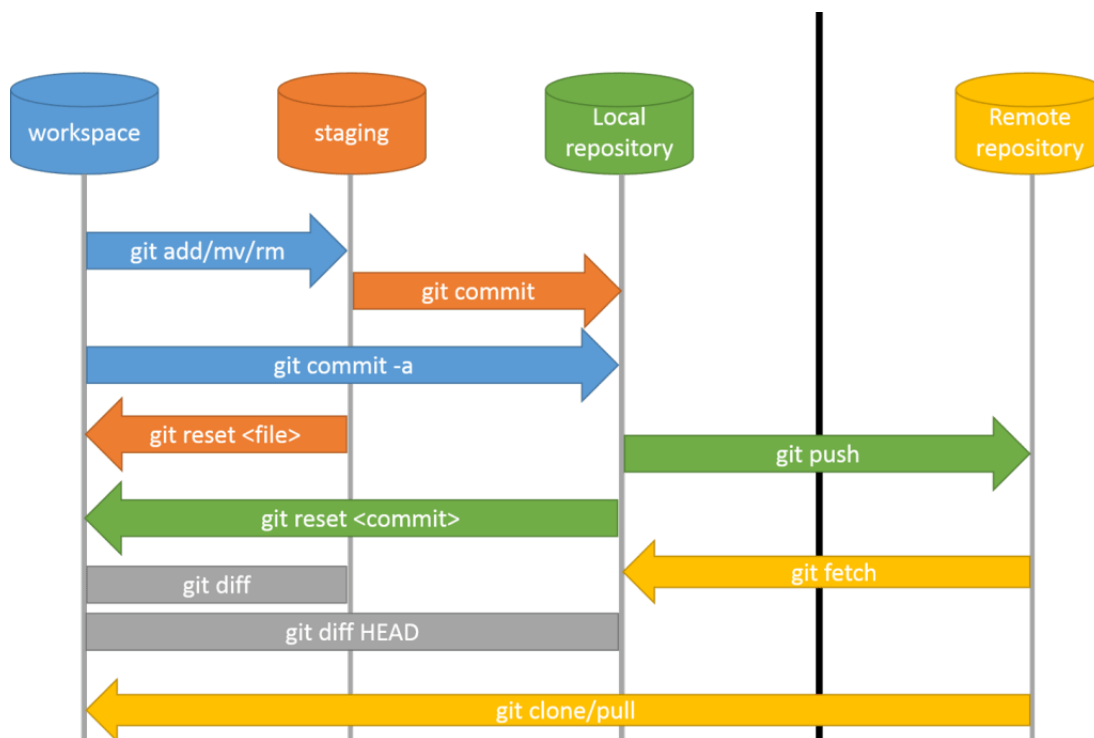


Figure 3.5: Typical local Git workflow [56]

To differentiate between different development stages, Git implements a branching model. Each branch is created from a commit and allows the user to develop the code from that commit without affecting the original branch. This allows each user to have their own development without conflicts, and when the development is done allow them to merge it signaling the differences to avoid conflicts.

The typical workflow when working with git can be seen in Fig. 3.5. Each user has a local repository, composed of its workspace with the local data, an intermediate *staging* stage and the local git repository with branches. The user adds the files from its current workspace to the staging stage, which allows storing the changes into a commit to its local branch.

That local branch is linked with a remote branch located in the Git repository, where the code can be accessed, and the user can push the changes into the remote repository to make them available there. Then, the other users can *pull* or *clone* the changes into their local repositories to have them.

In our project, the code for all the services deployed is available at the Grupo de Sistemas Inteligentes (GSI) GitLab server. It allowed us to keep versions of the code while development was going and store it securely. Also, it is used by some of the tools of the environment to get files needed for the use cases.

3.2.4 Docker

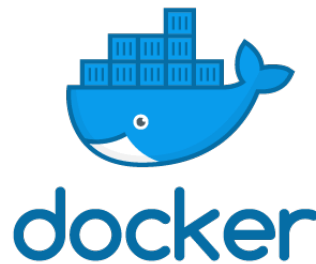


Figure 3.6: Docker logo [31]

As computing grows in capabilities and complexity, new technologies emerge to supply the crescent computing necessity. Virtualization allows building complex systems within a single computer, giving the user the capacity to create virtual networks and interconnected systems quickly.

In this context, container technology make a profound impact on development environments. While virtual machines focus on giving abstraction at the hardware level, containers virtualize the operating system. This difference makes containers a much more lightweight virtualization technology, allowing quick deployments, modifications and rebuilds of services.

Docker [31] is a Platform-as-a-Service (PaaS) virtualization service that allows building a standardized unit of software, called containers. Released in 2013, it meant an evolution from other virtualization technologies like *LXC* [53] or virtual machines, primarily for the possibility of packing into a container a fully functional service with all the needed dependencies.

To manage the containers, the Docker platform integrates an API for managing the

execution and creation. This allows developers to simplify the process of deploying a service and give additional features such as automation or high availability. Some of the advantages of using docker are:

- Allows to rapidly deploy applications, as containers include the minimal runtime requirements. This let to reduce the size and simplifies the architecture.
- Containers are portable, as all the dependencies are bundled into them independently of where they are running. This means that a container image can be deployed in another machine using Docker and executed in the same terms.
- Docker facilitates the sharing of images via Docker Hub [33], a free-to-use library with thousands of images. This makes it easier to pull commonly used containers.

To deploy a Docker container, there are several options. Users can pull an image from a repository like the mentioned Docker Hub or personalize it via configuration files called *Dockerfiles*. These files define how a docker image is built, from the parent image to the commands to run at launch, the files that must be copied to the container volume or the ports to expose to the host. Building these files to an image allows Docker to deploy a container with the options described in the file.

As containers are lightweight and support quick deployments, they allow to deploy services in *high availability* and quickly deploy replicated instances of a service in case of errors. To do so, there are tools like *Docker swarm* [32] or third-party orchestrating tools like *Kubernetes* [5] or the Red Hat service *Openshift* [46], that allows checking the health of containers to keep a high availability environment.

In the use case, we use Docker as the platform to deploy the whole backend side of the environment. We will deploy containers to run the Jenkins instance, a deployment slave managed by Jenkins and the development machine where we will perform the changes and updates the user specifies by the chatbot. Docker allows us to make the solution portable and replicate it, independently of the operative system or the machine where it is deployed.

3.2.5 Jenkins

As development environments get more complex and involve more services, operations like an update may be delicate. A failure in the process may incur long times of rollback to a functional version and redeployment, with losses in time and service. Tools that enable continuous integration can help in providing a reliable way of automating deployments.



Figure 3.7: Jenkins logo [48]

Jenkins [48] is a self-contained open source automation server that can be used to automate tasks related to building, testing and delivering software. Originally named Hudson, it is available in Windows, Linux, macOS or running it in a Docker container.

Jenkins allows incorporating continuous integration in a DevOps environment by launching automatic operations over it. It is based on jobs, where the user indicates the operations to be run. These jobs, also called *freestyle jobs*, can be automatically triggered by external sources such as changes in a *Version Control System* server or changes in the state of services, which allows the environment to quickly adapt to changes.

To ensure that the automation processes are running as expected, Jenkins offers logging about the operations performed, and metrics about the results. It is also capable of sending alerts to the users in case of errors in the realization of the tasks, allowing different user roles (like developer or administrator) to get the needed information depending on them.

To perform the jobs, Jenkins offers different tools that can be extended with the installation of *plugins*. Out of the box, Jenkins allows different functions. For example, it can use SSH to configure machines or to perform tests over the code collected from VCS, but by installing plugins it can acquire functions like creating Docker containers to perform tasks.

Apart from jobs, Jenkins allows configuring *pipelines*, consisting of several tasks performed in order. These pipelines can also be triggered by external changes, and unlike freestyle jobs, allow to break out tasks into smaller stages. Jenkins offers information and logging from each stage and can alter the workflow depending on stage results, making them more adaptable and allowing them to perform more complex tasks keeping the information flow with the user.

Pipelines can be described using a *Jenkinsfile*. These files describe the different stages of a pipeline, how are they called and what they perform. Jenkinsfile also indicates which

agent should run the pipeline, whether it may be any available agent or specify one of a cluster of them. Jenkinsfile allows creating a single source of truth for the pipeline, easing the creation of a sustainable environment where the deploying or testing steps can be tracked and replicated.

Among the benefits of using pipelines, we may highlight the versatility that pipelines give with options like pausing the execution until an action is performed, the information given to users in each stage and the ease of writing a script to automate a whole DevOps environment deploy.

In this project, Jenkins is used within a Docker container. It will communicate with Rasa to perform the actions requested by the user. This will allow us to add an intermediate layer between the chatbot and the environment, with logs and a history of the executions which will help us to simplify the environment.

It will be responsible for performing the operations asked by the user and processed by the chatbot application. To do so, Jenkins will use a deploy agent machine deployed beforehand. This agent will contact the machine and perform the changes or deploy it if needed, simplifying the architecture and allowing us to reduce the time of execution by reusing the same agent instead of deploying them on-demand. The details of the agent will be detailed in the following chapters, as well as the configuration made in Jenkins.

3.3 Conclusions

In this chapter, we have introduced some of the main technologies used in this project and the motivation behind choosing them. As there are many options in the DevOps market, we made our decision by how the tools can inter-operate and how they helped us to reach the objectives fixed in the Chap. 2.

The understanding of the tools depicted in this chapter is necessary to work with the environment proposed in this project. We detailed some of the most important features and characteristics of the main tools to give an insight into the different parts of the environment, which will be described in the following chapters.

Also, we introduced some key concepts of these tools that will be used in the use case. The details given in the previous section will allow getting to know how the environment works at a lower level while keeping an end-to-end vision of the objectives.

Architecture and Methodology

This chapter presents the methodology used in this work. It describes the overall architecture of the project, with the connections between the different components involved in the development of the project.

4.1 Introduction

The main purpose of this chapter is to explain the architecture of the solution developed in this project. To do so, we are going to cover the design of each component, its function, and how it is implemented. Firstly, we will display a diagram to show the different parts of it and how they interact. Then, we will focus on how that modules work in this project and their functions.

In this section, we will present the global architecture of the project, defining the different parts and components that conform the whole system. The system consists of a chatbot developed using the RASA Open Source framework that provides ChapOps functionalities in a DevOps environment. As shown in Fig 4.1, the DevOps environment consists of a dockerized development server managed by a Jenkins instance. In particular, the messages processed by Rasa will trigger operations performed on the development server by Jenkins. End users (i.e. DevOps developers and operation responsible users) can access the ChatOps functionality through the Telegram messaging system. A more detailed description of these components is provided below:

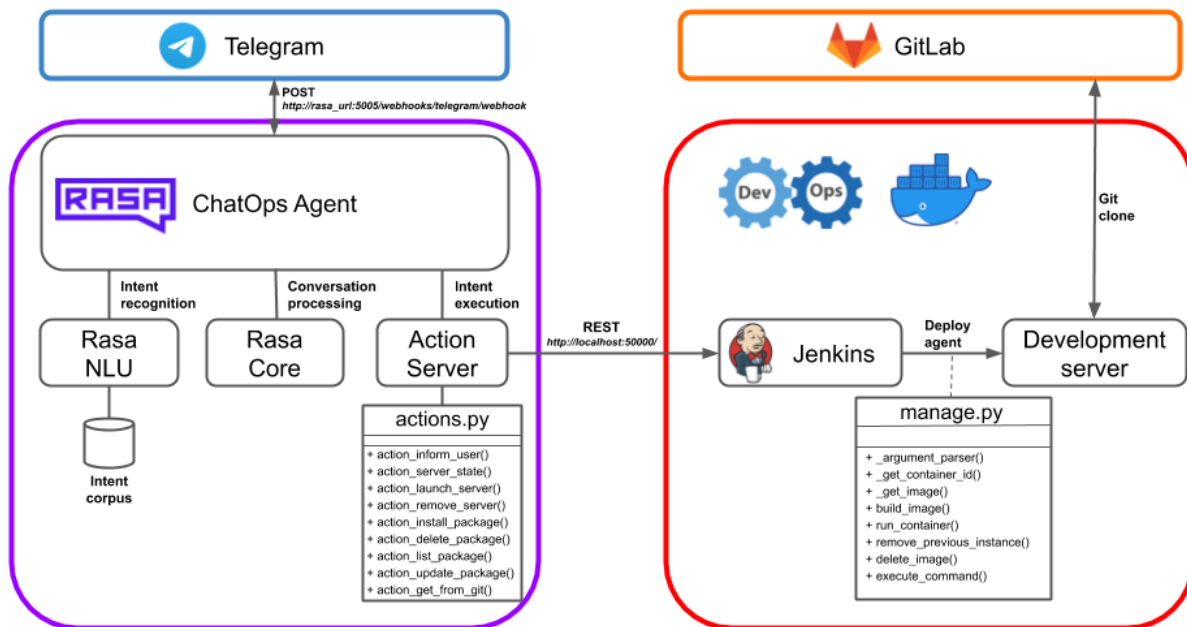


Figure 4.1: Architecture of the project

- **Telegram client:** The application chosen to connect with the chatbot environment. It can be run on different devices, depending on the user's needs.

- **Conversation system:** This system is responsible for managing the conversation with the user and processing its inquiries to interact with the backend part of the system, making it the center of the ChatOps environment. Among its functions, we can highlight the user intention recognition, the entity extraction from the information given by the user or the action launch based on the user necessities.

It is implemented using the Rasa open source chatbot framework, which offers a ChatOps agent that interconnects its different modules. These modules are the *Rasa NLU*, used to recognize the intents in the conversation with the user, *Rasa Core*, which allows processing the conversation and implements the methods to train the conversation models, and the *action server*, used to launch code depending on the conversation flow.

- **Jenkins server:** This server implements a *Jenkins* instance. Jenkins allows the environment to have *Continuous integration/Continuous delivery* automated workflows, accessible for the Rasa action server and the user. To simplify its deployment and maintenance, this server is containerized on a Docker container.
- **Deploy agent:** This server, managed by Jenkins, holds the methods to manage the Development server. Jenkins connects to it to perform the changes required by the user. The agent has control over the Docker socket, allowing it to deploy the development server and launch commands in it.
- **Development server:** This server is the one that is managed by the ChatOps environment. It is a *Python* development environment, with *Pip* managing the packages installed, accessible for the users via *SSH* with the user “developer” and use the server shell remotely. This server is managed by the deploy agent via Docker to deploy it and execute commands on it. It also can communicate with GitLab to update the packages list obtaining it from the remote server.

All the services displayed in the architecture, excepting the Telegram client and Gitlab, are deployed in a virtual machine running Ubuntu 18.04. This allows us to work with some of these services with more control over the processes and replicate the environment in case of need.

To simplify the architecture description, we are going to separate it into two parts. Firstly, the **user interactive side**, composed by the Telegram client and the conversation system, and the **textbfbackend side** with the development server and the tools used to manage it. This way, we can differentiate the part that interacts with the user from the part used to deploy and maintain the server.

4.2 User interactive side

As the architecture is based on the ChatOps paradigm, it is divided into components that interact with the user and components not meant to be managed by the user. This separation allows the user to operate a complex DevOps environment interacting with a cognitive system that offers a more human-like interface through conversation, simplifying the operation.

In this section, we are going to introduce the modules that conform the conversation system used in this part of the architecture. It will be formed by two modules, Telegram as the chatbot application to connect with the chatbot through *HTTP webhooks* and RASA as the chatbot module holding the conversation system and connected with the backend side of the architecture.

4.2.1 Telegram

At the time of deciding the technologies used in a ChatOps environment, the chat application used is one of the first decisions to take. It will be the access point of the system, so accessibility is important. Also, it is important to take account of allowing the users to access from different places and platforms to make the environment more flexible. To fulfill these necessities, we decided to use Telegram as the chat application to communicate with the environment.

Telegram offers to integrate a bot into the platform to chat with it, either in a private chat or integrating the bot in a group chat. The integration is supported by the bot API, which allows the developers to connect their bots to Telegram system. A Telegram bot is a special type of account that does not require a telephone number to set up. Instead of that, they are created by registering it in Telegram.

This is achieved by chatting with a special bot account made by Telegram, “Botfather”. To create a bot, the developer must give Botfather a name which will be displayed in the chat window and a username, used to search it in the Telegram users directory and that must end in “bot”. The developer can also adjust some settings of the generated chatbot, such as giving it a profile picture or change the behavior of the bot in a group chat. When the configuration is finished, a unique authorization token is provided.

This authorization token will be used to communicate with the Telegram bot API. This is achieved by using a “webhook”, consisting in HTTP request to an API endpoint given by Telegram that contains the unique token. The URL of the endpoint will have the

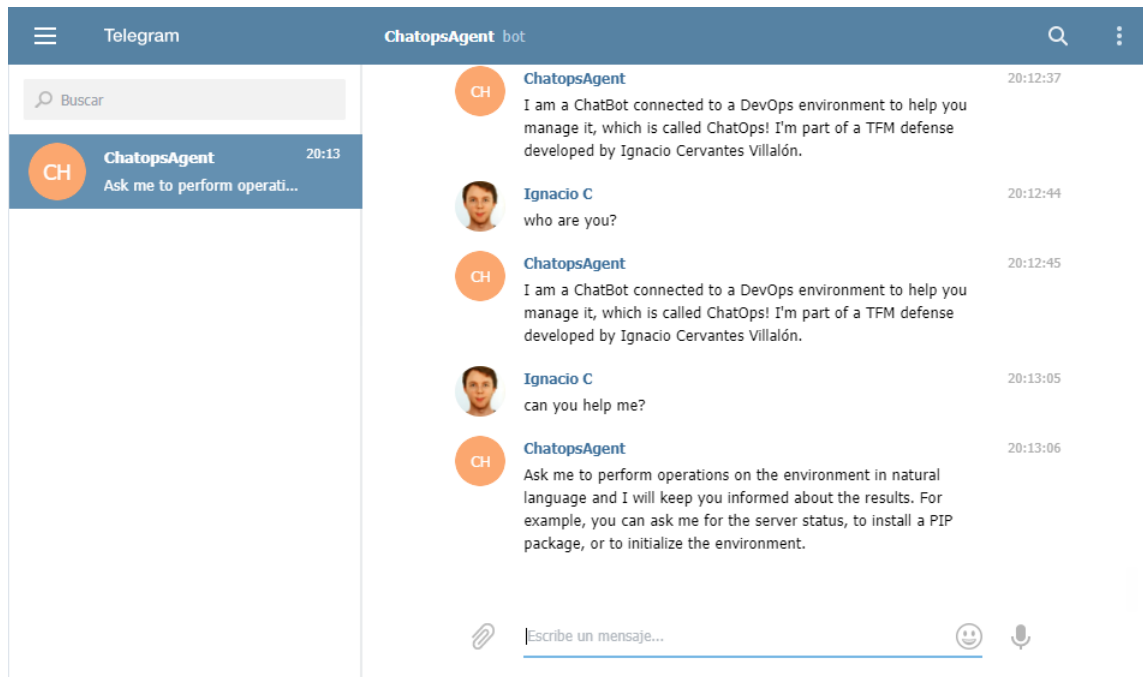


Figure 4.2: Telegram web user interface

form `https://api.telegram.org/bot<token>/method_name`. The supported HTTP methods to communicate with the endpoint are GET and POST, and the responses contain a JSON object with the result of the operation in the bot and the data returned. An example of a message towards the API is displayed in the List. 4.1:

Listing 4.1: HTTP petition example towards the api

```
POST /webhooks/telegram/webhook HTTP/1.1
Host: 5b8678613052.ngrok.io
Content-Length: 278
Accept-Encoding: gzip, deflate
Content-Type: application/json
X-Forwarded-For: 91.108.6.95
X-Forwarded-Proto: https
{
  "update_id":838224377,
  "message":{
    "message_id":80,
    "from":{
      "id":1125860467,
      "is_bot":false,
      "first_name":"Ignacio C",
      "username":"nasio13",
```

```
    "language_code": "es"
  },
  "chat": {
    "id": 1125860467,
    "first_name": "Ignacio C",
    "username": "nasiol3",
    "type": "private"
  },
  "date": 1609095880,
  "text": "hi!"
}
```

The chatbot also communicates with the Telegram API endpoint defined with the token to communicate with the user. The bot posts the response with HTTP methods towards the endpoint, which sends it to the user. This response can send additional data aside from text, like images, buttons or even stickers. A diagram showing the flow is shown in the following Fig. 4.3:

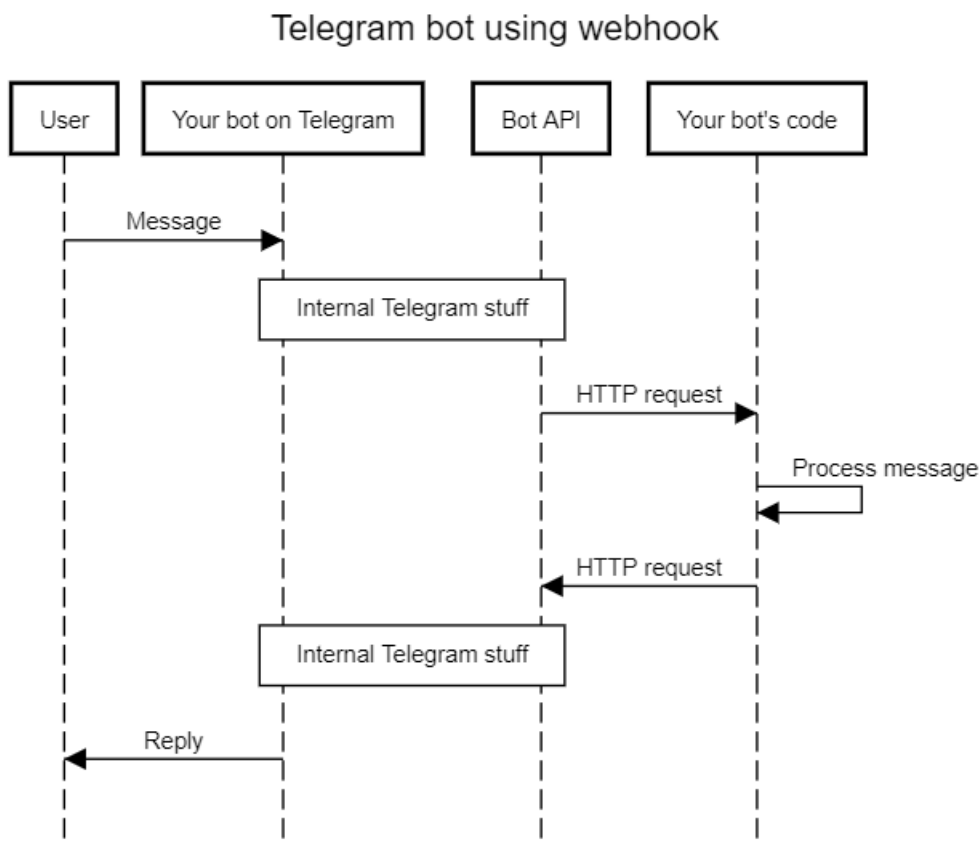


Figure 4.3: Telegram webhook message flow [85]

The messages sent by users through the client pass to intermediary servers that encrypts the messages and communicates with the Bot API server managed by Telegram. This server has some limitations in terms of processing messages in short periods or in large groups. To avoid these limitations, Telegram offers the code to deploy a local Bot API server, that allows extending the functionalities and offers more control over the communication. For our project, we will use the default Telegram Bot API Server as any of the extended options are necessary for achieving our objectives.

4.2.2 RASA Open Source

RASA Open Source, or from now Rasa, will allow us to develop a machine learning assistant capable of chatting with a user in natural language to manage a DevOps environment. Due to the functions and customization that RASA offers, it fits in the middle of the proposed environment, where plenty of services must be interconnected.

As it is the main service to be implemented in this project, we are going to take a closer look at the different components of RASA and how they interact. To visualize the parts that compound the RASA module, we are going to show the interactions of these parts in Fig. 4.4:

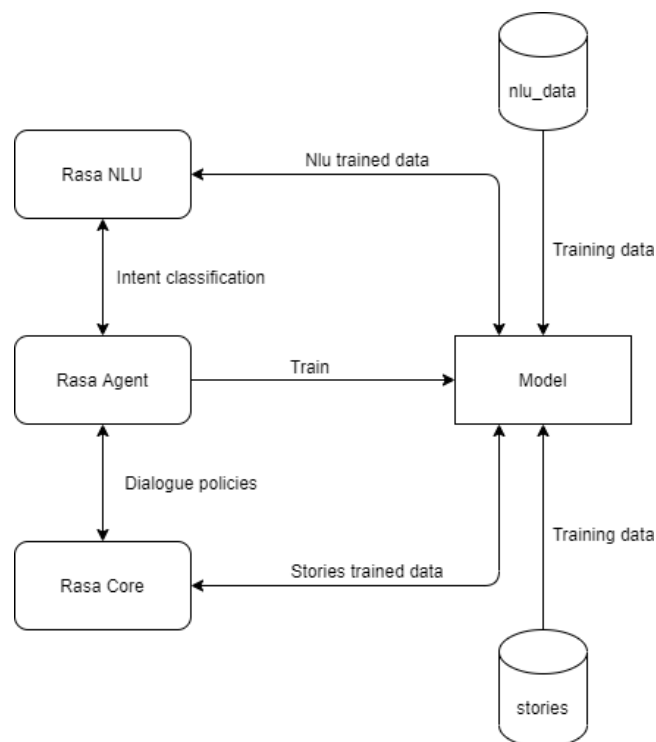


Figure 4.4: Interaction between Rasa components.

As Rasa functionality is achieved by the interaction of its components, we are going to explain the implementations on them and how they impact the use of the chatbot.

4.2.2.1 ChatOps Agent

Rasa is a complex chatbot service, composed of many components interconnected. These components have different functions, from dialogue processing to *Natural Language Processing* model training and recognition.

The Rasa Agent allows the user to interact with these components by providing an interface to the most important Rasa functionality. In Fig. 4.5 we can see how the agent acts as an essential interface in the different Rasa components:

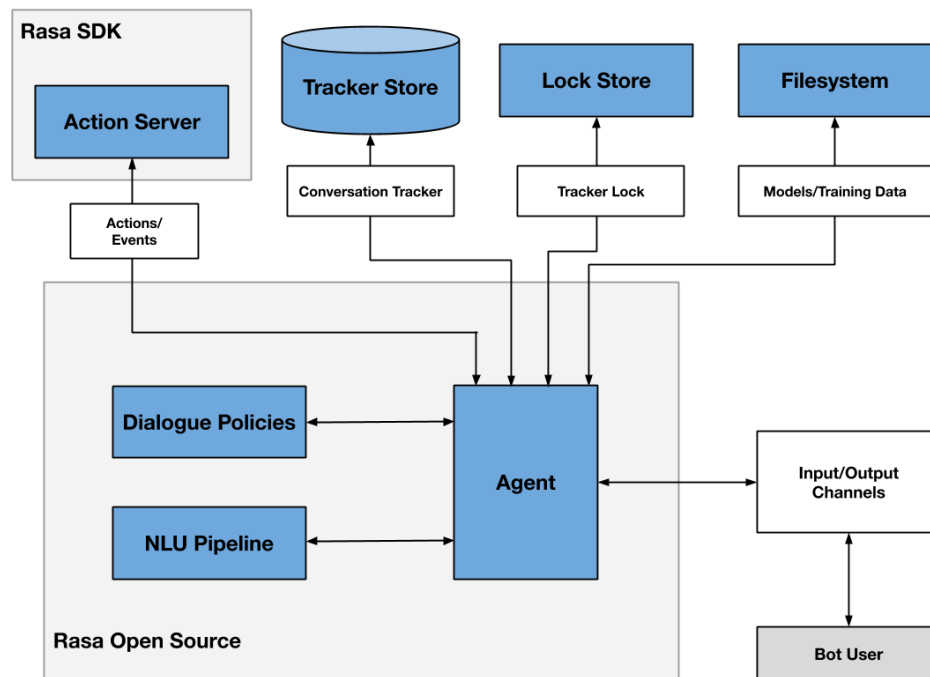


Figure 4.5: Detailed Rasa architecture [40]

The agent is implemented in Python like the rest of the components, and running an instance of it will build a functional Rasa chatbot with the existing data. By launching the Rasa agent with the command “`rasa run`”, the agent performs the following actions to deploy the service:

- Check the endpoints and credentials files to check the outbound connections configured.

- If there are connections to webhooks, check the health of the endpoints and the connections.
- Load the latest Natural Language Understanding (NLU) model generated.
- Connect to the Rasa Core services *Tracker store* and *Lock store*
- Build TensorFlow prediction graph based on the NLU model.

For the current project, the Telegram integration is checked at the first steps of the service deployment. To connect to the Telegram Bot API, we have to specify some parameters in the *credentials.yml* file with the following format:

Listing 4.2: Telegram integration in Rasa

```
telegram:
  access_token: "<telegram_api_token>"
  verify: "ChatopsAgentbot"
  webhook_url: "https://<rasa_url>:5005/webhooks/telegram/webhook"
```

The values configured in this project were the Telegram Bot API authorization token created in the Sect. 4.2.1, the chatbot name “ChatopsAgentbot” defined in the creation and the URL of the Rasa server. As this project is deployed in a local virtual machine, port 5005 where Rasa is listening to HTTP methods is not reachable from external networks. Therefore, Telegram is not able to communicate directly with our Rasa server. To allow communication, we used the service **Ngrok**. Ngrok [61] allows us to expose a local port to a specific protocol calls to the Internet, and allow communication with remote servers.

Ngrok assigns a URL to the specified port and protocol, forwarding the petitions to it while the daemon is running. It offers different billing plans as well as a free plan, which is enough for our requirements. The console user interface of the Ngrok daemon can be seen in Fig. 4.6, which offers information about the requests and the URL that forwards to the local direction:

Moreover, the Rasa agent also exposes a method that allows training the policies ensemble by using the command “rasa train”. It creates training data from the user stories and the NLU intents data and creates a model that will be used by the assistant to answer the user. The data used in this training will be detailed in the following sections, as well as the components that conform the pipelines used to train the model.

```

ngrok by @inconshreveable

Session Status      online
Account             Ignacio Cervantes (Plan: Free)
Version             2.3.35
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://5b8678613052.ngrok.io -> http://localhost:5005
Forwarding           https://5b8678613052.ngrok.io -> http://localhost:5005

Connections          ttl      opn      rt1      rt5      p50      p90
                    13        0        0.00     0.00     5.35     10.36

HTTP Requests
-----
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK
POST /webhooks/telegram/webhook 200 OK

```

Figure 4.6: Ngrok console user interface

4.2.2.2 RASA NLU

Rasa NLU is an open-source natural language processing tool used for intent classification, entity extraction and response retrieval. Rasa NLU is part of the Rasa Open Source framework, but it also can be used to generate structured data to build other chatbot solutions.

Rasa NLU distinguishes from other *Natural Language Processing* (NLP) solutions like *Microsoft's LUIS* in the customization. As Rasa is open source and all their tools are accessible to the developer and customizable, the flexibility of the training to get the needed models is ensured. Also, the training data used with the bot is only handled by the developer, so this data stills private.

The NLU module main functionality is to detect **intents**, which consist in utterances given by the user that the bot should be able to categorize into a predefined model, and **entities**, which consist of chunks of specific information that the bot should be able to recognize. This model will be trained with a list of user utterances categorized by intent. The sentences will be processed with *Natural Language Processing* tools defined in a pipeline to build the model that will be used to talk with the user. An intent definition example can be seen in the List. 4.3:

Listing 4.3: Intent format example

```
- intent: greet
```

```
examples: |
  - hey
  - hello
  - hi
  - hello there
  - good morning
  - good evening
  - moin
  - hey there
  - let's go
  - hey dude
  - goodmorning
  - goodevening
  - good afternoon
```

For this project, we have defined 12 different intents. With these, we aim to provide the user with the capability of managing a development server through conversation using natural language. We can organize the intents using the following criteria:

- **General intents** used to catch usual user interactions. These intents will launch responses with a short message.
 - Greetings from the user, such as “hi” or “hello”.
 - Thank you messages.
- **Support intents** used to guide the user. They catch messages that ask about the chatbot.
 - Asking who the user is talking to, with sentences like “are you a robot?” or “who are you”.
 - Asking for help with the usage, for example “what can you do?”.
- **DevOps intents**. These intents will be used to detect the operations the users want to perform on the development server. These intents can include entities to give the bot additional information, such as the package to install.
 - Asking for the server current state, with sentences like “is the server up?”.
 - Launch the development server (i.e. “deploy the server”).
 - Delete the current development server instance, with utterances like “delete the server instance”.
 - Asking for installation of a specific package, for example “install scipy==1.0.0”.

- Asking for removing a package, using “uninstall pandas”.
- Listing the installed packages, for example “show me the installed packages”.
- Update the installed packages, saying “update the packages”.
- Get from Gitlab the packages defined in the requirements file, using sentences like “get the changes from git”

These intents are defined in a *YAML* file called “nlu.yml”, located in the data folder of the project. Each of the intents has a list of sentences with examples with the format shown in List. 4.3, used to train the model. It is important to build a corpus of examples to improve the results of the intent recognition, so at least 5 examples per intent are recommended and 10 may be sufficient for simple sentences. For training the NLU model, we used a minimum of 10 examples for each intent.

It is important to highlight that the bot should be able to get some additional information for some of these intents. For example, when installing a package, the bot should be able to recognize the package and the version that the user wants to install to perform the action. To do so, these intents catch the entity “package”, that is passed to the actions launched by the intents to apply the changes. The List. 4.4 shows the recognition of the intent “install_package” and the entity “package” when the user uttered “install scipy==1.0.0” to the bot:

Listing 4.4: Entity extraction example

```
Received user message 'install scipy==1.0.0' with intent '{'id':  
-2441886824623672267, 'name': 'install_package', 'confidence':  
0.999893069267273}' and entities '[{'entity': 'package', 'start': 8, '  
end': 20, 'confidence_entity': 0.9992488026618958, 'value': 'scipy  
==1.0.0', 'extractor': 'DIETClassifier}]'
```

The entities that the developer wants to be recognized by the NLU should be tagged in the training data. These tags consist of surrounding the example entity with box brackets symbols and indicating the name of the entity between parentheses. The training data for the intent “install_package” is shown in Lst. 4.5. Note how we gave several examples of packages with different formats to allow the chatbot to extract the entity with the two expected formats, with the version and without it.

Listing 4.5: Entity extraction example

```
- intent: install_package
  examples: |
    - i want to install [numpy==2] (package)
    - install the package [test==1.0.0] (package)
    - install [testpy==0.2.0] (package)
    - can you install [pip==2.0.1] (package)
    - install the package [pandas==3.0.2] (package)
    - install [scipy==3.5.1] (package)
    - install the package [cowsay] (package)
    - install [standalone] (package)
    - install [pytest] (package)
    - install the package [scipy==1.0.0] (package)
```

As we mentioned before, to train the NLU model, Rasa allows the user to build a pipeline of *Natural Language Processing* tools. The tools cover different kinds of natural language processing tools that allow the developer to tune the pipeline to its needs, and allow for example to load pre-trained models to enrich the corpus.

As the intents and examples used in this project are intendedly low on numbers, we will use the default NLU pipeline as they offered good results at predicting the intents. The components used to train the model in the project are executed in the following order:

- **WhitespaceTokenizer:** Component that tokenizes whitespaces as a separator for words.
- **RegexFeaturizer:** This module creates a vector representation of user message using regular expressions. Used to extract entities in the components DIETClassifier and CRFEntityExtractor.
- **LexicalSyntacticFeaturizer:** Featurizer that creates lexical and syntactic features for a user message to support entity extraction. Moves with a sliding window over the tokens to create the features.
- **CountVectorsFeaturizer:** This component creates a *bag-of-words* representation of user messages, intents and responses using the sklearn CountVectorizer module.
- **DIETClassifier:** Dual Intent Entity Transformer (DIET) [67] is a transformer architecture that can handle both intent classification and entity recognition. It consists of a multi-task architecture capable of predicting entity labels through a Conditional Random Field (CRF) layer and getting the intents through comparing the intent labels embedded through a single semantic vector space with the target label. It can take as input different features:

- Dense features from pre-trained embeddings, such as *Spacy* models.
- Sparse features from the training data, like the features provided by the component *CountVectorsFeaturizer*.

The DIET classifier can take both types of features or just one of them. In this project, as we count with a low number of intents, we will just use the sparse features provided by *CountVectorsFeaturizer*. A representation of the vector space showing the intent detection is shown in Fig. 4.7:

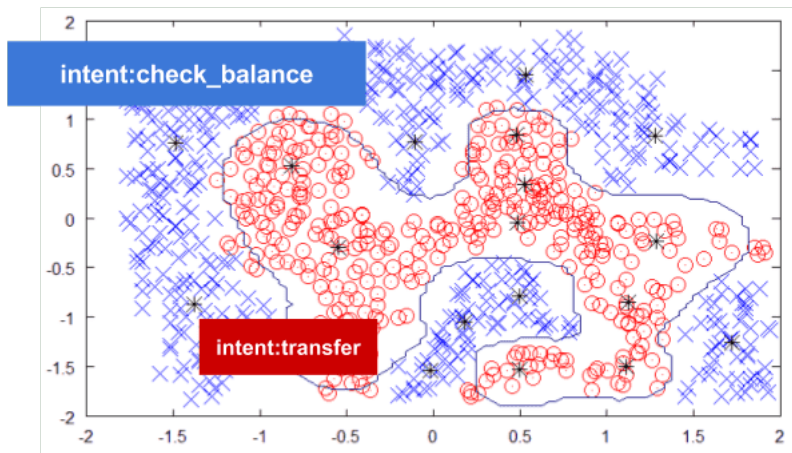


Figure 4.7: Representation of vector space used to classify the intents [66]

To classify the entities in the data, DIET uses the tags in the training data, as shown in Lst. 4.5. To evaluate if a word should be tagged with an entity, the classifier will look at the features of the word being evaluated, but also the preceding and the following words. This way, it will take into consideration the use of connectors and prepositions to decide if a word is an entity.

The behavior of the classifier can be tuned to adjust to the bot necessities. The developer can adjust the number of times that the training goes through the data by changing the *epoch* value, with a default value of 100. In our case, we left it at the default value as increasing it may incur in overfitting the model to the training data. It also can be adjusted to only recognize intents or entities, in case the developer wants to use other classifiers.

- **EntitySynonymMapper:** Module that maps synonymous entity values to the same value. It requires to define previously the synonyms in the training data.
- **ResponseSelector:** Component that chooses the predicted response. It is only used in training data with retrieval intents, a special type of intent that can be divided into

sub-intents.

- **FallbackClassifier:** This module classifies a message with the intent “`nlu_fallback`” if the classification scores are ambiguous. This allows, if the fallback action is implemented, to handle uncertain NLU predictions in the user stories.

To train the NLU model, we use the method “`rasa train`” introduced in Sect. 4.2.2.1 that creates a model by processing the information located in the file *nlu.yml*. This file contains the generated corpus with the intents and the entities that will be recognized along with the examples used for training the model. To create the model, the training data will be processed by the pipeline presented earlier in this section. This model will be used by Rasa NLU to recognize the intents and entities in the messages that the Agent receives. The information extracted from the user messages will be passed to Rasa Core, which we will introduce in the following section.

4.2.2.3 Rasa Core

By now, we reviewed the Rasa agent and NLU module. As we saw, the agent represents an interface to communicate the user with the different modules that compose Rasa, and the NLU module is the one in charge of recognizing the user messages patterns to reply in an expected way.

Taking a look into Fig. 4.5, that references the Rasa architecture, we do not find the Rasa core as a single module. Instead, the Rasa core is composed of different modules and services, and is responsible for choosing the right actions to take when a user message arrives at the chatbot. In this section we will explain the role of these modules, as well as explain how core components decide the next steps in conversation.

To introduce the core decision-taking, Fig. 4.8 presents a simplified view of the message processing flow. When a message from the user arrives, it is passed to the NLU interpreter introduced in the previous section. It will extract the intent and entities based on the trained model generated with the NLU pipeline. That data then is stored in a *tracker* object, that keeps a registry of the conversation state with information such as the previous intents and actions taken. That tracker state is then passed to the policy, which decides the next step action to take based on the *stories*. When the next action to take is decided, the tracker is updated with that action and the response is delivered to the user.

As seen in the message flow, the tracker makes a relevant part in decision making. It keeps tracking of the user intents recognized by the NLU module and the actions taken

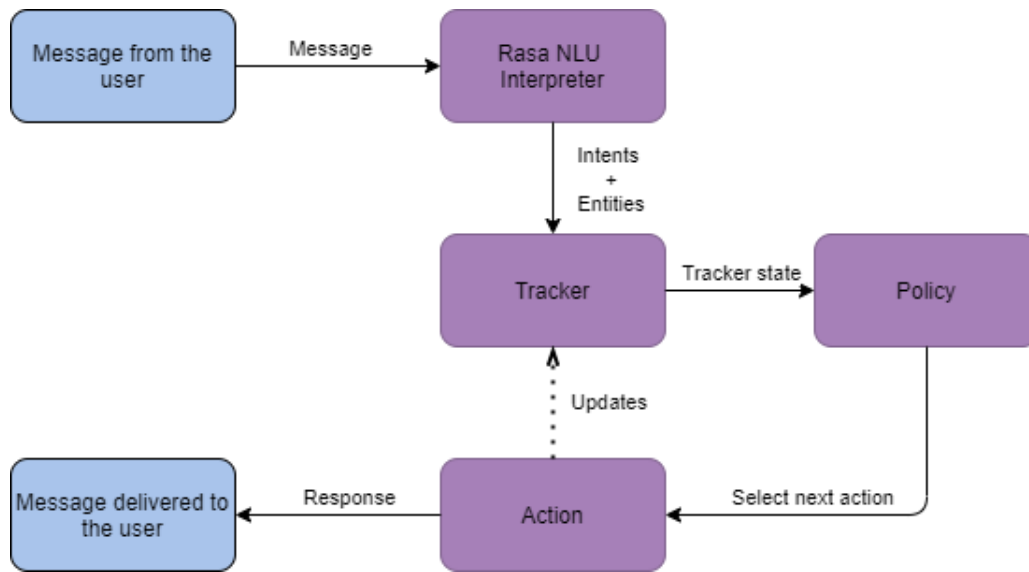


Figure 4.8: Steps taken by Rasa to respond a message

when those intents were detected. This allows the policy to decide which action should be taken next. In the List. 4.6 the information stored in the tracker is shown:

Listing 4.6: Tracker information example

```

2020-12-30 04:28:41 DEBUG    rasa.core.policies.rule_policy - Current
                             tracker state:
[state 1] user intent: greet | previous action name: action_listen
[state 2] user intent: greet | previous action name: utter_greet
[state 3] user intent: list_package | previous action name: action_listen
[state 4] user intent: list_package | previous action name:
         action_inform_user
[state 5] user intent: list_package | previous action name:
         action_list_package
  
```

Each conversation with the bot generates a unique tracker. It is linked to a unique conversation ID, as well as a *ticket lock*. This ticket lock mechanism ensures that incoming messages from the same conversation are processed in the intended order, locking conversations while messages are actively processed. As seen in Fig. 4.5, there are stores for trackers and locks, that enable running multiple Rasa servers in parallel to replicate the service and offer high availability. However, for this project we deploy a single Rasa server so we will use the default tracker store and lock stores, known as *InMemoryTrackerStore* and *InMemoryLockStore* respectively.

To train the dialogue management model, Rasa uses **stories**. A story is a representation

of a conversation between the bot and a user, where the user messages are represented by the intents and entities that match the message and the bot responses are expressed by the action names. The List. 4.7 represents a story used in this project to install a specific package in the development server:

Listing 4.7: Story that describes a package installation

```
- story: install_package
  steps:
  - intent: install_package
    entities:
    - package: "scipy==1.0.0"
  - action: action_inform_user
  - action: action_install_package
```

This story begins when the user utters that it wants to install a package, with an example entity to help train the model. Then, a custom action called “action_inform_user” is launched, sending a message to the user to advise of the action taken. Finally, the action that performs the changes in the server is launched. In this example, the action installs the requested package and replies to the user with the console output from the operation to inform it.

There are many additional options to refine the stories. For example, the developer can add checkpoints to simplify the training data processing, and “OR” statements to handle multiple intents in a single story. Nevertheless, as stories are meant for training purposes, the action server is not called and the assistant dialogue management model is unaware of the events that a custom action will return. Because of this, events that may be launched by actions such as slot setting or managing forms have to be explicitly indicated in the story. However, in this project neither of these additional options is needed to fulfill our objectives.

Aside from stories, *rules* are another training data for the dialogue manager. Unlike stories, they strictly determine the conversation path with the defined steps instead of generalizing to unseen conversations. Rules allow simplifying complex conversation models through defining rules that apply to intents that don’t need context or additional information. However, in multiple-turn interactions, it is recommendable to define a story. To give an example of rule usage, a simple rule used in the project to reply to a thank message is shown in the following List. 4.8:

Listing 4.8: Rule to reply to thank you messages

```

- rule: Reply to the thank you messages
  steps:
  - intent: thank
  - action: utter_welcome

```

For training the assistant, we generated the training files *stories.yml* and *rules.yml*. The stories file contains 11 stories like the one shown in List. 4.7, which relates the intent passed by Rasa NLU module with the actions to perform. For our chatbot, they always trigger the action “action_inform_user” first, which sends the user a message to inform it that which action the chatbot is running, and an action that performs the operation related to the intent. In the rules file, we have the rule shown in the List. 4.8.

```

10                                     install the package [scipy==1.0.0](package)
                                     intent: install_package 1.00

11  action_inform_user 1.00
    Installing the requested package...
    action_install_package 0.92
    Execution finished! Console output:
    Collecting scipy==1.0.0
      Downloading scipy-1.0.0-cp36-cp36m-manylinux1_x86_64.whl (50.0 MB)
    Collecting
    numpy>=1.8.2
      Downloading numpy-1.19.5-cp36-cp36m-manylinux2010_x86_64.whl (14.8 MB)
    Installing collected packages: numpy, scipy
    Successfully
    installed numpy-1.19.5 scipy-1.0.0
    action_listen 1.00

```

Figure 4.9: Interactive learning user interface

Aside from the manually generated stories, we also trained the dialogue policies by using the **interactive learning** [43] module. This module allows the developer to write stories by talking to the chatbot and correct the predictions made in real-time. To launch the interactive learning, the developer should run “rasa interactive” in the shell. This launches a conversation with the assistance and asks the user to make an input. After each input, the chatbot will predict the intent in the message received with the NLU classification model,

just as in a normal conversation, but will ask the user for confirmation about the predicted intent as shown in Fig. 4.9. If the intent predicted by the assistant is not correct, the user can select the correct intent. After the training is complete, the story created from the conversation with the bot will be added to the *stories.yml* file as shown in Lst. 4.9, and considered when building the conversational model.

Listing 4.9: Story generated from interactive learning

```
- story: interactive_story_1
  steps:
    - intent: greet
    - action: utter_greet
    - intent: install_package
      entities:
        - package: cowsay
    - action: action_inform_user
    - action: action_install_package
    - intent: install_package
      entities:
        - package: standalone
    - action: action_inform_user
    - action: action_install_package
    - intent: install_package
      entities:
        - package: pytest
    - action: action_inform_user
    - action: action_install_package
    - intent: install_package
      entities:
        - package: scipy==1.0.0
    - action: action_inform_user
    - action: action_install_package
    - intent: list_package
    - action: action_inform_user
    - action: action_list_package
    - intent: delete_package
      entities:
        - package: cowsay
    - action: action_inform_user
    - action: action_delete_package
    - intent: delete_package
      entities:
        - package: scipy
    - action: action_inform_user
```

```
- action: action_delete_package
- intent: launch_server
- action: action_inform_user
- action: action_launch_server
```

To train the dialogue policies, Rasa creates training data from the stories and rules created by the developer and trains a model on that data. Then, the policies decide which action to take based on that model. These policies are determined in the same file where the NLU training components pipeline were, “config.yml”. For our project, we used the default policies to determine the actions based on the model created by the stories we have defined. These policies are the following:

- **Memoization Policy:** This policy is based on the *memoization* technique, which consists of saving the results of computations so that future executions can be omitted when the same inputs repeat [78]. The policy checks the stories taken from the training data. If the current conversation matches the stories that we have defined, it will predict the next action from the matching stories with a confidence of 1. If there is not any matching conversation, it will predict “None” with a confidence equal to 0.

A *max_history* hyperparameter defines the maximum number of “turns” (a message sent by the user and any actions performed by the bot before waiting for the next user message) to take into consideration for deciding the next actions. By default, it takes into account 3 turns.

- **Rule Policy:** It handles the rules that the developer has defined in the “rules.yml” file. After training, it can check if there are no contradicting rules or loops that may affect the operation of the assistant.
- **TED Policy:** The Transformer Embedding Dialogue (TED) policy is a multi-task architecture that allows to predict next action and recognize entities. Based on machine learning, it performs the following operations [41] to get the confidence:
 1. Concatenate features from intents and entities, previous bot actions and slots, and put it into an input vector to the embedding layer before the dialogue transformer.
 2. Feed the embedding of the input vector into the dialogue transformer encoder.
 3. Apply a dense layer to the output to get embeddings of the dialogue.
 4. Apply a dense layer to create embeddings for system actions.
 5. Calculate the similarity between the dialogue and system actions embeddings.

6. Concatenate the output of the user sequence transformer encoder with the output of the dialogue transformer encoder for each time step
7. Apply Conditional Random Field (CRF) algorithm to predict entities for each user text input.

The steps performed to calculate the similarity through two dialogue turns can be seen in Fig. 4.10:

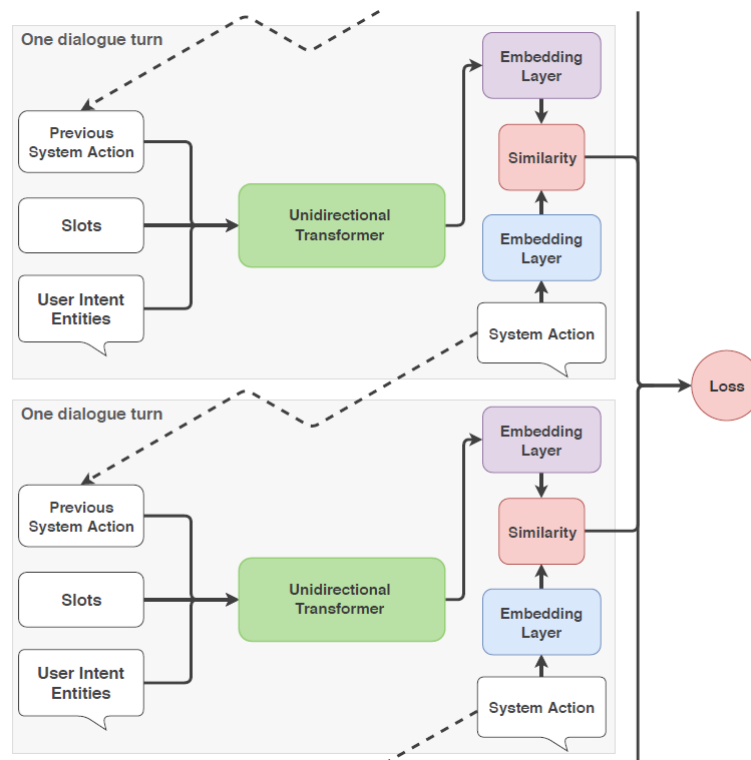


Figure 4.10: Representation of two iterations of TED policy [84]

In the case that two policies predict with equal confidence, Rasa will take the policy with a higher priority. The default priorities are 6 for Rule Policy, 3 for Memoization Policy and 1 for TED Policy. For example, as Memoization and Rule Policy may predict with confidence 1, if a story and a rule both match with the model, the action performed will be decided by the rule. To create the model used by the dialogue policies, the developer should launch the “`rasa train`” method, which is also used to train the NLU module. This will build a model using the policy pipeline defined in the `config.yml` file and the training data located in the `stories.yml` and `rules.yml`, and will be used to decide which actions should be launched after recognizing an intent.

Finally, as Rasa handles data used by different components, it is important to have a file where the whole chatbot data is accessible. That file is “domain.yml”, which defines the whole environment where the assistant operates. In this file, the intents, entities, slots, responses, forms and actions that the bot can perform are listed, as well as chat session configuration. The responses are displayed in the format shown in the List. 4.10:

Listing 4.10: Response section in domain.yml

```
responses:
  utter_greet:
    - text: "Hey! How are you today?"
    - text: "Hi! Let's get some work done today!"

  utter_who_are_you:
    - text: "I am a ChatBot connected to a DevOps environment to help you
      manage it, which is called ChatOps! I'm part of a TFM defense
      developed by Ignacio Cervantes Villalon."

  utter_help_me:
    - text: "Ask me to perform operations on the environment in natural
      language and I will keep you informed about the results. For example,
      you can ask me for the server status, to install a PIP package, or
      to initialize the environment."

  utter_welcome:
    - text: "You are welcome!"
    - text: "No problem!"
```

These responses will be launched as an action when the policy predicts a rule or story that fits. These actions send the user a message with the text detailed in the response, picking randomly one of the “text” values if there is more than one to vary the responses given.

4.2.2.4 Action server

By now, we have reviewed the components used to build up the models to converse with the user. First, we talked about how Rasa NLU allows extracting the user intents and entities and continued with Rasa core components that allow choosing the right response to the information collected from the user messages. After showing how we used these tools in our project, the next step is to explain how Rasa replies to the user.

Actions are the processes that the bot runs in response to user input. They are predicted by the policies, which decide based on the dialogue model what the bot should perform after catching a certain intent. There are different kinds of actions that Rasa could perform:

- **Default actions:** These actions are built into the dialogue manager by default. Most of them are automatically predicted based on conversation situations. Some of the most important are “*action_listen*”, launched to make the assistant wait for the next user message, “*action_restart*”, used to reset the conversation with the user, or “*action_default_fallback*” to ignore the last user message and utter a message indicating that the bot did not understand the received message.
- **Utter actions:** Send a message to the user. These actions are also built in the dialogue manager, and are defined in the bot domain file as shown in the List. 4.10.
- **Custom actions:** These actions can run any code. Custom actions should be defined in the domain file, and they are executed in an action server that communicates with the agent through an endpoint specified in the “*endpoints.yml*” file that listens for calls.

The logic used to connect Rasa to the DevOps environment shown in the architecture diagram is implemented with custom actions and executed via a Rasa Action server. The server is deployed in the local Ubuntu machine where the whole environment is implemented.

To execute the action server, use the command “*rasa run actions*”. That command will start the server, that will register the custom actions located in the file “*actions.py*” and make available the endpoint where Rasa will communicate to execute the actions when the policy predicts them. The endpoint in this project is *http://localhost:5055*.

When a custom action is predicted by the Rasa Core, an HTTP request is sent to the endpoint. The request includes in the payload the following information:

1. A *name_action* field with the name of the action to be executed.
2. The *sender_id*, a unique identifier of the user having the conversation with the chatbot.
3. The *conversation tracker* object content, which reflects the conversation state at the point of launching the action. It contains the messages, intents, entities and previous actions performed.
4. The chatbot *domain*, extracted from the “*domain.yml*” file.

Until the action is completed, the chatbot will not be responding to the user. When the action is completed, the server replies to the Rasa agent with a dictionary with two objects: an *events* entry with the changes on the conversation slots if they were changed and a *responses* entry with the messages that will be sent to the user. Then, the Rasa action server will change into an idle state and wait for the next calls to custom actions.

As long as it implements procedures to get the calls in the endpoint and reply to the agent, the Rasa action server can be developed in every language. However, the default Rasa action server and the SDK is implemented in Python, as well as the default custom actions file “actions.py”. To explain the main parts of a custom action, in the List. 4.11 the code for “action_list_package” is shown:

Listing 4.11: Code for custom action action_list_package

```
class action_list_package(Action):

    def name(self) -> Text:
        return "action_list_package"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:

        jenkins_server = _get_server_client()
        job_name = 'get_package_list'
        # Log the action execution in the action server console
        print('Accesed the action ' + self.name())

        current_job = _get_current_execution_number(jenkins_server,
            job_name)
        _launch_jenkins_job(job_name)
        job_console_results = _get_job_results(jenkins_server, job_name,
            current_job)
        dispatcher.utter_message(text="Execution finished! Console output:\n" +
            job_console_results)

        return []
```

Each custom action must be defined as a class that extends the Action class, and must have two methods. The *action.name* method defines the action name. This name is used by the Rasa agent to call the action, and it must match the name used in the bot domain file. The method *action.run* executes the code of the action. It has the following parameters:

- The **dispatcher**, used to send back messages to the user.
- The **tracker** with the state of the current user conversation. It implements methods to get the last message, slots or entities of the conversation.
- The bot's **domain**.

The developed code is inside the method *run*. This action is used for getting the list of packages of the development server. To do so, it calls a function *_get_server_client* to get the *jenkins_server* object that will perform the configuration. Then, it gets the job number with the method *_get_current_execution_number* and launches the job with *_launch_jenkins_job* in the Jenkins server. When it is finished, it will retrieve the console output with the method *_get_job_results*, to finally utter the formatted console results to the user.

To simplify the action part and delegate the deploying tasks to Jenkins, the rest of the actions that call Jenkins have a similar structure and call to the same methods that start with an underscore, meant to simplify the actions and reuse the code that uses Jenkins methods. The 9 custom actions implemented in this project to manage the DevOps environment are the following:

- **Action_inform_user:** Auxiliary action used to give information to the user about the next action to be executed. It gives feedback to the user about the action that the action server is about to perform, as the server blocks until the action is complete. It checks the last intent found in the tracker and gives a fitting response.
- **Action_server_state:** Calls the Jenkins job “check_server_state”. Checks the development server’s health and replies to the user with the current state of the server.
- **Action_launch_server:** Calls the Jenkins job “deploy_server”. This action initializes the development server, and if it is already created, destroys the current container and deploys a new one with the default configuration. If it is stopped, the server would be started again without affecting the configuration. Replies the user with the operations performed and the result.
- **Action_stop_server:** Calls the Jenkins job “stop_server”. This action will stop the docker container, but it will not affect the content of the development server. Replies the user with the result of the operations.
- **Action_install_package:** Calls the Jenkins job “install_package”. This action gets the entity *package* from the tracker of the conversation and installs the package in the development server. Replies the user with the result.

- **Action_delete_package:** Calls the Jenkins job “delete_package”. Gets the entity *package* and removes the package from the development server. Replies the user with the result.
- **Action_list_package:** Calls the Jenkins job “get_package_list”. It gets a list of the packages installed and utters it to the user.
- **Action_update_packages:** Calls the Jenkins job “update_packages”. Updates all the packages installed in the development server to the latest version available.
- **Action_get_from_git:** Calls the Jenkins job “get_from_git”. It downloads the package list to be installed in the server from GSI GitLab and proceeds to install them. When it is done, utters the user the packages gotten from GitLab and the results of the installation.

The Jenkins integration is based on the library Python Jenkins [18]. It allows the action server to connect to a Jenkins service by its URL, in our project `http://localhost:8080`, and perform operations in the instance such as creating jobs or checking their results.

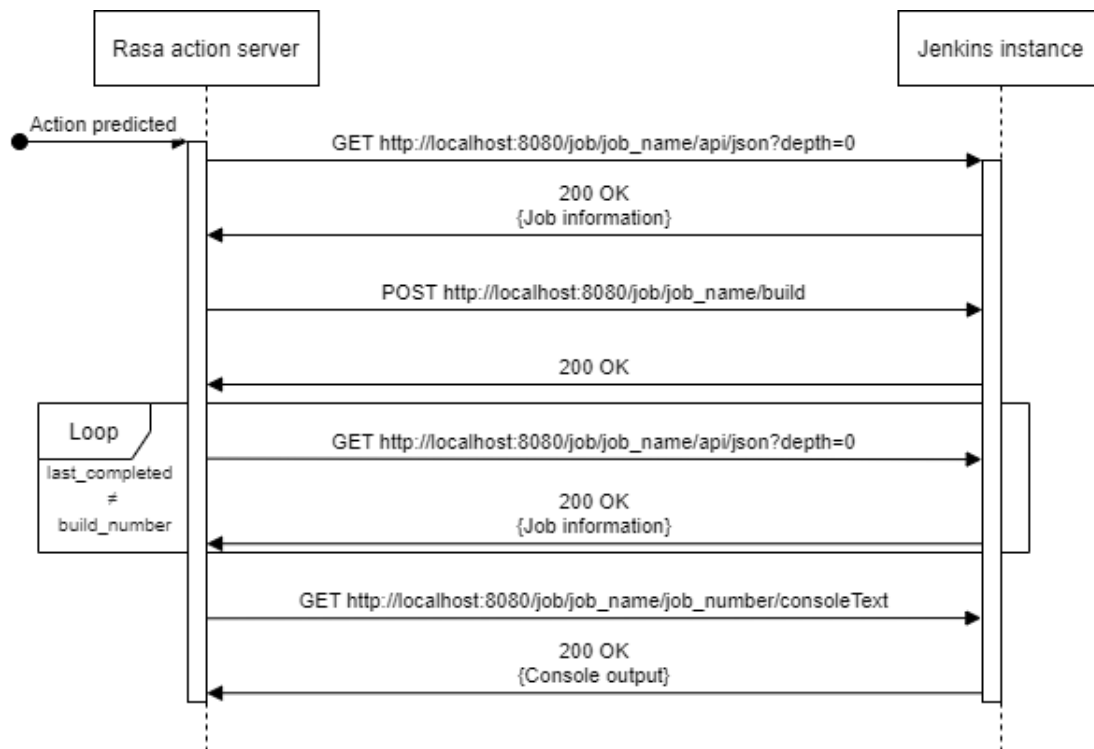


Figure 4.11: Sequence diagram for the Python Jenkins build job process.

Fig. 4.11 shows the interaction between the Rasa action server and the Jenkins instance. This is achieved by using the Python Jenkins library, imported in our `actions.py` file, which

is used by the actions that call Jenkins jobs. The steps taken when an action that calls a Jenkins job is predicted by the dialogue policies are the following:

- First, the action calls an internal method “_get_current_execution_number” that calls the Jenkins instance and retrieves a JSON with information from the job that is going to be launched. From that JSON, a variable with the next build number, *current_job* is stored.
- Then, the internal method “_get_current_execution_number” is launched. It sends an HTTP POST request to the URL with the job name to be built and the parameters if the job needs them (i.e. the package to be installed in the “install_package” job). Jenkins returns an “OK” message with the queue number assigned to the job.
- To check if the build is completed, the action will poll the job information with the method “_get_job_results”. This method checks if the build number stored before, *current_job*, is equal to the JSON field “last_completed_build”. If it is equal, it means that the build has finished, and a final GET petition is sent to retrieve the console output.
- Finally, the console output is processed to remove the build information. The post-processed console output is then uttered to the user, ending the custom action.

4.3 Backend side

In this section, we are going to introduce the components that perform the changes in the Development Server, as well as the server itself. The objective of this side of the architecture is to automatize the operations on the development server and to communicate with the chatbot module to convert the messages of the user into operations on the environment.

The backend side of the architecture is fully deployed in Docker containers, running in the Ubuntu virtual machine where the Rasa chatbot is executed. The decision of using Docker to deploy the main components of the backend is based on the following points:

- It allows us to quickly replicate the environment in case of need, as the images are built from *dockerfiles* developed by us that can be ported to launch the containerized services in other systems.
- As the services are containerized, the developed solution is **system agnostic**. This means that the modules can be run on different platforms that can run Docker. For

example, it can be run in other operative systems like *Microsoft Windows*, *Mac OS* or even in cloud platforms like *RedHat Openshift*.

- As Docker allows to forward the ports of the containers and expose them, it simplifies the management of the network connections needed to develop the ChatOps environment.

The modules that compose the backend side of the architecture can be seen in Fig. 4.12. These components are the following:

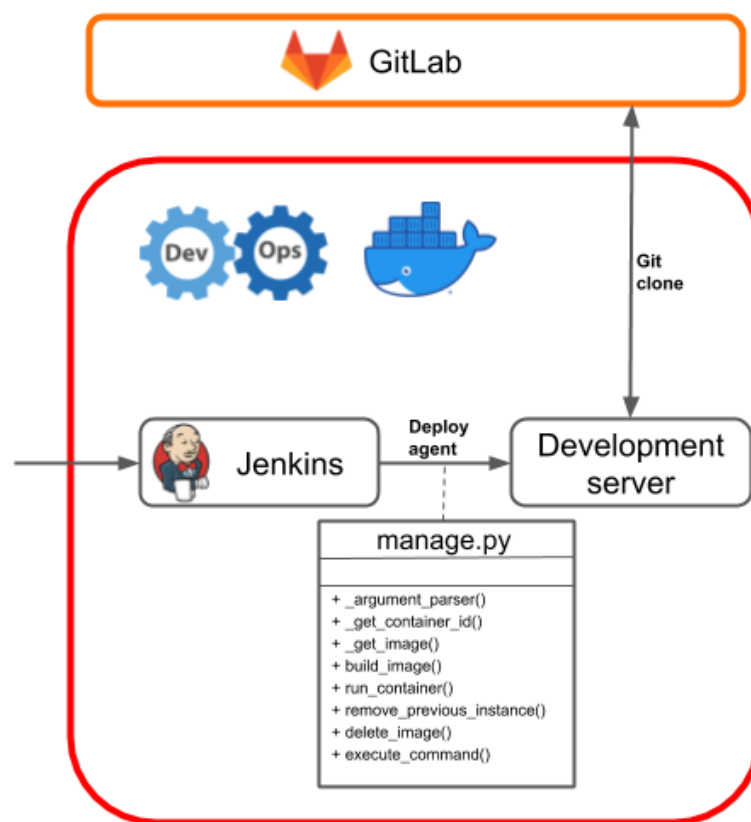


Figure 4.12: Backend section of the architecture

1. A **Jenkins instance** connected with the RASA action server. It will perform the changes over the Development server by launching *jobs* whenever the action server triggers them.
2. A **Deploy agent**, managed by Jenkins. It will be in charge of executing the changes ordered by Jenkins over the Development server.
3. The **Development server** that is configured and managed by the user messages.

4. The **GSI GitLab**, used by the Development server to pull information when the user demands it.

In the following sections, we are going to describe the configuration of these modules and how they interact with each other.

4.3.1 Jenkins

When setting up a DevOps environment, automation is one of the most important pillars. Having automation flows simplifies managing a complex environment with many tools that interact with each other. As DevOps environments grow in functionalities and complexity, automation tools get more relevance in the scene.

As we introduced in the Sect. 3.2.5, Jenkins is an automation server that allows automating tasks, simplifying the implementation of *Continuous integration/Continuous delivery* (CI/CD), one of the milestones of a DevOps environment. It allows the developer to automate tasks like deploying the environment, running tests and build pipelines to run a series of tasks while having control of the operations made.

In our project, Jenkins makes the central part of the backend side of the architecture. It is the connection point between the chatbot module, in charge of talking to the user and process the conversation, and the development server, the objective to be managed by the environment. From the petitions of the user, Jenkins launches automation jobs that execute the changes requested.

Using Jenkins in our environment provides several key features that make possible the consecution of this project. Some of the most important characteristics are the following:

- Using the official **Jenkins Docker image**, deploying a new Jenkins instance can be done fast.
- Jenkins offers a **web user interface** that allows us to quickly configure the server and personalize it to the environment needs.
- As the **python library** we introduced in the Sect. 4.2.2.4 exists, using it we could develop methods to communicate the Rasa Action Server with the Jenkins server directly.
- Jenkins offers a **history of executions and logs** that makes it easier to administrate the environment.

- The developer can extent the Jenkins out-of-box functionalities using **plugins**.

To run the Jenkins instance, the following command must be launched:

Listing 4.12: Command to deploy Jenkins container

```
docker run --privileged -d -p 50000:50000 -p 8080:8080 -v jenkins-data:/var
  /jenkins/_home -v /var/run/docker.sock:/var/run/docker.sock --name
  jenkins-master myjenkins
```

This command launches the container and exposes the port 8080 where the webhook is deployed, the port 50000 for Java API and passes the docker socket to the container to allow Jenkins to operate with Docker. In the project, this container is launched in the Ubuntu virtual machine, along with the other services.

To manage the server, Jenkins offers a web user interface at “http://localhost:8080”. There, the user can create jobs and pipelines, install plugins or configure the instance. The portal also gives information about the status of the jobs and pipelines, such as the last successful build, the duration of the last failed build. Fig. 4.13 shows a snapshot of the user portal:

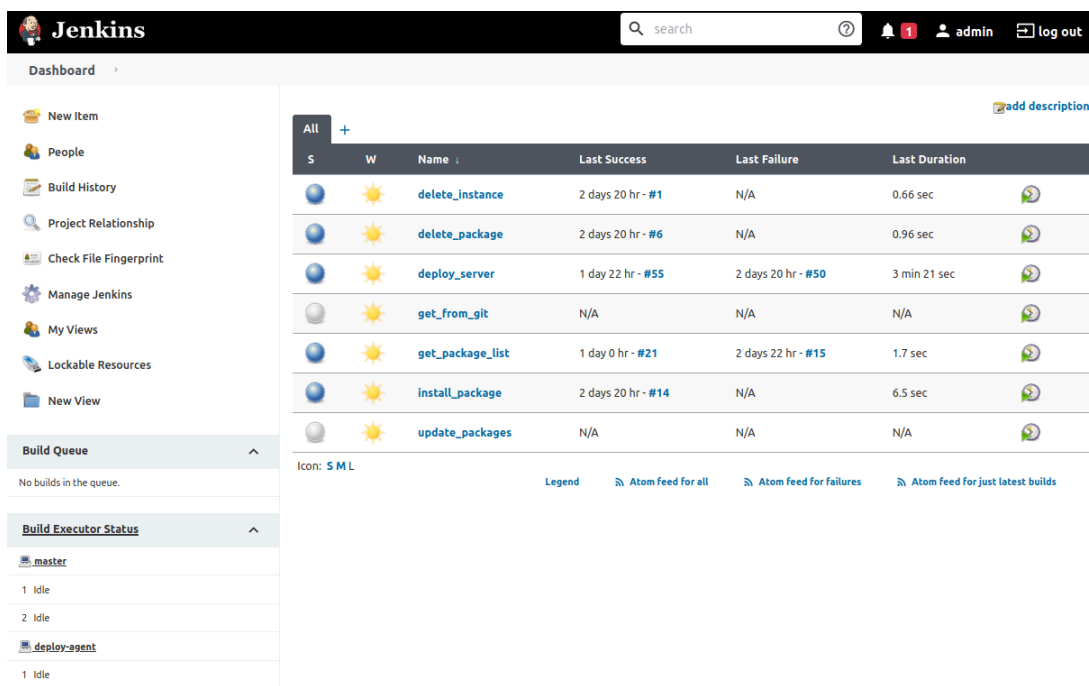


Figure 4.13: Jenkins web user interface.

Jenkins automates tasks by executing previously defined activities. These activities

may be *jobs* or *pipelines*, which are multi-stepped jobs defined by a text file that defines the operations to perform called *Jenkinsfile*. In this project, we decided to implement one job per operation to perform in the Development server to minimize the complexity of the builds and offer a lesser response time as possible.

Jobs are created via the web user interface. At the job creation, the user can adjust some of the parameters like inserting a description, take objects from Git or allow automatic triggers. However, the main part of a job configuration is the “build” step. There, the user can specify what the job does. It has options like invoking scripts, operations over docker clouds or simply execute commands in the shell. In this project, all jobs use the build step *execute shell*, which allows running a command in the Jenkins agent executing the job.

To perform the changes in the development server, we created a job for each custom action in Rasa Action server that makes changes in the server. The jobs developed are the following:

- **Check_server_state:** Checks the state of the Development server.
- **Deploy_server:** Initializes the development server or resets its content if it is already running. If it is stopped, it resumes its execution.
- **Stop_server:** Stops the development server.
- **Install_package:** Install a package passed from the Action server.
- **Delete_package:** Removes a package passed from the Action server.
- **Get_package_list:** Get a list with the installed packages.
- **Update_packages:** Updates the packages installed in the server.
- **Get_from_git:** Downloads the package list from GitLab and installs the packages

These jobs are triggered by the Rasa Action server when the actions described in the Sect. 4.2.2.4 are predicted by the dialogue policies. When an action is triggered, it is assigned to a Jenkins agent that performs the commands specified in the “build” step defined at the creation. The results of the jobs are visible from the Jenkins web interface by clicking on the job name. There, in the left column a history with the builds can be found.

The user can also get information about a specific build by clicking on it in the build history column. Jenkins offers information such as the time it took to complete, which agent executed the job or when it was launched. The user can also get the console output generated

when the job was built. In the List. 4.13, the console output for a “install_package” build is shown:

Listing 4.13: Console output for a ”install_package” build

```
Started by user admin
Running as SYSTEM
Building remotely on deploy-agent in workspace /home/jenkins/workspace/
install_package
[install_package] $ /bin/sh -xe /tmp/jenkins8609956344400738137.sh
+ sudo python /home/jenkins/scripts/manage.py -m install_package -p
standalone==1.0.1
Collecting standalone==1.0.1
  Downloading standalone-1.0.1.tar.gz (2.0 kB)
Collecting django
  Downloading Django-3.1.4-py3-none-any.whl (7.8 MB)
Collecting asgiref<4,>=3.2.10
  Downloading asgiref-3.3.1-py3-none-any.whl (19 kB)
Collecting sqlparse>=0.2.2
  Downloading sqlparse-0.4.1-py3-none-any.whl (42 kB)
Collecting pytz
  Downloading pytz-2020.5-py2.py3-none-any.whl (510 kB)
Building wheels for collected packages: standalone
  Building wheel for standalone (setup.py): started
  Building wheel for standalone (setup.py): finished with status 'done'
  Created wheel for standalone: filename=standalone-1.0.1-py3-none-any.whl
  size=2715 sha256=
  ee6482fe6accbbf39380a54e0deb02f0ed45b80ce4873d32f9394707e1edc1d1
  Stored in directory: /root/.cache/pip/wheels/da/7f/c8/6
  c62017267ee50e6dec22b0152178daabbd1752b9bebfd3b5b
Successfully built standalone
Installing collected packages: sqlparse, pytz, asgiref, django, standalone
Successfully installed asgiref-3.3.1 django-3.1.4 pytz-2020.5 sqlparse
-0.4.1 standalone-1.0.1

Finished: SUCCESS
```

The console output also gives information about the build. In the List. 4.13, we can see that the job was launched by user “admin”, which is the one configured in the Rasa Action server to call Jenkins, the agent that executed the job or the command launched. This console output will be gotten by the Action server through calls to the Python API, and after processing, will be uttered to the user to give the result of the requested operation. For executing all the jobs, we created a Jenkins agent that will be responsible for executing

a Python script to make the changes into the Development server.

4.3.2 Deploy agent

To perform jobs, Jenkins can either execute them by itself or delegate the task to an agent. Using agents as job executors have many advantages, such as balancing the charge between different nodes instead of the Jenkins server or allowing to specialize the agents to improve the performance. As Jenkins gives information about the load in the agents, temporary agents could be created on-demand when orders arrive to fulfill the demand.

For this project, we decided to build a custom deploy agent to simplify the logic of the Jenkins jobs. This is achieved by using the same agent with a unique Python script to perform all the changes in the development server. This allows us to configure once the agent with the needed dependencies, packages and permissions and delegate all the logic into it, instead of making all these configurations in the Jenkins server.

The *Deploy agent* is a docker container based on an *Ubuntu 18.04* docker image. This agent is meant to be accessed by Jenkins to execute the jobs that the Rasa Action server triggers by launching custom actions. To deploy the agent, we wrote a *dockerfile* that can be seen in the List. 4.14:

Listing 4.14: Dockerfile for the Deploy agent

```
FROM ubuntu:18.04

# Make sure the package repository is up to date.
RUN apt-get update && \
    apt-get -qy full-upgrade && \
    apt-get install -qy git && \
    apt-get install -qy sudo && \
    apt-get install -qy vim && \
# Install a basic SSH server
    apt-get install -qy openssh-server && \
    sed -i 's|session    required    pam_loginuid.so|session    optional\n        pam_loginuid.so|g' /etc/pam.d/sshd && \
    mkdir -p /var/run/sshd && \
# Install jdk-8 to configure as Jenkins build agent
    apt-get install -qy openjdk-8-jdk && \
# Cleanup old packages
    apt-get -qy autoremove && \
# Add user jenkins to the image
    adduser --quiet jenkins && \
```

```
# Set password for the jenkins user (you may want to alter this).
    echo "jenkins:jenkins" | chpasswd && adduser jenkins sudo &&\
    mkdir /home/jenkins/.m2

# Install python
RUN apt-get update \
    && apt-get install -qy --force-yes python3-pip python3-dev \
    && cd /usr/local/bin \
    && ln -s /usr/bin/python3 python \
    && pip3 install --upgrade pip \
    && pip3 install docker-py

#ADD settings.xml /home/jenkins/.m2/
# Copy ssh keys
COPY .ssh /home/jenkins/.ssh

# Copy scripts and development machine deployment data
COPY scripts /home/jenkins/scripts
COPY development_server /home/jenkins/development_server

RUN chown -R jenkins:jenkins /home/jenkins/.m2/ && \
    chown -R jenkins:jenkins /home/jenkins/.ssh/

# Standard SSH port
EXPOSE 22

CMD ["/usr/sbin/sshd", "-D"]
```

This dockerfile is used to generate a docker image ready to be launched with some previous configuration:

- Install the packages git, sudo and vim, used to perform configurations in the agent after the deployment in case of need.
- Install an ssh server to access the agent from Jenkins.
- Install **openjdk-8** to configure this container as a Jenkins build agent.
- Add a “jenkins” user to the container.
- Install Python 3 and *docker-py* package to execute the script used to manage the Development server.
- Copy files from the project to the container, such as the Development server dockerfile, scripts and ssh keys.

- Expose the port 22 and run the ssh service as a daemon.

Then, we build the image and run it with the command “`docker run -p 22222:22 -v /var/run/docker.sock:/var/run/docker.sock deploy-agent`”, that forwards the port 22222 of the virtual machine with the port 22 of the container and mounts the docker socket as a volume to allow the agent to make changes on containers.

To make this container a Jenkins build agent, some configuration must be performed through Jenkins’ web interface. The developer must register the agent as a “Node”, and give information such as the connection method to the agent or the availability policy for the agent, aside from having *Java JDK* installed. For our deploy agent, we called the node “deploy-agent” and specified to connect with Jenkins using SSH at the port 22222 and with the user “jenkins”. Also, we modified the jobs created in Jenkins to always be executed by this build agent.

The deploy agent uses a Python script to perform all the changes. This file, called “manage.py”, is meant to implement all the methods to interact with the Developer server container through the Python Docker API [30]. The script, located in `/home/jenkins/scripts`, is executed by the Jenkins jobs with the following command:

Listing 4.15: Command used to launch the script “manage.py”

```
sudo python /home/jenkins/scripts/manage.py -m "mode" -p "package"
```

The “mode” parameter in the command indicates the script the operation to be performed. Each one of supported modes are introduced in the “build” step of the jobs, depending on which job is executed. The relation between Rasa actions, Jenkins job launched and the mode selected is shown in the Table 4.1:

Therefore, the *manage.py* script is in charge of performing the changes in the Development server. As it is the main actor in the Development server configuration, we are going to describe the code structure and use. The script is structured in three main blocks of methods:

1. **Inner methods** used to retrieve information about the Development server container state.
2. **Containers manipulation methods**, used to perform changes over the Development server container.
3. The **main method** which is executed when the script is called by the job.

Rasa Action	Job name	Mode
action_server_state	check_server_state	state
action_launch_server	deploy_server	init
action_remove_server	delete_instance	remove_server
action_update_packages	update_packages	update_packages
action_get_from_git	get_from_git	get_from_git
action_install_package	install_package	install_package
action_delete_package	delete_package	delete_package
action_list_package	get_package_list	get_packages

Table 4.1: Relation between Rasa action, Jenkins job and manage.py mode.

As we stated before, the interaction with the Development server will be achieved via the Docker python library. The *inner methods* are auxiliary functions that primarily focus on retrieving information from the Docker platform about the Development server container. These methods are the following:

- *_argument_parser()*: It allows to add arguments to the command, such as the -m argument used to set the mode or -p in case of a package is passed by the action server, for example, to install a package asked by the user.
- *_get_container_id(client)*: Loops through the running containers and returns the id of the container with the “development-server” tag. This id will be used by other commands to operate on the container.
- *_get_image(client)*: Checks if an image with the name ‘development-server’ exists in the Docker platform, and returns True or False depending on it. This will be used in tasks like initializing the development server.

The code for these methods can be seen in the List. 4.16:

Listing 4.16: Inner methods in the script “manage.py”

```

# Parser for arguments at script launch
def _argument_parser():
    parser = argparse.ArgumentParser()
    # Add long and short argument
    parser.add_argument("--mode", "-m", help="set command mode")
    parser.add_argument("--package", "-p", help="package name")
    # Read arguments from the command line
    args = parser.parse_args()
    return args

# Returns the ID of a container if it's running. If it can't be find,
    return False
def _get_container_id(client):
    for container in client.containers():
        if 'development-server' in container['Image']:
            found_container = True
            running_container = container['Id']
            return running_container
    running_container = 'NONE'
    return running_container

# Returns true or false if the image for the given environment exists
def _get_image(client):
    image_name = 'development-server'
    if image_name in str(json.dumps(client.images())):
        print(image_name + ' image found')
        return True
    else:
        print(image_name + ' image not found')
        return False

```

Using these methods, we can get information to operate with the Development server docker. The *container manipulation methods* are in charge of interacting with the Docker platform to operate with the Development server. The methods developed are listed below:

- *build_image(client, dockerfile_path)*: Given the dockerfile path, executes the Docker *build* method, which generates a Docker image of the Development server tagged “development-server”. Uses the Docker method *build*, equivalent of running “docker build path/to/dockerfile -t development-server”
- *run_container(client)*: Creates and runs the Development server container. The created container will have the name ‘development-server’ and will forward port 22 to

the virtual machine port 22000. It uses the Docker method *start*, equivalent to the bash command “docker run -p 22000:22 development-server”.

- *remove_previous_instance(client)*: Using the inner method *_get_container_id*, checks if there is any Development server container running. If it finds any, kills the container and removes it. It uses the *remove_container* method, equivalent to “docker rm *container_id*”.
- *delete_image(client)*: Using the inner method *_get_image*, search for a “development-server” image to delete it. If it finds it, checks if there is any running container to remove it with the method *remove_previous_instance*. Uses the method *remove_image*, which is equivalent to running “docker rmi development-server” in the console.
- *execute_commands(client, command, workdir)*: Using the container id, launches the passed command in the container. If a *workdir* is passed as an input, the command will be launched at the given location inside the container. It is equivalent to using “docker exec *container_id* *command*” in the console.

The code for these commands is shown in the List. 4.17:

Listing 4.17: Container manipulation methods implemented in the script “manage.py”

```
# Generates a docker image from a provided dockerfile
def build_image(client, dockerfile_path):
    # Append the datetime of the last build
    # _append_datetime(deploy_env)
    # Build the image
    print('Building image, this may take some time...')
    response = [line for line in client.build(
        path=dockerfile_path, rm=True, tag='development-server'
    )]
    # Format of last response line expected: {"stream":"Successfully built
    # 032b8b2855fc\\n"}
    print('Dockerfile build result: ' + str(response[-1].decode('utf-8'))

# Creates and runs a docker container
def run_container(client):
    # Export port for ssh
    exposed_port = 22000

    container = client.create_container(
        image='development-server',
        name='development-server',
```

```

        detach=True,
        ports=[22],
        host_config=cli.create_host_config(port_bindings={
            22:exposed_port
        })
    )
    client.start(container=container.get('Id'))
    return 0, exposed_port

# Stops and removes a container
def remove_previous_instance(client):
    running_container = _get_container_id(client)
    if running_container != 'NONE':
        print('Killing local container')
        client.kill(running_container)
        print('Removing container')
        client.remove_container(running_container)
        return 0
    else:
        print('No previous containers found.')

# Deletes the existing image of a container
def delete_image(client):
    if _get_image(client):
        if _get_container_id(client) == 'NONE':
            print('No running containers found, deleting image...')
            client.remove_image('development-server')
        else:
            print('Found running containers, deleting image...')
            remove_previous_instance(client)
            client.remove_image('development-server')
        print('Image succesfully deleted')
        return True
    else:
        print('Image "development-server" not found')
        return False

# Execute commands towards the development server
def execute_commands(client, command, workdir=''):
    # Get the ID of the container
    container_id = _get_container_id(client)
    if container_id != 'NONE':
        # Create exec instance with a directory if it is passed.
        if workdir:
            exec_id = client.exec_create(container_id, command, workdir=

```

```
        workdir)
    else:
        exec_id = client.exec_create(container_id, command)
        # Launch exec
        command_result = client.exec_start(exec_id)
    else:
        command_result = 'Container not found. {} was not launched'.format(
            command)
    return command_result
```

After listing the auxiliary methods used to interact with the Docker platform, we will describe the *main method*. This method is run when a Jenkins job executes the script. Firstly, it defines a variable *cli*, which instantiates a Docker client and is used by any method that communicates with docker. This allows the script to communicate with the Docker platform and is created with the *base_url* field with value *unix://var/run/docker.sock*. This is the Docker socket running in our virtual machine, which is mounted as a volume when the Deploy agent is launched. Executing the commands over the shared Docker socket allows us to have all the Docker containers running on the same platform, easing the communication between them.

The next step would be to check the arguments used to launch the script. As shown in the Table 4.1, every job uses a different *-m* mode parameter. This simplifies the logic of the main method and use if statements depending on the mode to perform the actions required. The code of the main method is available in the List. 4.18.

Listing 4.18: Main method of "manage.py"

```
if __name__ == '__main__':
    cli = Client(base_url='unix://var/run/docker.sock')
    # Argument parsing
    args = _argument_parser()
    deploy_mode = args.mode
    if deploy_mode == 'install_package' or deploy_mode == 'delete_package':
        package = args.package

    if deploy_mode == 'status':
        if _get_container_id(cli) != 'NONE':
            print('Development server is running!')
        else:
            print('Development server is not currently running.')
    if deploy_mode == 'init':
        # Clean up older images
```



```

        delete_image(cli)
        # Build new images
        build_image(cli, '/home/jenkins/development_server')
        # Create and run new container
        result, exposed_port = run_container(cli)
        print('Created container with the image development-server,
              exposing ssh at port {}'.format(str(exposed_port)))
    elif deploy_mode == 'remove_server':
        remove_previous_instance(cli)
        print('Removed development-server instance')
    elif deploy_mode == 'update_packages':
        print(execute_commands(cli, "pip list --outdated --format=freeze |
              grep -v '^\\-e' | cut -d = -f 1 | xargs -n1 pip install -U").
              decode('utf-8'))
    elif deploy_mode == 'get_from_git':
        print(execute_commands(cli, "sh bash_git_clone.sh", workdir='/home/
              developer'))
        print('Successfully installed packages from GitLab')
    elif deploy_mode == 'install_package':
        print(execute_commands(cli, 'pip install {}'.format(package)).
              decode('utf-8'))
    elif deploy_mode == 'delete_package':
        print(execute_commands(cli, 'pip uninstall -y {}'.format(package)).
              decode('utf-8'))
    elif deploy_mode == 'get_packages':
        print(execute_commands(cli, 'pip freeze').decode('utf-8'))
    else:
        print('Unsupported mode {}'.format(deploy_mode))

```

The commands and methods invoked in the main method are listed in the Table 4.2. It is important to note that the majority of the options can be fulfilled by launching a command in the Development server container with the method *execute_commands*. Another important detail is that, as the Development server is meant to be a Python development machine, the package management is made via Pip and the actions that manage packages are implemented using it.

However, the “init” mode, launched when the Jenkins job “deploy_server” is executed, inquires in cleaning up the previous execution with *execute_commands*, build a fresh Development server docker image and run the container. The other exception is the *get_from_git* mode. This mode, launched when the Jenkins job “get_from_git” is requested by the Action server, gets from GitLab a *requirements* file with a package list and installs it in the Development Server. In this case, as it is necessary to execute several commands, the *execute_commands* method was not ideal. To perform this task, we created a bash script,

Script mode	Methods invoked	Parameters
init	delete_image build_image run_container	Docker client, docker-build_path = /home-/jenkins/development_server
remove_server	remove_previous_instance	Docker client
update_packages	execute_commands	Docker client, command = TODO
get_from_git	execute_commands	Docker client, command = sh bash_git_clone.sh, workdir = /home/developer
install_package	execute_commands	Docker client, command = pip install package
delete_package	execute_commands	Docker client, command = pip uninstall package
get_packages	execute_commands	Docker client, command = pip freeze

Table 4.2: Commands launched for the supported modes.

bash_git_clone.sh. This script is copied to the Development machine when the Docker image is built, and allows to clone the *development_requirements.txt* file from the GitLab server and install it in the server via “pip install -r development_requirements.txt”. The content of the bash script is shown in the List. 4.19:

Listing 4.19: Content of “bash_git_clone”

```
#!/bin/bash
# Cleanup old files
echo `rm -f requirements.txt`
echo `rm -rf tfm-ignaciocervantes`
# Clone the repo and checkout the development requirements file
```

```
echo `git clone -n http://icervillalon:$TOKEN@lab.gsi.upm.es/TFM/tfm-
    ignaciocervantes.git --depth 1`
echo `git --git-dir /home/developer/tfm-ignaciocervantes/.git checkout HEAD
    development_requirements.txt`
echo `pip install -r development_requirements.txt`
```

The results of executing any of the *manage.py* will be printed in the mentioned Jenkins “console output” section of the corresponding job. That information will also be replied to the user, as the actions implemented in the Rasa Action server extracts the results from the console output to build the response.

4.3.3 Development server

In a corporate environment, it is usual to divide the accesses and management between different departments. This allows each team to specialize in their tasks and accelerate the possible problems that may appear. For example, in an IT company it is common to have development teams with limited access to the operational tools and give them just the permissions needed to operate, while the maintenance and development of these tools rely on other teams. In that kind of environment, giving the users tools to work while assuring that the tools are safe is an important feature.

The ChatOps paradigm, and therefore this project, aim to this objective. In our project, we implemented the Development server as the component that the whole environment is meant to manage by processing the user messages in a chat app. This emulates a usual corporate environment, where a machine used to execute code is given to the developers but managed externally, so the developer only gets limited access to management methods like installing packages or getting the status.

The designed Development server is a Python 3 development machine based on an Ubuntu 18.04 Docker image. To describe its functionalities, the dockerfile used to build the container is shown in the List. 4.20:

Listing 4.20: Dockerfile used to deploy Development server container

```
FROM ubuntu:18.04

# Make sure the package repository is up to date.
RUN apt-get update && \
    apt-get -qy full-upgrade && \
    apt-get install -qy git && \
```

```
# Install a basic SSH server
apt-get install -qy openssh-server && \
sed -i 's|session    required    pam_loginuid.so|session    optional
    pam_loginuid.so|g' /etc/pam.d/sshd && \
mkdir -p /var/run/sshd && \
# Cleanup old packages
apt-get -qy autoremove && \
# Add user developer to the image
adduser --quiet developer && \
# Set password for the developer user
echo "developer:developer" | chpasswd && \
mkdir /home/developer/.m2

RUN apt-get update \
&& apt-get install -qy --force-yes python3-pip python3-dev \
&& cd /usr/local/bin \
&& ln -s /usr/bin/python3 python \
&& pip3 install --upgrade pip

#ADD settings.xml /home/developer/.m2/
# Copy authorized keys
COPY .ssh /home/developer/.ssh
# Copy bash script to get git changes
COPY bash_git_clone.sh /home/developer/bash_git_clone.sh

# Standard SSH port
EXPOSE 22

CMD ["/usr/sbin/sshd", "-D"]
```

The dockerfile will build an image following the steps below:

- Install git to get the packages from GSI GitLab.
- Install an SSH server, that will be used by the developers to access to the server remotely.
- Create a “developer” user.
- Install Python 3 and Pip as the package manager.
- Copy the .ssh folder that contains the public key to connect with the server and the bash_git_clone.sh shell script used to retrieve the package list from GitLab.

- Expose the port 22 of the container.

These steps will allow the server to have the needed packages and make the server accessible for the *developer* user using SSH. This dockerfile is meant to be built by the deploy agent when the Jenkins job *deploy_server* is executed, as well as running the container. After running the container, it will publish the port 22 in the virtual machine port 22000, so the machine could be reachable using the command “ssh -p 22000 developer@localhost:22000”.

The complete backend side flow to make operations in the development server can be seen in Fig. 4.14. After an action is predicted by the Rasa dialogue policies, the custom action calls to a Jenkins job. That job will launch the script *manage.py* using the command shown in the List. 4.15 with a specific mode and package if the Rasa action passed it. Then, the Deploy agent will connect to the Development Server through the Docker socket and apply the changes using Docker methods depending on the desired changes.

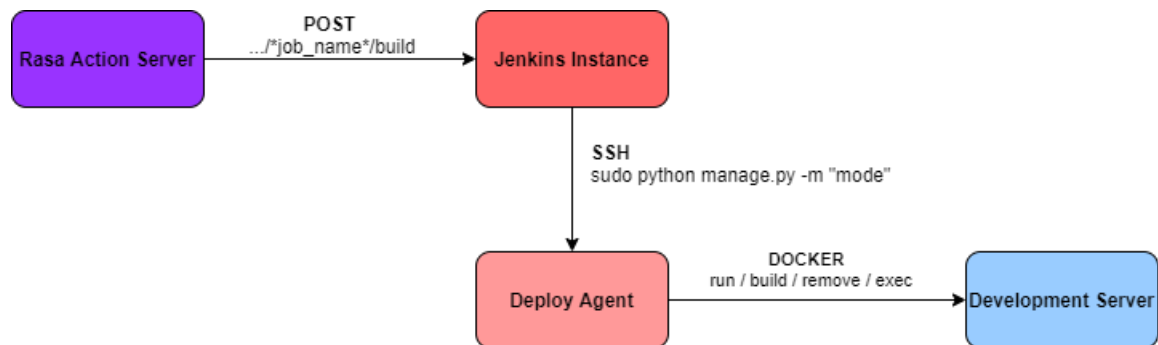


Figure 4.14: Configuration flow from Rasa Action server towards Development Server.

4.4 Conclusions

In this chapter, we described the architecture of the solution developed for this project. First, a diagram with the architecture is shown to give a global vision of the development and how the components interact with each other.

From the architecture, we examined the components one by one showing the developments made for each one. To simplify the explanation, we divided the architecture into the *user interactive side*, consisting of the Telegram and Rasa implementations, and the *backend side* with Jenkins, the deploy agent and the Development server.

For the *user interactive side*, we detailed the Telegram integration for our chatbot. For Rasa Open Source, we examined the parts involved in the conversation with the user and

exposed the work made to train the chatbot Natural Language Understanding module, the dialogue policies and the actions performed in the Action server.

Finally, we described the *backend side* of the architecture. To do so, we covered how the chatbot could reach the Jenkins instance to configure the rest of the components. Also, we described the code used to manage the Development server, as well as the configuration needed to deploy each one of the containers used to build the environment.

CHAPTER 5

Case study

In this chapter, we are going to describe a real-world use scenario of the project. To do so, the interactions between the agents and tools will be explained, as well as the steps which are taken to perform a use case of the environment.

5.1 Scenario overview

The proposed scenario is an *Information Technology* consultancy company. This company is a medium-sized business that offers software solutions to its clients. The software developed vary in characteristics and languages, so there are several teams focused on different technologies to specialize and offer short delivery times. This company embraces some of the state-of-the-art development paradigms like DevOps, and has a wide set of automation tools available for the teams and managed by an IT organization department. These tools range from automation servers to servers to test the developments locally before releasing them. As these tools are managed by the IT department, the developers must pass through the IT crew to validate the changes and apply them to the systems, which slows the releases.

To accelerate the time to market, the IT department wants to implement some tools to give the developers more independence to the development teams. To do so, they choose to implement a ChatOps environment. With it, the IT department aims to provide developers a bounded interface with the operation tools that can be accessed from different systems and can be extended in the future to provide further functionality. The objective is to allow the developers to reach the development servers used to test the packages via a chatbot that can perform a limited set of operations determined by the IT team, so some of the tickets that had to be managed by them can be automated.

After talking with the development teams, they determined some of the key features that the ChatOps environment should have. First, the chatbot should be reachable from a chat application, so it could be integrated into the team conversations easily. Also, the chatbot should be able to reset the environment and to install the packages requested by the user. Finally, it would be desirable to allow the chatbot to install packages retrieved from the company Git server.

After exposing the case study, we will list the actors implied in the scenario in Table 5.1. The *developer* is a member of a Python development team that uses a development server managed by the IT department. Before the new ChatOps environment, the developer had to open a ticket to IT to perform operations like installing a package in the server. The second actor is the *chatbot* deployed by the IT department. It is meant to interact with the ACT-1 to allow it to make some operations into the development server without the action of the IT team to accelerate the developer workflow.

The modules implied in the scenario are the following:

- A **Telegram** client. Could be used in any app or operative system, but for this case

Actor	Role	Description
ACT-1	Developer	Representative of a development team. Uses the automation tools provided by the IT team to test the code before the release.
ACT-2	Chatbot	AI conversational assistant developed by the IT team. Connects the developers with the DevOps tools by conversation through a chat application by launching pre-defined methods.

Table 5.1: Actors involved in the scenario.

study we will use the web client.

- The **Rasa Open Source** framework, deployed in an Ubuntu 18.04 virtual machine.
- The **Jenkins** instance deployed in a Docker container.
- The **Development Server**, also deployed in a Docker container.

In this scenario, we will describe some use cases that involve the agents shown in the Table 5.1 and use the ChatOps environment developed in this project. To show how the operations are done, we will attach screen captures to picture the user experience.

5.2 Development server deploy

The IT department has finished the development and implementation of the new ChatOps environment. Now, they want the development team to start using it to deploy the development server used to test the code, instead of the former servers. To make the First Office Application (FOA), the IT department asked the Python development team to deploy the server and perform an initial configuration. This first execution involves checking the current status of the server to see if any instance exists. As the environment has not been used yet, the developer should launch a new instance of the development server.

As we mentioned, this FOA will involve a developer and the chatbot environment. The first step that the developer must take consists of beginning a conversation with the chatbot. To do so, the developer should use a Telegram account to log in to it from the preferred application, in our case, the web UI accessible from the URL <https://web.telegram.org/>. There, the user should search from the chatbot using the Telegram search engine by “ChatopsAgent”, as seen in Fig. 5.1:

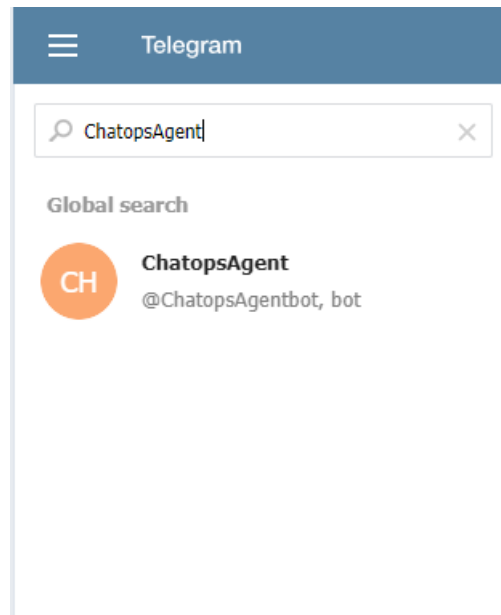


Figure 5.1: Search made to found *ChatopsAgent* bot

By clicking on the *ChatopsAgent* user in the list and clicking on the “Start” button in the right column, a new conversation with the bot will be launched, and the conversation will be shown in the right column of the interface. There, a “\start” message is automatically launched by Telegram to initialize the bot if it is not already launched in their API. Now, the developer can communicate with the chatbot using the chat box shown in Fig. 5.2, found at the bottom of the right column.

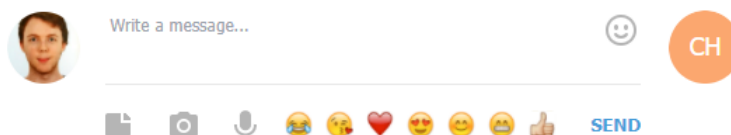


Figure 5.2: Chat box used to send messages.

Once the conversation is open, the developer will launch the development. To do so, it sends a message asking for it in natural language. In this case, as seen in Fig. 5.3, the message was “launch the server”, but using other words like “deploy the development

server” or “launch the server instance” would be equally recognized by the Natural Language Understanding (NLU) module as the intent *launch_server*.

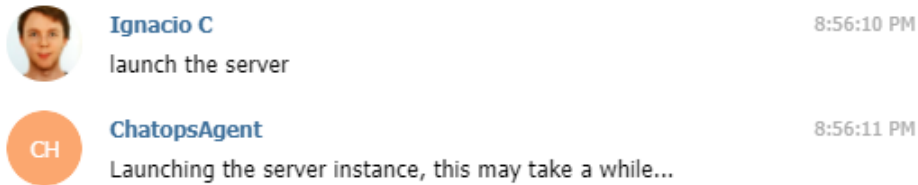


Figure 5.3: Message used to launch the development server.

This message arrives at the Rasa Agent via the configured endpoint in the configuration. Then, the Rasa NLU classifier predicts the intent of the message by using the model created with the training data. As seen in the Fig. 5.4, it correctly predicted the intent “launch_server”. Then, the Rasa core will predict the action to take based on the training data used to train the core. In this case, it launched the action “action_inform_user” to inform the developer that the server was being launched, as operations performed may take a while. After this message, the chatbot will be busy and will not respond to the user until the operation is finished or get timed out.

```
2021-01-06 20:56:09 DEBUG rasa.core.processor - Received user message 'launch the server'
with intent '{id: -6050623937800831848, name: 'launch_server', confidence: 0.99885100126
26648}' and entities '[]'
```

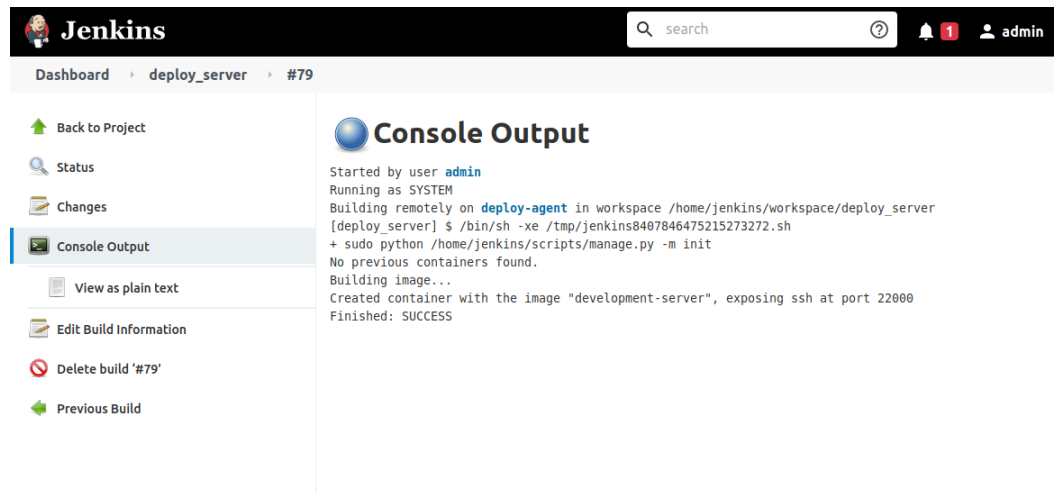
Figure 5.4: Intent recognition from the user message.

Then, the action “action_launch_server” is executed. This action, performed by the Rasa Action server, calls the Jenkins instance to perform the operation requested. In this case, the action called the Jenkins job “deploy_server”. This job is executed by a deploy agent, which launches a python script to do the configuration using Docker commands.

When the Jenkins Job is finished, a console output can be seen through the Jenkins web interface as shown in Fig. 5.5. This console output offers information about the operations performed to deploy the development server. This output is replied to the Rasa action petition and used to compose the message sent back to the developer after the job execution ended.

When the Rasa action server gets the response from Jenkins, ends the action by dispatching a message to the developer. This message is delivered by the Rasa Agent, which sends the message to the Telegram Bot API. Then, the API delivers the message to the developer with the execution results.

This message, shown in Fig. 5.6, offers the console output from the Jenkins job. Its

Figure 5.5: Jenkins interface after executing *deploy_server*

content states the steps taken to deploy the server and a port where the developer can connect to the development server. To check that everything went as expected and the server is correctly running, the developer sends a message to check the status of the development server. As Fig. 5.7 shows, by sending the message “check the server”, the action “action_server_state” is launched by Rasa and triggering a job in Jenkins that checks the health of the server. The response “Development server is running!” means that the server is correctly deployed and running.

Execution finished! Console output:
No previous containers found.
Building image...
Created container with the image "development-server", exposing
ssh at port 22000

8:56:24 PM



Ignacio C

check the server

8:56:37 PM



ChatopsAgent

Checking server state...

8:56:38 PM

Development server is running!

8:56:47 PM

Figure 5.6: Message replied by Rasa

Figure 5.7: Checking server status

5.3 Package installation

As presented in the subsection 5.2, the developer has deployed the server and checked that it was running. However, the Development server that was deployed only has installed Git, an SSH server, Python 3 and the Pip package manager. To use it, the developer needs to install packages such as *pytest* to check the code quality before releasing it. To do so, the IT department developed methods to allow the chatbot to manage the packages installed in the development servers. By requesting it to the chatbot, the developer can install, remove, list or even get from Git the packages that it needs.

To show the interactions for managing the packages, in this section we will show how the developer can install a package and list the installed packages to check that the package is correctly installed. The developer wants to install the testing suite *pytest* in the development server. To do so, it sends the message “install pytest” to the ChatopsAgent via Telegram.

In this case, the intent caught from the message received is “install_package”. In this case, Rasa NLU also caught the entity *package* with the value “pytest”. This information will be passed by the dialogue policies to the action “action_inform_user” to inform the developer that the operation is being performed and “action_install_package”, which will call a Jenkins job to install the package detected in the message received from the user. In this case, the response to the developer will consist of the console output as if the developer itself installed the package, as shown in Fig. 5.8. As the figure shows, the command installed the package required, as well as all the dependencies needed.

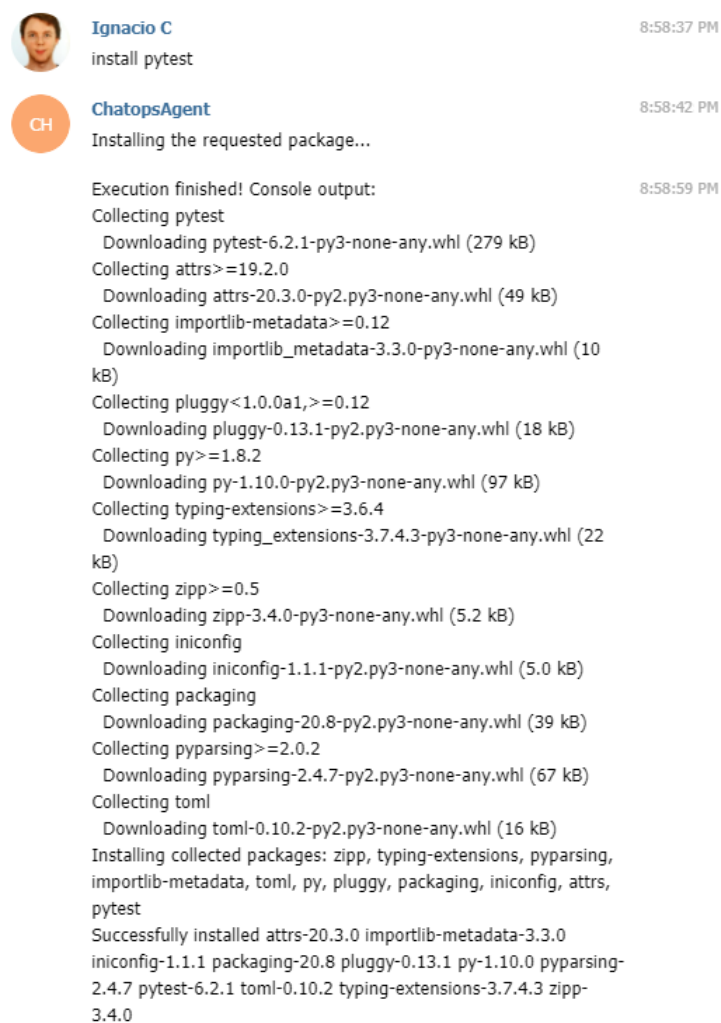


Figure 5.8: Response from Rasa after installing *pytest*

After receiving the response from the chatbot, the developer wants to know which packages are installed in the development server. To obtain the list, the developer sends the message “list the packages”. As the other messages, it is processed by Rasa and Jenkins to get the information from the development server. When the Jenkins job finishes, it sends the list with each package installed and their versions as presented in Fig. 5.9:

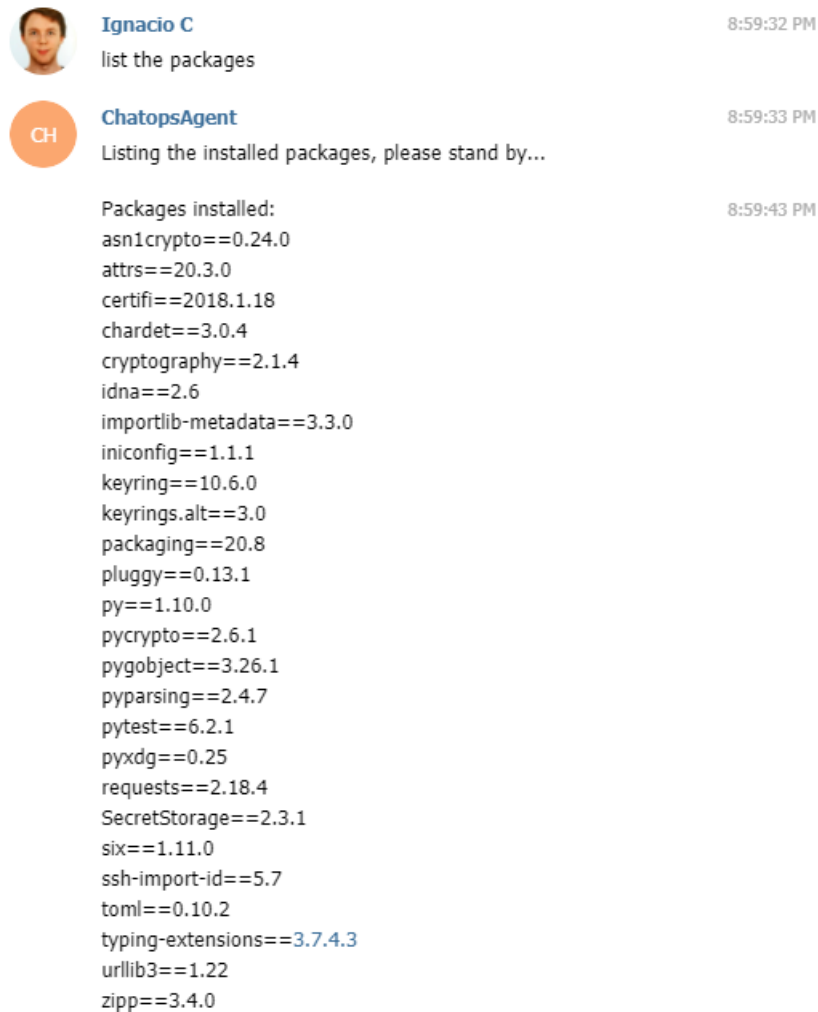


Figure 5.9: Package list after installing *pytest*

Finally, the developer wants to access the server via SSH to test if the server is reachable and the packages were successfully installed. To connect with the development server, as seen in the response message shown in Fig. 5.6, port 22000 is exposed. At deployment, the user *developer* is created inside the server, and the developer’s SSH public key is copied. To access the development server with his user, the developer launches the command “ssh -p 22000 developer@172.17.0.1” and introduces his password.

Once connected with the development server, the developer prints the installed package list to check if the list retrieved by the chatbot is correct. As shown in Fig. 5.10, the package list is correct and the package *pytest 6.2.1* is installed:

```

developer@f531d188dded:~$ pip list
Package            Version
-----
asn1crypto          0.24.0
attrs               20.3.0
certifi             2018.1.18
chardet             3.0.4
cryptography        2.1.4
idna                2.6
importlib-metadata  3.3.0
iniconfig           1.1.1
keyring             10.6.0
keyrings.alt        3.0
packaging           20.8
pip                 20.3.3
pluggy             0.13.1
py                  1.10.0
pycrypto            2.6.1
pygobject           3.26.1
pyparsing           2.4.7
pytest              6.2.1
pyxdg               0.25
requests            2.18.4
SecretStorage        2.3.1
setuptools           39.0.1
six                 1.11.0
ssh-import-id       5.7
toml                0.10.2
typing-extensions   3.7.4.3
urllib3             1.22
wheel               0.30.0
zipper              3.4.0
developer@f531d188dded:~$ pytest --version
pytest 6.2.1

```

Figure 5.10: Installed packages checked in development server.

5.4 Conclusions

In the case study, we have detailed the implementation of the current project inside a software development environment, concretely an AI consultancy company that wants to improve the efficiency. We show how the implementation of the *ChatOps* architecture developed offers new ways to manage a DevOps environment by giving use cases that perform operations usual in a real-world development team.

The scenarios presented proved the swiftness of the automatism developed versus deploying and maintaining a DevOps environment manually. The proposed examples performed

the tasks intended within seconds, in comparison with opening a ticket to ask as it is usual in a company that may take hours. The modular architecture used in the project also allows extending the functionalities supported with ease. Moreover, as the systems and platforms deployed are based on virtualization and containerized, the solution can be quickly deployed and replicated in other environments. Further conclusions will be exposed in the following chapter.

Conclusions

To conclude this project, we will resume the principal concepts that are detailed in this document by evaluating the achieved goals, review the problem faced and setting some future work lines to expand the development made for this master thesis.

6.1 Introduction

In this chapter we will expose the conclusions extracted from this master thesis. We will discuss the goals achieved, the problems faced and give insight into future work.

6.2 Conclusions

To conclude this thesis, we are going to summarize some of the keys to the project developed. We have designed a complete ChatOps environment by integrating the natural language AI framework Rasa into a state-of-art DevOps platform. The selected chatbot agent allowed us to develop a flexible assistant that can be accessed from different operative systems, and easy to interact with by using natural language.

The chatbot framework election allowed us to build an adaptable environment based on processing the messages sent by the user instead of launching fixed commands. This makes a difference with other ChatOps implementations where the interaction is made by sending specific commands, as the bot allows to be flexible with the user to achieve the same objectives without having to learn the concrete methods. This way, our design softens the interaction with the ChatOps environment by offering a human-like interface to the user.

Furthermore, the environment design allows being personalized in order to be integrated into different applications, as the systems used rely on public APIs and technologies like REST. Moreover, as all the solutions used are open source, the services themselves can also be tweaked to fit in a wide variety of applications. This also facilitates implementing this project in different environments without the necessity of acquiring licenses.

All the system architecture is meant to be adaptable to different platforms and environments in terms of deployment. The decision of implementing most of the components in *Docker* containers was made with the idea of deploying the solution swiftly. This allows the developers to launch this environment in other contexts such as cloud computing platforms like *Kubernetes* or *Amazon Web Services*.

Additionally, the environment is designed to be easy to extend its functionality. The components used in the architecture make use of easy to read files like *YAML* and Python as the main coding language. Also, the inclusion of Jenkins facilitates the automation, and gives the developer logs and a history of the operations made which simplifies the debugging.

Finally, the complete development presents a detailed architecture that implements an innovative environment, with a state-of-art Natural Language Understanding assistant

accessible from a wide variety of systems which connects with a functional DevOps environment, extending the current paradigms used in the IT sector with a trained AI.

6.3 Achieved Goals

The achieved goals for this project are the following ones:

Implement an AI open source assistant able to interact with the user to automatize tasks using natural language instead of commands.

Develop a functional ChatOps environment based on the AI assistant and DevOps tools. This environment will be oriented to manage a development server to reflect a common IT company environment.

Investigate the software development paradigms to get an insight into how the DevOps tools should be picked and how they impact the results.

Analyse the state-of-the-art chatbots and their main features to choose the solution that fits our environment, finally choosing *Rasa Open Source* as the framework for its flexibility and possibilities.

Create the DevOps side of the architecture by selecting the tools needed to connect with the assistant. The selected tools were Jenkins as the automation server and Docker as the platform to deploy the services.

Configure the components to interact with each other by the use of webhooks and REST API.

Develop the services used to fulfill the objectives marked for this project. This incurred in developing custom actions for Rasa chatbot and scripts used by Jenkins agents to deploy the development server.

Train the chatbot with Machine Learning techniques to allow the intent and entities extraction from the messages received by the user, which is used to trigger the automation methods developed.

Make the chatbot available from different platforms by integrating the chatbot with Telegram Bot API. This gives the environment more flexibility as it can be reached seamlessly from a PC or a mobile device.

6.4 Problems faced

Despite reaching the proposed objectives, we reached some difficulties in the realization of this project. In this section we will list some of the challenges faced, as well as how we overcame them:

- The use of an **AI chatbot** instead of a command-based assistant. As using a Natural Language Understanding chatbot benefits the environment by making it more flexible and easy to use, the intent recognition and entity extraction required more work than coding a set of commands without worrying about the message processing. However, the developed chatbot offers a satisfactory recognition of intents thanks to the training data we generated and used to train the models.
- While the realization of this project, *Rasa Open Source* released a major update of the framework to the 2.0 version. This update introduced enhancements and functions that simplify the development of bots but required to modify some of the files such as the training data or the pipelines we had already created.
- As the development was made in a local virtual machine, **exposing the chatbot to Internet** to make it available from a chat application required some investigation. The final solution was to use *Ngrok*, a solution that exposes local services through NAT and offers a URL forwarding the petitions to the local servers. This allowed us to integrate our chatbot with Telegram and connect with its API.

6.5 Future work

As this project was built with the idea of simplifying future extensions in its functionalities, there are some lines of work that can be followed in order to expand the development:

- Deploy the environment in a cloud platform such as *Amazon Web Services* or *Azure*. This would allow to make the environment available 24/7, and simplify implementing desirable features like redundancy or load balancing.
- Extend the functionalities implemented in the project to achieve a higher grade of automation. Some of the possible functions would be automatically pulling the changes from Git, integrate unitary testing of the code or the support of different users inside the development server with different roles and permissions.

- Add support to different languages in the chatbot. This feature is supported out of the box by Rasa but requires creating new training data and configure the machine learning training pipeline components to adapt to the languages.
- Integrate our chatbot into more chat applications and services, such as *Whatsapp* or *Slack*. This would give the environment more flexibility to adapt to the users by giving them more options.
- Include the option to use voice inputs to talk to the assistant. As most of the chat applications integrate voice messaging, it would require to develop a speech-to-text component that would produce a message to be processed by the current Rasa implementation. Based on this, a text-to-speech component could also be included to give audio responses and emulate a voice conversation.
- Rearrange the current environment to other applications. As the architecture is modular and the components communicate with each other using common protocols, the service managed and the functions performed could adapt to many applications. For example, the ChatOps environment created would be used to manage a computing cluster by adding methods to create several instances, launch services in them and processing the results.
- Adapt the development to be part of a subject in the *Máster Universitario en Ingeniería de Telecomunicación*. As the project covers topics from Machine learning to virtualized services, the development made in this project would fit a practical application in subjects like Ciencia de Datos y Aprendizaje automatico en la Web de datos (CDAW) or Computacion en Nube y Virtualizacion de Redes y servicios (CNVR).

Project impact

This appendix describes the social, economic, environmental and ethical impact related to the project.

A.1 Context

This project is a proposition from the Intelligent System Group (GSI) of the Polytechnic University of Madrid (UPM), with the purpose of implementing a ChatOps environment, consisting of an *Artificial Intelligence* (AI) chatbot which manages a DevOps environment through conversation. This project has been fully developed by the author, with the assistance of a member of the department, Carlos Ángel Iglesias.

The following sections will expose the impact of this thesis on diverse topics.

A.2 Social Impact

The social impact caused by this project is the automation of the process of managing a DevOps environment, which is one of the most used software development paradigms. As the environment developed in this thesis is based on real-world use cases, implementing it in *Information Technology* companies would have a positive impact in terms of efficiency.

Therefore, the development shown in this thesis would have a positive impact to reduce costs in companies, as it would allow the employees to save time operating with the environment.

A.3 Economic Impact

Following the analysis of the previous section, the implementation of this environment in a company would incur automatizing tasks that are done by employees. This may cause the dismissal of employees in charge of the management of the development environments, as the solution would automate the operations made on them. Therefore, the implementation of the solution detailed in this project may save costs for companies.

However, the environment developed in this project requires a specialized team to deploy and maintain, as well as developing new automatisms. This would create new jobs and opportunities, allowing companies to grow while creating jobs.

A.4 Environmental Impact

Regarding the environment. As this project is designed to be light-weight and easily ported to different platforms, the environmental impact is low. During the development, this impact was caused by the energy consumption from my personal computer and two monitors, but in a real-world scenario the energy consumption would be lower deploying the environment in a cloud service.

A.5 Ethical Impact

The ethical impact concerns were mentioned in the previous sections as they affected the economic and social impact. The main concern is the fact that substituting human tasks for artificial intelligence to improve efficiency may cause the destruction of jobs.

However, as the deployment and management of environments like the one developed in this project require human interaction, new jobs would be created.

Project budget

This appendix covers the expenses derived from this project.

B.1 Hardware Expenses

For the development of this project, the main hardware requirement is a computer capable of running the developed environment, as well as the editors used to code and write this thesis. The computer used is an *Asus* laptop, with an *Intel* i5 6300hq processor, 8 GB RAM and 1 TB SSD Hard Disk. To optimize the work, an *AOC* monitor was used aside from the laptop screen.

B.2 Software Expenses

In terms of software, we used Windows 10 as the operative system that hosted the virtual machine where we deployed the environment, which ran Ubuntu 18.04. The rest of the components are open-source and free to use.

B.3 Payroll Expenses

To calculate the expenses related to the development time, we will use the European Credit Transfer and Accumulation System (ECTS), the European standard used to calculate the workload in education. As a Master thesis has 30 ECTS credits, with one credit being equal to 30 working hours, following this standard the hours associated to develop this project are 900. A person working 8 hours per day, 22 working days per month, would take about six months of work. Based on a regular payroll of a junior engineer of 24.000 euros per year, the expenses would be 12.000 euros.

B.4 Indirect Expenses

As indirect expenses, we will consider some of the bills associated with the development done. For this project, we will consider:

- The *Internet bill* for a 100 Mbps line, costing 40 euros per month. As the Master thesis is meant to take 6 months, the total cost would be 240 euros.
- The *electricity bill*, which a cost of 0.1199 kWh. The average consumption of the computer and peripherals used in this project is about 500 watts, which taking into consideration the hours cost about 54 euros.

B.5 Total expenses

The summary of the project budget is shown in the Tab. B.1:

Element	Cost (euros)
Laptop	700
Monitor	200
Windows 10 license	145
Payroll	12.000
Internet bill	240
Electricity bill	54
Total	13.339

Table B.1: Actors involved in the scenario.

Bibliography

- [1] Sameera A Abdul-Kader and JC Woods. Survey on chatbot design techniques in speech conversation systems. *International Journal of Advanced Computer Science and Applications*, 6(7), 2015.
- [2] Atlassian. *Trello*, 2020 (accessed November 24, 2020). <https://trello.com/>.
- [3] Atlassian. *What is ChatOps? A guide to its evolution and adoption*, 2020 (accessed November 8, 2020). <https://www.atlassian.com/blog/software-teams/what-is-chatops-adoption-guide>.
- [4] Prometheus Authors. *Prometheus - Monitoring system & time series database*, 2020 (accessed December 1, 2020). <https://prometheus.io/>.
- [5] The Kubernetes Authors. *Kubernetes*, 2020 (accessed November 19, 2020). <https://kubernetes.io/>.
- [6] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 1997.
- [7] S Balaji and M Sundararajan Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.
- [8] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.
- [9] Jack Cahn. Chatbot: Architecture, design, & development. *University of Pennsylvania School of Engineering and Applied Science Department of Computer and Information Science*, 2017.
- [10] Rocket Chat. *Rocket.Chat - The Leading Communication Hub*, 2020 (accessed November 24, 2020). <https://rocket.chat/>.
- [11] Software Freedom Conservancy. *Git*, 2020 (accessed December 5, 2020). <https://git-scm.com/>.
- [12] Cognitive Computing Consortium. *Cognitive Computing Defined – Cognitive Computing Consortium*, 2020 (accessed December 10, 2020). <https://cognitivecomputingconsortium.com/resources/cognitive-computing-defined/#1467829079735-c0934399-599a>.
- [13] Menal Dahiya. A tool of conversation: Chatbot. *International Journal of Computer Sciences and Engineering*, 5(5):158–161, 2017.

- [14] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. Devops. *Ieee Software*, 33(3):94–100, 2016.
- [15] Nagios Enterprises. *Nagios - The Industry Standard In IT Infrastructure Monitoring*, 2020 (accessed December 1, 2020). <https://www.nagios.org/>.
- [16] Python Software Foundation. *Welcome to Python.org*, 2020 (accessed November 18, 2020). <https://www.python.org/>.
- [17] The Apache Source Foundation. *Maven – Welcome to Apache Maven*, 2020 (accessed November 27, 2020). <https://maven.apache.org/>.
- [18] Willow Garage. *Python Jenkins — Python Jenkins 1.1.1.dev1 documentation*, 2020 (accessed December 12, 2020). <https://python-jenkins.readthedocs.io/en/latest/>.
- [19] Inc Github. *HUBOT / Hubot is your friendly robot sidekick. Install him in your company to dramatically improve employee efficiency.*, 2020 (accessed November 16, 2020). <https://hubot.github.com/>.
- [20] GitLab. *What is GitLab? / GitLab*, 2020 (accessed December 5, 2020). <https://about.gitlab.com/what-is-gitlab/>.
- [21] Nick Groenen Guillaume Binet, Tali Davidovich Petrover. *Errbot — Err 9.9.9 documentation*, 2020 (accessed November 16, 2020). <https://errbot.readthedocs.io/en/latest/>.
- [22] J. O. Gutierrez-Garcia and E. López-Neri. Cognitive computing: A brief survey and open research challenges. In *2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, pages 328–333, 2015.
- [23] Jason Hand. *ChatOps essential guide: The basics, benefits, and challenges / TechBeacon*, 2020 (accessed November 9, 2020). <https://techbeacon.com/enterprise-it/chatops-essential-guide-basics-benefits-challenges>.
- [24] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.
- [25] IBM. *The DeepQA Research Team - IBM*, 2020 (accessed December 1, 2020). https://researcher.watson.ibm.com/researcher/view_group.php?id=2099.
- [26] IBM. *Watson Assistant for customer self-service*, 2020 (accessed December 2, 2020). https://www.ibm.com/watson/assets/duo/pdf/190905_WatsonAssistant_ChildPagePDF_CSS_.pdf.
- [27] Apple Inc. *Usar Face ID en el iPhone o iPad Pro - Soporte técnico de Apple*, 2020 (accessed December 3, 2020). <https://support.apple.com/es-es/HT208109>.
- [28] Apple Inc. *iOS 14 - Apple (ES)*, 2020 (accessed November 21, 2020). <https://www.apple.com/es/ios/ios-14/>.
- [29] Apple Inc. *macOS Big Sur - Apple (ES)*, 2020 (accessed November 21, 2020). <https://www.apple.com/es/macos/big-sur/>.

- [30] Docker Inc. *Docker SDK for Python — Docker SDK for Python 4.4.1 documentation*, 2020 (accessed December 12, 2020). <https://docker-py.readthedocs.io/en/stable/index.html>.
- [31] Docker Inc. *Why Docker? / Docker*, 2020 (accessed December 7, 2020). <https://www.docker.com/why-docker>.
- [32] Docker Inc. *Swarm mode overview / Docker Documentation Containers*, 2020 (accessed November 18, 2020). <https://docs.docker.com/engine/swarm/>.
- [33] Docker Inc. *Docker Hub*, 2020 (accessed November 19, 2020). <https://hub.docker.com/>.
- [34] Github Inc. *About continuous integration - GitHub Docs*, 2020 (accessed December 1, 2020). <https://docs.github.com/es/free-pro-team@latest/actions/guides/about-continuous-integration>.
- [35] Github Inc. *GitHub: Where the world builds software · GitHub*, 2020 (accessed November 24, 2020). <https://github.com/>.
- [36] Google Inc. *Dialogflow use cases*, 2020 (accessed December 2, 2020). <https://dialogflow.com/case-studies/dominos/>.
- [37] Google Inc. *Dialogflow / Google Cloud*, 2020 (accessed December 2, 2020). <https://cloud.google.com/dialogflow>.
- [38] Google Inc. *Fulfillment / Dialogflow Documentation / Google Cloud*, 2020 (accessed December 2, 2020). <https://cloud.google.com/dialogflow/docs/fulfillment-overview>.
- [39] Google Inc. *Android / La plataforma que amplía los límites de lo posible*, 2020 (accessed November 20, 2020). <https://www.android.com/>.
- [40] Rasa Technologies Inc. *Rasa Architecture Overview*, 2020 (accessed December 11, 2020). <https://rasa.com/docs/rasa/arch-overview>.
- [41] Rasa Technologies Inc. *Policies*, 2020 (accessed December 12, 2020). <https://rasa.com/docs/rasa/policies>.
- [42] Rasa Technologies Inc. *Improve your contextual assistant with Rasa X*, 2020 (accessed December 2, 2020). <https://rasa.com/docs/rasa-x/>.
- [43] Rasa Technologies Inc. *Writing Conversation Data*, 2020 (accessed January 8, 2021). <https://rasa.com/docs/rasa/writing-stories/#using-interactive-learning>.
- [44] Rasa Technologies Inc. *AI assistants that go beyond basic FAQs. - Rasa*, 2020 (accessed November 3, 2020). <https://rasa.com/product/why-rasa/>.
- [45] Red Hat Inc. *Open Source Cloud Computing Infrastructure - OpenStack*, 2020 (accessed December 1, 2020). <https://www.openstack.org/>.
- [46] Red Hat Inc. *Red Hat OpenShift, the open hybrid cloud platform built on Kubernetes*, 2020 (accessed November 20, 2020). <https://www.openshift.com/>.

- [47] SonarSource Inc. *Code Quality and Security / SonarQube*, 2020 (accessed November 26, 2020). <https://www.sonarqube.org/>.
- [48] Jenkins. *Jenkins User Documentation*, 2020 (accessed December 8, 2020). <https://www.jenkins.io/doc/>.
- [49] Conny Johansson and Christian Bucanac. The v-model. *IDE, University Of Karlskrona, Ronneby*, 1999.
- [50] Markus Juopperi. *Deployment automation with ChatOps and Ansible*, 2020 (accessed November 15, 2020). <https://www.theseus.fi/bitstream/handle/10024/127446/Deployment%20automation%20with%20ChatOps%20and%20Ansible.pdf?sequence=1&isAllowed=y>.
- [51] Lorenz Cuno Klopfenstein, Saverio Delpriori, Silvia Malatini, and Alessandro Bogliolo. The rise of bots: A survey of conversational interfaces, patterns, and paradigms. In *Proceedings of the 2017 Conference on Designing Interactive Systems*, DIS '17, page 555–565, New York, NY, USA, 2017. Association for Computing Machinery.
- [52] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.
- [53] Canonical Ltd. *Linux Containers*, 2020 (accessed November 18, 2020). <https://linuxcontainers.org/>.
- [54] Lucy Ellen Lwakatare, Pasi Kuvaja, and Markku Oivo. Dimensions of devops. In *International conference on agile software development*, pages 212–217. Springer, 2015.
- [55] M Mahalakshmi and Mukund Sundararajan. Traditional sdlc vs scrum methodology—a comparative study. *International Journal of Emerging Technology and Advanced Engineering*, 3(6):192–196, 2013.
- [56] Diego C. Martín. *Tutorial de Git. Manual básico con ejemplos*, 2020 (accessed November 26, 2020). <https://www.diegocmartin.com/tutorial-git/>.
- [57] Microsoft. *Microsoft HoloLens*, 2020 (accessed December 1, 2020). <https://www.microsoft.com/en-gb/hololens>.
- [58] Microsoft. *Microsoft HoloLens / Mixed Reality Technology for Business*, 2020 (accessed December 1, 2020). <https://www.microsoft.com/en-gb/hololens>.
- [59] Microsoft. *Explora el sistema operativo, los equipos, las apps y más con Windows 10 / Microsoft*, 2020 (accessed November 21, 2020). <https://www.microsoft.com/es-es/windows/>.
- [60] Eueung Mulyana, Rifqy Hakimi, et al. Bringing automation to the classroom: A chatops-based approach. In *2018 4th International Conference on Wireless and Telematics (ICWT)*, pages 1–6. IEEE, 2018.
- [61] Ngrok. *ngrok – documentation*, 2020 (accessed December 11, 2020). <https://ngrok.com/docs>.

-
- [62] Nub8. *¿Qué es DevOps? – Nub8*, 2020 (accessed November 9, 2020). <https://nub8.net/es/que-es-devops/>.
- [63] United States. Navy Mathematical Computing Advisory Panel. Symposium on advanced programming methods for digital computers : Washington, d.c., june 28, 29, 1956, June 1956.
- [64] Inc Red Hat. *Ansible is Simple IT Automation*, 2020 (accessed November 15, 2020). <https://www.ansible.com/>.
- [65] Winston W Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338, 1987.
- [66] Mohit Saini. *Better Intent Classification And Entity Extraction with DIETClassifier Pipeline Optimization / Conversational AI platfrom built on Rasa for teams*, 2020 (accessed December 3, 2020). <https://botfront.io/blog/better-intent-classification-and-entity-extraction-with-diet-classifier-pipeline-optimization>.
- [67] Mohit Saini. *Using the DIET classifier for intent classification in dialogue*, 2020 (accessed December 3, 2020). <https://medium.com/the-research-nest/using-the-diet-classifier-for-intent-classification-in-dialogue-489c76e62804>.
- [68] Outi Salo and Pekka Abrahamsson. Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum. *IET software*, 2(1):58–64, 2008.
- [69] Ayse Pinar Saygin, Ilyas Cicekli, and Varol Akman. Turing test: 50 years later. *Minds and machines*, 10(4):463–518, 2000.
- [70] Eitan Schichmanter. *How ChatOps practices improved communication on a DevOps team*, 2020 (accessed November 9, 2020). <https://techbeacon.com/devops/chatops-whats-all-chatter-about>.
- [71] Patrick Schueffel. *A guide through the Fintech jungle. More than 130 Fintech terms, acronyms and abbreviations explained in plain English*, 2020 (accessed December 1, 2020). <https://web.archive.org/web/20120405071414/http://www.augmentedrealityon.com/>.
- [72] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.
- [73] Paul Semaan. Natural language generation: an overview. *J Comput Sci Res*, 1(3):50–57, 2012.
- [74] Sajjad Hussain Shah and Ilyas Yaqoob. A survey: Internet of things (iot) technologies, applications and challenges. In *2016 IEEE Smart Energy Grid Engineering (SEGE)*, pages 381–385. IEEE, 2016.
- [75] Ayat Shukairy. *Chatbots In Customer Service - Statistics and Trends [Infographic]*, 2020 (accessed December 1, 2020). <https://www.invespcro.com/blog/chatbots-customer-service/>.

- [76] Heung-Yeung Shum, Xiao-dong He, and Di Li. From eliza to xiaoice: challenges and opportunities with social chatbots. *Frontiers of Information Technology & Electronic Engineering*, 19(1):10–26, 2018.
- [77] A. Srivastava, S. Bhardwaj, and S. Saraswat. Scrum model for agile methodology. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 864–869, 2017.
- [78] Arjun Suresh, Erven Rohou, and André Seznec. Compile-time function memoization. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 45–54, 2017.
- [79] CA Technologies. *Flowdock: Group chat for teams. Integrates with GitHub, Jira, Trello.*, 2020 (accessed November 15, 2020). <https://www.ansible.com/>.
- [80] Telegram. *Telegram Messengers*, 2020 (accessed December 4, 2020). <https://telegram.org/>.
- [81] Telegram. *Bots: An introduction for developers*, 2020 (accessed December 5, 2020). <https://core.telegram.org/bots>.
- [82] Telegram. *Telegram Web*, 2020 (accessed November 24, 2020). <https://web.telegram.org>.
- [83] Userlike. *What is ChatOps? A guide to its evolution and adoption*, 2020 (accessed November 8, 2020). <https://www.userlike.com/en/blog/consumer-chatbot-perceptions>.
- [84] Vladimir Vlasov, Johannes E. M. Mosig, and Alan Nichol. Dialogue transformers, 2020.
- [85] Jiayu Yi. *Introduction to the Telegram Bot API, Part 1 by Jiayu Yi / Chatbots Life*, 2020 (accessed December 10, 2020). <https://chatbotslife.com/introduction-to-the-telegram-bot-api-part-1-2ae36f7b30a4>.