Politecnico di Milano

School of Industrial and Information Engineering
Department of Electronics, Information and Bioengineering
Master Degree in Computer Science and Engineering

POLITECNICO
MILANO 1863

A Virtual Assistant To Manage Cloud Performance Monitoring Tools

Supervisor: Prof.ssa Maristella Matera
Co-supervisors: Dr. Federico Marcianò
Dr. Stefano Orciari

Master Thesis of:
Giuseppe Taddeo, 928360

Academic Year: 2020-2021

Più difficile è la vittoria,
più sarà grande
la felicità nel vincere.

Sommario

L'utilizzo di assistenti virtuali basati sul Machine Learning si è diffuso parecchio negli ultimi anni. È nota l'importanza di avere un'entità che riesca a comprendere le intenzioni dell'utente e gli fornisca delle risposte in maniera accurata, tempestiva, aggiornata e affidabile e permetta di effettuare operazioni contestualmente ai problemi riscontrati.
In questo lavoro di tesi il campo di applicazione di tale assistente virtuale è quello del monitoraggio delle performance. Al giorno d'oggi è sempre più in espansione l'utilizzo di varie piattaforme di monitoraggio infrastrutturale e applicativo che permettono la segnalazione di errori o comportamenti indesiderati. È in questo ambito che *Communication Valley Reply* opera, con lo scopo di innovare e migliorare sempre più la consultazione e operatività verso tali strumenti.
Lo scopo di questo lavoro è dunque creare un chatbot personalizzato che possa essere utilizzato tramite varie piattaforme di comunicazione (Telegram, Microsoft Teams..) e che riesca, una volta comprese le intenzioni dell'utente, a facilitare la consultazione verso le piattaforme di monitoraggio utilizzate (Checkmk, AppDynamics..) afferenti al mondo cloud e alla gestione di servizi applicativi e infrastrutturali complessi.
È importante precisare che l'intento è quello di creare un unico luogo centralizzato dove l'utente ha la possibilità, in base ai suoi privilegi, sia di avere una visione generale dei problemi sia di essere operativo sfruttando i consigli che lo stesso chatbot gli fornisce in base ad esperienze passate.
Il sistema di feedback permette di aggiornare l'esperienza del chatbot che proporrà dunque una nuova risoluzione del problema in caso si presenti una situazione analoga in futuro.
Un'altra componente fondamentale che ha incoraggiato questo lavoro è quella di ideare un chatbot che abbia memoria di azioni passate, quantomeno nel breve periodo, per avere un paradigma conversazionale che si avvicini maggiormente a quello umano, anziché avere un sistema di comando-risposta senza memoria che necessita la ripetizione del contesto. La tesi discute come sia possibile sfruttare le funzionalità del chatbot sia tramite frasi testuali in linguaggio naturale, sia tramite comandi.

Abstract

The use of virtual assistants based on Conversational AI has become very popular in recent years. The importance of having an entity that can understand the user's intentions and provides answers in an accurate, timely, up-to-date and reliable way, and allows operations to be carried out when there are problems, is well known.
In this thesis work, the field of application of such virtual assistant is performance monitoring. Nowadays, the use of various infrastructural and application monitoring platforms that allow the reporting of errors and undesired behaviour is increasingly in expansion. It's in this field that *Communication Valley Reply* operates, with the aim of innovating and improving the consultation and operativeness of such tools.
The aim of this work is therefore to create a custom chatbot that can be used via various communication platforms (Telegram, Microsoft Teams, etc.) and that, once the user's intentions are understood, can facilitate consultation with the monitoring platforms used (Checkmk, AppDynamics, etc.) for the cloud and the management of complex services and infrastructures.
It is important to point out that the intention is to create a single centralized place where the user has the possibility, according to his privileges, both to have an overview of the problems and to be operational by exploiting the advice that the chatbot itself provides based on past experience.
The feedback system allows the chatbot's experience to be updated and it will then propose a new solution to the problem if a similar situation arises in the future.
Another fundamental aspect that encouraged this work is that of developing a chatbot that has memory of past actions, at least in the short term, in order to have a conversational paradigm that is closer to the human one, instead of having a command-response system without memory that requires the repetition by the user of contextual information every time a new command is issued. The thesis will show how it is possible to exploit certain functionalities both through textual sentences and through commands.

Table of Contents

List of Figures

Acronyms

CO ChatOps

AI Artificial Intelligence

NLU Natural Language Understanding

NLP Natural Language Processing

DM Dialog Manager

NLG Natural Language Generation

VM Virtual Machine

SNOW ServiceNow

CMK Checkmk

AD AppDynamics

HR Health Rule

EUM End User Monitoring

APM Application Performance Monitoring

ITSM Information Technology Service Management

Chapter 1

Introduction

Automating specific tasks and having guidelines to be directed and correctly advised during complex activities has always been a clear need in people's lives. Over the years, thanks to the advent of Machine Learning and Artificial Intelligence, humans have started preferring a more human-like virtual support, based on the adoption of *natural-language interaction*, which appears more friendly as it allows the users to express their needs in various, more natural ways.
This simpler and more conversational way to interact with the virtual assistant has favored the use of complex tools even for less technical users.
In fact, the virtual assistant provides a contextualized, accurate and timely response following a request expressed conversationally by the user, without the use of particular technicalities or complex commands expressed in formal languages.

To satisfy these requirements it arises a new concept of *ChatOps*, that is a new model that facilitates and speeds up operation, monitoring and development activities. It enables to join users, tools, tasks and tips in the same place. Thanks to that, all the activities are organized in a persistent location where all the actors publish their contents. This scenario improves information sharing and team collaboration. [1]

Conversation-driven cooperation is not a new topic, but CO combines the old form with a new technology, and this permits a radical change in the way of working. Conversation is the power that lets people work and learn together and create new things, so it's a key step for human progress. [2]

Moreover the automation of repetitive manual tasks can avoid human errors and the collaborative workspace environment provides instant visibility and documentation for what is being done, because it supplies a real-time and historical log of activities in the environment.
By helping the development of more sharing and task automation and less feedback loops, CO is more and more central to many organizations that have begun to evolve to a DevOps model. [3]

1.1 Aim of the work

Given the premises illustrated above, the purpose of this thesis is to facilitate and speed up the consultation and operability on top of performance monitoring platforms like AppDynamics and Checkmk as well as other types of platforms like ServiceNow or Azure (discussed in detail in chapter 4), by creating a single place for the user to centrally manage all the information received and by making it possible to move quickly to different contexts with a meaningful general overview. *Communication Valley Reply*, the company in which this work has been carried out, focuses on improving all monitoring aspects. Thus, it plays a fundamental role in this continuous research and development of new approaches to facilitate and speed up monitoring processes.

The user will therefore be able, through a communication channel such as Telegram, Microsoft Teams etc., to interact with the backend of a chatbot, mainly developed in Python language, which will be able to understand his needs and to interact with the platforms mentioned above.
The goal is to create a chatbot that is operational but at the same time consultative, therefore suitable for both a technical user who needs to perform operations on these platforms, and a user that, for informative purposes, only wants to have a clear view of his infrastructure and immediately wants to have evidence of possible problems.
The conversational assistant will therefore have to respond to the user in a clear and timely manner, interpreting the data it will collect and responding to the user's needs.

A further goal is to keep the structure as modular as possible to allow future integrations both in terms of new communication channels (Webex, Facebook, Skype) and in terms of future application fields.

As a result, the integration is possible without upsetting the backend and the core of the existing program, but only adding new components.

1.2 Approach overview

A way to explain how the flow of the requests to the bot are handled is summarized in 4 main steps:

1. Fetching: the moment in which events start and end their lives into the *Connectors*, a term we use to describe the entity which interfaces with the chat service of our choice and triggers events whenever people send messages to the bot.
   In this project, the Telegram and Microsoft Teams connectors play a principal role, as they are two communication channels that are widely used by companies and are easy to install and configure.

2. Parsing: where events are parsed by *Parser* modules which take an event and run it through various services to add more information and context to it. This could be as simple as testing a chat message against various regular expressions to see if there is a match, or running text through third-party AI inferencing services to perform Natural Language Understanding or object recognition tasks.
   In this case it was decided to use the Rasa NLU component, which allows the recognition of intents and the extraction of entities from user sentences.

3. Matching: where we compare the event with all the skills (actions) which have been configured. The most suitable skill (if any) is run to handle the event based on various criteria.
   We are talking about the Rasa Core component, which allows to redirect to the skill to be executed, going to run the Python code expressed in the appropriate module (AppDynamics.py, Servicenow.py, Checkmk.py etc..)

4. Execution: in which skills are run to process the event in whatever way we like. We define these as Python functions which take an action Rasa object and run any arbitrary code. This may be to respond to a chat message with another chat message or to take some more advanced

action such as restart a server, show problems, or anything you can do in Python.

As described in steps 2 and 3, to achieve the desired solution are required some components of the *Rasa* architecture, the standard infrastructure layer for developers to build, improve, and deploy better AI assistants.
Rasa provides the infrastructure and tools needed for high-performance, resilient, and contextual assistants. [4]
In particular, the *Rasa NLU* module was exploited for the extraction of the intent, that is the understanding of the action the user wants to execute, and entities, that is the objects that enrich his request. (i.e. "I want to order a pizza" has as intent 'askForOrder' and as entity 'foodType' the value 'pizza')

Next, the *Rasa Core* module was used in order to predict the following action to be performed by the bot.
This module chooses the next action influenced by various factors such as the actions performed previously (with a short term memory system) and by the value of some stored placeholders.

The great advantage of Rasa is that it mimics a human-like dialogue, since it saves the information and previous actions, so the conversation appears friendlier and less repetitive.
Taking as an example the use case of a user who asks about the health of an application's servers, for subsequent actions there will no longer be the need to specify which application he's talking about, since it is stored into the memory of the bot, and the dialogue flow will appear more natural and realistic.
This is one of the reasons that led to choose the Rasa architecture in preference to another equally valid technology such as Wit.ai, a natural language interface for applications capable of turning sentences into structured data.[5]
The latter is an excellent tool for extracting intent and entities from users' sentences, but it does not allow the creation of a conversational bot and is limited to the 'command and response' paradigm.
For the use case mentioned before, if we use Wit.ai as a tool to extract intent and entities, we need to specify which application the user is talking about, as if all requests were disconnected from each other.

The entire bot code resides on a virtual machine that allows Telegram servers, in a specific case, to communicate with it and, thanks to *Nginx*, a lightweight and high-performance web server that can also be used as a reverse proxy,

requests will be redirected to a fixed port where a listening *Flask* is running.

The ability to easily integrate new communication channels is helped by an initial fetching phase, which interprets the data sent by the specific communication channel and transforms them in a standard format before forwarding them to the next phase.
This allows an easy way to use a new communication channel by simply creating a new custom fetching function that will take care of transforming data from the new channel format into the standard format, leaving everything else unchanged.
The same thing happens to what we call skills, that are the actions that the bot performs, whether they are simple response messages rather than complex operations. These are written into respective platform modules.
For example, the skill to modify an open ticket will reside within the ServiceNow module which will have all the supporting functions to make API calls or to store useful login information.

Database The use of a database is very important, both to manage tables of enabled users, to regulate the management system of permissions and privileges, and to load other information useful like feedbacks and contexts. Choosing the database is always an important choice. The choice was to use *MongoDB*: a cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB stores data in flexible, JSON-like documents. Meaning fields can vary from document to document and data structure can be changed over time. The document model maps to the objects in the application code, making data easy to work with.

Cache In addition to a real Database, it was useful to use something more quickly accessible and searchable, very close to a cache, which is *Redis*, an open-source in-memory data structure store that helpes saving and loading data such as users' IDs rather than data structures related to the information collected.

1.3 Document outline

The remainder of this thesis is structured as follows. Chapter 2 introduces the state of art of the NLP and NLU problem and the existing solutions of chatbots. Development environment in chapter 3 enters into detail of the Rasa architecture with its functionalities and pros as well as the external tools used in the project. Chapter 4 presents an overview of all the platforms which interact with the bot. The entire architecture which illustrates the totally structure of the project is presented in chapter 5. Chapter 6 shows all the implemented use cases for the platforms of cloud monitoring plus other use cases for the remediation and feedback system. Finally, a general conclusion of this dissertation, along with perspectives for future work, is provided in chapter 7.

Chapter 2

State of Art

This chapter describes the distinction about NLP and NLU, the benefits of ChatOps that have led to the development of chatbot and the most popular virtual assistants created until today. In addition is shown the relevance of the natural language versus the command paradigm and the case study of the social chatbot Sounding Board.

2.1 NLP & NLU

As described in [6], conversational interfaces are powered primarily by natural language processing (NLP), and a key subset of NLP is natural language understanding (NLU).
The terms NLP and NLU are often used interchangeably, but they have different meanings. It's important to understand the difference between natural language processing and natural language understanding to build successful conversational applications.

Natural language processing is a subset of AI, and it involves programming computers to process massive volumes of language data. It involves numerous tasks that break down natural language into smaller elements in order to understand the relationships between those elements and how they work together. Common tasks include parsing, speech recognition, part-of-speech tagging, and information extraction. NLP focuses largely on converting text to structured data.

One of the primary goals of NLU is to teach machines how to interpret and

understand language inputted by humans. It aims to teach computers what a body of text or spoken speech means. NLU leverages AI algorithms to recognize attributes of language such as sentiment, semantics, context, and intent. It enables computers to understand the subtleties and variations of language.
For example, the questions "what's the state of the server X?" and "how's the server X?" are both asking the same thing. The question "what's the state of the server X?" can be asked in hundreds of ways. With NLU, computer applications can recognize the many variations in which humans say the same things. (we'll se how it works in chapter 3 with Rasa NLU)

NLP looks at *what was said*, and NLU looks at *what was meant*. People can say identical things in numerous ways, and they may make mistakes when writing or speaking. They may use the wrong words, write fragmented sentences, and misspell or mispronounce words. NLP can analyze text and speech, performing a wide range of tasks that focus primarily on language structure. However, it will not tell you what was meant or intended by specific language. NLU allows computer applications to infer intent from language even when the written or spoken language is flawed.

A simple chatbot doesn't necessarily need NLU. However, if a developer wants to build an intelligent contextual assistant capable of having sophisticated natural-sounding conversations with users, they would need NLU.
NLU is the component that allows the contextual assistant to understand the intent of each utterance by a user. Without it, the assistant won't be able to understand what a user means throughout a conversation. And if the assistant doesn't understand what the user means, it won't respond appropriately or at all in some cases.

2.2 ChatOps

As explained in [3], the benefits of CO can be broken down into two categories: social and technical. Varying members of a team and organization are going to be attracted to some benefits over others.
Social Benefits:

- Increased collaboration;
- Increased sharing of domain knowledge;

- Increased visibility and awareness;
- Improved empathy.

Engineers and members of technical teams within an organization will likely find greater value in the technical benefits. These more closely address the concerns they are faced with on a day-to-day basis. Technical Benefits:

- Increased automation;
- Increased speed of actions and executed commands;
- Automatic logging of conversations and actions;
- Synchronous communication.

At its core, CO is primarily about increase sharing and collaboration regarding efforts and actions taken each day. A higher focus on collaboration, automation, context, and shared institutional knowledge is at the heart of what DevOps has brought to teams and organizations. With very little effort, teams that use a persistent group chat (with powerful chatbots and third-party integrations) instead of email and private messages, begin to see the benefits outlined above. As a result, the organization begins to evolve into one that is efficient in its actions and inherently good at knowledge sharing.

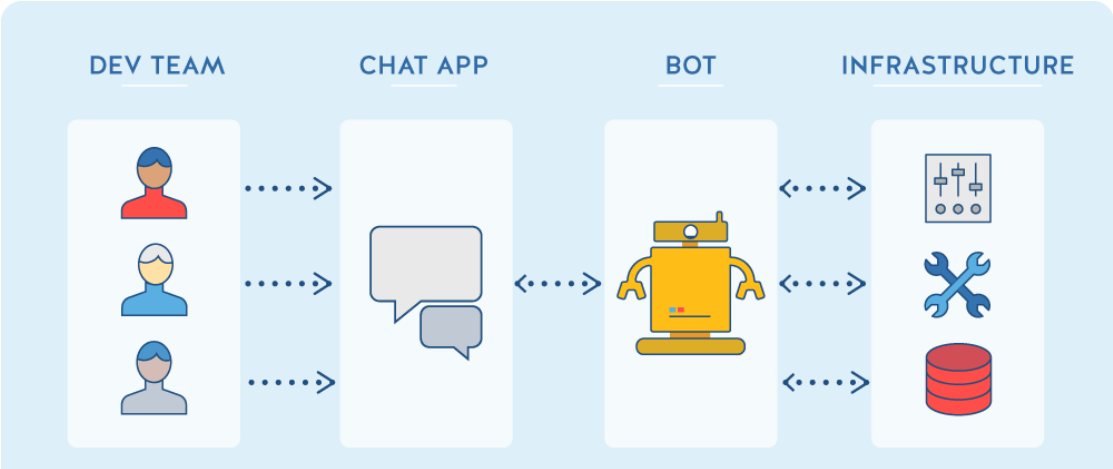


Figure 2.1: ChatOps Architecture

2.3 Chatbot

Although many teams may not need to host, configure, and support a chatbot, as a team's CO efforts get more advanced it may find that third-party integrations do not provide the full functionality or customization that it requires. It's at this point that a chatbot may become necessary to fully realize the capabilities of CO and allow teams to manage the interactions with tools and services in the way that works best for them.
It's time for a chatbot when the third-party integrations are:

- Not available;
- Not flexible enough to work with your special "snowflake";
- Not feature rich or missing core functionality your team needs.

Currently there are several well-known chatbots available, but much of the media attention and conversations online about chatbots are more focused on the business-to-consumer (B2C) style of chatbot development. Retail companies are experimenting with bots that integrate with their ecommerce stores, allowing customers to interact from within chat. While this discussion around chatbots and the related development targeting social media and retail is interesting and gaining much attention, these are not the types of bots that we are referring to when we talk about CO. For our purposes we are primarily focused on chatbots that help our teams manage their day-to-day work lives rather than facilitating interaction with consumers.

2.3.1 Hubot

"I didn't invent Hubot as much as he spawned into our existence and invented himself. He began as a coding assistant. No one expected him to evolve beyond a helper bot and understand us better than we could ever understand ourselves. No one expected him to learn. He is indeed a curious, spectacular, and, dare I say, frightening machine."[7]

The most well-known chatbot available today is Hubot. A Node.js application extensible with CoffeeScript or JavaScript, Hubot was originally developed by GitHub for internal use only. Later, GitHub rebuilt Hubot from the ground up and open-sourced the project, making it available to the public and allowing others to contribute to its ongoing development. Because it has

been around the longest, the number of contributors to the core application as well as the growing list of scripts that manage the interactions with third-party services, the infrastructure, and the codebase is larger than for any other chatbot available today. [8]

2.3.2 Lita

A chatbot written in Ruby with persistent storage provided by Redis that uses a plugin system to connect to different chat services and to provide new behavior. The plugin system uses the familiar tools of the Ruby ecosystem: RubyGems and Bundler.

This type of bot is named Lita and it has caught the attention of teams around the world. Definition files and instructions are written in Ruby in the form of modules that allow much of the same functionality that Hubot provides. With a strong and growing community consistently contributing to the source code and modules, this chatbot has become very popular and easy to implement. [9]

Figure 2.2: Lita Example

2.3.3 Errbot

"Errbot is a chatbot, a daemon that connects to your favorite chat service and brings your tools into the conversation."[10]

Errbot is an open source project written in Python. It has a number advantages over both Lita and Hubot, the most notable one being that the application does not need to be restarted each time a new script is added to the library. Errbot is able to recognize new Python scripts and begin running them on behalf of users as soon as they have been placed in the correct directory. Following its nearly complete redesign a year ago, Errbot has a gentle learning curve and intriguing features to consider.
These include:

- Ability to "pipe" commands, automation, and conversations;
- Ability to manage the bot through chat;
- Built in ACLs for security.

2.3.4 Cog

Cog is another new player in the CO space, but it's much more than simply a bot. Cog is engineered to be more of a framework that addresses a number of concerns many teams have, such as security. With built-in access control and audit logging functionality, Cog allows teams to collaborate on sensitive tasks with higher confidence. Taking inspiration from the command-line interface, Cog has a "pipe" operator that allows users to run a command and use that output as the input for another command in a process. [3]

2.4 Command VS Natural Language

One of the more interesting discussions surrounding chatbots and the future of their development is around the use of natural language versus command language. At present, all the chatbots we've looked at here require a very specific syntax in order to execute commands. Much like when inputting commands with variables, triggers, and flags from the command line, the bots will only execute your commands if they are typed or pasted into the chat client in a very specific way. Typing errors, missing variables, or an errant

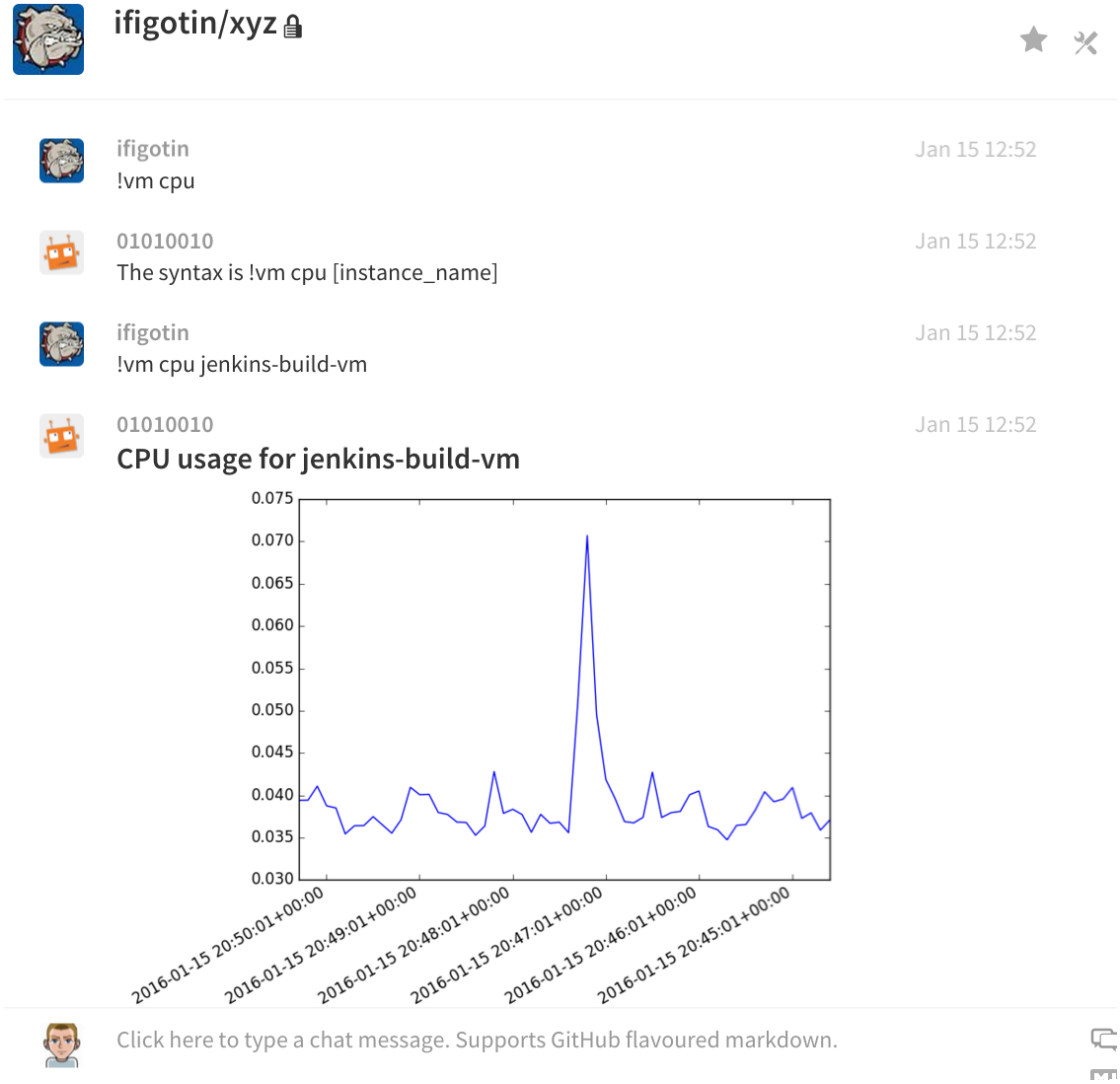


Figure 2.3: Only Command Paradigm Bot

space will prevent the bot from executing anything. Accidents are reduced because the bots will only respond and act on your behalf if a command is provided with complete accuracy.

With that said, some teams are experimenting with ways to create a more "human-like" interaction with their preferred chatbots. Natural language processing (NLP) and the development of associated APIs to allow more natural interactions with applications (such as chatbots) is an area of technology that is gaining more and more attention. With the surge in

interest in chatbots that interact with social media services such as Facebook and Twitter, developers are seeking out ways to make the "back and forth" conversation between a chatbot and a human more natural. This requires enhanced algorithms that can take in many considerations. Additionally, it requires that that the algorithms are able to learn and optimize the experience over time for the end user.

Until operators familiarize themselves with the required syntax understood by their bots, interactions are less fluid and conversational. Also, context is not carried and disseminated throughout a conversation. However, by allowing the bot to store and maintain a history of previous conversations, a scenario arises where clarification on interactions is available. For example, if an operator attempts to execute a command but leaves an important piece of information out of the syntax, the bot can respond asking for further clarification.

2.5 Sounding Board

In [11] is presented a social chatbot that won the 2017 Amazon Alexa Prize. The system architecture consists of several components including spoken language processing, dialogue management, language generation, and content management, with emphasis on user-centric and content-driven design.
A modular framework as shown in Fig. 2.4 is used. When a user speaks, the system produces a response using three modules: natural language understanding (NLU), dialog manager (DM) and natural language generation (NLG).

The NLU is meant to produce a representation of the current event by analyzing the user's speech at the current dialogue state; based on this representation, the dialog manager executes the dialog policy and it has to decide the next dialog state.
The NLG, finally, uses the content selected by the dialog manager in order to return the response to the user and stores it as context in the dialog manager.
During the conversation, the dialog manager also communicates with a knowledge graph that is stored in the backend and updated daily by the content management module.

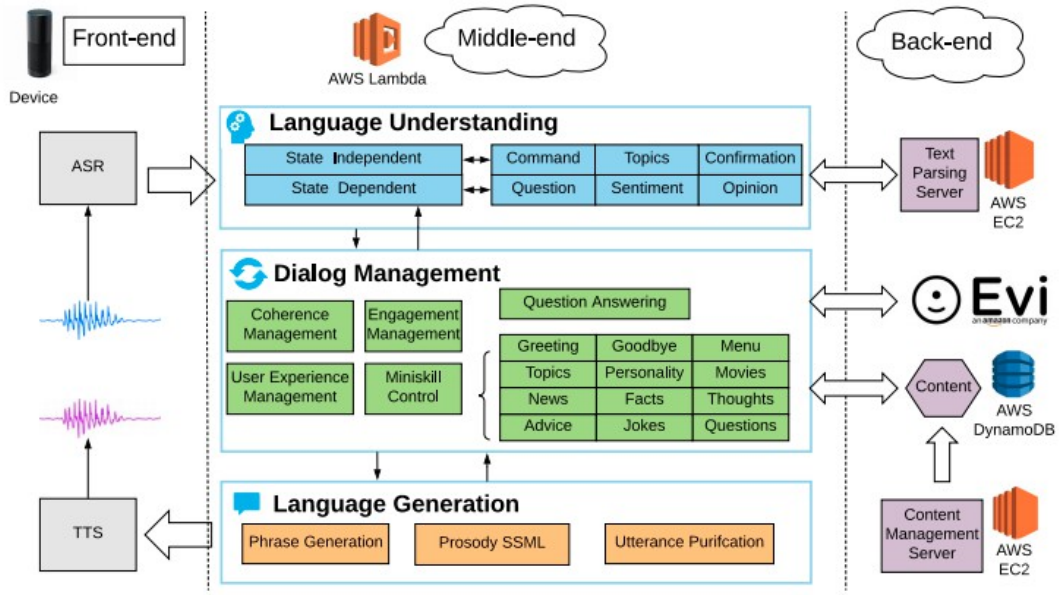


Figure 2.4: Modular Framework For Sounding Board

The dialog management is designed according to three high-level objectives: engagement, coherence, and user experience. The Dialog Manager takes into account user engagement based on components of the NLU output and tries to maintain user interest with different interaction strategies.
The DM tries to maintain dialog coherence by choosing content on the same or a related topic and it does not present topics or content that were already shared with that user.
To improve the user experience, the dialog manager uses conversation grounding acts to explain the system's action and to instruct the user with available options. Every conversation turn, a sequence of processing steps are executed in order to identify the best response that addresses the user's intent.
In fact, at first a state-independent processing step checks if the speaker is initiating a new conversation. If not, a second processing stage executes state dependent dialog policies.
After that, the dialog manager produces a list of speech acts and corresponding content to be used for NLG and at the end it updates the dialog state.

The Natural Language Generation module is composed by three steps:

1. **Utterance Purification** if an offensive word is detected, the utterance purifier will replace it with a random and often humorous non-offensive word chosen from a list of innocuous words.

2. **Prosody** here speech synthesis markup language (SSML) is used to improve the naturalness of concatenated speech acts and to emphasize suggested topics.

3. **Phrase Generation** the response consists of speech acts from four broad categories: grounding, inform, request, and instruction. The response is split into a message and a re-prompt. The device always reads the message; the re-prompt is optionally used if the device does not detect a user's response.

To analyze the system performance, a study over one month has been done. In this period, Sounding Board had 160210 conversations with users that lasted 3 or more turns. At the end of the session, the systems asks the user to rate it on a scale of 1 (bad) to 5 (excellent). The 43% of the users accept to rate the system and the mean score was 3.65. Of the rated conversation, 23% received a score of 1 or 2, 37% received a score of 3 or 4 and 40% received a score of 5. Furthermore the average conversation length was 19.4 turns with the longest one of 772 turns.
Longer conversations tended to get higher rated but while the conversation length is an important factor, it is not enough to have a good measure of the conversation quality.
So, in order to have a good measure the conversation was segmented into sub-dialogues based on the system-identified topic and annotate each sub-dialog as engaged or not depending on the number of turns where the system detects that the user is engaged.The average topic engagement percentages differ significantly (62.5% for high scoring vs. 28.6% for low), but the number of engaged sub-dialogues were similar (4.2 for high and 4.1 for low). The average depth of the sub-dialog was higher for the high conversations (4.0 vs. 3.8 turns).

Chapter 3

Adopted Technologies

3.1 Rasa

We can categorize three main type of assistants:

- Notification assistant: it sends a simple notification through a text message into our app;
- FAQ assistant: the most used at the moment. It allows users to ask a simple question and get the answer using a set of rules or a state machine. It's very easy to build this type of assistant, but at the same time is more likely to make mistakes once the conversation takes a different turn from what is enforced by the defined rules;
- Contextual assistant: it creates a human-like dialog where the context matters. What a user said before influences how the conversation goes on.

For our goal it's better to use the third type, because we want to create an assistant that is more friendly and has memory of past actions in order to make it smarter and therefore more collaborative.
In this context we can exploit Rasa, a set of open source machine learning tools for developers for conversational AI. [4]
For this work, we have specifically used the two main component of this framework: *Rasa NLU* and *Rasa Core*. The first is responsible to recognize intent and entities of the user input, the latter is a dialog manager that chooses which action to perform.

3.1.1 Rasa NLU

This component defines what the user requires and captures key contextual information thanks to the intent classification and entity extraction. So it transforms natural language into structured data using existing NLU and machine learning libraries that support any language. Moreover, It gives the possibility to choose between two components to be inserted in its pipeline:

1. Intent Classifier Sklearn (Pretrained Embeddings).
   This classifier utilizes the spaCy library to load pre-trained language models and each word in the message is seen as a word embedding, that is a vector representation that collects the semantic and syntactic features of words. This means that similar words are represented by similar vectors.
   Word embeddings are language-specific. So, we have to choose between different models depending on the language we are using.
   Since the embeddings are already trained, it requires little training effort to not make mistakes on predictions. So this classifier is suitable for situation in which we have small amounts of training data.

2. Intent Classifier TensorFlow Embedding (Supervised Embeddings).
   This classifier trains word embeddings from scratch. The 'intent featurizer count vectors component' (Fig. 3.1) counts how often different words of the training data appear in a sentence and forwards it to the intent classifier. This classifier also supports messages with multiple intents. Since it is based on the training data, it adapts to the domain specific messages and it's language independent. [12]

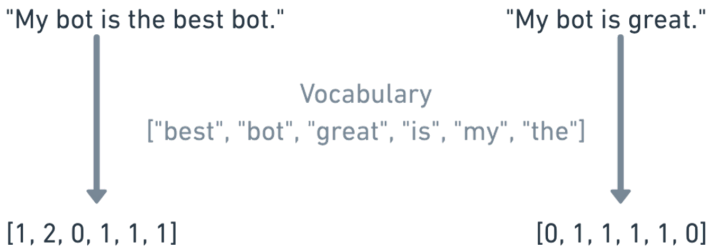


Figure 3.1: Count Vector For Supervised Embeddings

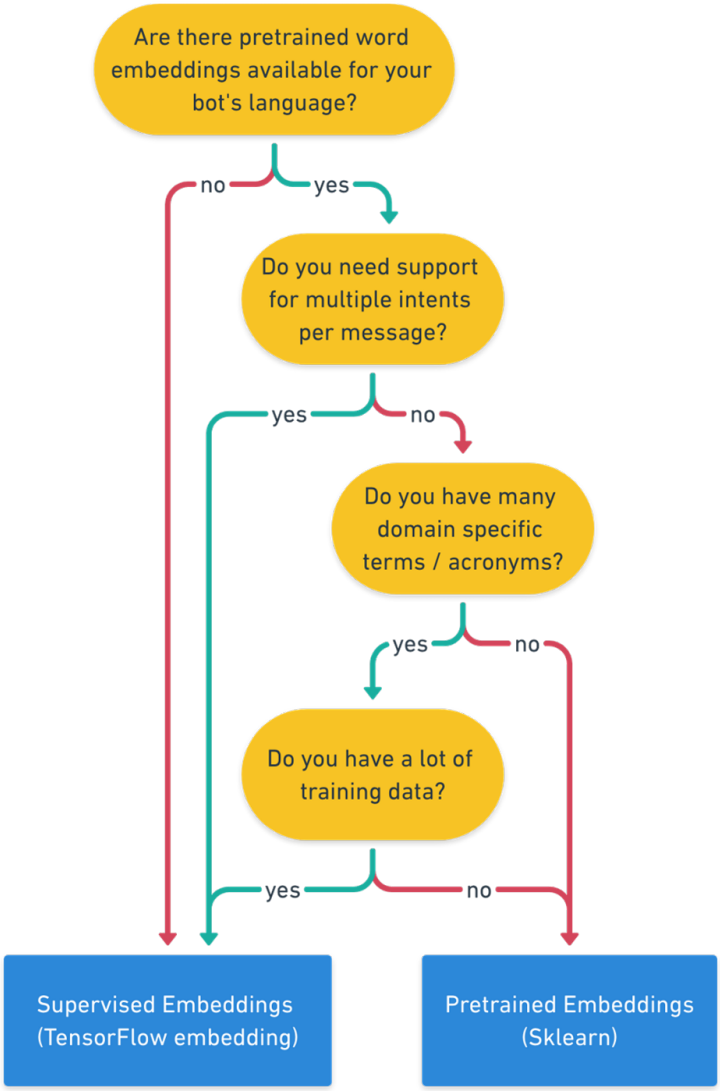


Figure 3.2: Guide For Classifier Choice

We have used the pretrained embeddings and the flow chart in Fig. 3.2 explains well the motivation of the decision, because Italy and English are languages with available pretrained word embedding and we don't need a support for multiple intents per message. This choice also permits to have less training examples.

Data There is a json training file where we insert all the possible intent of the user and the relative sentences specifying the entities of them. It's important to avoid unbalances for an intent, i.e. if we list a very big set of phrases for one intent and a smaller one for another, otherwise the prediction of the intent could be incorrect. Since the file must be in json format and as we see in Fig. 3.3 it is quite complex and consequently not very quick to create a large number of examples in a short time, we can create the same file into the yaml format and then convert it to json.

Figure 3.3: Utterance In Json Format

3.1.2 Rasa Core

For a good user experience, taking context into account is always a good idea. To construct a context-aware conversational assistant, you need to define how the conversation history affects the next response. The *tracker* is the object which keeps track of conversation state. Then, the *policy* receives the current state from the tracker and chooses which action to take next. Finally, the chosen action is logged by the tracker. [13]

Slots Are the bot's memory. They store pieces of information that the assistant needs to refer later and direct the flow of the conversation based on 'slot was set' event (if a slot is set or not, and eventually with which value). There are different types of slots, and each affects the conversation flow in its own way. In the 'status hosts of application' example, the 'application' slot is a textual type. If it is set, the bot knows in that moment we are talking about that specific application, so if the user asks another request to the bot, he does not need to specify the name of the application again. The slots are defined in the domain file, together with entities, intents and the location of custom actions (to allow the bot to go in that location and search the action when it's predicted). If an entity and a slot are configured with the same name, the slot is automatically set when an entity is recognized.

Stories They are examples of how conversations should go. The bot chooses which action to perform based on whether there is an appropriate story that matches the user's intent with the action to be performed, as well as the slots set. In Fig. 3.4 there is an example of story in which, after a greet by the user, he asks to the bot which is the status of the hosts of application 'polifemo' and then it chooses to open a ticket. We can create many stories for the same intent that match different actions based on set slots and what is the previous action.

Policies For this work, we have chosen two specific policies:

1. Memoization Policy. It mimics the stories it was trained on. Depending on what *maxHistory* parameter is set, it tries to match the fragment of the current story with stories provided into the training data file. If it finds one, it predicts the next action from the matched story with a confidence one, otherwise it predicts None with confidence zero.

2. Keras Policy. It uses the neural network implemented in a Python deep learning library called *Keras* to predict the next action.
   It considers:

   - what last action was;
   - what intents and entities were predicted for the current user input;
   - what slots are set at the moment;
   - *maxHistory* states of the dialogue.

3.2 MongoDB

MongoDB is a not relational database management system (DBMS) that uses a document oriented database model which supports various types of data. It is categorized as NoSql database, so instead of using tables and rows, its architecture is made up of collections. A record in MongoDB is a document, which is a data structure composed of field and value pairs. It's similar to JSON object, but it's a variant called Binary JSON (BSON), a binary format utilized to represent simple data structure and associative arrays. [14]

The fields in documents are similar to the columns in a relational database, and the values they contain can be a variety of data types, including other documents, arrays and arrays of documents. Like other NoSQL databases, MongoDB doesn't require predefined schemes and it stores any type of data. This gives users the possibility to create any number of fields in a document. MongoDb was chosen for this project for 4 main reason:

```
## Story application's hosts on Checkmk
* greet
  - utter_greet
* ask_status_application_appdynamics{"application": "polifemo"}
  - action_status_service_appd
* open_ticket
  - action_open_ticket
```

Figure 3.4: Rasa Story

1. It can handle direct JSON input and therefore it's an optimal solution that easily integrates with the Python code.
2. The document structure is more in line with classes and objects of the programming language.
3. The documents doesn't need to have a schema defined in advance. Instead, the fields can be created at the moment. And this is great when a user wants to save a feedback on database.
4. The data model available within MongoDB allows us to represent hierarchical relationships, to store arrays and other more complex structures more easily.

The most important collections are:

- Users and Roles, which define the enabled members and their privileges;
- Resolution, which saves for each specific service a link to a documentation or a popular remediation, i.e restartService if the service is restartable;
- Feedback, which saves the feedbacks of the users. They can be custom feedback, so a snapshot of the problem situation with a phrase written by the user.

Figure 3.5: MongoDB User

3.3 Nginx

Nginx is an open source software that can perform the function of web server, reverse proxy, cache and other features.
With Nginx, one master process can control multiple work processes. The master is responsible for maintaining the work processes, while the workers do the actual processing. Since Nginx is asynchronous, each request can be processed by the worker at the same moment without blocking other requests. [15]
In the following we will consider the Telegram connector as the output of the bot. Each discussion can then be generalized to other types of connectors, with little differences and modifications.

For our purpose, we use Nginx as a reverse proxy because it's the tool that allows the server Telegram to communicate with the backend of the bot. We have configured the Nginx on the virtual machine where is present the program.

The flow then is as follows:
the server Telegram knows, through a webhook set up, that all the messages received by the bot must be forwarded to the IP address of the VM where our nginx in turn forwards them to our bot (more precisely converts the https request into an http one and adds the relative endpoint for the Flask that is listening on a specific port, like 5020). In Fig. 3.6 we can see the entire flow.

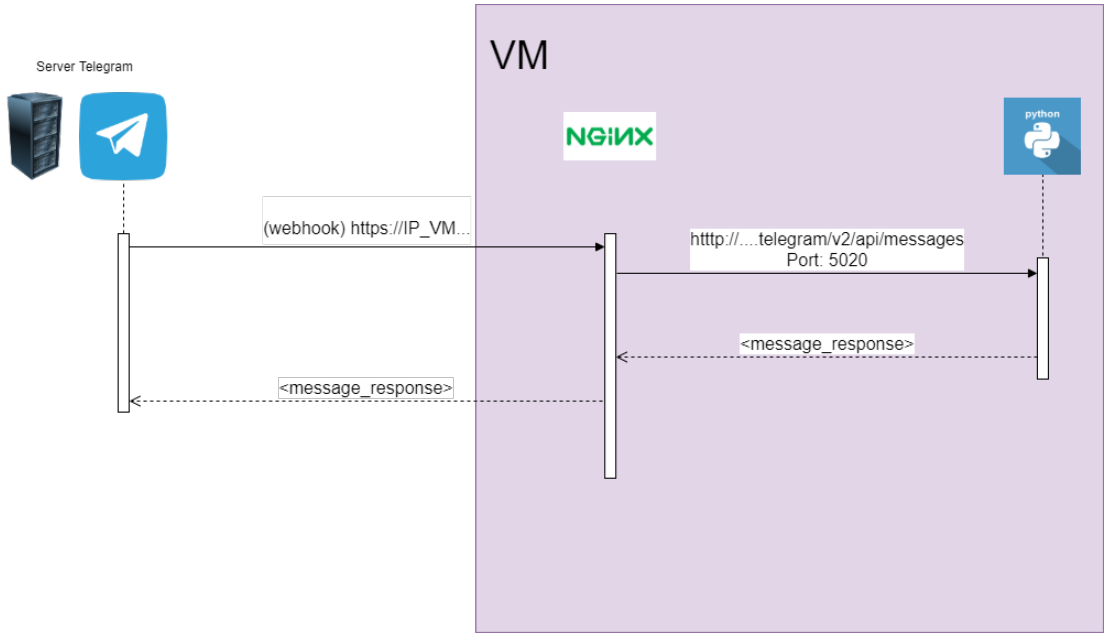


Figure 3.6: Nginx Flow

3.4 Redis

Redis (Remote Dictionary Server) is a fast, open source, in-memory, key-value data store. It enables millions of requests per second for real-time applications and it's a popular choice thanks to its fast performances.

All Redis data resides in memory and this enables low-latency and high-speed effective data access. Unlike traditional databases, in-memory data stores do not require disk access, reducing engine latency to microseconds.
The result is incredibly fast performance with average read and write operations taking less than a millisecond and support for millions of operations per second.
Redis is the best solution for deploying high-availability in-memory caching to minimize data access latency, maximize throughput, and alleviate the load on relational or NoSQL applications and databases. [16]

For our goal, we exploited Redis to save and retrieve data in a quick and persistent way. In particular, we save on Redis two particular object, which are ChannelActivity and Context.

The first one is the main object that permits the scalability of the connectors, because is the entity that saves all the data of the user, like ID Telegram, name, privileges and so on. Thanks to this object, regardless of the location where we are in the code, we can simply write 'sendMessage' without specify nothing more, then it will be the object itself to decide if send such message towards a connector rather than another one (Telegram, Teams etc.)

The latter is the object that saves the entire context, so which host has been visited rather than the list of services in critical/warning status at the moment and so on. It's therefore a support that increases the background of the conversation.

All key-value pairs must be saved as strings, so before retrieving and saving the information the application converts these two objects to json and then to strings and vice versa.

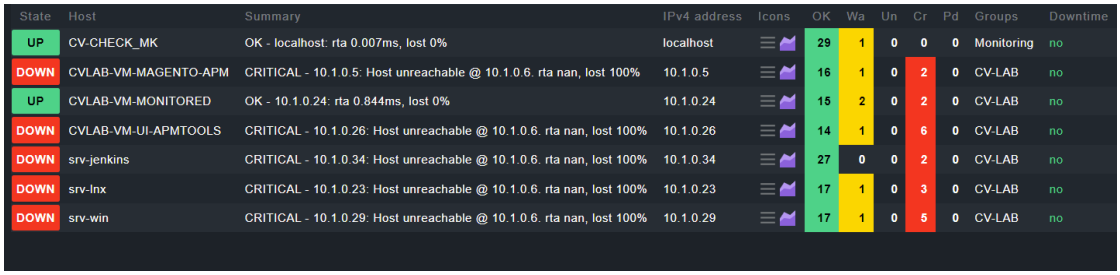
Chapter 4

Cloud Monitoring Tools

Trying to reduce or eliminate problems of IT systems has always been a challenge. Both the complexity of hardware and software stacks and the needs of users continue to grow, whether they are working with real hardware or with cloud solutions. This is why detailed and complete IT monitoring is fundamental nowadays.
Both Checkmk and AppDynamics can perform infrastructural and application monitoring, but we preferred to use the first one to check the issues of VM and the second one to control the end user experience and application level, because both are the best tools in these areas.

4.1 Checkmk

All the monitoring is based on an agent, which is a normal process that must be installed on the VM to be monitored and it's the component that transmits the analyzed metrics.
Everything in Checkmk revolves around hosts and services.



State	Host	Summary	IPv4 address	Icons	OK	Wa	Un	Cr	Pd	Groups	Downtime
UP	CV-CHECK_MK	OK - localhost: rta 0.007ms, lost 0%	localhost		29	1	0	0	0	Monitoring	no
DOWN	CVLAB-VM-MAGENTO-APM	CRITICAL - 10.1.0.5: Host unreachable @ 10.1.0.6. rta nan, lost 100%	10.1.0.5		16	1	0	2	0	CV-LAB	no
UP	CVLAB-VM-MONITORED	OK - 10.1.0.24: rta 0.844ms, lost 0%	10.1.0.24		15	2	0	2	0	CV-LAB	no
DOWN	CVLAB-VM-UI-APMTOOLS	CRITICAL - 10.1.0.26: Host unreachable @ 10.1.0.6. rta nan, lost 100%	10.1.0.26		14	1	0	6	0	CV-LAB	no
DOWN	srv-jenkins	CRITICAL - 10.1.0.34: Host unreachable @ 10.1.0.6. rta nan, lost 100%	10.1.0.34		27	0	0	2	0	CV-LAB	no
DOWN	srv-lnx	CRITICAL - 10.1.0.23: Host unreachable @ 10.1.0.6. rta nan, lost 100%	10.1.0.23		17	1	0	3	0	CV-LAB	no
DOWN	srv-win	CRITICAL - 10.1.0.29: Host unreachable @ 10.1.0.6. rta nan, lost 100%	10.1.0.29		17	1	0	5	0	CV-LAB	no

Figure 4.1: Overview Hosts On Checkmk

Hosts An host can be:

- A server;
- A network device (switch, router, load balancer);
- A measuring device with an IP connection;
- Anything else with an IP address;
- A cluster of several Hosts;
- A virtual machine;
- A Docker container.

It always has one of the following states:

- UP: The host is accessible via the network (this generally means that it answers a PING.);
- DOWN: The host does not answer network inquiries, is not accessible.

Fig. 4.1 shows the hosts in the our lab site of CMK.

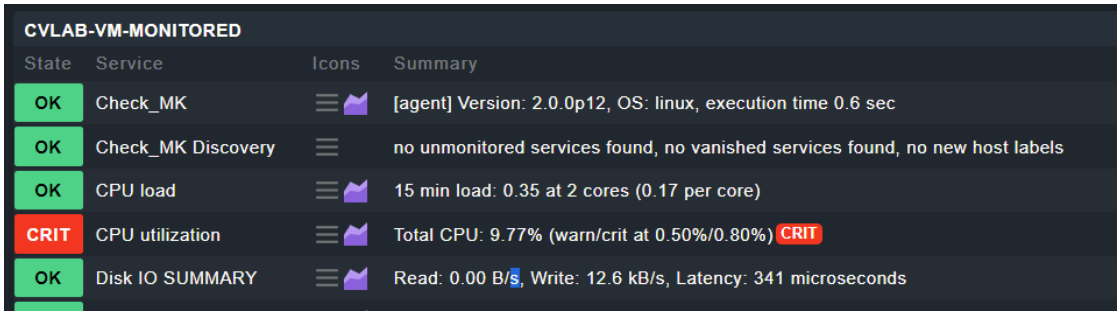


Figure 4.2: Services On Checkmk

Services An host has a number of services. A service is any part or aspect of the host that can be OK, or not OK. In the most cases this is decided by a threshold, which can be either a standard and predetermined, for services like CPU Utilization, Throughput etc., or a custom threshold set by the user. Naturally the state can only be determined if the host is in an UP condition. In 4.2 we can see an example for some services of the host CVLAB-VM-MONITORED.
A service being monitored can have the following states:

- OK: The service is fully in order. All values are in their allowed range;
- WARNING: The service is functioning normally, but its parameters are outside their optimal range;
- CRITICAL: The service has failed.

Downtime 'If an operator performs maintenance work on a server, device or software, we want to avoid potential problem notifications during this time. In addition, we will probably want to advise our colleagues that problems appearing in monitoring during this time may be temporarily ignored. For this purpose you can enter a condition of scheduled downtimes on an host or service'. [17]
While an host or service has a scheduled downtime:

- No notifications will be sent;
- Problems will not be shown in the Overview.

Additionally, when you wish to later document statistics on the availability of hosts and services it is a good idea to include scheduled downtimes.

4.2 AppDynamics

'AppDynamics is an application intelligence company that provides Application Performance Monitoring (APM) to monitor, manage, analyze and optimize customer experience and complex software environments.'[18]
Its special feature is that it offers a real-time, end-to-end vision of application performance from end users to the data centre, together with analysis of customer behaviour and correlation between application performance and business performance.
Also AD is based on an agent that must be installed and configured on the VM to be monitored.

In this thesis we utilize AppDynamics for an application level of monitoring. A user which asks to the bot the health of an application will visualize informations extracted from this platform.
A typical application environment consists of different components that interact in a variety of ways to fulfill requests from the application's users. As explained in [19], AppDynamics app agents automatically discover the most common application frameworks and services. Using built-in application detection and configuration settings, agents collect application data and metrics to build flow maps.
A flow map visually represents the components of your application to help you understand how data flows among the application components. For example, the business transaction flow map for a simple e-commerce application shows data flowing between web services, message queues, and databases (Fig. 4.3).

Health Rules Health rules allows us to specify the parameters and thresholds that represent what we consider normal or expected operations for your environment. The parameters rely on metric values, for example, the average response time for a business transaction or CPU utilization for a node. When the performance of an entity affected by the health rule violates the rule's conditions, a health rule violation occurs. The health statuses are represented as critical, warning, normal, and unknown. An health rule violation event can also be used to trigger a policy, which can initiate automatic actions, such as, sending alerting emails or running remedial scripts. [20]

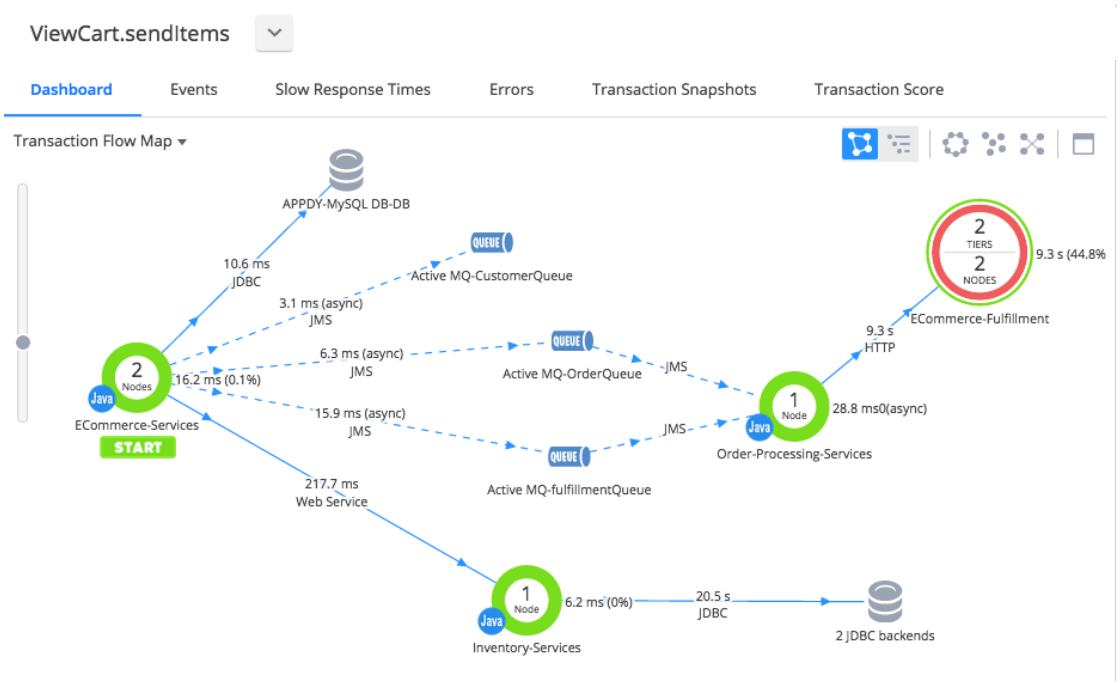


Figure 4.3: Flowmap AppDynamics

Synthetics AppDynamics End User Monitoring (EUM) gives us visibility on the performance of our application from the viewpoint of the end user. We call *synthetic agents* the components which has the control of this type of monitoring.
While Application Performance Monitoring (APM) measures user interaction starting at the web server or application server entry point, EUM extends that visibility all the way to the web browser, mobile, or IoT application. As a result, EUM reveals the impact the network and browser rendering time have on the user experience of our application.

4.3 ServiceNow

As organizations expand, they need a tool of managing employee issues outside of emailing and calling IT departments with requests. Ticketing software takes all service requests and converts them into a single point of contact.
'IT service management (ITSM) is a framework that focuses on employee

needs, offering planning, delivery, and support of IT services through integrated people, process, and technology.'[21]
With respect to traditional ticketing systems, ITSM is more inclusive and illustrates the processes and tools IT teams use to handle all IT services, end to end. It allows that the employees' issues are being resolved quickly and effectively.
SNOW offers this type of service and allows the users to create, modify and monitor all the tickets and workflows created for many different issues.

Ticket 'A ticket is a special document or record that represents an incident, alert, request, or event that requires action from the IT department.' [21]
It often includes further contextual details and can also contain relevant contact information of the user who created the ticket. In Fig. 4.4 we can see the table of the tickets which are automatically created by the bot, with all the parameters set (some of these are not present on the overview page, but we can see a more in-depth view in the chapter of use cases).

Number ▼	Short description	Contact	Account	Consumer	Channel	State	Severity	Assigned to	Updated	Created
CS0015415	CVLAB-VM-MONITORED	(empty)	(empty)	(empty)	Web	Open	1 - Critical	(empty)	03/03/2022 16:23:14	03/03/2022 16:23:14
CS0015414	CVLAB-VM-UI-APMTOOLS	(empty)	(empty)	(empty)	Web	Closed	1 - Critical	(empty)	03/03/2022 00:31:03	03/03/2022 00:30:20
CS0015413	proserpina	(empty)	(empty)	(empty)	Web	Closed	1 - Critical	(empty)	02/03/2022 19:14:21	02/03/2022 19:12:23

Figure 4.4: Tickets Table

Chapter 5

Project Architecture

As introduced in the previous chapters, the structure of the project is modular and scalable. First of all, before starting the program, we compute two models: NLU and dialogue.
The first one in order to understand the intentions of the user, the latter to redirect the flow of the program through the predicted action. Both models will then be used by an object named *agent*.

Everything starts from the Flask which, according to the endpoint specified by Nginx, directs the json file received by the connector to its fetching function. In the specific case of Telegram we are directed to the function 'Telegramfetcher' where the unpacking of the file received from the Telegram server takes place. In this way we know TelegramID, name and message of the user who just sent a message to the bot.

Interpreter Each thread of a message post is linked to an interpreter, which is connected to the NLU model previously computed to extract all the relevant elements of the input. The model gives in output all recognized entities and an intent ranking, i.e. each intent with its confidence.

Agent At this point, an agent is assigned to the user which will be a newly initialized agent if the user has written to the bot for the first time, or it will be retrieved from a data structure if the user has already used the bot.
An agent is a Python object which connects the two component Rasa NLU and Rasa Core, in fact it combines the interpreter and the dialogue model. It has a data structure containing all currently set slots and an own tracker

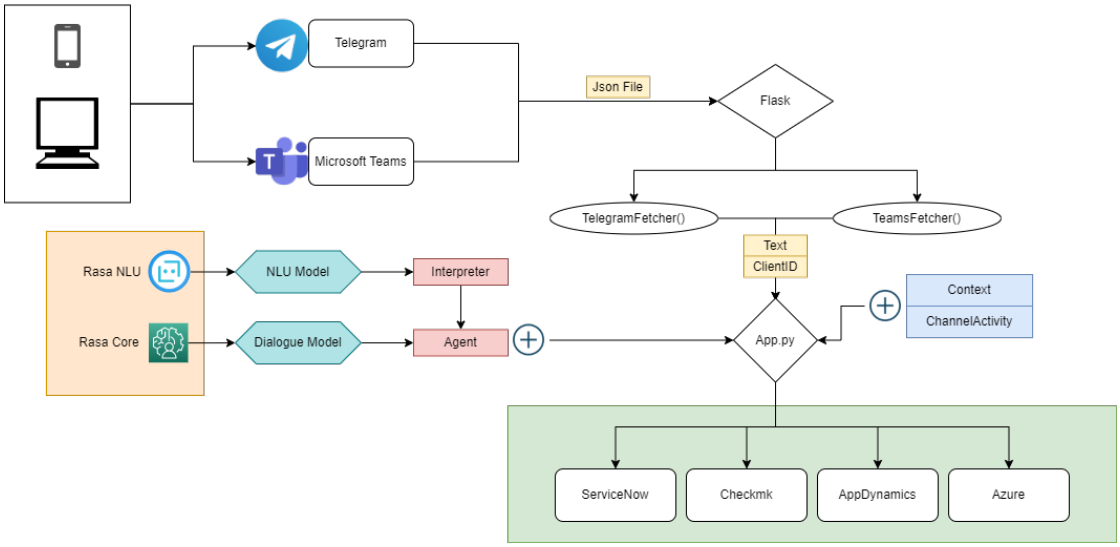


Figure 5.1: General Structure

and dispatcher to keep track of the actions performed and the status of the conversation.
This allows each TelegramID to have a different agent, so that the actions performed by one user and the state of the conversation will not affect those of another user.

After the fetching phase, the message text passes through the two modules Rasa Core and Rasa NLU as previously explained. Then, the flow is redirected to the relevant module.
Fig. 5.1 shows the general structure from the client to the skills and highlights its modularity.

Modules Every component that interacts with the bot is represented by a Python module (Checkmk.py, AppDynamics.py etc.) where there are all the relative functions. This structure facilitates modularity and the possibility of quickly integrating a new component.
In each of these modules there are login function, endpoint for API calls and data structures to save the information requested.
In addition to these functions, there are specific classes that inherit the object Rasa 'Action' and they are particular object with a tracker and dispatcher in their constructor. These classes represent the skills of the bot: the code to be executed once specific actions are predicted.

ChannelActivity This component facilitates the independence from communication channels, in the sense that regardless of the channel used, the functionality of the modules is not affected.If at some point there is a need to reply to the user with a message, we will simply use the generic sendMessage function without specifying the channel. This is feasible because among the attributes of ChannelActivity there is 'channelRequested' which is set consistently during the fetching function. Other important attributes of this object are TelegramID, TeamsID, language and the token of the Telegram bot.

Context This Python object, saved and retrieved from Redis like ChannelActivity, is important to save context data in addition to the ones saved in the agent slots.
For example, if a ticket is opened for a particular host, the ticket number will be saved along with the sysID that must be specified in the API call to ServiceNow to close the same ticket later.

5.1 Localization

For the question of the language, a distinction needs to be made. The bot's understanding of the user's language and the language used by the bot in its replies.

For the latter, we have created a file called language.yaml where all possible answers of the bot are rewritten in other languages and the same type of answer will have the same label regardless of the language. We just need to specify the language (which can also be set by asking the bot directly) and the bot will pick up the answer consistently.
In Fig. 5.2 and 5.3 we can see the structure of the localization file.

For the first one, we are forced to create a different model for each language, with different training file. The only thing that remains unchanged is the dialogue model and the domain with the various intents and entities.
So we have to link a new interpreter to this new NLU model and a new agent will be needed.

```
it-IT:
  FallbackResponse: "Scusami, Non ho capito la tua richiesta "
  CriticalStatus: "risulta in CRITICAL"
  WarningStatus: "risulta in WARNING"
  NormalStatus: "non risulta criticità"
  NotScheduledStatus: "non è attivo in questo momento"
  UnknownStatus: "Sconosciuta "
```

Figure 5.2: Italian Part Of Localization File

```
en-EN:
  FallbackResponse: "Sorry, I don't understand your request "
  CriticalStatus: "is in a critical state"
  WarningStatus: "is in a warning state"
  NormalStatus: "there is no criticality"
  NotScheduledStatus: "is not currently scheduled"
  UnknownStatus: "unknown"
```

Figure 5.3: English Part Of Localization File

5.2 Multiusers

For each user which write to the bot there is a personal context, channelActivity and agent connected to him. This is to create different and separate environments depending on the user and to avoid interference between them. Each user is registered on the MongoDB with its role and according to it he can exploit some use cases rather than others.

As a future evolution it would be interesting to create groups of users who contaminate the answers of the bot towards other users belonging to the same group. With the goal of extending the environment separation to group level instead of user level.

5.3 Multiconnectors

As mentioned above, linking different communication channels is not difficult thanks to this structure.
All we need is a new fetching function, to which you will be redirected according to the Flask endpoint and where it will interpret the data received from the new channel. After this step, everything that comes after remains unchanged.

Chapter 6

Use Cases

In this chapter we illustrate the possible use cases and potentialities of the bot. For some of them we can trigger the action either with a natural language and with a command. A general overview is presented in Fig. 6.1. Instead, the diagram of possible dialogue paths offered by the bot is shown in Fig. 6.10.

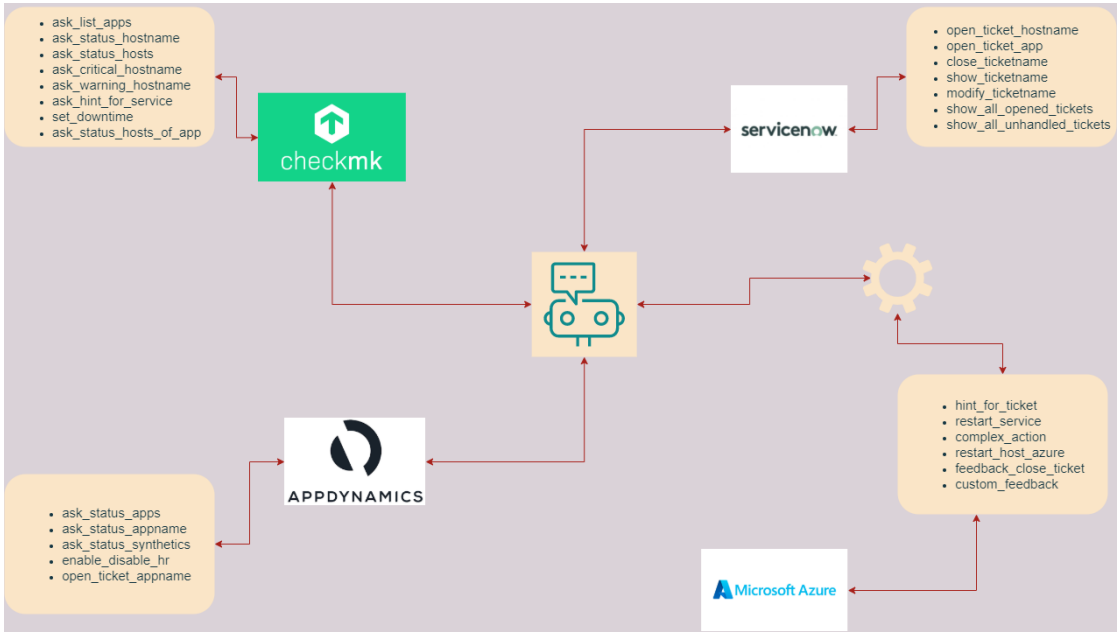


Figure 6.1: Use Cases Overview

6.1 Checkmk

All the skills concerning this platform are grouped in the module Checkmk.py which creates the object *Checkmk*. The latter has as attributes:

- the endpoint for API calls;
- the username used in the get/post request;
- the secret used in the get/post request.

6.1.1 List Apps

The user can have a summary of all the monitored applications. Each application is supported by different hosts, which are the main component monitored by CMK, with an overview of all the services of the hosts in critical or warning status.

Why useful? To know the name of the application and then make more specific requests on them. The action can be also triggered with a command '/app' which appears in the drop-down menu of Telegram.

6.1.2 Host Status

The user can request the status of a specific host or all the monitored hosts. The bot will response with a compact recap that shows the number of critical and warning services of them.

Why useful? To have an high level overview of the health of the host and then to decide if open a ticket on SNOW or ask about the name of services and a comment of the highlighted problem.

6.1.3 Criticals & Warnings

Each time we ask about the health of an host, then we can ask about which are the critical or warning services. As previously stated, the bot has a memory and seems intelligent. In fact, if we are talking about a specific host or application, there is no need to repeat it again.

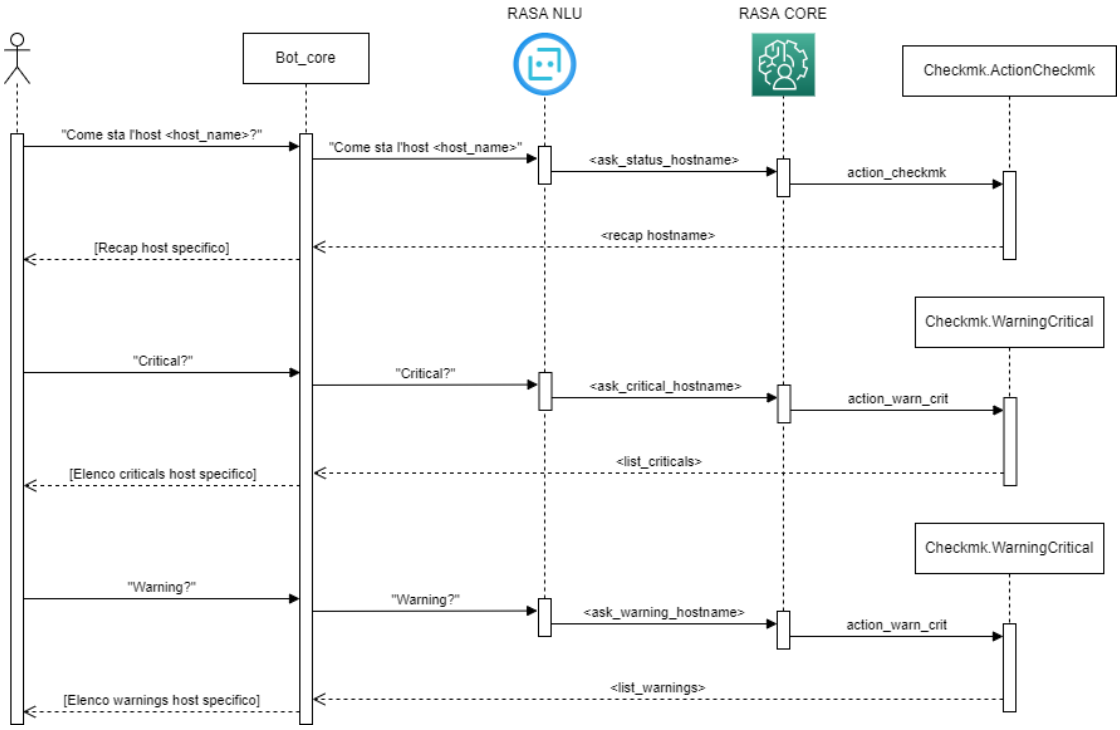


Figure 6.2: Critical Warning Diagram

As we see in Fig. 6.2 we can simply write the keywoard 'warning' and 'critical' to let the bot understand that we want to know more about the services of the host specified in the previous request. Of course there are many phrases to indicate to the bot the same intention.

Why useful? To understand more specifically which are the services of the host that are not performing as they should, and then to choose the next action based on this information.

6.1.4 Application Hosts

In addition to asking the health of specific hosts, you can also ask for information about the hosts of an application. Then the bot will be in charge of retrieve, interfacing with the MongoDB, which are the hosts connected to that application and will show a summary.

In Fig. 6.3 is shown an example in which a user asks the health status of the hosts of an application called 'proserpina' and the bot shows an

Figure 6.3: Hosts Of Application

overview of the two servers 'CVLAB-VM-MONITORED' (with 'UP' status, that means it's running and reachable) and 'CVLAB-VM-UI-APMTOOLS' (with 'DOWN' status, which means it's turned off or unreachable) with their respective numbers of critical and warning services.

Why useful? In this use case the name of the requested application will be saved in a slot, which can be used in future questions (e.g. to open a ticket). The user have to know only the name of the application, whereas it may not even know the name of the hosts.

6.1.5 Service Hint

When the user knows which services are not performing, he can ask to the bot if it has any suggestion about this bad situation. The bot sees which are the name of the services and if there is a relative remediation saved on MongoDB by someone else in the past. If so, it sends a link to a documentation to resolve the problem.

Why useful? If a user finds itself in a situation which has happened before and has been solved, he can look the documentation to solve the same

problem.

6.1.6 Downtime

On CMK is possible to set a downtime for an host or service. It means that for that element are disactivated all the alarms. We can set the downtime with the bot, by specifying for which element and for how long, and an API call will be launched to set up what is required.

Why useful? If there are any hosts or services which are unreachable or have problems because some operator is working on them, it makes no sense to categorize them as the others. Having the possibility to set the downtime via the bot simplifies and speeds up the process.

6.2 ServiceNow

Also for this platform, all the skills are grouped in the module ServiceNow.py where we create the object 'ServiceNow' which has as attributes:

- The endpoint for API calls;
- The endpoints for the different tables (Tickets, Users etc.);
- The username and password used in the get/post/put requests.

6.2.1 Open Ticket

We can open a ticket for a specific host or application. Usually after we asked informations about hosts of an application, we can decide to open a ticket. At this point, after making our intention clear to the bot, a drop-down menu will open with a list of candidate hosts, which will intelligently be those consulted earlier. The menu is a shortcut to not having to write the entire name of the host, but it is also possible to write it by hand. Once selected from the menu, the latter will disappear and a ticket will be opened for the selected host.

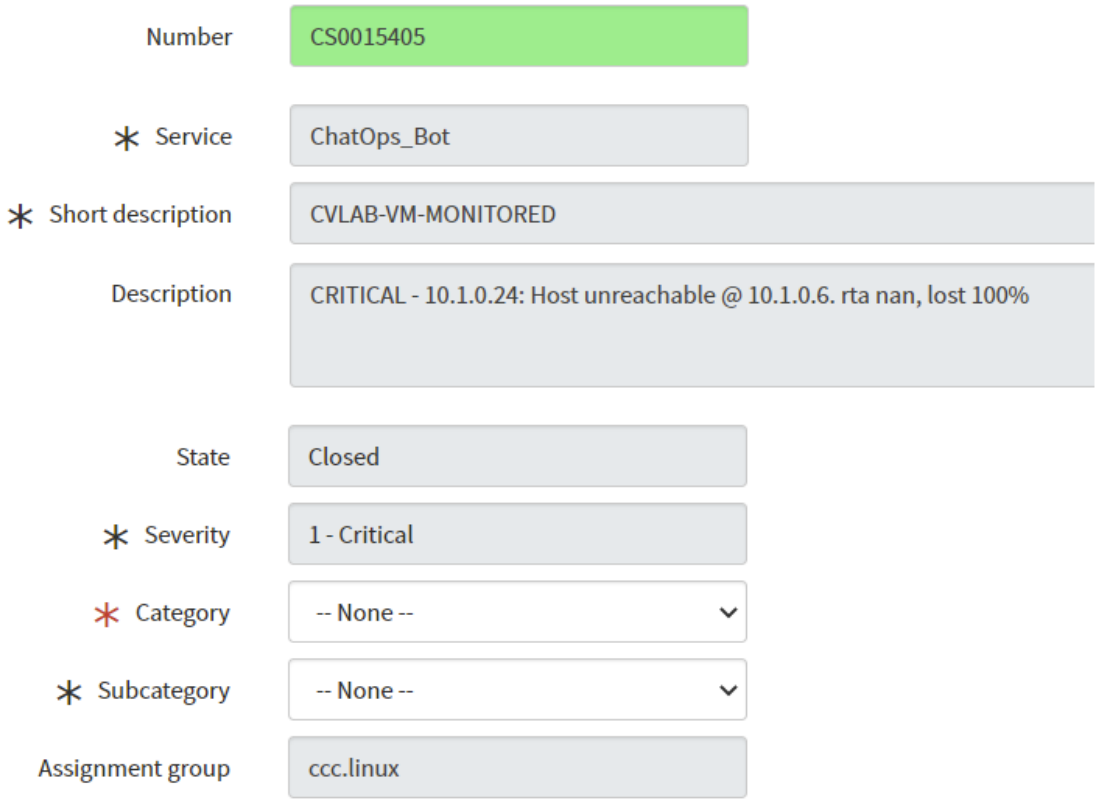


Figure 6.4: Fields Of Ticket

Why useful? When a ticket is opened, the bot initialize many parameters like priority, assignment group, and set a description with the highlighted problem. For example, if the host for which the ticket is opened is a windows machine, the assignment group will be *ccc.windows* and if it's a linux machine it will be *ccc.linux*. This speeds up the ticketing operations, because the bot is like taking a screenshot of the problem and setting all the default fields automatically.
In Fig. 6.4, for example, we opened a ticket for the linux host 'CVLAB-VM-MONITORED' and the assignment group is set to *ccc.linux*. The severity is *1-Critical* because the host is unreachable, as we can see by the description.

6.2.2 Close Ticket

Just as it is important to open a ticket, it is very useful to have the possibility to close it in the same place. When a ticket is opened, the ticket number is saved in the context object with all its parameters and the reference host, among which the value of the sys_id is fundamental because the ticket closing API call needs this parameter.
Also in this case, the bot, as a shortcut, will propose to the user open ticket numbers with a menu, but you can also specify by hand the number of the ticket to be closed.

Why useful? Having the possibility to close tickets in the same place where they have been opened centralises the control and tracking of what has been done.

6.2.3 Show Ticket

The possibility to ask the bot for a printout of a specific ticket can be obtained either in natural language or via command */show <num_ticket>*. We can also ask the printing of a particular group of tickets, like ticket unhandled or ticket opened.

Why useful? to check that the open ticket has the parameters set consistently or to verify that changes have been made to it. Moreover, we can know which are the tickets unhandled and work on them.

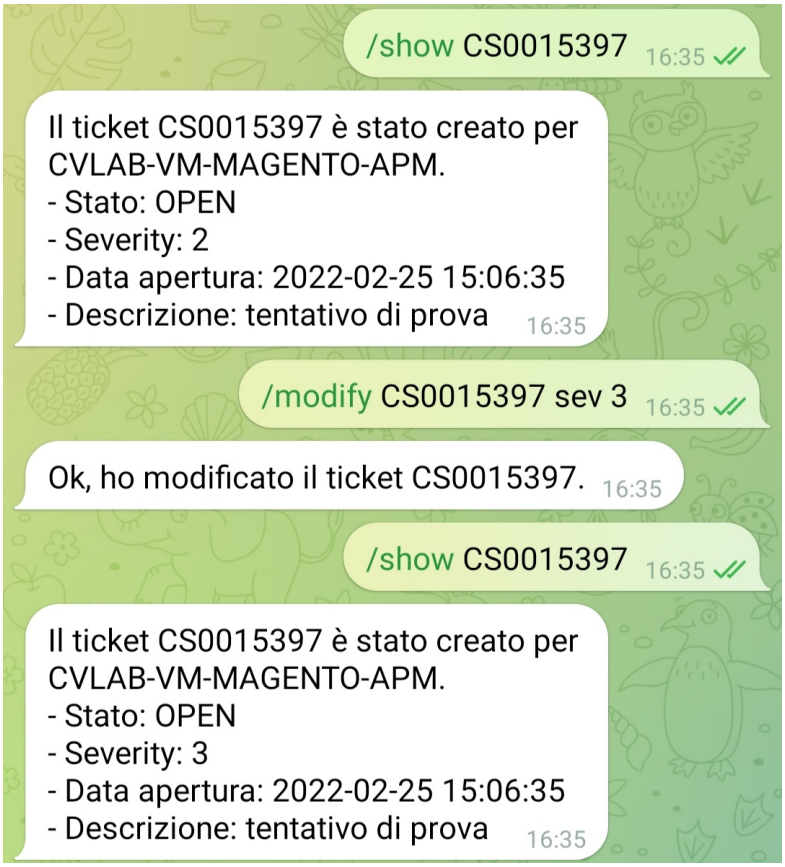


Figure 6.5: Modify Ticket

6.2.4 Modify Ticket

In Fig. 6.5 we can look how to modify a ticket. This is the only intent that it's possible to achieve exclusively via the command */modify* <*num_ticket*> <*parameter*> <*value*> and not with natural language, because it's a particular executive command that needs three clear and unambiguous parameters.

Why useful? A ticket maybe has parameters set automatically by the bot that don't match the user's wishes because of some special situation. For example, I want to assign the ticket to a group other than *ccc.linux* even though the problem host is a linux machine.

6.3 Remediation

The part of remediation is an important feature of the bot because it mixes the advice part with the operational part. In fact, the bot will propose known remedies to the user, which can be wheter simple links to in-depth documentation or operations that have helped solve the same problem in the past.

6.3.1 Restart Service

If for a specific service, for example a *Process Tomcat*, is set a well-known remedy like the restart, which is one of the most popular and widely used, the bot will ask to the user if he want to restart the service, explaining that this remedy has been successful a certain number of times (based on the feedback from other users, as shown in Fig. 6.6).
We have talked about restart services, but this use case can be extended to any operational action that the bot can perform. It will be enough to add a new operation on the MongoDB in the remediation collection (Fig. 6.8).

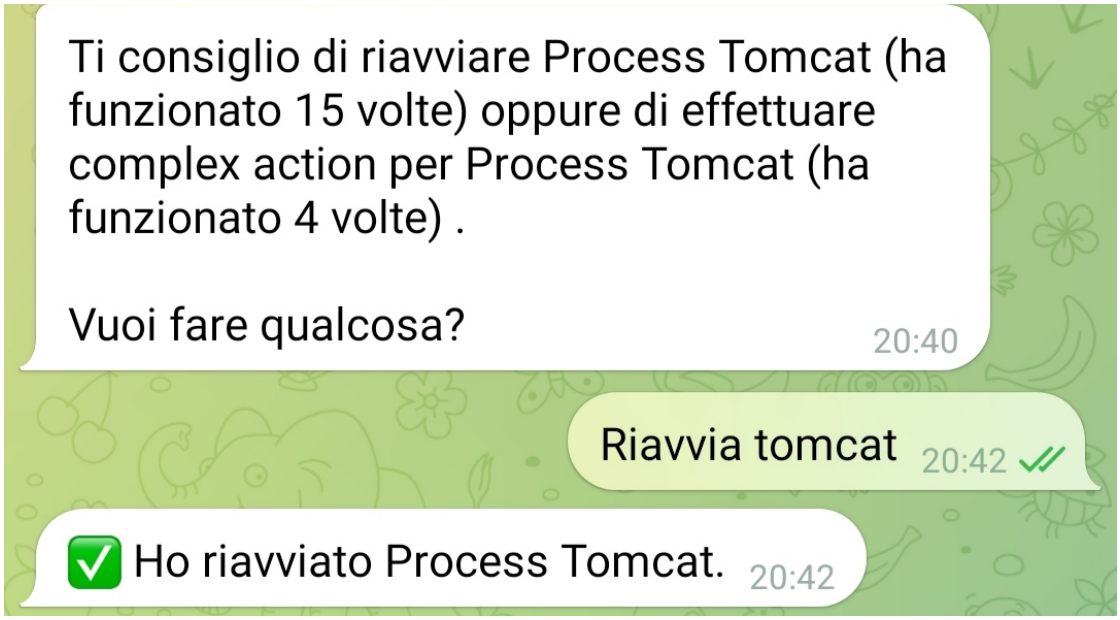


Figure 6.6: Restart Tomcat

Why useful? This feature allows the bot to be operative, obviously only when used by enabled users. It's very useful to know how often this remediation has worked in the past.

6.3.2 Restart Host On Azure

When we open a ticket because an host is in down status, first of all the bot will suggest a restart of the server. Also in this case the bot can be operative and is able to access the VM of a specific Azure subscription and perform the start, stop, restart operations.

Why useful? Many times a simple restart of the server can bring it into an reachable state, although this does not mean that there are no problems in the related services. By quickly restarting the host we can immediately focus on the real problems

6.3.3 Feedback

As we seen in section 6.3.1, there are some numbers for each tuple service-application-host. In Fig. 6.8 we have the service called *Process Tomcat* for which the restart has been successful 17 times for the application *proserpina* and 8 times for *polifemo*.
When we closed a ticket, the bot asks us how the problem was solved, proposing in a drop-down menu the popular remediations . If we click on one of these, the counter on the MongoDB will be incremented accordingly. There is another important type of feedback that we can nominate *custom feedback*, which is, unlike the previous one that is guided by the menu, a free way of saving the record. In fact, by choosing this type, we have the possibility of expressing with a sentence how we solved the problem and this phrase will be saved on the MongoDB along with a screenshot of the problem, with the relative name of the host, critical and warning services.

Why useful? The feedback helps us to have a bot that learns from problems that happen often and for which a remedy is quickly suggested. The possibility of the custom feedback is fundamental because an administrator can look what the user has written and for which situation. Then, he can decide to add this new type of remedy and make it a standard remediation in the future.

Figure 6.7: Remediations On MongoDB

6.4 AppDynamics

All the skills for AD are located in AppDynamics.py, where there is an object called *AppDynamicAPI* which has all the relative endpoints and the token for the API calls. As already mentioned, we use AD for an application level of monitoring, so when an user asks about an application the bot redirects the request to this module.

6.4.1 Applications Status

With this intent, an user wants to know the general health of an application, and the bot gives him an overview based on the health rules triggered. If at the moment there is a logged HR, the bot will translate the problem returned by the API request into natural language by creating a structured response providing HR name and description. If, on the other hand, no HR has been triggered, it will inform the user that the application is in a good state.
The fact that the application is fine on the AD side doesn't imply that the servers are fine too, in fact the bot will eventually ask the user if he wants

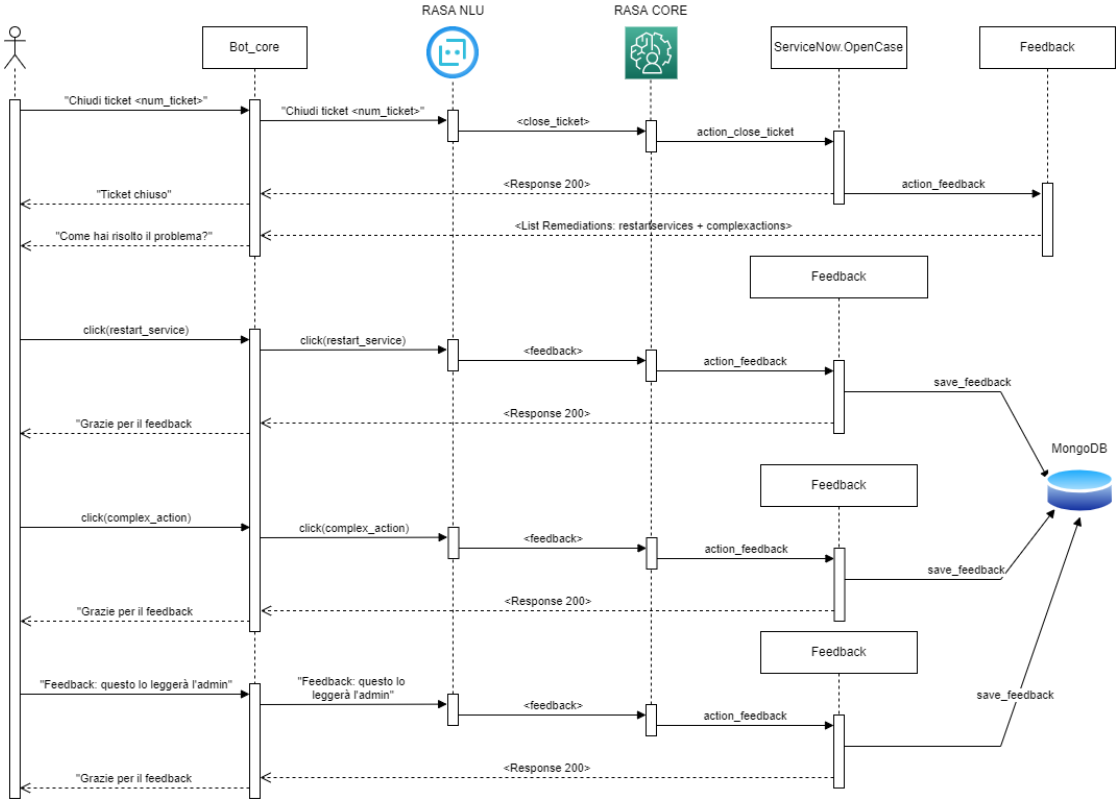


Figure 6.8: FeedbackDiagram

to take a look at the application's servers too, moving to CMK.

Why useful? For an AD user is important to have a general view of the health of monitored applications, with the names of the triggered HR which can address towards a resolution or to open a ticket for the application. If we decide to open a ticket for the application, the bot will compile the 'description' field of ticket with the comment of the HR.

6.4.2 Synthetics Status

For what concerns the monitoring of the synthetics, the bot, as we see in Fig. 6.9, interprets the data from the API call and shows three possible types of output:

- A red X symbol: the synthetic is disabled, so the monitoring is not possible;

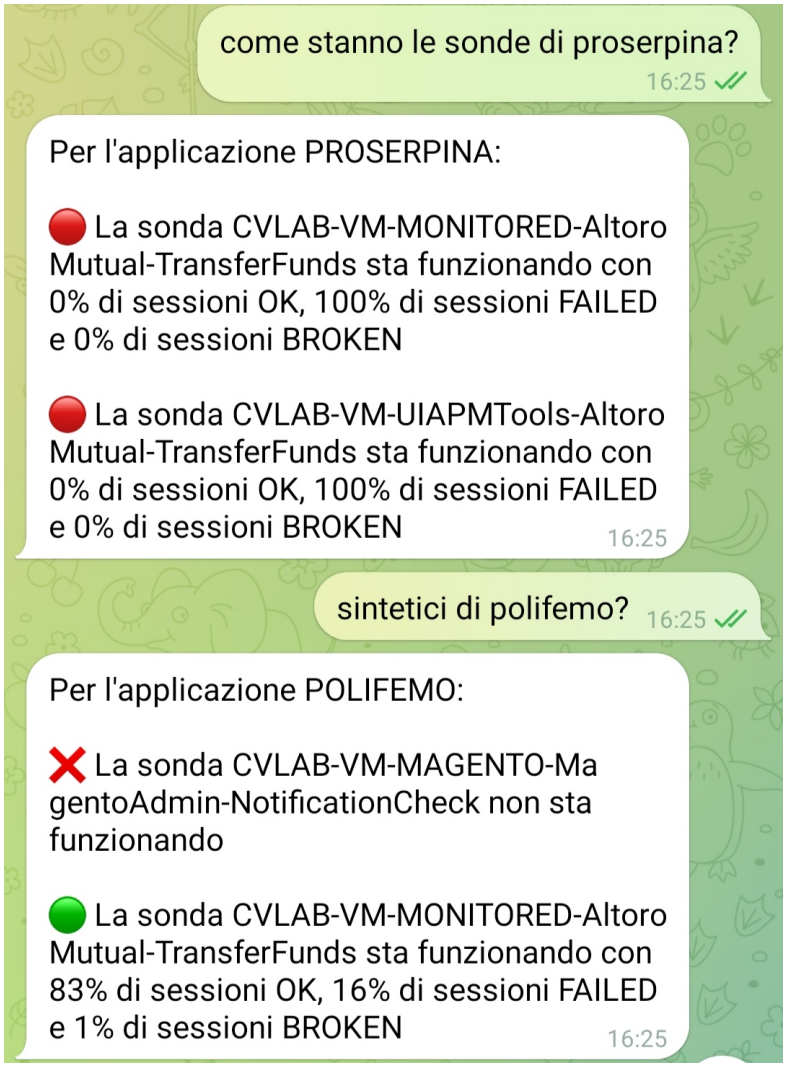


Figure 6.9: Synthetics Output

- A green bullet: the synthetic is enabled and the majority of sessions are OK;
- A red bullet: the synthetic is enabled and the majority of sessions are FAILED.

Why useful? For and end user experience is important to have, in a quick way, an overview that illustrates the screenshot of the actual situation. Also in this case, it's possible to open a ticket if there are unexpected problems.

6.4.3 Health Rule

Like on CMK, also with AD is possible to deactivate or activate alarms. In this case, does not exist the concept of downtime, but we can disable and enable the HR.

Why useful? It's fundamental to have the possibility of turning off the alarms so that applications are not in critical states any time it is normal for them to be so, perhaps due to periods of maintenance or inactivity.

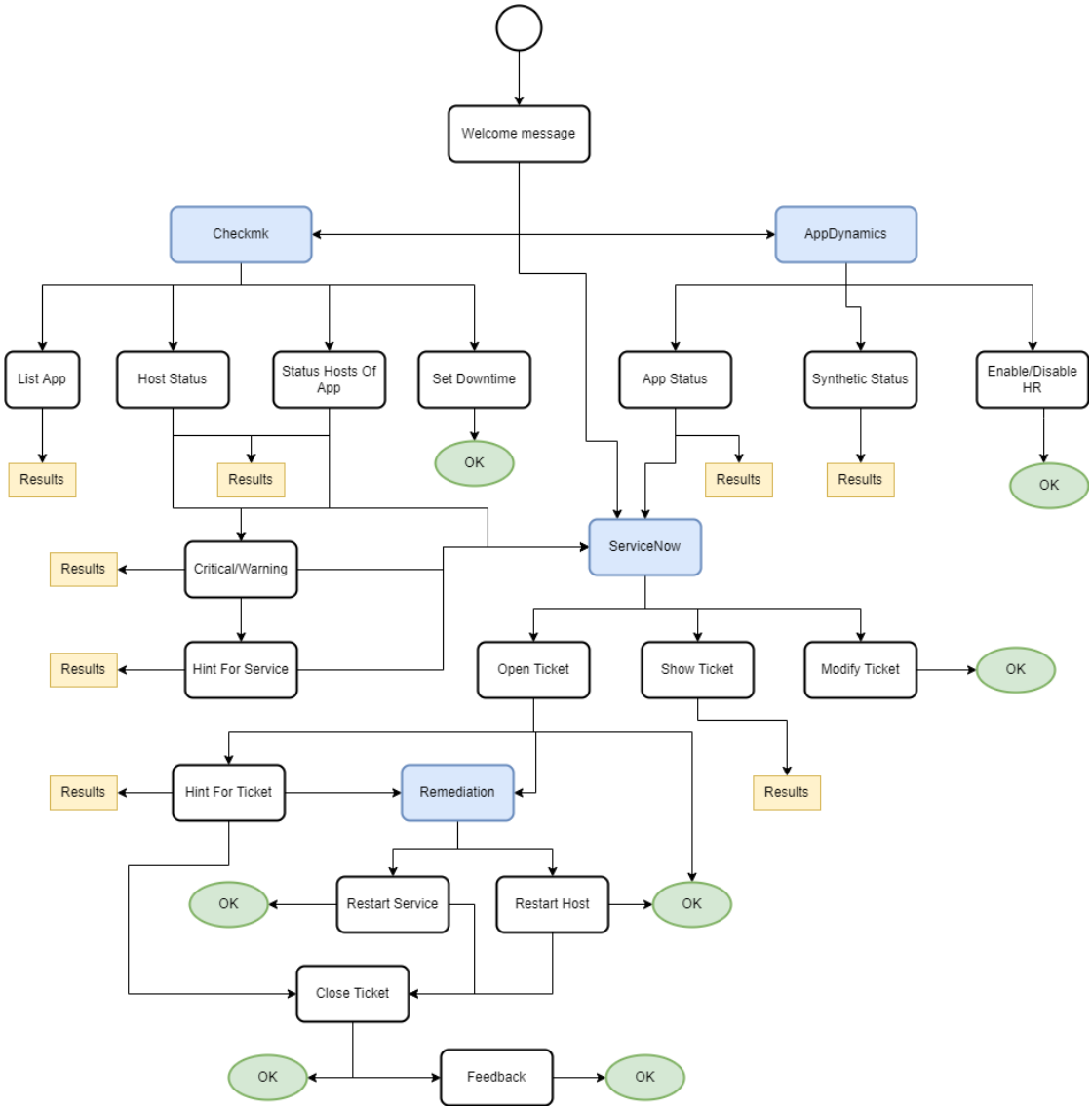


Figure 6.10: Intents Diagram

Chapter 7

Conclusions

The world of chatbots is a growing world that has already demonstrated its importance in many contexts over the last few years, so our aim was to find the best from all existing technologies and adapt them to the company's business needs.
We took advantage of the structure of some of the bots mentioned in chapter 2 to create a custom assistant that could take advantage of all their best qualities.

As a result we got a virtual assistant that can understand the user's intention in several different ways and is well adapted to the needs of our company.
It has a short-term memory that improves conversation and makes it more intelligent, as well as having the feedback system described above that facilitates future suggestions to situations already encountered.
The first development was intended for internal use, to improve the work of the company's operators and facilitate problem solving, but in the future it will also be possible to export the product to the company's customers to allow them a consultative view only, so that they can have an up-to-date view of their infrastructure.
The bot, despite being ready for many different types of chat connectors, has been refined and improved especially for Telegram, where all possible features have been exploited, such as drop-down menus or buttons that appear in chat and then disappear once clicked.

7.1 Future Works

This thesis was designed to meet the needs of performance monitoring, but the same bot backend can also be used for other platforms and other targets. As already mentioned on many occasions, the modular and scalable structure of the project allows for various adaptations.

Authentication As regards access policies, it's necessary to insert an ID (Telegram or Teams) on the MongoDB to be enabled, but in the future it will be possible to create an authentication system with passwords and a profiling of users who will be assigned to different groups and will have the relevant privileges.

Training An interesting evolution is to create an automatic training system with tools that create, depending on the language, a large number of sentences sufficient to make the bot able to easily understand the intentions of user. For now, in the training file, many different sentences have been created (about 200 per intent) in a smart way, knowing how the bot's learning works and trying to avoid imbalances, as explained in the chapter 3.

NLU We have decided to use Rasa's NLU and action prediction capabilities as the core, but nothing prevents us from using new tools in the future. In that case, it will be useful to recycle the various modules, but we will have to change the core of the bot, which was designed and adapted for Rasa.

Voice Another important feature may be to add the possibility of making requests via voice messages instead of text messages, either on the platforms already used such as Telegram and Teams, or via Amazon's Alexa.

Bibliography

[1] *Definition of ChatOps*. Addteq, 2017. URL: https://www.addteq.com/blog/2017/08/chatops-more-than-just-a-buzzword (cit. on p. 1).

[2] *Definition of ChatOps*. Atlassian, 2016. URL: https://www.atlassian.com/blog/software-teams/what-is-chatops-adoption-guide (cit. on p. 1).

[3] Jason Hand. *ChatOps*. O'Reilly, 2016 (cit. on pp. 2, 8, 12).

[4] *Rasa*. Rasa, 2022. URL: https://rasa.com (cit. on pp. 4, 17).

[5] *Wit.ai*. Facebook, 2022. URL: https://wit.ai (cit. on p. 4).

[6] *Nlp vs Nlu*. Rasa, 2019. URL: https://rasa.com/blog/nlp-vs-nlu-whats-the-difference (cit. on p. 7).

[7] Pierre Berthet-Rayne, Maura Power, Hawkeye King, and Guang-Zhong Yang. «Hubot: A three state Human-Robot collaborative framework for bimanual surgical tasks based on learned models». In: (2016), pp. 715–722. DOI: 10.1109/ICRA.2016.7487198 (cit. on p. 10).

[8] *Hubot*. GitHub, 2020. URL: https://hubot.github.com (cit. on p. 11).

[9] *Lita*. Lita.io, 2020. URL: https://docs.lita.io (cit. on p. 11).

[10] *Errbot*. Guillaume Binet, Tali Davidovich Petrover and Nick Groenen, 2022. URL: https://errbot.readthedocs.io/en/latest (cit. on p. 12).

[11] Hao Fang, Hao Cheng, Maarten Sap, Elizabeth Clark, Ari Holtzman, Yejin Choi, Noah A. Smith, and Mari Ostendorf. *Sounding Board: A User-Centric and Content-Driven Social Chatbot*. New Orleans, Louisiana, 2018. DOI: 10.18653/v1/N18-5020. URL: https://aclanthology.org/N18-5020 (cit. on p. 14).

[12] *Rasa NLU*. Rasa, 2022. URL: https://rasa.com/docs/rasa/nlu-training-data/ (cit. on p. 18).

[13] *Rasa Core*. Rasa, 2022. URL: https://rasa.com/docs/rasa/contextual-conversations/ (cit. on p. 21).

[14] *MongoDB*. MongoDB, 2022. URL: https://www.mongodb.com/ (cit. on p. 22).

[15] *What Is Nginx*. Kinsta, 2022. URL: https://kinsta.com/knowledgebase/what-is-nginx (cit. on p. 24).

[16] *What Is Redis*. Amazon, 2022. URL: https://aws.amazon.com/it/redis/ (cit. on p. 25).

[17] *Checkmk*. Checkmk, 2022. URL: https://docs.checkmk.com/latest/en/monitoring_basics.html (cit. on p. 29).

[18] *AppDynamics*. Kiratech, 2022. URL: https://www.kiratech.it/devops/appdynamics (cit. on p. 30).

[19] *AppDynamics Doc*. AppDynamics, 2022. URL: https://www.kiratech.it/devops/appdynamicshttps://docs.appdynamics.com/21.9/en/application-monitoring/overview-of-application-monitoring (cit. on p. 30).

[20] *Health Rules*. AppDynamics, 2022. URL: https://docs.appdynamics.com/21.9/en/appdynamics-essentials/alert-and-respond/health-rules (cit. on p. 30).

[21] *ServiceNow*. ServiceNow, 2022. URL: https://www.servicenow.com/products/itsm/what-is-it-ticketing-system.html (cit. on p. 32).

Ringraziamenti

Un ringraziamento speciale va alla Communication Valley Reply, in particolare ai manager Federico e Stefano, che hanno creduto in me e mi hanno dato la possibilità di effettuare la mia prima esperienza lavorativa, oltre ai colleghi Giovanni, Giorgio, Luca, Alex e Roberto che mi hanno aiutato tanto.

Un sentito ringraziamento alla professoressa Maristella Matera per la fiducia riposta in me e per tutti i consigli che mi hanno guidato nella creazione di questo lavoro.

Grazie a tutti i miei colleghi universitari, in particolare Sacco, Tino, Leo, Tone, Zazzi, Arianna e Totta, con i quali ho condiviso gioie e momenti difficili e che hanno reso questo percorso più leggero.

Grazie ai miei compagni di squadra della Sporting C.B. che mi hanno fatto apprezzare il valore di appartenere a un gruppo e lottare tutti insieme per un obiettivo sportivo.

Grazie ai miei amici storici Cristian e Simone, i quali mi hanno garantito svago e spensieratezza in momenti difficili e sui quali so sempre di poter contare.

Grazie al gruppo AS Pes che mi ha assicurato divertimento e leggerezza in tante fasi della mia vita.

Grazie a mio fratello Renato che si è sempre mostrato disponibile per ogni tipo di consiglio e mi ha sempre dato il sostegno necessario.

Infine grazie di cuore alla mia famiglia, ai miei zii e ai miei nonni, ma soprattutto ai miei genitori Maurizio e Antonella che mi hanno sempre supportato in tutto e per tutto e ai quali sarò riconoscente a vita, questo traguardo è dedicato a voi.