FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

IPBRICK OS – AI Voice/Chatbots

Bruno Miguel Costa Baptista

Msc Dissertation

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor - Ana Cristina Costa Aguiar
Co-Supervisor - Luís Miguel Ramalhão Ribeiro

March 2022

© Bruno Miguel Costa Baptista, 2022

Abstract

The application of technologies that require little to no human intervention have been applied in several domains at a worldwide scale, ever more so with the implementation of Artificial Intelligence (AI), making services more available and efficient. However, in fields such as Call center/Contact center operations, the challenge relies on building a service capable of impersonating human interactions, simulating natural actions and features that we possess.

The overall theme for the development of this dissertation consists of developing a system capable of automating the communication flows of both voice (Voice over Internet Protocol) and chat (Instant Messaging) of the IPBRICK Operating System, using AI, allowing for the consequent utilization of virtual assistants. In the context of this dissertation, the assembly of a commercial support bot using AI frameworks is required, granting IPBRICK clients and partners the ability to report technical issues directly to their services, along with the development of two independent interfaces, according to the necessary communication flows, relying on the same AI engine; and the implementation of a dashboard interface to observe their performance.

This dissertation provides a detailed description of all implementations that were carried out for all objectives. In a first instance, a literature review was conducted, in order to make sure that the conceptualization phase would be performed in the best possible manner. Then, the implementation phase begins by performing a thorough analysis on every architectural structure that each solution is set upon, determining the best course of action, followed by describing all applied techniques that led to a successful implementation. The execution of virtual assistants used Rasa, a framework for building contextual AI assistants, where IPBRICK requested the implementation of a hand-off mechanism, where human assistance would be provided in a last resort scenario. For voice-based bots, Asterisk was used to perform all telephony business logic, where the Google Cloud Platform was fundamental when introducing Speech-to-Text and Text-to-Speech capabilities to the system. On the other hand, chat-based bots used React, a JavaScript library dedicated to building user interfaces, along with *ejabberd*, an application server that allowed users to interact with operators using IPBRICK CAFE, in the act of a human hand-off. Regarding the conception of the dashboard interface, KoolReport was used for data delivery, transforming information stored in a database into an understandable format through charts and graphs.

After all solutions were developed, a controlled experiment was conducted, interacting with virtual assistants through both communication flows, observing the performance of the deployed bots through the dashboard interface. Given two instances, at the beginning and at the end of this study, these unique time frames describe a gradual learning process, where bots initially hand over the majority of conversations to a designated operator, failing to fulfill their task. After a consecutive learning process, the last scenario showcases the performance of improved bots, where at the end of the study they are capable of autonomously conducting most conversations without requiring human intervention. In parallel, an analysis to voice-based components is performed, concluding that the proposed solutions resulted in a more efficient system.

At the end of this work, the proposed solution contributed to the optimization of the performance regarding voice-based assistants, by effectively improving speech recognition, and devising a mechanism that reduces the overall execution time and monetary costs. Additionally, the implementation of a configurable and non-intrusive chat interface introduced a new perspective towards text-based assistants, capable of being deployed onto any web page. A dashboard interface was also developed, monitoring the performance of virtual assistants, facilitating the process of a proper and gradual improvement on all bots. The devised solutions contributed to the overall reduction of required human resources, converging all developments into an AI subsystem capable that performs tasks without human intervention.

Keywords: Rasa, Chatbot, Voicebot, Dashboard, Artificial Intelligence

Resumo

As tecnologias que requerem pouca ou nenhuma intervenção humana têm vindo a ser implementadas em vários domínios a uma escala mundial, sendo cada vez mais frequente a utilização de Inteligência Artificial (IA), resultando em serviços mais eficientes e acessíveis. Contudo, em áreas de *Call center/Contact center* surge o desafio de desenvolver serviços personificáveis, simulando as características naturais que o ser humano exprime regularmente.

O objetivo desta dissertação incide no desenvolvimento de um sistema capaz de automatizar os fluxos de comunicação por voz (*Voice Over Internet Protocol*) e por *chat* (*Instant Messaging*) do Sistema Operativo IPBRICK, recorrendo a IA, por forma a possibilitar que os atendimentos de voz e chat sejam efetuados com recurso a assistentes virtuais. No contexto da dissertação, é necessário a construção de um *bot* comercial de suporte usando ferramentas de IA, para que clientes e parceiros IPBRICK sejam capazes de reportar problemas técnicos de forma autónoma, juntamente com o desenvolvimento de duas interfaces (para cada fluxo de comunicação), recorrendo à mesma ferramenta de IA; e a implementação de uma *dashboard* capaz de monitorizar a *performance* dos mesmos.

Esta dissertação descreve, de forma detalhada, as implementações necessárias para que todos os objetivos sejam realizados. Em primeiro lugar, realizou-se uma revisão da literatura, com o intuito de consolidar conhecimentos, garantindo um desenvolvimento fundamentado. Seguidamente, na fase de implementação procedeu-se uma análise prévia à arquitetura de cada solução, determinando os focos de incidência para que todos os desenvolvimentos sejam implementados com sucesso. Os assistentes virtuais desenvolvidos recorreram à ferramenta *Rasa*, que permite construir assistentes contextuais utilizando IA, onde se inferiu a implementação de um mecanismo de *hand-off*, recorrendo a intervenção humana numa situação de último caso. Nos *Voicebots*, utilizou-se o *software Asterisk* para executar a lógica referente a um sistema de telefonia, recorrendo ao *Google Cloud Platform* para introduzir os módulos de *Speech-to-Text* e *Text-to-Speech*. A implementação de *Chatbots* foi desenvolvida utilizando *React*, uma biblioteca *JavaScript* dedicada a construir interfaces, juntamente com a comunicação com um servidor *ejabberd*, o que permite uma comunicação interpessoal entre utilizadores da interface *chat* e indivíduos que utilizem a plataforma IPBRICK CAFE, em circunstâncias de *hand-off*. Relativamente à construção da *dashboard*, foi utilizada a ferramenta *KoolReport*, permitindo a visualização de informação proveniente de uma base de dados.

Após a implementação das soluções, realizou-se um estudo em ambiente controlado, onde se testou os assistentes virtuais através de cada interface construída, e observou-se a *performance* dos *bots* pela *dashboard*. Analisados dois instantes temporais, referentes ao primeiro e ao último dia de estudo, observou-se uma aprendizagem gradual por parte dos *bots*, onde numa primeira análise os assistentes transferiram a maioria dos utilizadores para um operador, devido a não conseguirem concluir a tarefa proposta de forma autónoma. Após um processo de ajuste e correção por parte dos assistentes, observou-se que no último dia, grande parte das conversas foram realizadas sem intervenção humana, sendo possível identificar esta evolução na *dashboard*. Simultaneamente, analisou-se a *performance* dos componentes constituintes da interface voz, onde se concluiu que as soluções propostas resultaram num sistema mais eficiente.

A solução proposta contribuiu para a otimização da *performance* de assistentes que comunicam por voz, melhorando o reconhecimento de fala e introduzindo um mecanismo que diminuiu os custos inerentes a este serviço, e simultaneamente reduziu o tempo de execução. Adicionalmente, a implementação de uma interface *chat* configurável e não intrusiva conferiu uma nova perspetiva a assistentes que comunicam neste formato, e pode ser introduzida em qualquer página *web*. O desenvolvimento de uma interface *dashboard* permitiu a monitorização de assistentes virtuais e contribuiu para o melhoramento de cada *bot*. As soluções implementadas podem contribuir para a redução de recursos humanos, uma vez que o desenvolvimento de um subsistema de IA trata de realizar tarefas de forma autónoma.

Palavras-chave: *Rasa*, *Chatbot*, *Voicebot*, *Dashboard*, Inteligência Artificial

Acknowledgements

First and foremost, I would like to thank my family, for supporting me at times when I felt lost, always presenting a positive approach to all hurdles and hardships that came along the way. To this day, Union stands as the word that best describes our companionship and unconditional love, always demonstrating that together we become stronger and better individuals. This is a motto that I will carry with me at all times, never forgetting my true roots.
To my close friends, *Zangaleões*, that continuously gave me a sense of community. Always carrying a cheerful presence, my life-long friends always put me in a better mood, no matter the occasion. On stressful occasions, you always helped me relax and loosen up a bit, hearing my complaints and always being present in difficult times.
To my dear friends and confidants throughout my University experience, Tozé, Sérgio, Roque, Lúcia, and especially Daniela. Words cannot describe our journey throughout these years, to which I am grateful for taking part in each one of your lives. Being there in times of need, whenever I felt misguided, this work is devoted to you. I am sure that I will always save our experiences close to my heart, and I am excited for the future that awaits us.
I would also like to thank Prof.ª Ana Aguiar, who provided helpful insights and advice that were steer-defining for this dissertation. I am very grateful for her mentorship and for the opportunity to work and learn from her.
For one of the most rewarding challenges and experiences, I would like to thank Eng. Luís Miguel Ribeiro, for welcoming me into the vibrant atmosphere that is IPBRICK, and for all the guidance and support that I had the pleasure to be a part of. All of your teachings provided me with an unforgettable experience that I will personally carry and grow as a person.
I owe a very special thanks to all the IPBRICK team, who so warm-heartedly welcomed me in their environment, for always giving me a helping hand when things went unexpected, and giving me support during my time at IPBRICK.

This work is dedicated wholeheartedly to Holi.

虎口拔牙。

”*Pull a tooth from the tiger’s mouth.*”

Ancient chinese proverb

Contents

1 Introduction **1**
1.1 Context & Motivation . . . 1
1.2 Goals . . . 1
1.3 Structure . . . 2

2 Literature Review **3**
2.1 Problem Definition . . . 3
2.2 Problem Characterization . . . 4
2.2.1 Bot Classification . . . 4
2.2.2 Architecture . . . 5
2.2.3 Response generation . . . 7
2.2.4 Dialogue Strategies . . . 9
2.3 Available Technologies . . . 10
2.3.1 Artifical Intelligence Bot Frameworks . . . 10
2.4 Devised Solution . . . 14

3 IPBRICK Group **15**
3.1 History . . . 15
3.2 IPBRICK Operating System . . . 15
3.2.1 iPortalDoc . . . 15
3.2.2 IPBRICK MAIL . . . 15
3.2.3 IPBRICK UCoIP . . . 16
3.2.4 IPBRICK CAFE . . . 16

4 Solution Planning and Structure **17**
4.1 Background on IPBRICK solutions . . . 17
4.1.1 Asterisk . . . 17
4.1.2 Google Cloud Platform . . . 17
4.1.3 ejabberd . . . 17
4.1.4 KoolReport . . . 18
4.2 Solution integration in IPBRICK OS . . . 19
4.3 Rasa Framework . . . 21
4.3.1 Core purpose . . . 21
4.3.2 Architecture . . . 21
4.3.3 IPBRICK Commercial Support Bot . . . 28
4.4 Voice-based Bot . . . 33
4.4.1 Overview . . . 33
4.4.2 Voicebot Architecture . . . 33
4.4.3 Voicebot implementation . . . 38
4.5 Chat-based Bot . . . 44
4.5.1 Overview . . . 44
4.5.2 Chatbot Architecture . . . 44
4.5.3 Chatbot Implementation . . . 46
4.6 Dashboard . . . 49
4.6.1 Overview . . . 49
4.6.2 Implementations . . . 49

5 Validation and Evaluation **52**
5.1 Evaluated Metrics . . . 52
5.2 Validation & Results . . . 53

6 Conclusion and Future Work **59**
6.1 Accomplished Goals 59
6.2 Future Work 60

7 References **62**

List of Figures

1 Chatbot Classification 4
2 Chatbot Architecture 5
3 AIML code 7
4 Seq2Seq RNN model 8
5 Example of error handling and confirmation 9
6 Rasa architecture 10
7 Training data formats 11
8 Training dialogue in Rasa Stories 11
9 Rasa Core correction example 11
10 Microsoft Bot Framework architecture 12
11 Dialogflow conversation 13
12 Simplified human-to-bot communication flow 19
13 IPBRICK OS Web Interface 20
14 Rasa Open Source architectural diagram 21
15 Rasa project file structure 24
16 Default Rasa project NLU and Stories configuration 25
17 Rasa interactive 26
18 Rasa X interface 27
19 Input and Output response from REST endpoint 27
20 Diagram of proposed AI solution 28
21 Custom channel connector message example 29
22 Overwritten action that initiates the section 30
23 Hand-off custom action 31
24 Hand-off request from a user perspective 31
25 NLU fallback rule 32
26 Outline of the voicebot architecture 34
27 In-depth overview of the voicebot architecture 35
28 Requests from Google Cloud Platform 38
29 Enhancements regarding voice recognition 39
30 Complete voicebot architecture 42
31 Requests from Google Cloud Platform - audio file reusage 43
32 Baseline chatbot architecture 44
33 Complete overview of the chatbot architecture 45
34 Customized user interface 46
35 Hand-off from an IPBRICK CAFE perspective 48
36 Database for virtual assistants 50
37 Number of calls for each assistant - day 1 53
38 Call treatment for each communication flow, English assistant - day 1 54
39 Call treatment for each communication flow, Portuguese assistant - day 1 54
40 Number of calls to each assistant, per day 55
41 Call treatment for each communication flow, English assistant - day 5 56
42 Call treatment for each communication flow, Portuguese assistant - day 5 56
43 Google Cloud API requests, case study 57

List of Tables

1 Default performance of every component - 50 interactions 36
2 Default minimum and maximum values for every component 36
3 Performance of every component, after implementations - 50 interactions 42
4 Minimum and maximum values for every component, after implementations 43
5 Call distribution by time of day, case study 57
6 Voicebot component duration, case study 57

Abbreviations and Acronyms

3P	Third Party
AI	Artificial Intelligence
AIML	Artificial Intelligence Markup Language
API	Application Programming Interface
CDD	Conversation-Driven Development
CLI	Command Line Interface
EN - BOT	English Commercial Support Bot
EOL	End of Line
HTTP	Hypertext Transfer Protocol
IA	Inteligência Artificial
JID	Jabber Identity
JSON	JavaScript Object Notation
LSTM	Long Short-Term Memory
MQTT	Message Queuing Telemetry Transport
NLP	Natural Language Processing
NLU	Natural Language Understanding
OS	Operating System
PHP	Hypertext Preprocessor
PT - BOT	Portuguese Commercial Support Bot
REST	Representational State Transfer
RNN	Recurrent Neural Networks
SDK	Software Development Kit
SIP	Session Initiation Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
STT	Speech-to-Text
Seq2Seq	Sequence to Sequence
TCP	Transmission Control Protocol
TTS	Text-to-Speech
UCoIP	Unified Communications over Internet Protocol
URL	Uniform Resource Locator
VoIP	Voice over Internet Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
YAML	Yet Another Markup Language

1 Introduction

1.1 Context & Motivation

In recent years, there has been an exponential application of conversational Artificial Intelligence (AI) in every aspect that demands interpersonal communication, becoming ubiquitous in our daily lives, ever more distinct in the social domain. In a world that is continuously developing itself towards automated services, this technology takes a fundamental role on impersonating human interactions as naturally as possible, thereby allowing services to be fully independent, without the need for human intervention during this process. The application of Machine Learning and Natural Language interface has allowed for this innovative technology to be introduced to several areas of development, such as healthcare and education, due to the advancement of computing power and the nature of its design [1]. The interaction of these conversational agents, like chatbots or voice assistants, can be speech or text-based, being classified by their autonomous level, where the lower level forces customers to follow a rigid path, and higher levels are capable of engaging in a conversation about a specific topic. In this regard, implementing a service that is capable of maintaining an active dialogue by processing the input, understanding it, and formulating a response without breaking the natural flow of the conversation in real-time is fundamental to good execution [2].

The overall theme for the development of the dissertation is properly introduced, joined by COMCIBER - COMUNICAÇÕES E CIBERNÉTICA LDA to devise a solution that focuses on the component of unified communications, more specifically on voice and text services of IPBRICK Operating System (OS), both of which hold significant roles in the solutions of Call center/Contact center of IPBRICK OS.

1.2 Goals

The objective of this project consists of developing a system capable of automating the communication flows of both voice (Voice over Internet Protocol - VoIP) and chat (Instant Messaging) of the IPBRICK OS, using AI, allowing for the consequent utilization of virtual assistants. In the context of this dissertation, the assembly of a commercial support bot using AI frameworks is required, granting IPBRICK clients and partners the ability to report technical issues directly to their services, along with the development of two independent interfaces, according to the necessary communication flows, relying on the same AI engine, and the implementation of a dashboard interface to observe their performance.
For voice-based bots, it is crucial to establish an interface capable of communicating with the bot via speech, where the bot formulates an audible response that is sent back to the listener. Being provided with IPBRICK's private telephony system, the focus of this work is set to enhance the baseline configurations that have been implemented for AI communication, as well as to provide a solution that improves the performance of a voice-based system.
Whereas voice-based bots operate via audible enquiries and replies, a text-based bot approach requires an interface that communicates with the AI engine through text. To this end, an additional objective stands on creating a text-based interface, that not only communicates with the bot but gives users the ability to exchange information with an individual that uses the IPBRICK CAFE instant messaging system, if requested. Moreover, this interface needs to be bot-agnostic, meaning that it needs to function regardless of any bot that is developed under the Rasa bot Framework, as it is essential to be deployable on any website, as a service, in order to fit any use case.
In addition to these prior objectives and the overall core development of the bot, it is also required to implement a dashboard that resides within the IPBRICK OS, enabling clients to evaluate the performance of their bots by analyzing key factors, as well as monitoring their performance to determine if the deployed bots have proven to be successful. In conjunction with Rasa X, a graphical user interface tool provided by the Rasa bot Framework, this dashboard is set to point out any imperfections that occurred when interacting through any of the preferred communication flows, whether by voice or chat, indicating

possible focal points that need to be adjusted in order to yield optimal results.
Finally, supplementary configurations on the IPBRICK OS are required to automate the process of creating both voice and text-based interfaces, simplifying these procedures in a general use case that is adaptable to any instance.
An AI subsystem within the IPBRICK OS is crucial, making available all necessary management and configuration interfaces of voice and chat-based bots, for the creation of detailed and personalized stories.

1.3 Structure

The following sections explore each major component in further detail. The current knowledge about the implementation of a bot through the analysis of similarly published works is presented in chapter 2, where a brief overview of the architecture of a bot is introduced. Then, a modular view of the necessary components that comprise this system is demonstrated, including different classifications of a bot, exposing some constraints and different implementations that are characterized as an obstacle for the devised solutions, as well as describing the different models that have been applied in the past for response generation and some dialogue strategies to better humanize the program's behavior. Additionally, the current knowledge about the technologies that are relevant for the devised solution is described, in hopes of understanding the best approach to its implementation.
Then, section 3 details the architecture of the IPBRICK system, with emphasis on the subsystems that this assignment will be focused on.
Subsequently, section 4 showcases the methodology that was applied throughout the progress of the proposed work in this dissertation, describing all major developments, delineating the thought process behind every given solution. An in-depth explanation is given to every devised interface, as well as the construction of the commercial support bot, exposing their architectures.
In chapter 5, a critical analysis of the results obtained during the implementation phase is made, where the commercial support bot is put into practice under a controlled environment, testing the developed interfaces in the IPBRICK network, observing a simulated gradual development of the assistant through the implemented dashboard.
Lastly, section 6 provides a conclusion to the work's development, followed by future enhancements to this matter.

2 Literature Review

2.1 Problem Definition

When planning an AI subsystem, the proposed challenge relies on constructing each major component of this said subsystem, the automation of both VoIP and Instant Messaging, and selecting appropriate technologies that ensure proper interactions and quality standards between the user and its intelligent counterpart.
In 1950, Alan Turing presented the question ”Can machines think?”, along with the Imitation Game, in order to shed some light on that matter. Turing's Imitation Game aims to see whether a machine displays intelligent behavior by fooling the interrogator into thinking that utterances from the computer program were actually from an individual [3].

For both voice-assisted bots and conversational agents alike, a dialogue system is a core element when establishing these components, considering that it is defined as a computer program that supports spoken, text-based, or multi-modal conversational interactions between itself and the users [4]. These can be specified as task-oriented, if the interaction is set to accomplish a specific task, or non-task-oriented if the conversation holds no particular purpose. These systems have been around for some time, recently popularized with virtual assistants such as Apple's Siri and Amazon's Alexa, or chatbots like Cleverbot, that can generate an answer to any given domain.
A chatbot is properly defined as an online human-computer dialog system with natural language [5], or a software application that is able to communicate with users, by conducting a text-based conversation using natural language [6], with the purpose of understanding the user's intent and providing accurate responses that are relevant to the impending problem. It is generally viewed that the first-ever conceived chatbot is ELIZA, an early computer program conceived in 1966 that convincingly portrays the work of a human psychiatrist. Given an input sentence, ELIZA would identify keywords and apply pattern matching techniques to a set of pre-programmed rules, in order to generate a response [7].
When applying a dialogue system to a program that is able to recognize and translate voice-enabled commands, the concept of a voicebot is founded. A voicebot must incorporate speech processing, converting vocal input from a user to text, translating the text to an intention, and finally transforming this intention into action. The final setup of this process results in the conversion of the generated response to speech [8]. When giving a voice to a program, this application is personified and given character, thus conveying an emotional bond with the user, improving quality of service. This hands-free approach turns out to be, not only convenient, but very useful in business fields where both hands are constantly occupied. The simple act of turning on the light in a room, while simultaneously typing on the computer, can be easily performed by a voice assistant with a single voice command. This technology has been actively developed over the years, providing solutions for intricate tasks up to a great extent.

2.2 Problem Characterization

2.2.1 Bot Classification

When first introducing the concept of a chatbot, it is noticeable that there are many properties that may differ for each application. In order to bring clarification, a generalized list of categories that classify a bot in detail is proposed by [9], in figure 1. Some of these criteria are further uncovered in the next chapters, such as the Response Generation Method in 2.2.3.

Chatbot Categories		
	Knowledge domain	Generic
		Open Domain
		Closed Domain
	Service provided	Interpersonal
		Intrapersonal
		Inter-agent
	Goals	Informative
		Chat based/Conversational
		Task based
	Response Generation Method	Rule based
		Retrieval based
		Generative
	Human-aid	Human-mediated
		Autonomous
	Permissions	Open-source
		Commercial
	Communication channel	Text
		Voice
		Image

Figure 1: Chatbot Classification [9]

A chatbot's Knowledge Domain is determined by the range of its cognitive content, varying from generic, if the bot can assert any type of sentence from whichever domain; Open Domain, if the chatbot only provides answers of a particular domain; or a Closed Domain, if it focuses on a particular set of topics within a domain, having a limited number of responses.
When providing services, such as giving simple responses regarding Frequently Asked Questions, with no emotional attachment to the user it is considered an Interpersonal bot. On the other hand, an Intrapersonal bot acts as a companion that understands the user's needs, usually interacting with other applications or programs. An Inter-Agent chatbot communicates with other chatbots, sharing resources between them.
A chatbot can have three sets of purposes: Informative, Chat-based/Conversational, or Task-based. The first aims to, as the name suggests, provide information. This bot tends to follow a strict path, since it works inside a reduced domain to supply specific information. In contrast, a Chat-based/Conversational is usually applied to a slightly larger domain, in order to respond to different kinds of questions that the user may query. It also inherently expresses itself, to appear more human-like. The latter is used to perform a multitude of functions, excelling at requesting information and responding accordingly.

A bot's mode of operation is Human-mediated, if any human intervention is involved in the process of supplying responses, or can be considered Autonomous if no assistance is required. This criterion is usually connected to the chatbot's decision-making and overall intelligence, since it can hinder or promote its ability to respond in a flexible manner.
Lastly, as with every piece of technology, it can be considered Open-source or Commercial. When transmitting information to the user, it can communicate via text, voice, or images. In this Communication category, more than one criteria can be attributed to a chatbot [9].

2.2.2 Architecture

An in-depth layout of a chatbot's architecture gives a good overview of its components, where a detailed summary of each module is fundamental to the comprehension of this system. Multiple designs have been suggested for a chatbot's architecture. However, none of them presented a generalized solution. Therefore, an architectural design that includes all components was proposed, detailing each element when devising a chatbot solution [9],[10]. This architecture is comprised of 5 components: User Interface Component, User Message Analysis Component, Dialog Management Component, Backend and Response Generation Component, presented in figure 2.

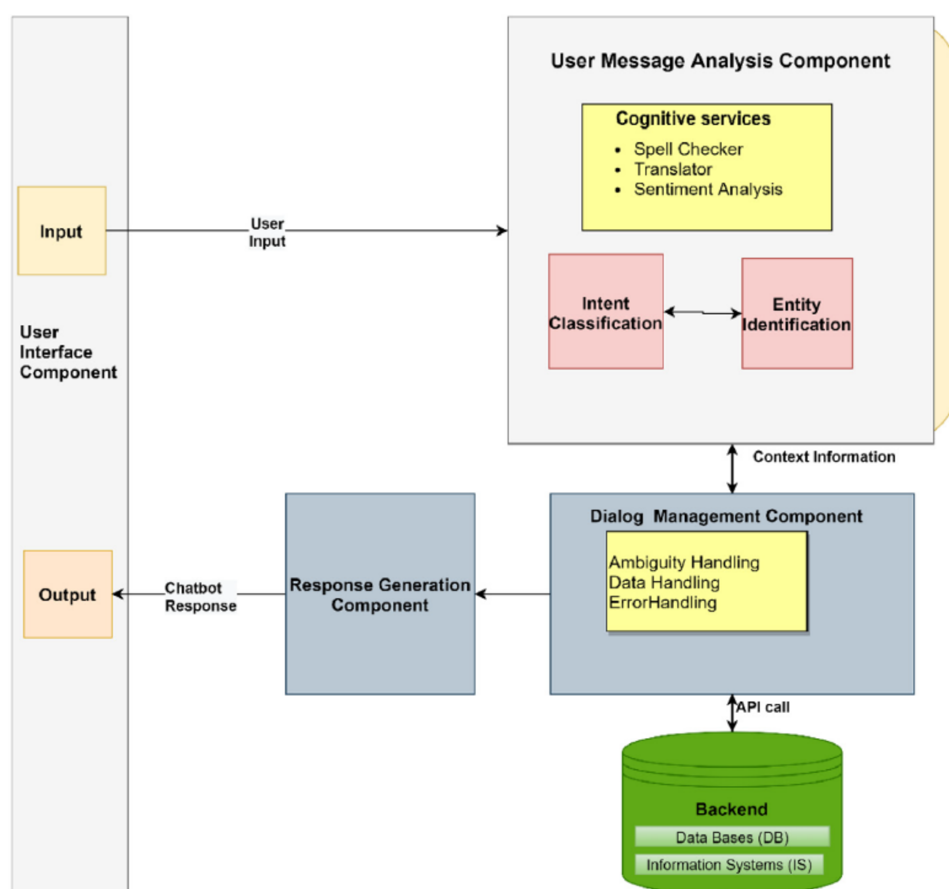

Figure 2: Chatbot Architecture [9]

User Interface Component

This interface is a fundamental component of the system, displaying the exchange of interactions between the user and the chatbot, where all inputs are introduced and the corresponding responses are received [9]. A proper implementation of this component is essential, given that the user must be able to instinctively recognize its mode of operation, as well as its limitations, without compromising quality of service.

User Message Analysis Component

When a user transmits an utterance, he expresses an intent or an action that is to be performed by the chatbot. The User Message Analysis Component is used to examine the input, capture the user's intent, and extract certain parameters, called entities, through pattern matching or machine learning techniques. The input can be retained as plain text, or processed by Natural Language Processing (NLP), to obtain the correct entities [9].
The goal of NLP is to take in the provided input, extract the semantic information, and create a grammatical data structure to be further processed by the Dialog Management Component [8].
Additionally, cognitive services may be added, such as a spell checker, translator, and sentiment analysis, to achieve better accuracy. A spell checker, as the name suggests, checks for spelling errors to better understand the user's intent. Translators are used in chatbots that are capable of translating more than one language at a time, converting all expressions to the bot's Natural Language Understanding (NLU). The latter, sentiment analysis, is used to observe the emotional state of the user through his input [9].

Dialog Management Component

This component conducts conversational context, by holding current intents and entities. If the chatbot requires additional information to perform a requested user action, it will ask for further questions as a mean to collect data, thus filling any missing entities. Dialog Management is usually comprised of three modules: Ambiguity Handling, which aims to provide answers if neither intent nor input is recognized; Data Handling, whose goal is saving and updating information as the dialogue progresses; and Error Handling module, which introduces mechanisms that handle unanticipated errors [9].

Backend

When processing the user's request, the chatbot relies on the back end to obtain the correct information and formulate an accurate response. When the required data is extracted through external resources, such as Application Programming Interfaces (API) or Databases, the information is forwarded to the Dialog Management Module and the Response Generation Module. If there is a need to recall previous conversations, a Relational Database may be used.

Response Generation Component

The Response Generation Component is essential when formulating understandable responses, consequently forwarding its content to the User Interface Component. The methods for developing replies consist of three approaches, which are further detailed in section 2.2.3.

2.2.3 Response generation

When researching this type of software, one must take into account how the chatbot takes the input, provided by a user, and outputs a response that fulfills a set of requirements that were previously intended. The methods for constructing the chatbot's dialogue system are essentially comprised of three approaches: rule-based, retrieval-based, and generative models [8], [9], [10], holding a fundamental task in NLP.

Rule-based Models

The first-most approach uses a set of hard-coded templates and rules that apply simple pattern matching or keyword retrieval techniques that consequently generate responses. This output is based on a predefined set of rules, when the lexical form of the input is recognized [10]. In a general case, rule-based models do not create new answers, as the knowledge base is hand-crafted by developers, in the form of conversational patterns [9].
The following example showcases a chatbot implemented with Artificial Intelligence Markup Language (AIML), where its underlying principle is put into practice when a bot holds a list of hand-crafted categories. Each document contains a <pattern> and <template> tags [8]. The considered AIML code shown in figure 3 illustrates the aforementioned principle:

```
1 <aiml version="1.0.1" encoding="UTF-8"?>
2 <category>
3    <pattern> HELLO BOT </pattern>
4    <template>
5       Hello my new friend!
6    </template>
7 </category>
8 </aiml>
```

Figure 3: Example of AIML code [11]

When the input matches the text stored inside <pattern>, the generated response corresponds to the message inside <template>, granting a fast response time [9]. For rule-based models, the problem relies on the fact that the programmer must issue all possible rules for a specified domain, and is prone to failure when spelling and grammatical errors occur [10]. For this pattern matching approach, the dialogue usually takes a single-turn communication, where responses are predefined and automated, thereby deprived of originality and flexibility that resides in human-like dialogue [9], including the ability to learn from conversations.

Retrieval-based Models

To overcome the hurdle of hand-coding all pattern matching pairs, the information retrieval-based model analyzes external data sources using an algorithm [10]. Unlike rule-based approaches, neural network-based models contain a machine learning algorithm [6], extracting content from the input using NLP, learning from the dialogue and maintaining conversational context [9]. This model grants more flexibility, as it queries external resources, recurring to a neural network to retrieve and assign a scoring function to multiple response candidates, returning the highest graded reply [12].

Yan et al. [12] proposes the following response ranking function:

$$Rank(S,Q) = \sum_{k} \lambda_k * h_k(S,Q)$$

Given Q as the user's reply, and S as its response candidate, the overall rank is defined, for a given k, as the summation of the multiplication between the scoring function $h_k(S,Q)$ and its corresponding scoring weight λ_k [12].

Generative-based Models

The latter approach, the generative models, do not require a dialogue database when providing responses. Instead, the replies are synthesized one word at a time, generating responses on the spur of the moment [13]. According to Cahn J. [8], one of the most recent models that generate an effective response is the Statistical Machine Translation, where key phrasal classifiers are identified based on real dialogues, allowing for multiple language translation. This model was first introduced by Ritter et al. in 2011 in a chatbot [14], and its benefit over human translation is mostly due to resource savings [8], [14]. A typical generative-based model is Sequence to Sequence (Seq2Seq), firstly introduced by Cho et al. in 2014 [15] as a variation of Ritter's [14] generative model that uses deep learning to achieve higher levels of accuracy[8]. In this Encoder-Decoder model, two Recurrent Neural Networks (RNN) are implemented, using a Long Short-Term Memory (LSTM) architecture. The first RNN performs the function of encoding the input sequence, and the second RNN decodes the same input, generating a reply. Figure 4 shows an example of an RNN introduced in Seq2Seq, where <EOL> is a symbol that delimits the end of the sentence.

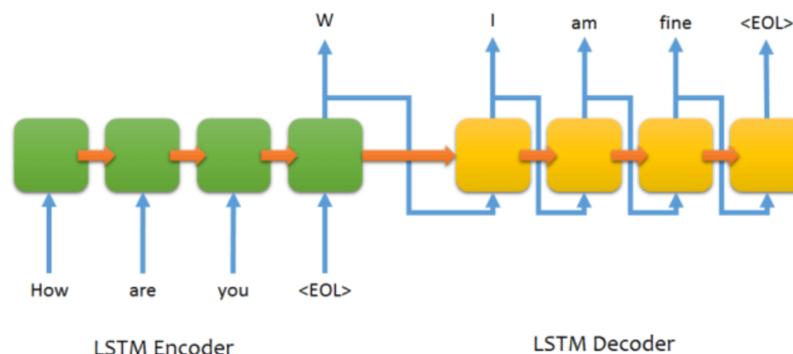

Figure 4: Example of Seq2Seq RNN model [8]

This model has proven to be effective and widely used, being considered as the industry's best practice for response generation [15]. However, it presents some drawbacks, as Cahn J. [8] refers that bots with this implementation fail to simulate human-to-human interaction, frequently responding to inputs with "I don't know" since this response is used quite often. Secondly, in a conversation between two Seq2Seq bots, the conversation also showed to get stuck in an infinite loop.

2.2.4 Dialogue Strategies

When a response is selected, the Dialogue Management Component must choose a set of strategies to create a response that, along with maintaining grammatical structure, must deliver a message that mimics human behavior and at the same time is understandable to the user. An interaction strategy is comprised of three approaches, which are determined by which party dominates the course of the conversation: the user, the system, or both. In the first approach, the user leads the dialogue, but a problem arises: if the user inputs an ambiguous response or an ill-defined input, the system will not know how to react. On the other hand, a system-led approach reduces the occurrence of ill-defined inputs but compromises dialogue flexibility. The latter method is defined by a mixed combination of both previous approaches. A second strategy consists of error handling and query confirmation, showcased in figure 5, providing a simple example that presents this strategy in motion, repeating important elements of the input [8]. By doing so, the user is able to verify if the original intent matches the response given by the bot.

User: "I want to go to New York."
System: "You want directions to New York?
This is time consuming, but ensures the correct response. With implicit confirmation, the system includes what it thinks it heard back to the user as part of the next question.
User: "I want to go to New York."
System: "What type of transportation do you want to use to get to New York?"

Figure 5: Example of error handling and confirmation [16]

Another set of strategies is Reinforcement Learning and Active Learning. These are quite different from the previous strategies, as the bot is continuously trained to find the optimal interaction and generate effective responses. They work with a scoring function according to the system's behavior. Additionally, in the Active Learning approach, a set of ranked responses are shown to the user, where he selects the preferred reply. In this case, the system learns in real-time. However, this breaks the natural flow of the dialogue and, if the user selects an incorrect option, the strategy may not prove to be efficient [8].

To some of the previous strategies, a set of ranked responses is applied, where its scoring function indicates the likelihood of the bot to consider that answer as appropriate. For responses with a lower score, Yu et al. [17] suggests a set of strategies, such as asking open questions, in order to elicit more information if the first input was not clear, providing an additional opportunity to continue the conversation.

In order to appear more human, some strategies can also be applied. Developing a personality and directing the conversations are some examples that ELIZA implemented, where giving a name and personality traits make the bot more human-like. Some other approaches are the ability to make small talk when non-task-oriented conversations are active, and the ability to commit human mistakes, such as replying with a disfluent message [18].

2.3 Available Technologies

2.3.1 Artifical Intelligence Bot Frameworks

Rasa

Rasa is an intuitive framework for developing personalized conversational AI. This platform introduces, similarly to other conversational systems, NLU (Rasa NLU), converting short user messages into dialogues that are comprised of intent and multiple entities, and Dialogue Management - Rasa Core [19]. Rasa's text classification is based on fastText approach, a library for efficient baseline for text classification [20]. There is no support for end-to-end learning, since some components such as NLU, Dialogue Management, and Response Generation are learned from dialogue transcripts, where the last two are fully decoupled, allowing for the same dialogue model to be reused across different languages [19].
Rasa's architecture follows a modular approach, allowing for easier integration with external systems, where Rasa Core can be used as a dialogue manager with multiple NLU services. This proves to be an advantage when we wish to create a generalized solution that can be configured to work in a specific domain. Figure 6 depicts the planned architecture, from the moment when the user inputs a message to the generated response:

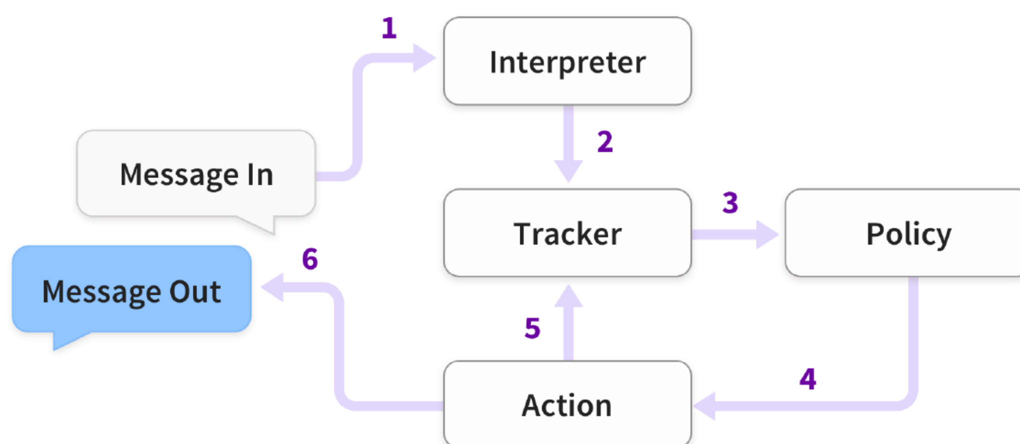

Figure 6: Rasa architecture [19]

In this architecture, a single dialogue state is saved in a tracker object per conversation, being the only stateful component. This tracker object works as a log file, registering all previous events. From figure 6, only the first step is executed by Rasa NLU or another NLU module, where the following steps are performed by Rasa Core. The Interpreter extracts the intent, entities, and structured information from the message, where the Tracker will save the state. Then, the Policy will determine the best course of action that will be replied to the user, where the Tracker will subsequently register the action. If the action was not carried out, the system returns to step 3 [19].

In the matter of training data formats, Rasa Core and NLU work with human-readable formats, presented in a JavaScript Object Notation (JSON) structure or a markdown format. Figure 7 presents two examples that showcase each format.

```
{
  "text": "show me chinese restaurants",
  "intent": "restaurant_search",
  "entities": [
    {
      "start": 8,
      "end": 15,
      "value": "chinese",
      "entity": "cuisine"
    }
  ]
}
```

(a) JSON format

```
## intent: restaurant_search
- show me [chinese](cuisine) restaurants
```

(b) Markdown format

Figure 7: Training data formats [19]

Regarding dialogue stories, Rasa used the same markdown format when implementing each story, denoting a sequence of events where entities are key-value pairs separated by commas, which are presented in figure 8:

```
## story_07715946
* greet
  - utter_ask_howcanhelp
* inform{"location":"rome","price":"cheap"}
  - utter_on_it
  - utter_ask_cuisine
* inform{"cuisine":"spanish"}
  - utter_ask_numpeople
* inform{"people":"six"}
  - action_ack_dosearch
```

Figure 8: Training dialogue - Stories [19]

Rasa Core also supports a machine teaching method, where the developers correct system-made actions. This is presented as a practical approach, but somewhat exhausting for systems at a larger scale.

```
------
Chat history:

    bot did:        []

    bot did:        action_listen

    user said:        /greet

        whose intent is:        greet

we currently have slots:  cuisine: None, people: None,
            price: None, location: None

------
The bot wants to [utter_ask_howcanhelp] due to the intent. Is this correct?

        1.        Yes
        2.        No, intent is right but the action is wrong
        3.        The intent is wrong
        0.        Export current conversations as stories and quit
>>
```

(a)

```
------
what is the next action for the bot?

        0                        action_listen      0.12
                  ...
        8                    utter_ask_cuisine      0.03
        9                   utter_ask_helpmore      0.03
       10                 utter_ask_howcanhelp      0.19
       11                  utter_ask_location       0.04
       12                utter_ask_moreupdates      0.03
       13                  utter_ask_numpeople      0.05
       14           action_search_restaurants       0.03
                  ...
>>
```

(b)

Figure 9: Rasa Core correction example, upon selecting the option labeled as number 2 in 9(a) [19]

When correcting an action, Rasa Core trains the data policy. Once the dialogue ends, the set of policies is updated, changing the probability values of the previous policies to determine the best course of action. Rasa also offers the opportunity to gather analytics and observe the bot's performance.

Microsoft Bot Framework

Microsoft Bot Framework is a developer environment for designing conversational AI experiences. Along with Azure Bot Service, which is Microsoft's AI chatbot, it allows developers to create enterprise-grade bots that perform regular chatbot functions, such as the ability to communicate through speech and understand natural language through inputs. It includes a modular and extensible Software Development Kit (SDK) for building custom bots, templates and related AI services. Figure 10 showcases a clear view of the components that constitutes the Bot Framework's architecture, where a description of each each component is presented shortly after.

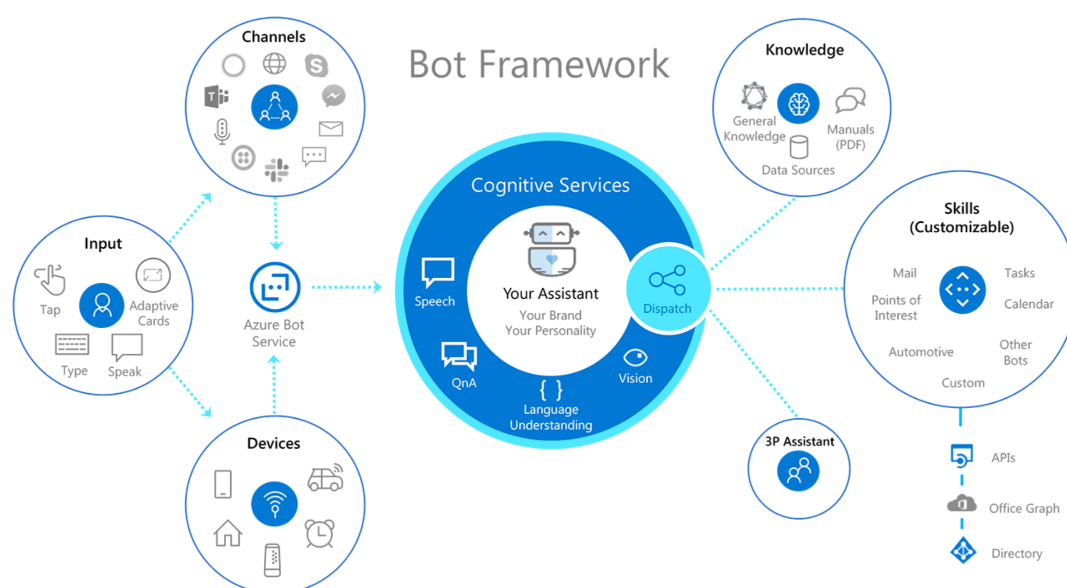

Figure 10: Microsoft Bot Framework architecture [21]

- Input - Comprised by the various forms that the user can interact with the system.
- Channels - Connection between communication services and a bot. The Bot Framework allows to create a bot in a channel-agnostic way, by normalizing messages that are sent to channels. It converts the messages from the Bot Framework schema to the specified channel's schema. If the channel does not support the Bot Framework's schema, the service tries to change the format in a way that is supported by both parties. For most channels, channel configuration information is required in order to run a bot. Additionally, a bot account is needed in most channels [22].
- Devices - Physical mediator between the user and the bot. The user may use it to introduce an input, or the bot may perform an action on it.
- Cognitive Services - AI capabilities that enhance the bot's overall performance, or give it additional features.
- 3P Assistant - As the name suggests, third party (3P) assistants are external assistants outside the Bot Framework environment, but can interact with it.

- Knowledge - Related to sources of information and external resources.
- Skills - A skill is a bot that can perform a set of tasks for another bot, where a bot can be both a skill and a user-facing bot. A skill consumer is defined as a bot that can call one or more skills, and a skill manifest is a JSON file that describes the skill's actions, input and output parameters, and endpoints [22].

Google Dialogflow

Dialogflow is Google's product for creating chatbot solutions. This platform is available in multiple editions, with some crucial attributes varying between them. It offers pre-built agents to cut down on development time, with a framework that takes a simplistic approach, being beginner friendly. It also contains multi-turn and multiple language support [23]. Figure 11 showcases the steps that take place in a conversation using Dialogflow:

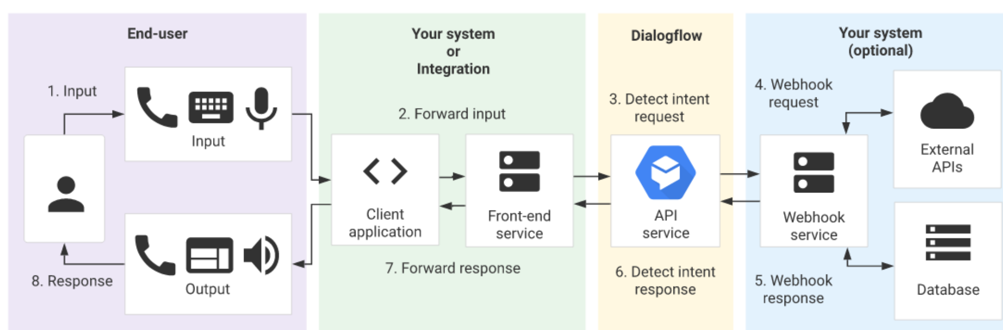

Figure 11: Dialogflow conversation example [23]

1. The user sends a message via the input, whether in text, speech, or any other means of communication.
2. The user interface then receives the information and forwards it to the Dialogflow API in a detect intent request.
3. The Dialogflow API receives the request. If the input matches to an intent or form parameter, session state is updated, as certain parameters are required. For a webhook-enabled parameter, it sends a webhook request to its service, otherwise jump to step 6.
4. The webhook service calls for external APIs or other external resources in order to satisfy the parameters.
5. The response is sent back to Dialogflow.
6. A detect intent message is created, sending the intent response to the user interface or integration system.
7. The user interface receives the response and forwards it to the end-user.
8. The message is sent to an output device [23].

Other alternatives

Botpress is an open-source platform for developers to create digital assistants. It is cloud-agnostic, meaning that the developer may run it on his cloud service of choice, and it supports more than 100

languages. This platform includes NLP tasks, a visual conversation studio to design multi-turn conversations, and provides an SDK. Additionally, it also includes tools to gather and analyze data, and a Human In The Loop feature that allows to delegate a chat session to a human [24].

Amazon Lex is Amazon's take on AI solutions, considered as a service that allows developers to create text or voice-based chatbots. It is powered by Amazon's deep learning, NLU and speech recognition that is used on other Amazon solutions, such as Alexa. However, Lex differs from Alexa since it can be integrated into services and devices. It can be used through Amazon Web Services-SDK, which means that developers can create a system that posts messages to an API and get a response from Lex [25].

2.4 Devised Solution

Upon reviewing the design and development of intelligent chatbots, a decision had to be made when selecting the appropriate technologies. All different technologies shown in 2.3 aim to provide a platform that allows for creating a bot, accessible through interfaces, where questions asked to the bot usually rely on third-party systems or external resources in order to respond accurately. This results in a seamless and good quality of service. In general terms, the expected project of this dissertation does not want to follow a black-box approach, where most of the information is concealed to the client, but rather the opposite. Therefore, the devised solution must be user and beginner-friendly, in order to accommodate the client's objectives when configuring the system, but at the same time, it must provide extended documentation and freedom for developers to create the desired product without any restraints.
Despite offering an excellent framework, as it provides deep learning functionalities, Amazon Lex only supports the English Language and United States Spanish. Another drawback is the fact that data preparation is somewhat complex. This is a major restriction, as the envisioned project aims to support multiple languages. Botpress offers a modular approach for building chatbots, but it contains a limited amount of features and has a hard time learning from user experience since it is mainly rule-based. Contrarily to Amazon Lex, Google's Dialogflow supports a wide amount of languages and offers a set of pre-built agents in order to facilitate the construction of a chatbot. However, it only supports one webhook per project, which hinders the solution if the consumer wishes to obtain data from various sources in real-time. Additionally, it does not provide live customer service, which can be a useful feature, since a human agent can intervene as a last-resort method, in case the user is not able to work with the software. Microsoft Bot Framework presents thorough and extended documentation and already integrates various systems that work within the Microsoft ecosystem. Despite this large integration, most of these functionalities are valuable if the final product is intended to function inside the same ecosystem.
Finally, Rasa presents a framework for building a contextual chatbot. One of Rasa's drawbacks is that is not very beginner-friendly, as it requires prior knowledge of NLU. However, a contextual chatbot approach makes the bot appear more sophisticated and intelligent, as it portrays human-like behavior. The presented framework is also highly customizable, which gives freedom to developers when creating chatbots with specific features.
Upon discussing these alternatives with IPBRICK, the chosen platform for the development of this project was Rasa, since it portrays the technology to provide several virtual assistants that differ in their autonomous levels, being aplicable to every use case.

3 IPBRICK Group

3.1 History

IPBRICK is a Portuguese software company that provides Corporate Communications, On-Premises, and Private Cloud solutions. Its genesis was in the year 2000, at a time when Linux OS was rising in popularity, as an answer to multiple focus points that were settled on Open Source solutions. While effective and customizable, the Linux experience can find itself to be somewhat daunting to non-tech-savvy individuals and corporations. Bearing that in mind, IPBRICK carried out a mission to provide simplified and accessible Information Technology experiences to both companies and public organizations. In 2002, IPBRICK's first product, iPortalDoc, a Document and Process Management system, is introduced to the market, and three years later, due to several requests from external markets, it began its globalization process. Then, in 2008, IPBRICK developed its own telephony subsystem by launching Unified Communications over Internet Protocol (UCoIP). Ahead of its time, the company predicted the replacement of traditional telephony lines, adopting the usage of Internet-connected lines as part of their telecommunication infrastructure, resulting in every communication to function over the Internet.
IPBRICK aims to be a key reference in Single-Tenant solutions, where companies with access to these services have full control over data, such as processed phone calls, e-mails, and documents. Unlike a Multi-Tenant solution, this approach grants more security, customization of services, and overall availability. This company also constructed a well-founded network with partners, capable of commercializing and implementing IPBRICK solutions, along with specialized teams that dedicate their focus to assisting and implementing these solutions on end clients.
Over the years, IPBRICK developed additional products, IPBRICK MAIL and IPBRICK CAFE, centering their activity into four major areas, known as the IPBRICK MAGIC WALL. These areas of development are housed in one single service, the IPBRICK OS.

3.2 IPBRICK Operating System

The IPBRICK OS is a Communications Platform directly aimed at businesses, based on Linux Debian. It is comprised of the four major solutions that make up the IPBRICK MAGIC WALL, featuring a management interface, enabling the user to perform system configurations that best fit their use case without manually running complex commands through the command-line interface. Moreover, they are able to install additional software modules and security updates directly through public IPBRICK repositories, granting additional security and stability throughout the whole system.

3.2.1 iPortalDoc

iPortalDoc is the only Document Management solution that natively integrates with a Unified Communications Center. By operating on a Private Cloud and On-premises, it creates an enclosed system where data is available throughout the network, allowing users to insert documents or trigger workflows by sending e-mails and associating documents with other users they relate. Additionally, it is also prepared to save Calls, E-mails, and Chat Conversations in the Document Management system, making information available for consultation at any time [26].

3.2.2 IPBRICK MAIL

The IPBRICK MAIL - Email and Collaborative Tools solution - provides secure and effective management of each user's e-mail client, along with each user's personal Calendar, Notes, Contacts, and other features. It is designed to be easily accessible on all platforms, whether it is through a Web Browser, Mobile Applications, or Desktop, ensuring a collaborative business environment. It is fully integrated with the Document and Process Management solution, since it allows to archive, classify and associate e-mails with documents and processes within the iPortalDoc environment [26].

3.2.3 IPBRICK UCoIP

IPBRICK UCoIP is a Unified Communications over Internet Protocol center that operates as a telephony subsystem, functioning as a private Internet Protocol Centrex solution. Running inside IPBRICK's ecosystem, it merges all forms of communication, whether it is through voice, video, e-mail, or professional chat, onto a single point of reference, using an e-mail address as a common identifier through all communication channels. Following a single-tenant architecture, each company is presented with its own independent system, creating a personalized and secure network. Additionally, a dedicated UCoIP web page for each user is provided, showcasing all forms of communication, for any individual outside the organization that wishes to contact a user within the IPBRICK network [26].

3.2.4 IPBRICK CAFE

IPBRICK CAFE is a digital workspace that aims to create a dynamic environment between co-workers, providing tools designed to simplify communication, as well as exchanging data with one another. This application is comprised of two major areas, Applications, and Communications, allowing users to access resources and applications in a centralized manner [26].

4 Solution Planning and Structure

4.1 Background on IPBRICK solutions

4.1.1 Asterisk

Asterisk is a software framework, conceived in 1999 by Mark Spencer, that converts an ordinary computer into a communication server. It is used to build real-time communications applications by providing a phone system that integrates multiple communication methods - Unified Communications - as a key concept [27].
Running on several operating systems, such as Microsoft Windows, Linux, and OS X, its flexibility and customization are complemented by Asterisk's free and open-source nature, which assisted on its implementation at a worldwide scale [28].

In the pretended context, Asterisk functions as the IPBRICK's private phone system, allowing for users within the network to connect with one another, as well as for users outside the network to dial several external incoming telephone numbers, that are used for specific purposes. Some of its underlying features, such as call queuing, music on hold, and the Asterisk Gateway Interface are already used in the IPBRICK's VoIP services, named UCoIP, which hold a fundamental role in the proposed voicebot solution.

4.1.2 Google Cloud Platform

The Google Cloud Platform is a collection of cloud computing services, consisting of both physical and virtual resources that are established inside Google's infrastructure. These resources are distributed through data centers dispersed across the world, providing some benefits such as redundancy in case of failure and reduced latency, since the users can choose the resources that are located within their region [29].
Alongside the previously mentioned benefits, additional advantages of using Cloud services are high availability to the end-user, considering that data can be easily replicated along all regions; as well as scalability and elasticity, since an application requires adaptable resources, from small to heavy loads, by provisioning the number of necessary servers to fulfill the task in hand [30].

Google Cloud Platform showcases a plethora of modular cloud services, such as Serverless Computing and Internet of Things where, in the context of this work, the product that emerges as being an essential part of the voicebot is the AI and Machine Learning module, more specifically the Speech-to-Text (STT) and Text-to-Speech (TTS) APIs. As the name suggests, Speech-to-Text tries to accurately transcribe any input given by the user through voice, generally through a microphone, converting speech into text. On the other hand, Text-to-Speech uses Google's AI technologies to provide a natural-sounding response by converting any given text into an audio output [29]. By combining these two APIs, a bot can actively listen to the person's intent and supply an audible response, delivering a lifelike experience.

4.1.3 ejabberd

The ejabberd software is a free and open-source messaging server, written in the Erlang programming language. It runs on all three major platforms, Microsoft Windows, Linux, and OS X, where its system architecture emphasizes being fault-tolerant and distributed, by allowing the user to deploy a cluster of machines that serve the same domain, that can be easily added or replaced, where the information is replicated throughout the network to ensure a proper working service whenever a node fails. It is able to support multiple communication protocols, such as Messaging Queuing Telemetry Transport (MQTT), Extensible Messaging and Presence Protocol (XMPP), and Session Initiation Protocol (SIP).
The first protocol, MQTT, primarily designed for the Internet of Things, relies on a publish/subscribe method for communication, where a client would effectively publish messages to an MQTT broker,

which is subscribed by other clients, following a one-to-one, one-to-many or many-to-one connection mechanisms. Secondly, XMPP finds its effectiveness on text messaging applications, using both publish/subscribe and request/response methods, where XMPP clients and servers communicate by exchanging data in the form of Extensible Markup Language (XML) stanzas. In addition, one of the key features of this protocol is security, excelling in creating secure messaging systems [31]. Whenever a user sends or receives a message, a standard XMPP connection lifecycle begins by initiating a Transmission Control Protocol (TCP) connection to the server, where a stream header is sent and authentication mechanisms are validated. Then, the client binds to a resource, sending its presence to the server, and requesting its contact list, or roster, thus being able to send and receive messages [32]. Lastly, the Session Initiation Protocol is an application-layer protocol designed to establish, modify, and terminate multimedia sessions, such as Internet telephony calls [33].
The ejabberd is prized not only for its versatility, due to its multi-protocol support, but also for its scalability and modularity, since its architecture allows the application to be deployed in both small and big enterprises, and the fact that its server code can be extended through powerful APIs, connecting to third-party extensions and web clients such as Hypertext Transfer Protocol (HTTP) Binding - Bidirectional-streams Over Synchronous HTTP- services [34] [35].
In the context of this work, an ejabberd server is already implemented within the IPBRICK system, enabling all users inside the IPBRICK network to communicate with one another, but will be enhanced to jointly cooperate with the chatbot interface. In these circumstances, when a user intends to communicate to a person instead of the bot, all messages will be forwarded to an IPBRICK account that serves the purpose of an assistant.

4.1.4 KoolReport

KoolReport is an open-source reporting framework programmed in Hypertext Preprocessor (PHP) language. It contains a large quantity of data visualization tools, all while giving the user full control of its data. In addition, multiple data sources are supported, from Structured Query Language (SQL) Databases to Microsoft Excel, and various features are available to extend the KoolReport's functionalities [36].
In this work, this tool will be used when showcasing the results of the overall bot performance, working in conjunction with a PostgreSQL database containing detailed information, describing the interaction between users and the bots, differing from both voice and chat, displaying all data in a dashboard-like concept that seeks to evaluate if the designed bots have proven to be successful. If the results show otherwise, it should indicate how and when the bot is failing, along with the Rasa X platform, enabling its further development.

4.2 Solution integration in IPBRICK OS

Upon reviewing the design of intelligent chatbots, a thorough analysis of IPBRICK's architecture was made, assessing how every system comes into play. When devising a solution to a fully-established environment, the understanding of its composition, as well as the comprehension of the assignment is key to a successful implementation. During this development phase, an isolated IPBRICK environment was provided, so that the OS could be analyzed and tested without compromising security and the workflow of active systems.
Considering the evaluated communication flows in the context of this work, figure 12 showcases a simplified version of how a standard bot interaction unfolds:

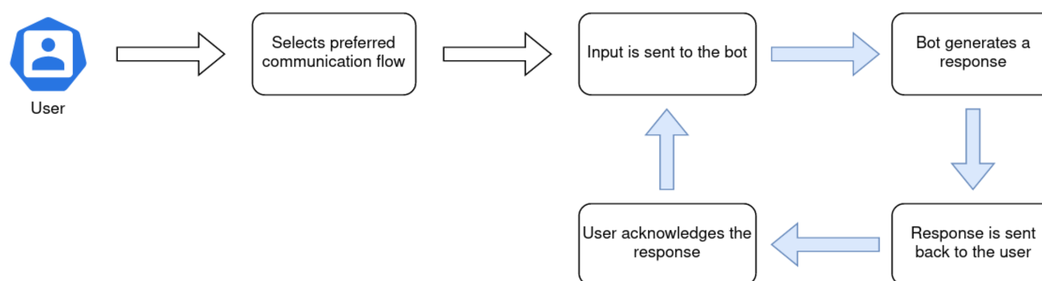

Figure 12: Simplified human-to-bot communication flow

In a regular interaction between two individuals, person A starts out by disclosing their preferred method of communication, to which then will transmit their intention, in an understandable format, towards the recipient, person B. Consequently, if the perceived information is in the right format, and the intent is recognized, person B will formulate a response, communicating it back to its original sender, person A. This ongoing, cyclical interaction between multiple individuals will occur until a goal has been reached.

In an analogous way to a standard interpersonal communication, one must first have the main intent, or goal, to a conversation, before developing a "story" with the target individual. In this case, the purpose of this said story is to report faults that occurred in the client's machines to the IPBRICK services. Then, after the main objective of the conversation has been set, one must first develop the required interfaces to transmit, for example, the information from person A, the client, to person B, the AI engine, and vice-versa. The IPBRICK services provide an interface to communicate via voice, namely the CAFE. Fully integrated with UCoIP, some advancements in this matter have been readjusted for bot interaction. However, despite having an additional instant-messaging sub-system, there is no dedicated interface for human-to-bot interaction. Moreover, just like in a regular conversation, every exchange of information is prone to misunderstandings and interruptions, so it is up to the listener to perform additional enquiries to better recognize the initial intent, in the first case, or be able to switch the original context in order to satisfy sudden demands in the latter. In order to understand these behaviors and react accordingly, the bot must be properly configured for this specific closed domain, as well as handle these interruptions in a natural way. In these circumstances, one interruption that later comes as a feature is the ability to transfer a conversation to a human employee, regardless of communication flow, if the bot results in failure due to intent incomprehension, errors, or simply if the user wishes to speak to a human operator instead. Given the unpredictability of the human mind, it is unfeasible to cover all possible scenarios that result in failure. Therefore, the bot's framework offers a solution to diminish these events, which are further explored in 4.3. Once the response is processed, it must be forwarded to its original sender, in the chosen platform, maintaining the dialogue.

IPBRICK OS Setup

After describing how a basic human-to-bot communication unfolds, the following step is to prepare the necessary arrangements in the IPBRICK OS web interface. Figure 13 depicts an example of the interface of the IPBRICK system software that is distributed to all clients and partners. Considering the opportunity to work alongside the IPBRICK team, all developments were carried out in the newest, unreleased version of this OS.

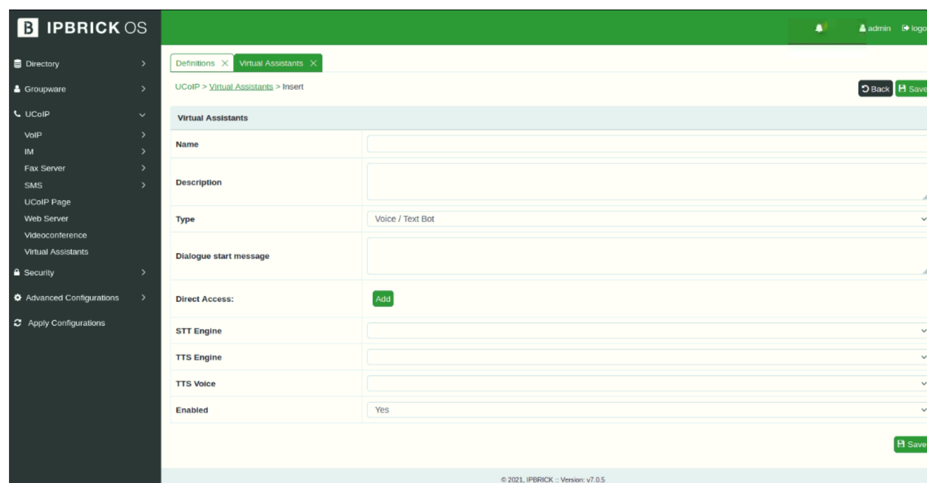

Figure 13: IPBRICK OS Web Interface

Given the nature of this work, most of the required configurations for the designated components were found under ”UCoIP”. In this tab, the procedures were as follows:

- For the voicebot component, **two Text-to-Speech engines** were configured, specified by language and gender.
- In addition, **two virtual assistants** were defined, each having their own SIP address.
- In order to implement the requested multi-lingual support, **two** default **Rasa projects** were created, for Portuguese and English languages.
- Given that the **chat interface** is yet to be implemented, two separate **web pages** were created for hosting the chatbot text component, one per language.
- An additional **web page** is intended for the implementation of the bot **dashboard**, showcasing measured bot performance and statistics.

Inside this closed environment, an extra user was defined, serving the purpose of an operational agent, when a given hand-off action occurs. Afterwards, in a live ecosystem, this user is naturally replaced with a real operator inside the IPBRICK services.
When reporting issues with the IPTicket reporting feature, entity validation is required prior to using this service. Therefore, along with the configuration of the IPTicket service, which consists of a reporting feature where users can issue faults that occurred in their systems, several fictional entities were created in IPBRICK Contacts - a dedicated space to define all types of entities, including customers, partners, and employees. This will be used in tandem with IPTickets, as it requires entity tax identification number for authentication measures.
With all system configurations complete, the following step is dedicated to constructing the bot within the Rasa Framework, that will interact with these external services, creating a functional and autonomous reporting feature.

4.3 Rasa Framework

4.3.1 Core purpose

When defining a virtual assistant in the IPBRICK OS, a new Rasa project is generated with default configurations, ensuring that each project is readily available to be implemented in any environment, where the automation of several tasks is requested. On the subject of this work, a commercial support bot was developed, allowing users to report faults that occurred on their systems. These faults can be differentiated into two types, depending on the associated services. For Document Management issues, these fall under the category of iPortalDoc, and the remainder, whether CAFE, UCoIP, or MAIL related, are listed as IPBRICK problems.
Apart from this reporting feature, IPBRICK also wants to provide additional alternatives to users, including multiple language support. To explore this concept, the assistants must support both English and the Portuguese language, in order to accommodate a larger user-base. Despite this bot solution being fully autonomous, IPBRICK requires a safety measure to be implemented, where human action intervenes in active chats. Hence, a human hand-off feature is to be developed, granting the user the ability to talk to a human operator when requested, regardless of communication type, or when this bot fails to deliver accurate responses, throughout the course of the conversation.
After outlining all major features, an analysis of the Rasa Framework is made, uncovering its architecture and procedures involved in constructing this bot. To best describe this system, the official Rasa website presents well-built and structured documentation.

4.3.2 Architecture

The Rasa Open Source architecture is designed with scalability in mind, where its modular approach allows for multiple technologies, fully integrated with this system, to replace the standard software in existing modules, accommodating the target requirements. Although highly configurable, the default Rasa configurations should suffice most services that desire a deployable and available on-the-go AI product. Figure 14 carefully exposes each module, highlighting its core components.

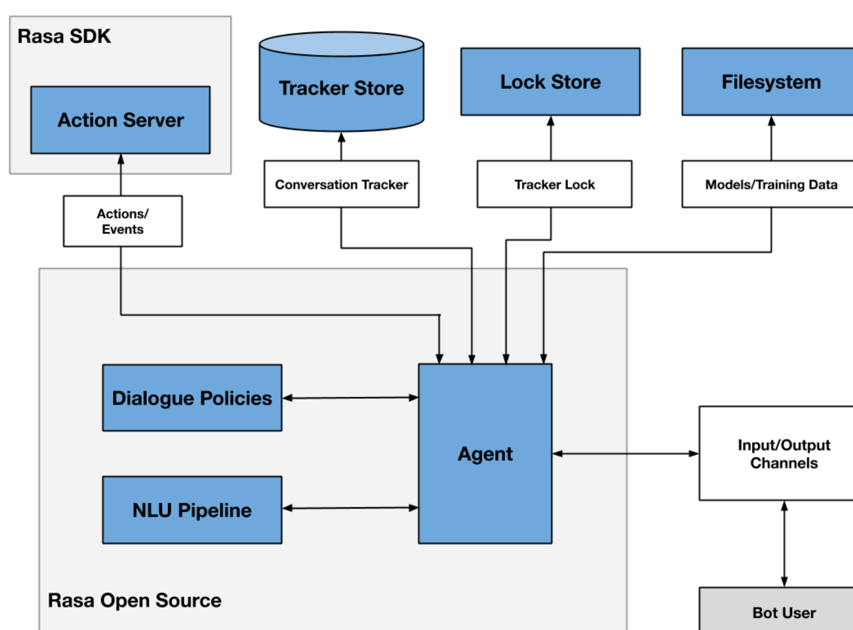

Figure 14: Rasa Open Source architectural diagram

From the previous diagram, there is a noticeable distinction from every module, where the most vital components are the NLU and Dialogue Management. After an input is sent from the bot user, the NLU, shown as the NLU Pipeline, is responsible for processing the user's utterance, gathering the intent behind the message, and extracting entities. Following a prior analogy of a regular conversation between two individuals, the intent is defined as the objective that person A is trying to convey or accomplish to person B, where entities are structural pieces of information that can be extracted in the user's message. This combination of entities and intents helps retrieve the context of the utterance accurately. Then, based on the current context of the conversation, the Dialogue Management, displayed as Dialogue Policies in the diagram, decides the best course of action, selecting the most appropriate response to be sent back to the bot user, continuing the conversation.

Although the rest of the modules are not considered as primary components, they certainly play an essential role in the Agent, or bot, as a whole. In the same way that we keep track of past conversations with other individuals, the Tracker Store is a modular component that focuses on mimicking that same behavior, saving the assistant's past conversations in the default *InMemoryTrackerStore*, storing the conversation history in memory.
When dealing with circumstances that involve multiple people talking to one individual at the same time, one may have trouble responding, depending on the number of people that initiated a conversation, the complexity of the utterance, and the actual order of responses that one individual is going to answer. Being just one individual, he is unable to respond to every single question simultaneously, finding himself physically limited to only provide one answer at a time. Rasa addressed this kind of parallelism using the default Lock Store configuration, *InMemoryLockStore*. This synchronization mechanism uses the concept of ticket lock, ensuring that each active conversation is given an identifier, so that these can be processed in the right order, locking the remaining conversations until their turn arrives. For small solutions where the demand is not high, this does not arise as an issue. In larger projects, server load is expected to increase, as there are too many incoming messages, resulting in a slower response time since these wait longer periods for their turn to be processed. With that in mind, multiple Rasa servers can be run in parallel as a replicated service, effectively balancing the load across this network, ensuring a faster service.
When developing an AI project, the bot is subject to multiple tests and implementations. As additional requirements arise, its adaptability must be maintained, and consequently need to be trained under recent circumstances, constantly updating the model to a newer version. Some of these tests are bound to occasional errors and mistakes, forcing the user to roll back to a previous version, in order to mitigate harsh consequences. Simply put, the FileSystem component is where all models, or versions, are placed, after the virtual assistant is trained.
In addition to standard outputs provided by the bot, the Rasa SDK Action Server offers additional customization when creating responses, called custom actions. Apart from the traditional responses, these custom actions can be used to access external services, such as querying a database or calling an API. When the bot predicts that a custom action will be called, an HTTP POST request with a JSON payload is sent to the action server, containing key data that helps execute the requested action, as well as identifying the user and the universe (or domain) that the bot is established. After running the custom action, a JSON payload is sent back to the Rasa server with responses and events. Afterwards, the Rasa server returns the same set of responses back to the user, adding the events to the conversation tracker [37].
By providing a modular approach, these components can be replaced with supporting technologies that best adapt to the overall scalability of each project, maintaining the same architectural structure, without configuring additional software for that sole purpose. In the context of this work, the default architectural configurations were used, since this is meant to serve as an example of a service that is provided to several clients, so simplicity and availability are key factors.

Before developing the commercial support AI, some research was made in figuring out the best practices when implementing a bot. Rasa strongly suggests the Conversation-Driven Development (CDD) approach. This is the process of analyzing interactions that occurred between users and the bot, adjusting the bot's actions to its intended behavior, effectively training the bot with real user data. Given the complexity and unpredictability of human language, every individual will approach each bot in a unique way, being impossible for the developer to anticipate every interaction. Therefore, listening to the user's insights and subsequently orienting the assistant towards real user language and behavior is a great approach to building a bot. For a proper virtual assistant design, CDD must be applied and often reviewed in every step of the process, from early stages in development to its implementation in production, consisting of the following actions:

- **Share** the assistant with users at the earliest oportunity;
- **Review** conversations frequently;
- **Annotate** user expressions and messages, using them as NLU training data;
- **Test** that the assistant always behaves accordingly;
- **Track** its failures and measure the bot's performance over time;
- **Fix** unsuccessful behaviors during conversations.

To accompany the bot's performance, Rasa developed its own CDD tool, Rasa X, allowing developers to carefully review every conversation, successfully training the assistant over real user data [37].
Similar to every other approach, CDD is no silver bullet when perfecting an AI solution, also presenting its downsides. It relies heavily on manpower, since human intervention is required to assess every existing conversation, resulting in a poor analysis when this resource is scarce, hindering performance and the overall bot improvement over time when faced against a large quantity of conversations yet to be reviewed. Additionally, for the circumstances where the developer wishes to observe specific edge cases with real user data, many regular conversations are firstly reviewed, as these occur more frequently, before addressing this issue, circling back to the same actions repeatedly. However, implementing human intervention in the process of correcting machine-made errors ensures a more lifelike assistant, guaranteeing constant instruction directly from the target audience.

Project Structure

When creating a new Rasa assistant using version 2.8.7, its default project follows the structure presented in figure 15.

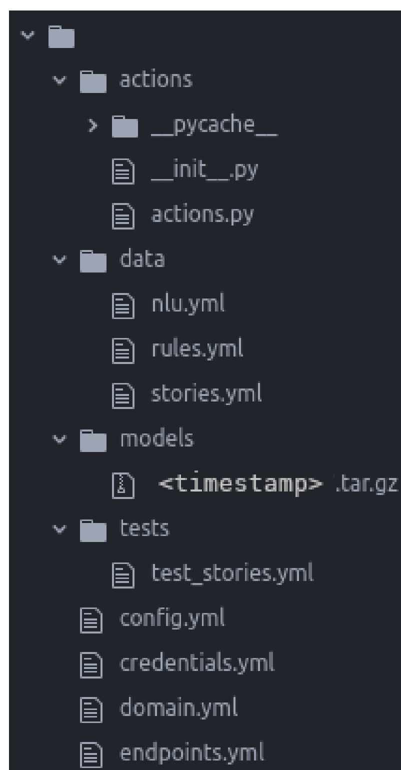

Figure 15: Rasa project file structure

Rasa Open Source uses Yet Another Markup Language (YAML), a human-readable data serialization language, across all files. This stands as a unified and extendable way of managing data, allowing developers to fully re-arrange the file structure, splitting training data over any number of YAML files. Some comparisons can be immediately drawn with Rasa architecture in figure 14, where some components can be identified.

- The **actions.py** file, written in the Python programming language, holds all custom actions written by developers, consequently executed under the Rasa Action Server.
- The data folder contains **nlu.yml**, **rules.yml** and **stories.yml** files. NLU Training data is comprised of example utterances that are classified by intent, including a list of synonyms, regular expressions, and lookup tables that facilitate intent and entity identification. Stories and rules represent conversations between the bot and users, functioning in tandem to train the Dialogue Management component. The foremost describes small conversations that should always follow a rigid path, whereas the latter contains lengthier conversations, each one is described as a story, listing the steps that the user and the assistant took during each interaction. Stories are used to train a machine learning model, identifying conversational patterns, as well as generalizing and predicting unseen conversational paths.

```
nlu:
- intent: affirm
  examples: |
    - yes
    - y
    - indeed
    - of course
    - that sounds good
    - correct

- intent: deny
  examples: |
    - no
    - n
    - never
    - I don't think so
    - don't like that
    - no way
    - not really
```

(a) Rasa NLU example

```
stories:

- story: happy path
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_great
  - action: utter_happy

- story: sad path 1
  steps:
  - intent: greet
  - action: utter_greet
  - intent: mood_unhappy
  - action: utter_cheer_up
  - action: utter_did_that_help
  - intent: affirm
  - action: utter_happy
```

(b) Rasa Story example

Figure 16: Default Rasa project NLU and Stories configuration

- Analogously to the Filesystem component, the **models directory** holds all trained versions of the bot, specified by the date and time of when the assistant was trained. All versions are compiled and stored in the same folder by default, switching to older versions when needed.

- Before implementing recent models into production, these versions should be properly tested, in order to confidently cover all scenarios without exposing an untested bot to the public. Therefore, **test_stories.yml** follows a similar structure to stories.yml, written in a modified story format, introducing entire conversations in order to test that, given certain user input, the model will behave as expected.

- The configuration file, **config.yml**, specifies the components and policies that are used to make predictions based on user input. The components are configured by the language and pipeline keys, working sequentially in order to make NLU predictions, whereas policies key is used by the model to determine the next course of action. These can be finely tuned or replaced with different models and policies in order to efficiently portray accurate results without compromising performance.

- The **credentials.yml** file is focused on configuring the credentials of dedicated communication channels to where the assistant will send and receive messages. By default, a Representational State Transfer (REST) channel is pre-configured, providing a REST endpoint where the user can interact with the assistant via HTTP POST requests, where Rasa Open Source returns a JSON body of bot responses.

- The **domain.yml** file describes the universe of the assistant, specifying all entities, intents, actions, and a set of all possible responses that the assistant may apply during an interaction. In addition, other means of extracting and processing user information are also defined, such as slots and forms, along with conversation sessions configuration. Within the system, slots are considered as the assistant's memory, storing data provided from the user or external sources, whereas forms are considered as a special way of extracting information, similarly to filling out a questionnaire.

- The **endpoints.yml** configuration file contains, as the name suggests, the different endpoints that the assistant is connected to. Rasa's modular architecture allows for their default systems to be used interchangeably with external services, as long as these are supported and properly identified in this file. Rasa Open Source's standard configuration utilizes system resources of the machine that this solution is hosted in, so by default these configurations are only set to use internal resources [37].

Standard Rasa interaction

After an analysis of Rasa's architecture and file structure, the following process focuses on observing standard human-to-bot interactions. This framework operates through the Command Line Interface (CLI), where all tasks are performed by executing specific commands. Rasa's default project comes with a pre-trained model, containing default configurations, NLU data, and stories.
From early development to production, the process of building a bot first starts by building an initial project using *rasa init* command, followed by modifying its core behavior, configuring its domain, adding responses, rules, and NLU data. With every major iteration in the adjustment of this system, a new model must be created, training the assistant under these new circumstances with *rasa train*. The assistant is then tested through conversations, covering edge cases, and introducing new expressions, before loading the updated model in a server with *rasa run*, placing the bot in production. Subsequently, the bot is subject to continuous testing and development, always following CDD methodology.
Through the CLI, several commands are available, including creating a visual representation of stories, running the action server, and testing Rasa models without requiring a separate graphical interface. One particular example is *rasa interactive*, a command that, after loading a model, starts an interactive session with the assistant, where the developer is able to accompany each step of the conversation, gathering new training data and evaluating its performance throughout this process. A simple interaction using *rasa interactive* is showcased in figure 17:

```
Chat History

 #    Bot                                You
 ────────────────────────────────────────────────
 1    action_listen
 ────────────────────────────────────────────────
 2                                        hello
                               intent: greet 1.00
 ────────────────────────────────────────────────
 3    utter_greet 1.00
      Hey! How are you?

Current slots:
        session_started_metadata: None

------
? The bot wants to run 'action_listen', correct?  Yes
? Your input -> I am doing great!
? Your NLU model classified 'I am doing great!' with intent 'mood_great' and there are no entities, is this correct?  Yes
------
```

Figure 17: Rasa interactive

For every session initiated with the bot, *action_session_start* marks the beginning of a conversation between the assistant and the user, followed by an action that awaits for user input, *action_listen*. The action *action_session_start* also reoccurs when the user fails to respond within a determined period of time, restarting the whole conversation under a different identification number. This is presented as the default behavior for all assistants. From this point forward, a regular interaction occurs, where sent messages are processed, the intent of these messages is collected, and a response is given.

Although the CLI is an effective way of testing the virtual assistant, some of its users may find it confusing and difficult, especially if technical background is lacking. Therefore, Rasa also provides its users with a CDD tool, Rasa X, containing a simple, browser-based Graphical User Interface. It performs the same as if one were using the CLI, but with a clearer interface, including observing past conversations, chatting with the bot, and correcting its mistakes. Executing the command *rasa x* in the CLI launches the interface in the web-browser, presented in figure 18.

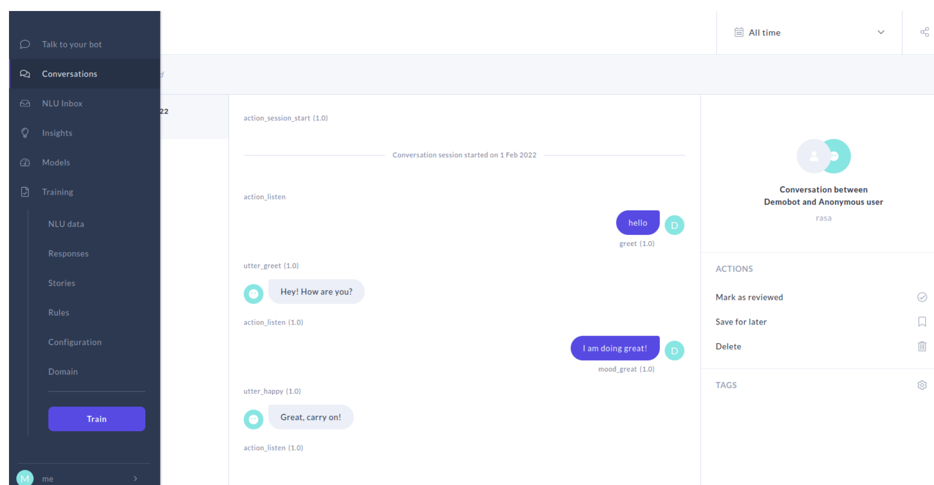

Figure 18: Rasa X interface

Rasa also provides an alternative way of interacting with its users, through endpoints. After initiating a Rasa server, the virtual assistant is able to communicate via dedicated channels, thus requiring a separate interface in order to communicate with these mechanisms. By default, a REST channel is configured, where users can send messages via HTTP POST method to:

*http://<host>:<port>/webhooks/***rest***/webhook*

Using this Uniform Resource Locator (URL), users can post messages with a payload formatted in JSON, receiving responses directly or asynchronously through webhooks, consequently receiving a JSON body of responses.

```
* REST channel input format
* POST message sent to http://<host>:<port>/webhooks/rest/webhook

{
  "sender": "test_user",
  "message": "Hello!"
}


* REST channel output format
[
    {
        "recipient_id": "test_user",
        "text": "Hey! How are you?"
    }
]
```

Figure 19: Input and Output response from REST endpoint

Two key values are sent, *sender* and *message*, identifying the current user, through a procedurally generated identification number, and the content of his message. Communication through endpoints is

presented as an essential feature for this work since there is a need to develop additional interfaces to interact with the bot. Both communications must adopt this format, in order to send messages and receive formulated responses. Once the assistant has interacted with multiple users, its behavior is reviewed under the Rasa X platform, observing past conversations and carefully correcting its mistakes.

4.3.3 IPBRICK Commercial Support Bot

After outlining Rasa's default project and analyzing a standard human-to-bot interaction, the process of developing an AI solution was set in motion. Given the requested implementation in 4.3.1, a diagram is presented in figure 20, indicating all major events in the proposed story that make up the submission of an IPTicket, independently of communication flow.

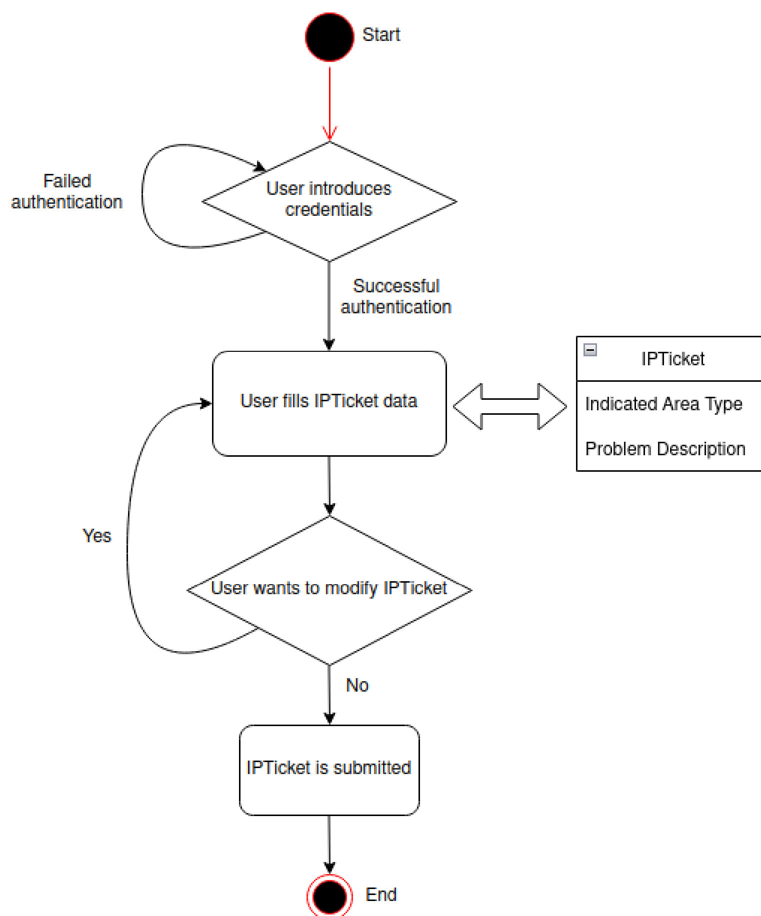

Figure 20: Diagram of proposed AI solution

When comparing the indicated solution with the previous diagram, some additional remarks are immediately recognized. Prior to filling out a form that makes up the IPTicket, a necessary authentication measure is required, in order to properly identify the client that is reporting the issue, as well as preventing unwanted users to utilize these services.

Upon completing the IPTicket form, one must be prepared for occasional human errors, transmitting incorrect data to the assistant, such as typographical errors or interruptions during speech, resulting in an erroneously filled form. Therefore, an opportunity to perform adjustments to the ticket must be given, before the said ticket is submitted, whether the user selected the wrong area type, failed to provide a complete description of the problem, or supplied an incorrect description. Implementing these safety measures reduces the amount of incorrect IPTickets in the system, as these would result in multiple reports that failed to address the same issue, since additional attempts would be necessary to submit a correct ticket, increasing system load.

Despite not being present in the diagram, the requested hand-off feature must be available at any given point during the conversation. This is presented as a last-resort method, when the bot fails to understand the messages, redirecting the conversation to a human operator that intervenes in due course, preventing an infinite loop of misguided responses. Additionally, this feature may also be requested by user preference, providing multiple approaches to this reporting service, thereby improving quality of service.

Communication flow differentiation

Following the demonstration of the universe, or domain, that the assistant is settled upon, one key challenge that arose was the differentiation of communication flows. Regarding Rasa Open Source architecture, the assistant communicates through endpoints, where its default configurations provide a REST channel, as shown in figure 19. Having a strict messaging format, only allowing *sender* and *message* fields to be sent, it implies that the standard endpoint is not suited for the intended purpose. Thus, a new channel connector is to be developed where, at the start of every conversation, the communication flow must be identified, maintaining the same REST architectural style endpoint, distinguishing users by their ID and their associated message. Regarding communication flow differentiation, the new connector addresses this issue by having an additional *metadata* key field, where extra information can be attached, sending through the Rasa server into the Rasa Action Server, for further processing. This custom channel connector, named *myio*, is implemented as a separate Python file, *custom_channel.py*, in the root directory of the project, and properly configured in *credentials.yml*, where it can be accessed from:

*http://<host>:<port>/webhooks/***myio***/webhook*

From the specified URL, HTTP methods are used, performing POST requests to send JSON-formatted messages, specifying the communication flow type in the metadata key filed, as shown in figure 21:

```
{
  "sender": "test_user",
  "message": "Hello!",
  "metadata": {
      "type": "CHAT"
  }
}
```

Figure 21: Custom channel connector message example

Integration with external services

The main objective of this solution finds itself naturally dependent on external services, such as accessing IPBRICK databases to verify user authentication or performing API calls to submit IPTickets. Therefore, connections to these web services were implemented in *actions.py*, where all custom actions are set. All IPBRICK operations, related to user authentication and the submission of IPTickets, use Simple Object Access Protocol (SOAP)-based web services. Moreover, a PostgreSQL database is used to register events during the conversation, hence an additional PostgreSQL connection is required. A supplementary connection to RESTful web services was developed but was not used in the context of this work.

Regarding event registration, this separate PostgreSQL database aims to record all major events, in hopes of determining if the assistant is functioning as intended, as well as gathering additional statistics,

such as the average duration of each conversation and call distribution by the time of day. In this work, three types of events that occur within the Rasa assistant are considered, indicating a specific occurrence during the conversation.

- **START** - Announces the beginning of a new conversation;
- **HANDOFF** - Indicates the human hand-off event;
- **SUCCESS** - The bot successfully submitted an IPTicket, completing the intended story.

The first event, START, must be implemented every time a new conversation takes place. Upon reviewing how a standard Rasa interaction unfolds, *action_session_start* always indicates a new session, before processing user input, fitting for this specific instance. Therefore, this action is overwritten in *actions.py*, inserting a new entry in the PostgreSQL database, specifying the date and time of said instant, along with the communication type sent through the metadata field and additional parameters regarding the selected interface. Figure 22 demonstrates this behavior.

```
class ActionSessionStart(Action):
    def name(self) -> Text:
        return "action_session_start"

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

        # the session should begin with a `session_started` event
        events = [SessionStarted()]

        time1 = datetime.datetime.now() # Obtain a timestamp of the ocurrence


        for i in tracker.current_state()['events']:

            if i['value']['type'] == 'VOICE':        #VOICEBOT

                query = (...)

                connect(query, "bots")      # Insert query in the database

            elif i['value']['type'] == 'CHAT':      #CHATBOT

                query = (...)

                connect(query, "bots")     # Insert query in the database

            else:
                print("ERROR - wrong metadata fields")

        events.append(ActionExecuted("action_listen"))  # Next following action

        return events
```

Figure 22: Overwritten action that initiates the section

For the remaining events, HANDOFF and SUCCESS, two custom actions were implemented, registering such events in the database, whenever the assistant processed a hand-off intent or successfully submitted an IPTicket.

As previously mentioned, the hand-off mechanism, in general terms, is used in two different states of the conversation, whether by user intent or as a safety measure when the bot fails to comprehend user utterances repeatedly. In this mechanism, not only the user needs to be informed of such event, but both interfaces also need to be notified with a special message, in order to identify and transition from the bot to a human-to-human conversation, redirecting all upcoming messages to this new recipient.
Rasa must send two different responses given a single intent, one to be understood by the user, and another to be processed by the chosen interface, since the latter will handle the switching mechanism. Given that the user communicates with the assistant using the interface, both messages are sent to the same recipient, using the same JSON format. The first Rasa message consists of a standard message,

whereas the second message must be invisible, per se, to the original sender, only being visible to the interface, keeping the natural flow of the conversation. Therefore, the assistant sends an additional custom nested JSON object back to the interface, `{"bot_handoff":"true"}`, communicating the handover intent, shown in figure 23 and 24.

```
class ActionHandoff(Action):
    def name(self) -> Text:
        return "action_handoff"

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


            dispatcher.utter_message(response = "utter_human_handoff") # Rasa notifying the user with a standard response

            dispatcher.utter_message(                                  # Rasa notifying the interface with a JSON message
                json_message={
                    "bot_handoff": "true",
                    }
            )


            time1 = datetime.datetime.now()                            # Obtaining the timestamp of the event
            query = (...)
            connect(query, "bots")                                     # Registers a new entry in the database

            return [ConversationPaused()]       #handoff action
```

Figure 23: Hand-off custom action

```
Chat History
#   Bot                                                                                  You
1   action_listen
2                                Hi, I am having some issues. I would like to talk to an operator, please.
                                                                                   intent: handoff 0.99
3   action_handoff 1.00
    You will be transferred to an IPBrick operator. Hold on for a few seconds,
    please.

    ConversationPaused()
```

Figure 24: Hand-off request from a user perspective

From the interface's perspective, it receives two messages from a Rasa assistant towards one single recipient. Thereafter, it is possible to filter which responses are consequently heard by the user, or processed by the interface:

```
[ {
"recipient_id": "test_user",
"text": "You will be transferred to an IPBRICK operator." },
{
"recipient_id": "test_user",
"custom": { "bot_handoff": "true" }
 } ]
```

This same concept of transmitting special messages to the interface is configured and later used in the voicebot architecture, sending the following JSON message to the interface:

```
dispatcher.utter_message(json_message={"flag":"true"})
```

Some improvements regarding the overall voice-based bot performance are addressed and thoroughly explained in 4.4.

Additional Rasa implementations

When devising a virtual assistant, independently of the design decisions, it is undoubtedly impossible to anticipate every possible action that a user will take. Without proper configuration, the assistant will provide incoherent responses when faced with a topic outside its domain. Therefore, to alleviate sudden halts due to incomprehension, a generic action that handles conversational fallbacks was implemented, handling the various failures of the assistant without breaking the natural flow of the conversation. In the same scope, the hand-off mechanism was integrated in this action, functioning as a last resort method.

In order to allow the virtual assistant to fail gracefully, one must first assess what is the supporting mechanism that enables the bot to recognize an intent. Inside the configuration file, *config.yml*, a list of policies and components defines the behavior of the bot when making predictions, given user input. Default configurations are set for every new Rasa project, where a pipeline of components works sequentially to accurately extract user input and generate an output. These components can be finely tuned to yield specific results, as tampering with these and their intricate configurations compromise intent precision, as well as affecting the duration of the bot in the act of delivering responses. At the beginning of the file, a two-letter code of the used language must be inserted. This specified language is set to inform the components inside the pipeline of how intent extraction should perform. In the context of this work, the standard Rasa pipeline supports both English and Portuguese language, as these follow a similar format. For different languages, such as Arabic or Mandarin, some components should be replaced in the pipeline in order to be supported.
Contrarily to user messages that the assistant is able to respond correctly, misinterpreted messages contain lower NLU prediction values, where these range from 0, if the intent is unable to be captured, to 1, if an utterance conveys a clear intent. Failing to accurately identify the intents, the bot automatically chooses from its domain the action with the highest prediction value, disregarding the state of the conversation. By default, the last component of the pipeline predicts *nlu_fallback* intent for every input that falls below a specified NLU confidence interval threshold. Given that the implemented goal-oriented bot is comprised of a closed and short domain, the value of this threshold was raised to a higher value, since a task-based assistant dictates the path that the user must follow, dismissing any enquiries that deviate from this path. After introducing a generalized intent that gathers all misinterpreted inputs, a universal fallback action must be performed for every *nlu_fallback* intent. Therefore, a rule is set to execute a custom action whenever this intent is predicted, shown in figure 25.

```
- rule: NLU Rule - fallback
  steps:
  - intent: nlu_fallback
  - action: action_default_fallback
```

Figure 25: NLU fallback rule

In this action, the objective is to handle messages where the intent was captured incorrectly, by observing previous actions between the bot and the user. In the first occurrence of a fallback, the assistant communicates that it was not able to completely understand the input, requesting further clarification. If this situation persists in the following interaction, the bot utters what was understood from his perspective, along with a brief description of the context that the user is supposed to follow. In a final failed attempt, it is interpreted that the interaction between a user and the assistant has proven unsuccessful after three consecutive intents have been identified. To overcome this indefinite cycle of fallback actions, the bot transfers the client to an operator, activating the hand-off mechanism.

After constructing all baseline mechanisms that handle the assistant's main purpose, two versions of the bot were built, one for each requested language. These versions hold the same model, as both assistants follow the same structure, rules, and intents. For both Portuguese and English bots alike, its structural behavior and content are identical, only differing in the selected language, regarding intent extraction, in the NLU pipeline, and in the responses that are communicated back to the user.

4.4 Voice-based Bot

4.4.1 Overview

When conceptualizing a voice-based bot, one must unravel innovative ways to perfect this system. Contrarily to a text-based bot, not only must its discourse be clear, but it should also try to impersonate human speech as closely as possible. Apart from its domain and the corresponding responses that carry cognitive features, carefully handled by a response-generation mechanism, some key characteristics should not be overlooked, such as clear comprehension of utterances from its conversational partner, as well as human personification and mannerisms portrayed in the act of delivering responses. These include selecting the appropriate region-based language, the presence of spoken punctuation, and multilingualism in the dialogue. These responses should be delivered in due course, maintaining the natural flow of a conversation, so its performance and duration are crucial. These are just a few of the several characteristics that make AI personification in spoken dialogue possible, and should not be neglected when devising a virtual assistant.
In the context of this work, this chapter is set to improve the quality of service of this specific communication flow. A general overview of the voicebot architecture within the IPBRICK OS is introduced, exploring the major components and key features that compose this system. After this analysis, some contributions to this structure are presented, regarding the enhancement of the overall user experience, as well as showcasing implemented solutions associated with the development of the IPBRICK commercial support bot.

4.4.2 Voicebot Architecture

When structuring the architecture of a voice-based bot, one must first analyze how this system functions in its entirety, observing how it unfolds into several major components, and how every component interacts with its neighbouring modules. For this type of communication, an input is sent via speech, generally through a microphone, where the assistant will subsequently process its request, providing an audible response. With IPBRICK UCoIP, users engage in voice calls with one another through a private telephony system, built under the Asterisk software, via an individual's UCoIP web page, or by dialing the corresponding SIP address in IPBRICK CAFE. Considering Asterisk's extended customizability, this concept of two-person interaction was modified, developing a specific dial plan script customized for human-to-bot communication, using the Rasa bot Framework. Figure 26 showcases an interaction between a user and the virtual assistant, using IPBRICK UCoIP, where this dial plan script is executed right after a call request to the bot is made:

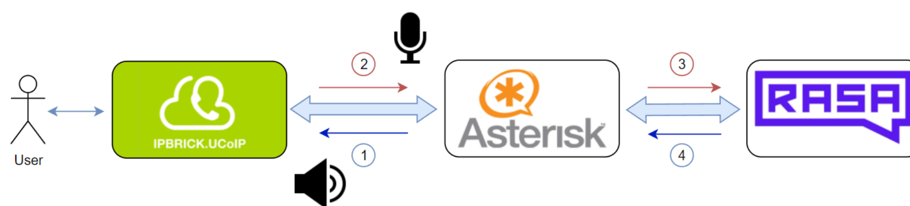

Figure 26: Outline of the voicebot architecture

1. Asterisk begins by sending an audible message through an audio output. This creates the initial environment conditions to be acknowledged by the user.
2. After hearing the contextual message, an audible notification is played, indicating that the user can proceed with providing a response. The voiced reply is generally captured with a microphone, which is subsequently handled by the Asterisk dial plan script.
3. Asterisk proceeds to run a custom program, forwarding the message to Rasa, which formulates an adequate answer by conducting the corresponding response generation mechanism, given said message as input.
4. After generating a response, Rasa sends the reply to Asterisk, which then prepares the message to be played back to the user, ending the custom program, repeating step 1 with the latest message. This cyclical process operates continuously, until the user decides to terminate the call.

To respond accordingly, the user must first be given a context at the beginning of the call, before providing a response. When configuring a virtual assistant in the IPBRICK OS, one of its parameters is the *Dialogue start message*, which converts any inserted text into an audible file, playing it in the start of a call. Every time a user calls the assistant, a new Rasa session is initiated, where it waits for user input. Therefore, in order to resolve the ambiguity present in the initial user utterance, Asterisk plays the audio file back to the user at the start of each call, providing the initial frame of reference, avoiding incoherent messages.

However, given that the Rasa bot Framework can only process text as input, the challenge relies on accurately converting speech into text, which can be consequently processed by the assistant, and converting Rasa replies into an audible format that can be played back to the user, via an audio output. In the aforementioned Asterisk custom program, ***system-bot.py***, additional components are present to solve this hindrance, namely Speech-to-Text and Text-to-Speech engines. With recourse to the Google Cloud Platform, some of its cloud computing services are used, namely the Google Cloud TTS and Google Cloud STT APIs, playing a key role in the conversion of both voice and text. To the overall voicebot structure, these engines give the bot the ability to comprehend human speech, as well as portraying human characteristics, such as giving a personified voice that ranges from language and gender.

Following the same *modus operandi* of the voicebot in figure 26, the TTS engine is firstly used when configuring the dialogue message of the virtual assistant in the IPBRICK OS web interface, converting the inserted text into an audio file. In the beginning of Asterisk's dial plan, the script starts by playing this file through an audio output. Afterwards, the user provides an audible input, which is then saved in a file and handled by the STT engine within Asterisk's custom program, translating the audio file into a text-based format. This text is inserted into a JSON type structure, following the same input format as described in 19, which is then sent to Rasa for processing. After generating a response, the content in these messages is sent to a TTS engine, transforming these into an audio file, storing it in a specified directory, and ending the custom program. Asterisk's dial plan script resumes, playing the newly generated file back to the user. Thereafter, all audio files are removed and a new communication cycle begins. A complete overview of the voicebot architecture is presented below, in figure 27.

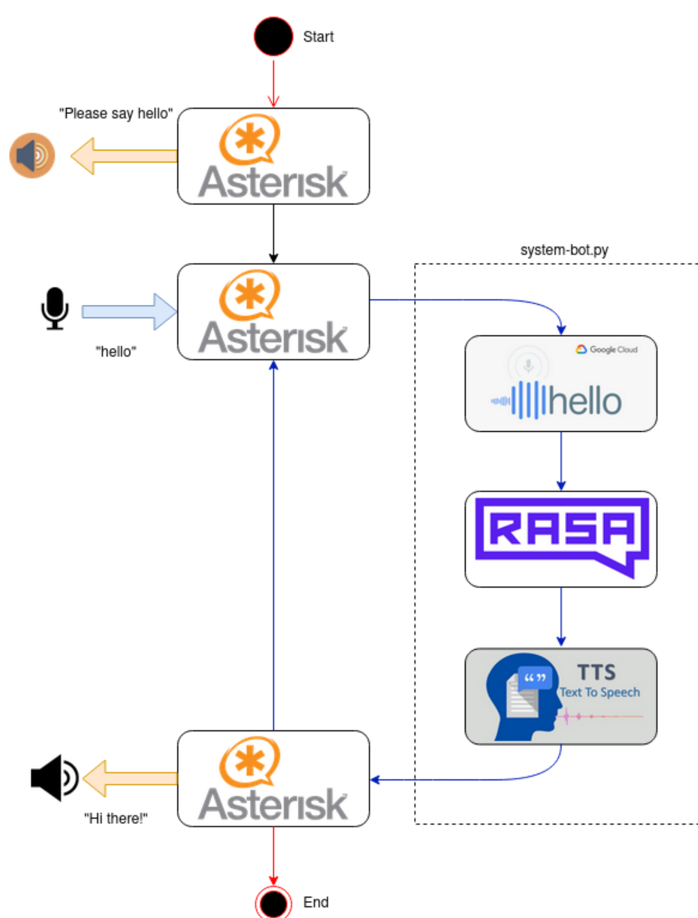

Figure 27: In-depth overview of the voicebot architecture

For every virtual assistant created in the IPBRICK OS interface, a dedicated dial plan script is developed, containing several parameters regarding their internal components and configurations. Some of these parameters refer to the used STT and TTS engines, including language and gender of the assistant, as well as indicating the Rasa server port to which messages are forwarded. Although this information is stored as a set of Asterisk-defined variables at the beginning of each dial plan script, they are mostly used in *system-bot.py*. The custom program follows a generalized structure, in order to be reused by any existing virtual assistant, along with their unique configurations. To effectively select the right parameters for a given assistant, Asterisk provides an interface between its dial plan script and external programs, the Asterisk Gateway Interface, where these variables are called and modified, manipulating the course of the communication channel as necessary.

Standard Voicebot interaction

Whenever a call with a virtual assistant takes place, *system-bot.py* is executed for each detected input, representing the time it takes for the assistant to comprehend given user utterance, and provide an audible response. To better monitor the performance of this system, the duration of every component is measured. Observing this not only represents the total period of the voicebot's execution, but also helps determine which modules are hindering the system. Drawing comparisons between each module is subjective, as each component plays different roles, where some are expected to take longer periods of time to perform the intended tasks, given their nature and complexity.

Some aspects of this dynamic are registered, in order to observe if these are subject to change, prior and after any enhancement is applied in the system. To obtain some consistency in these values, a series of calls were performed to a standard Rasa project, using a set of 5 English phrases, each varying in length, resulting in a total of 50 interactions with the assistant. In the same manner, the responses of the bot were changed, replying with extended sentences for longer utterances, and giving small responses for shorter utterances. This example aims to perform a series of simple enquiries to the bot, obtaining default performance values for the voicebot. After the implementations have been set, a new analysis to the same values is performed, observing the difference between the two.

Table 1: Default performance of every component - 50 interactions

Component duration for all 5 phrases (seconds)			
Phrase #	**STT**	**Rasa**	**TTS**
1	1.833	0.037	0.466
2	1.389	0.041	0.377
3	1.322	0.059	0.348
4	1.172	0.046	0.325
5	1.025	0.046	0.312
Mean execution time (seconds)			
-	1.348	0.046	0.366

Table 2: Default minimum and maximum values for every component

Mininmum and maximum component duration (seconds)						
-	**STT**		**Rasa**		**TTS**	
Phrase #	Min	Max	Min	Max	Min	Max
1	1.712	1.984	0.033	0.041	0.351	0.754
2	1.292	1.593	0.038	0.046	0.306	0.568
3	1.249	1.372	0.056	0.063	0.301	0.381
4	1.114	1.249	0.043	0.051	0.282	0.487
5	0.966	1.100	0.042	0.050	0.271	0.376

On a first analysis, there is a clear distinction on the execution time of the different components. Considering the total mean execution time in Table 1, **1.76 seconds**, it represents the average time a user waits to listen to a response, after giving an input. During this time interval, there is a constant silence period, where users that engage in the conversation can interpret it as errors that occurred during the call, such as sudden call drops or problems with the recording device, compromising quality of service, and the natural flow of the dialogue. Therefore, reducing the overall execution time is a fundamental part of improving this communication flow.

After Asterisk records user input and stores it, the STT component first starts by reading the resulting audio file, sending its content to Google Cloud API, converting the introduced speech into text. When observing the registered interactions in table 1 and 2, all 5 sentences are ordered by phrase length in a descending manner, from long to short. Comparing variable-length phrases to their corresponding duration in the STT component, one can observe that the time spent reading an audio file, together with audio conversion, is directly correlated with file size, meaning that for longer user utterances, the STT component will take longer to perform.
Before the audio file is sent for processing, some configurations are set for speech recognition. Choosing the sample rate of the audio file, specifying the spoken language for transcription and adding multichannel recognition are some of the many settings that can be applied. However, optional parameters can be configured in this system, such as recognizing and transcribing audio to multiple languages, as well as introducing automatic text punctuation and profanity filters. These should only be inserted when applicable, as extra requirements compromise execution time. Therefore, a balance between conversion time and accuracy is ever-present. Preferring a faster service compromises the text that is forwarded to Rasa, whereas the opposite configuration profile results in a slower but accurate service, so choosing optimal parameters is key to good execution. Moreover, relying on external services to perform audio conversion implies an additional latency to the system, considering the Round-Trip-Time values when sending audio files to Google services and obtaining the according responses. Given its worldwide distribution, selecting the nearest available region minimizes the delay when performing requests to Google Cloud services.

Using a modified version of the default Rasa project, its performance remains somewhat consistent throughout every interaction, always assuming values within the range of the average execution time, independently of phrase length. Having a reduced number of intents in the domain, the Rasa server is faster to predict and extract an intent in the user input, providing quick and simple utterances back to the user.
In this initial case, external services are not used, as custom actions are not set by default. When introducing the commercial support bot, some of the interactions are expected to perform substantially slower than the average responses, as these require access to IPBRICK services in order to perform specific functions. Additionally, its domain is expected to become more complex, having a larger set of intents, thus assumed to perform comparably slower, as intent prediction goes through a larger domain. Contrarily to other components, the Rasa server runs inside the machine that hosts the IPBRICK OS, relying heavily on processing power, therefore its performance may vary depending on the hardware components that this solution is set on.

The TTS component is expected to follow a similar behavior to STT, as there is a latency associated when requesting external services. Moreover, from the results presented in tables 1 and 2, one can observe that lengthier sentences, that are subsequently synthesized, generally impose larger execution times in the system. Given plain text as an input, some initial parameters are set to define the voice of the synthesized audio, such as the preferred language and gender, as well as optional settings to further configure audio data, including speaking rate, pitch, and volume gain.

Although Google Cloud services play a fundamental role in the voicebot, there is a fee associated with every request, for both STT and TTS APIs. This pricing is based on the amount of processed audio and synthesized characters for every month, respectively. Given the voicebot architecture, for each human-to-bot interaction, both of these services are requested once. From the Google Cloud Platform, one can observe the number of requests, as well as the mean latency of each service exclusively, presented in figure 28.

Name	↓ Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Cloud Speech-to-Text API	98	0	807	1,840
Cloud Text-to-Speech API	98	0	79	363

Figure 28: Requests from Google Cloud Platform

The voicebot solution is configured to be easily scalable and replicated across several machines that host the IPBRICK OS. In result, when distributing the voicebot solution to several IPBRICK clients, the usage of these external services is expected to grow exponentially, impacting economic resources in an unprecedented manner.

4.4.3 Voicebot implementation

Upon performing an analysis on a standard voicebot interaction, their major components are identified, carefully exposing their structural constraints. Regarding the implemented enhancements, the proposed solution addresses some of these architectural limitations, as well as introducing additional behaviors that complement the devised virtual assistant and its unique features.
The focus of these improvements are set to adapt this interface to multiple-language support, using English and Portuguese as an example, as well as adding domain-specific word recognition and some minor improvements. Additionally, some adjustments to the Asterisk dial plan script and its external program are made, in order to increase performance in the TTS component, as well as reducing the total amount of requests to Google services by introducing a system that re-purposes synthesized audio responses. In conjunction with the context of this work, supplementary configurations are set to the dial plan script and *system-bot.py*, the external program, taking the necessary actions whenever a hand-off intent is recognized.

Voice recognition

The STT component is responsible for accurately converting spoken utterances into plain text. Depending on the environment that surrounds the user, voice recognition mechanisms must be able to capture certain intents and entities that deviate from generalized vocabulary. Therefore, specific parameters must be configured, prior to audio transcription, as these particular words can easily be mistaken with expressions that portray a similar phonetic structure. Given the closed-domain characteristics of the virtual assistant, as it is solely developed for issue reporting purposes, formal language that is unique to a specific field must be prioritized over general expressions. To this end, Google Cloud STT uses the *model adaptation* feature to recognize particular phrases more frequently. In this case, expressions such as *IPBRICK*, *IPTicket* and *iPortalDoc* are added, expanding the vocabulary of words that are recognized by this component.
When devising the commercial support assistant, it is crucial that transcription results hold some lexical construction when forwarding text to the Rasa component. This allows for the assistant to better understand the intent of the phrase, as well as provide an understandable format to the developer for further comprehension when reviewing past conversations through the CDD approach. Therefore, automatic punctuation was enabled to improve phrasal structure, reducing misinterpreted intents.

Whenever a user calls a virtual assistant, their initial configurations are set in the Asterisk dial plan script. These settings include the spoken language, as well as the gender of the synthesized voice, for personification purposes, which are passed to the respective STT and TTS components through the Asterisk Gateway Interface, loading *system-bot.py*. In this case, if a user were to call an English-based bot speaking a different language, the STT component would try to transcribe the audio to a phrase in the English language, thus failing to capture the intent.
In order to address the support of both English and Portuguese given a single call to an assistant, the TTS component is configured to identify the spoken language within the audio file and, upon detection, loads the corresponding parameters, redirecting the conversation to the appropriate bot. Given an example where two virtual assistants are configured in the IPBRICK OS interface, their parameters are set to interact with Rasa P and Rasa E, if the user desires to interact using Portuguese and English language, respectively. In this instance, if a client calls the English-based voicebot using the Portuguese language, the STT component identifies this language, loading the remaining parameters as if one was dialing the Portuguese voicebot, forwarding the text to Rasa P, which will consequently use the TTS to generate an audible output in Portuguese.
This language identification mechanism is only performed in a single instance when the first audio input is introduced, and will not be performed again until a next conversation occurs. This prevents users from engaging with different virtual assistants in the midst of a conversation, as the state of the conversation is unique to each assistant. This implementation is showcased in figure 29:

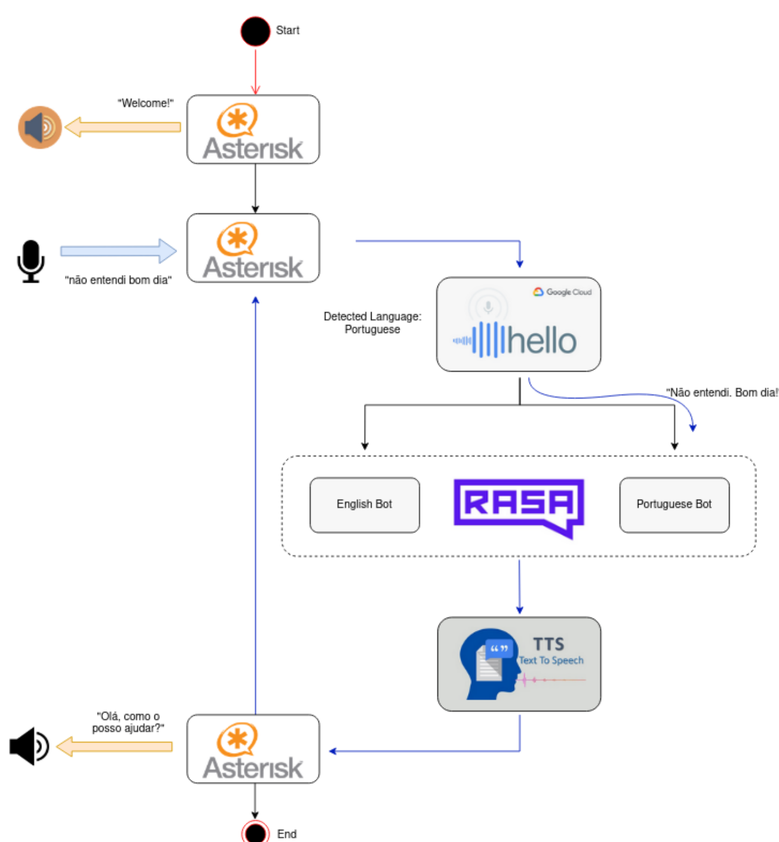

Figure 29: Enhancements regarding voice recognition

Synthesized audio re-purposing

After evaluating the default performance of a voice-based assistant, in tables 1, 2 and the corresponding Google Cloud requests in figure 28, one could observe that both STT and TTS components are accessed in every interaction. This imposes limitations concerning latency and economical resources, as the TTS is always requested, even if it has synthesized the same exact text previously. Given that the voicebot always produces a fixed amount of responses, exploring the concept of re-purposing synthesized audio files that match the exact reply of the assistant reduces the amount of requests being made to the TTS API.
Therefore, the challenge relies on the process of matching current Rasa responses with previously-stored audio files, addressing how, when and which responses should be synthesized and differentiated. Additionally, this new implementation should function interchangeably with Google TTS service, if such file is non-existent, thus requiring new audio files to be processed.
However, given the unpredictability of the human mind, this solution proves unfeasible to the STT component, as each individual possesses their own mannerisms and unique characteristics when producing speech.
In the process of generating responses, the developer must be aware that outputs from the Rasa component are differentiated into two types of replies: strict and dynamic. As the name suggests, the first type of messages are pre-determined and set by the developer, always remaining static independently of the state of the conversation. Given this example of a strict utterance:

```
utter_greet:
- text: How can I help you?
- text: Hey, how can I be useful?
- text: How can I be useful?
- text: Hi, how can I help?
```

Despite offering several alternatives to the *utter_greet* action, these serve the purpose of providing different responses and do not depend on the user that triggered the initial intent, thus being suitable for audio file re-purposing.

On the other hand, dynamic responses are pre-determined messages, carrying variable elements that are unique to every conversation. These are cases where the assistant requests data from the user, which are then mentioned further in the conversation. In the following example, the bot asks for the name of the user, providing a custom welcome message with the solicited information.

```
Virtual assistant: "Hey, what is your name?"
User: "My name is John Doe."
Virtual assistant: "Hello, John Doe!"
```

Portraying itself as a useful technique when giving context, this simple example quickly showcases that these type of replies, when synthesized to an audio file, should not be re-purposed, as every user transmits different data, compromising hardware storage resources.
Therefore, a mechanism that identifies specific messages must be set, preventing dynamic audible replies to be saved locally. Following a similar concept to the hand-off intent, Rasa sends the following JSON message:

```
dispatcher.utter_message(json_message={"flag":"true"})
```

In this case, similarly to a hand-off message in figure 23, this JSON message is imperceptible to the user and is sent alongside a visible response, being processed by *system-bot.py* and the Asterisk dial plan script, indicating that the resulting audio file from the visible message is to be removed afterwards.

When delivering a message to a user, a Rasa response passes through a TTS component, converting plain text into an audible file, storing it locally. In a default configuration, the recorded user input is identified with a sequence of numbers, where its corresponding response uses the same identifier, followed by an "*_answer*" tag:

```
User audio input file:  <UNIQUE-ID>.wav

Bot audio output file:  <UNIQUE-ID>_answer.wav
```

After this conversion, Asterisk plays the audio back to the user, discarding it after its usage. This naming convention is practical, yet not built to be re-used, as each identification number is unique to every interaction.
In order to introduce a file re-purposing mechanism, a new naming convention system is to be set, where local file verification must be performed prior to any Google TTS request, playing a previously stored file instead. If such file does not exist, it requests the audio from the TTS service, storing it locally under a new name.
When developing a naming convention system, one must be aware of the criterion that affects audio files. In this case, the language and gender of synthesized voices, deriving from TTS parameters, along with utterances from the Rasa assistant are what define an audible response. Hence, the following naming convention is proposed:

```
<SYSTEM_LANGUAGE>_<VOICE_GENDER>_<UTTER>.wav
```

The first parameter, *SYSTEM_LANGUAGE*, is classified by the IETF BCP 47 tag, a standardized code that is used to identify and distinguish languages according to their country, region, and writing system [38]. This structure is adopted by Google services, being a mandatory parameter when synthesizing audio, therefore no additional configurations are necessary. The second parameter is also required by Google services, specified by a single-letter code, differentiating the gender of the spoken language ("m" for male, and "f" for female).
The last parameter is solely dependent on Rasa, varying with each message from the assistant. With every reply being unique and directed to a user, each utterance is constructed towards human comprehension, containing key phrasal elements, such as punctuation and symbols that compromise file naming scheme, thus an additional step is necessary, removing all non-viable characters beforehand. Given this brief example for both languages:

```
Hey, how's it going?  → hey_hows_it_going

Ótimo, muito obrigado.  → otimo_muito_obrigado
```

This text normalization method converts every phrase into a single lowercase string of characters, replacing letters that contain accent marks with their corresponding non-accented version, removing punctuation, and substituting the space character with an underscore. This ensures that every utterance is properly formatted, as user-generated text often includes invalid characters that can incapacitate file path mapping, while simultaneously being arranged in an understandable format for both man and machine.
Upon configuring all three parameters, two distinct files containing synthesized audio using a female a male voice would be presented as:

```
en-US_f_hey_hows_it_going.wav

pt-PT_m_otimo_muito_obrigado.wav
```

When introducing the re-purposing mechanism, a final architectural overview of the voicebot is presented as the following:

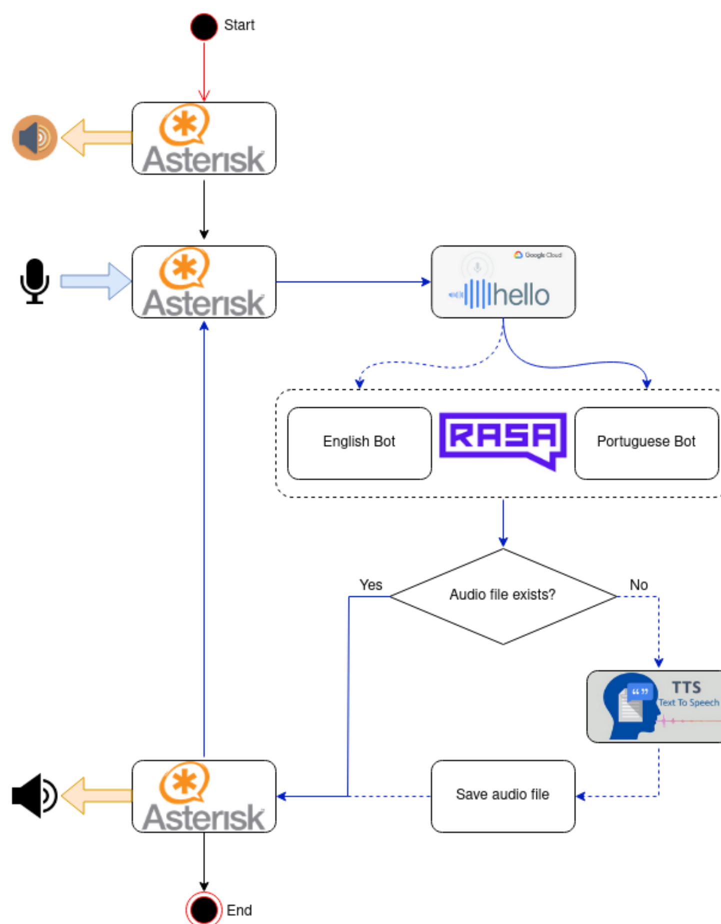

Figure 30: Complete voicebot architecture

After all implementations regarding this interface have been set, the second part of the experiment was conducted, using the same environmental factors, repeating the same 5 phrases up to a total of 50 interactions, retrieving 5 different responses from Rasa, observing the performance of every component in Table 3.

Table 3: Performance of every component, after implementations - 50 interactions

Component duration for all 5 phrases (seconds)			
Phrase #	**STT**	**Rasa**	**TTS**
1	2.034	0.039	0.071
2	1.404	0.042	0.055
3	1.341	0.060	0.048
4	1.227	0.047	0.039
5	1.030	0.049	0.041
Mean execution time (seconds)			
-	1.407	0.047	0.051

Table 4: Minimum and maximum values for every component, after implementations

Mininmum and maximum component duration (seconds)						
-	**STT**		**Rasa**		**TTS**	
Phrase #	Min	Max	Min	Max	Min	Max
1	1.882	2.336	0.035	0.049	0.001	0.698
2	1.312	1.487	0.036	0.049	0.001	0.537
3	1.308	1.426	0.055	0.065	0.002	0.461
4	1.132	1.451	0.043	0.050	0.001	0.384
5	0.956	1.074	0.046	0.051	0.002	0.395

Name	↓ Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Cloud Speech-to-Text API	50	0	835	1,855
Cloud Text-to-Speech API	5	0	98	242

Figure 31: Requests from Google Cloud Platform - audio file reusage

When drawing a comparison between the initial configuration and the proposed implementations regarding the STT component, there is a slight increase in the overall average performance time. Not only does the mean latency of the Google Cloud STT service increase, visible in figure 31, due to additional requirements such as alternate language recognition and automatic word punctuation, but changes performed to the external program, *system-bot.py*, also affected the STT performance.
In each conducted experiment, the execution time regarding the Rasa component maintained constant values, since no changes were performed between the two.
However, there was a significant improvement in the TTS component. Given that all responses were classified as strict messages, the re-purposing audio file mechanism was put into motion. In the first interactions with the virtual assistant, no audio files were stored locally. Due to the fact that the first 5 inputs consisted of the set of 5 unique English phrases, 5 different requests were performed to Google Cloud, creating the necessary synthesized audio files, explaining the maximum values for the TTS component in table 4. Afterwards, when repeating the same set of phrases, all audio files were re-used, showcasing values close to the minimum performance levels. In result, for a total of 50 interactions with the bot, only 5 requests were performed to the TTS API, drastically reducing monetary costs and performance time, confirming its effectiveness in figure 31.

Additional Implementations

In order to communicate to a database that a specific call has terminated, the Asterisk dial plan script was modified, inserting an entry to the database with **HANGUP** event, as well as identifying the user through its identification number right after a call is terminated.
This ensures that a call was deliberately terminated by the user, and not due to possible errors that occurred during this interaction. This same database is explored in section 4.6, when developing a dashboard-like interface to showcase statistical data regarding the performance of both communication flows.

4.5 Chat-based Bot

4.5.1 Overview

Communication that revolved around distant message exchange has suffered a continuous evolutionary process. From the early adoption of sending smoke signals to communicate an intent, to software applications that include instant-messaging capabilities for online interaction, humans have become ever so acquainted with remote communication. Never being fully aware of the identity of the recipient behind these messages, the introduction of AI to instant-messaging services became a fitting application in circumstances where tasks can be performed autonomously and handled remotely.
When designing a chat-based bot, an interface with transparency attributes must be chosen, as its communication method is rather simple, relying on exchanging plain text messages. There must be a clear comprehension of all actors that take part in the conversation, where users can instinctively interact with one another. A holistic approach to this system must be carefully addressed, as this chat-based communication method interacts with multiple external systems.
In the context of this work, a Graphical User Interface fit for instant-messaging communication is adapted for human-to-bot interaction, creating all back end functionalities that allow this interface to communicate with a Rasa virtual assistant. Additional support for human-to-human communication is developed, where a natural hand-off mechanism is to be implemented without breaking conversation flow, interacting with users within the IPBRICK CAFE without resorting to additional interfaces. The resulting solution is required to be fully customizable as a service, compressed into a bundle-like module that can be deployed into any web page.

4.5.2 Chatbot Architecture

For this type of communication, every interaction is transmitted through a readable, text-based format. Considering that the chat-based bot must be able to talk with two distinct recipients at different points in time, its architecture is first decomposed into baseline elements, according to the imposed requirements. A detailed analysis for both types of communication is performed, describing their modes of operation, thus establishing a complete structure of the chatbot.
In order to grant deployability, the interface will consist of a non-intrusive pop-up icon that resides in the bottom-right corner of a web page. When clicked, a chat box will appear, displaying an interface where the user transmits and receives messages from both recipients. From figure 32, its intended behavior is as follows:

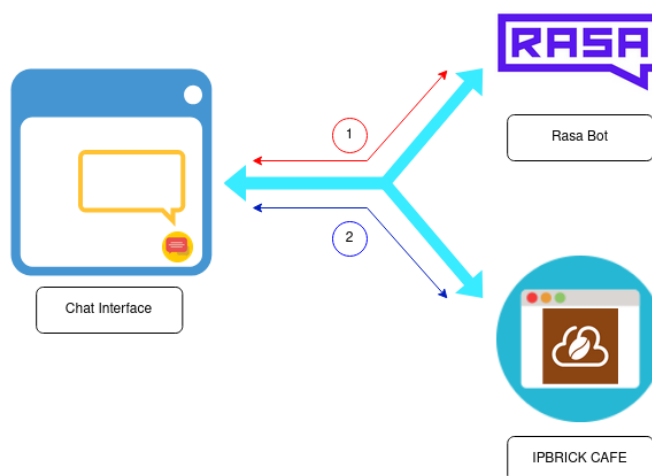

Figure 32: Baseline chatbot architecture

1. At first, the user is presented with a contextual message, stating the assistant's main purpose. By default, every input that is introduced to the interface is forwarded to a Rasa assistant, receiving messages and replying accordingly to all extracted intents.

2. Given a specific instance where the hand-off intent is captured by the assistant, this information is communicated to the interface, performing a protocol switching mechanism. Under these circumstances, all subsequent messages are then forwarded to a human-aided environment, where an operator using IPBRICK CAFE can provide assistance.

Similar to the initial audio output, *Dialogue start message*, applied in the voicebot, an introductory message is also presented, giving context in order to respond accordingly.
Having previously explored Rasa's communication procedure, whenever a user finishes typing a message, its content is displayed in the chat interface and simultaneously sent to the assistant, through an HTTP POST method.

As an equally significant aspect of this solution, in order to interact with a human operator, one must first analyze how users within the IPBRICK network interact with one another, specifically using IPBRICK CAFE.
In this network, all business logic regarding interpersonal communication is handled by ejabberd, an Extensible Messaging and Presence Protocol server application. In a similar approach to e-mail, every entity has its own address, or Jabber ID, sending and receiving messages to or from others, respectively. Whenever a client wishes to send a message, better known as stanzas, a connection to an XMPP server must be present, before sending any stanzas through a bi-directional XMPP stream. Therefore, an updated version of the architecture is presented in figure 33:

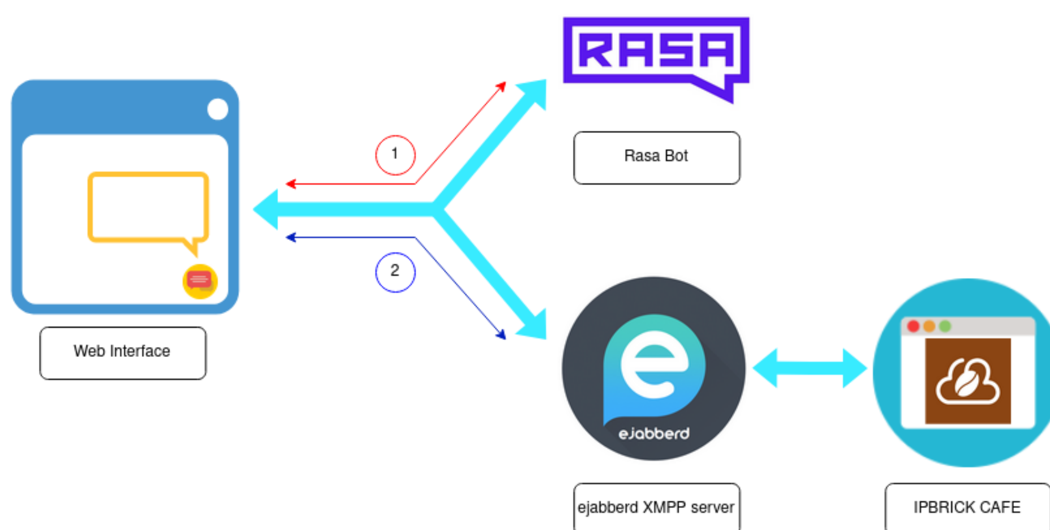

Figure 33: Complete overview of the chatbot architecture

Apart from implementing both communication protocols, as well as its inherent switching mechanism that alternates from the different recipients, the challenge resides in devising a customizable, bundle-like solution that is ultimately loaded onto any web page.

4.5.3 Chatbot Implementation

Graphical User Interface

In order to deploy this component in any website, without compromising the structure of the web page, a dedicated JavaScript library for building user interfaces was used, known as React, granting the ability to adopt full-fledged React projects while keeping a small footprint in the source code of the target web page.
For the Graphical User Interface component, the publicly available project *react-chat-popup* was used, not only for its simplistic design, but due to its customizable features [39]. Given every message from its two distinct entities, originating from the user or its recipient (a virtual assistant or a human operator), this project handles the front end logic, ensuring that messages are correctly displayed in chronological order while identifying its sender, as well as the button that portrays its pop-up feature. The baseline of this project is carefully addressed, customizing its interface according to the context of this work, and presenting a final design as shown in figure 34.

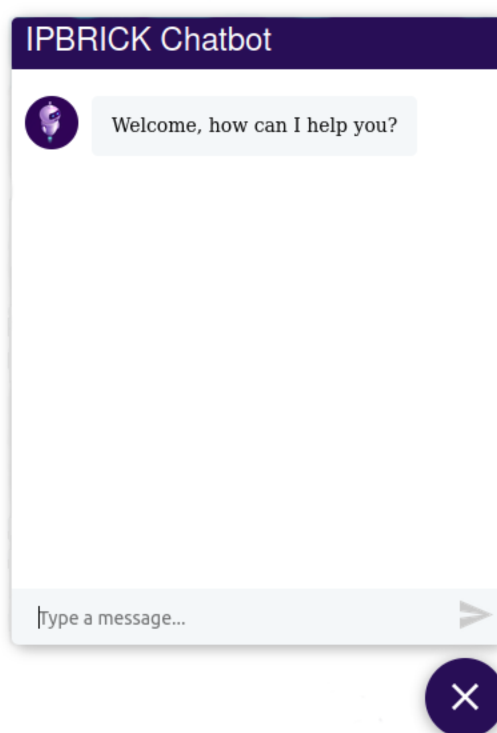

Figure 34: Customized user interface

In conjunction with this interface, the back end logic of this project is constructed from the ground-up, implementing both types of communication protocols, regarding human-to-bot interactions with a Rasa virtual assistant, and reciprocal human-to-human communication through ejabberd.

Back end

In the same manner that typed messages are displayed on the designated chat area, notifying the entity that their message has been sent, it is imperative that a background, behind-the-scene process must format the same content, according to each protocol standards, effectively forwarding the message to its intended receiver.
Analogously, given a situation where a Postal Service contains two collection boxes for two distinct zones, if an individual places a written letter in neither of these boxes, leaving the premises with all

parameters left unspecified, a postal carrier does not know to whom, or where, the letter supposed to be forwarded. From the individual's perspective, he deliberately took the action to send a message, oblivious to the fact that the Post Office is unable to address his request.

This back end dynamic is best described as having a stateful behavior, reacting differently to the same inputs, based on pre-determined events. In an initial state, the user first interacts with a Rasa assistant, thus encapsulating the message under a JSON-formatted payload. Under the new REST channel connector, **myio**, additional parameters are set, similarly to figure 21, indicating a different communication flow, "CHAT", amongst other key values included in the *metadata* field, to be sent through an HTTP POST method.
For every response that originated from the virtual assistant, the interface displays its plain-text content, while being observant for a specific JSON object.


```
[ {
"recipient_id": "test_user",
"text": "You will be transferred to an IPBRICK operator." },
{
"recipient_id": "test_user",
"custom": { "bot_handoff": "true" }
 } ]
```


Upon recognizing the hand-off intent from a user utterance, the Rasa assistant transmits an additional JSON object. Not being visible to the chat display area, the interface changes seamlessly to its second and final state: interpersonal communication.

In order to interact with operators using the IPBRICK CAFE, this project must also support communication through the XMPP protocol, connecting to the same ejabberd server. StanzaJS presents itself as the indicated solution, being a JavaScript/Typescript library dedicated to using XMPP in the browser, exposing all content in a JSON format [40].
Every interaction through the XMPP communication protocol requires identification from both participants. Prior to sending chat messages, or stanzas of type chat, both parties are required to be connected and properly identified to the same ejabberd server. Having specified the Jabber Identity (JID) address that will serve the purpose of a human operator, the challenge of identifying the entity using the chatbot interface arises. Given an instance where multiple individuals access the same web page, requesting assistance from the designated operator at the same time, their JID addresses should ideally be different from one another, so that the operator can provide undivided assistance in separate chat rooms. To overcome this issue, unregistered users can still engage with a specific ejabberd member using the visitor feature.

```
ejabberd user: <user>@<domain>

visitor for a given user: <visitor_name>_<user>_<visitor>@<domain>
```

In order to guarantee separate chat-rooms for every unregistered client, the **<visitor_name>** parameter is given a unique identifier, being randomly generated every time the web page hosting the chatbot interface is accessed, thus enabling communication between the interface and an IPBRICK operator using IPBRICK CAFE.

Additional implementations

Apart from communicating with a Rasa assistant in the first state, a JavaScript variable is stored in browser memory, keeping track of all interactions that occurred between the user and the assistant. Upon detecting a hand-off intent, the project switches to its second state, connecting the user to an ejabberd server, so that all upcoming messages are received by an operator, where he can respond accordingly. Prior to any interaction between the two, and once a bi-directional XMPP stream has been defined, the content of this JavaScript variable, holding all chat history, is sent to the operator. When receiving all text logs from the user and the bot, the operator is first given context as to why his assistance was requested. By analyzing the chat history, he can better identify the problem, as well as provide faster support to his client, showcasing this instance in figure 35.

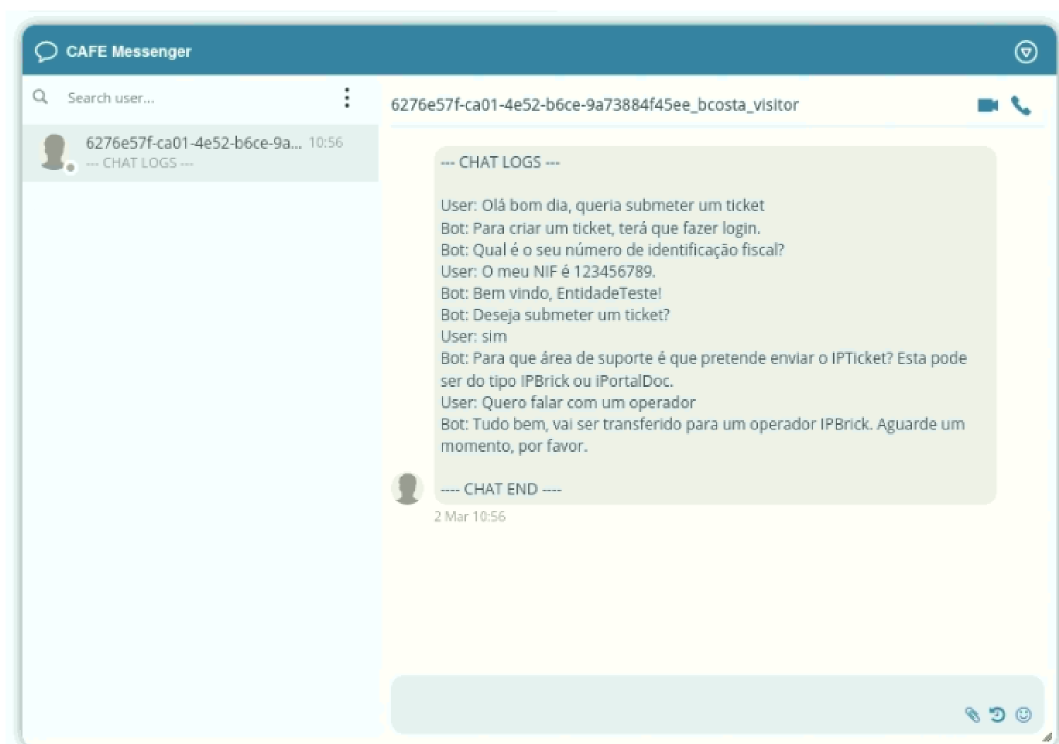

Figure 35: Hand-off from an IPBRICK CAFE perspective

In order to indicate whether a user has stopped communicating with the chat interface, similar to the behavior when a user deliberately hangs up a call regarding the voicebot infrastructure, additional features are implemented in this project. Having no "Hangup" button, one must devise a solution to perform the same mechanism in the chatbot interface. Therefore, *Event Listeners* are implemented in this module. When triggered, an entry in the database is registered with the **HANGUP** event.
From the moment where a user types a message to an assistant, two time-based events are initiated. These are present to detect motion, regarding mouse activity within the web page, along with keystroke detection, indicating that the user is still interacting with the web page hosting the chatbot component. If no activity is detected for a given amount of time, an event is triggered, registering a **HANGUP** event in the database, confirming user inactivity. In these circumstances, the chatbot interface sends a timeout message, declaring that the next sent message to the assistant will result in a new conversation, under a new identification number.
In addition, another *Event Listener* detects for browser or tab closing events. Whenever this window is closed, it is interpreted that the user no longer wishes to interact with the assistant, therefore the same **HANGUP** event is registered.

4.6 Dashboard

4.6.1 Overview

When implementing services that include virtual assistants, along with the proposed interfaces, it is imperative that the service provider maintains a constant awareness of the performance of this solution. For every interaction that occurs, its major events are registered in a dedicated database.
Having its core *modus operandi* revolved around AI, to accompany this evolutionary and continuous process, a proper tool must be designed to process data in a given database, showcasing the effectiveness of all virtual assistants, in order to assess if their implementation has proven to be successful. In conjunction with Rasa X, a CDD tool that observes past conversations with Rasa bots, it is possible to adjust an assistant's behavior by correcting its previous mistakes, ensuring that it is constantly being perfected towards the intended purpose.
In this section, the construction of a database is carefully addressed, detailing how its information is perceived, along with the development of a dashboard interface using KoolReport, a PHP Reporting Framework. It processes information from a data source, communicating valuable data insights regarding bot performance using graphs, charts, and tables.

4.6.2 Implementations

Regarding the implementation of the database, one must be aware that information derives from distinct environments. When devising a virtual assistant in the UCoIP section from the IPBRICK OS web interface, its configurations are set in the Asterisk dial plan script, which are forwarded to *system-bot.py* through the Asterisk Gateway Interface. Relevant parameters include the assistant's name and identification number, together with its dedicated Rasa port to which all spoken utterances are redirected, as well as settings inherent to Google Cloud services. These define the core characteristics of a virtual assistant and should not go overlooked.
When specifying a voice-based communication flow, all aforementioned parameters are included in the payload of a message and sent at the beginning of a human-to-bot interaction, through the *metadata* field. Subsequently, the first Rasa action, *action_session_start*, handles this information by inserting an entry in the database, registering a new conversation with the **START** event.
In the same manner, for chat-based communication, UCoIP matching parameters are set as variables in the chatbot React project, also being sent in the *metadata* field at the start of a conversation, along with its specified communication type.
Besides both interfaces handling circumstances where a user terminates a conversation from either voice or chat, the remaining events, such as successfully issuing an IPTicket or transferring current users to a human operator, are handled by the Rasa assistant in the midst of a conversation. Every event, when inserted into the database, is identified by date and time, ensuring a chronological order. The proposed relational database is built under PostgreSQL, introduced in figure 36:

```
id serial PRIMARY KEY,
--TIME CONFIGURATIONS
datetime timestamp with time zone,
year int,
month int,
day int,
hour int,
minute int,
second numeric,
--BOT CONFIGURATIONS
user_id varchar(255),         --user id
bot_type varchar(255),        --CHAT/VOICE type
event varchar(255),           -- START/HANGUP/HANDOFF/SUCCESS events
extra_details varchar(255),
--UCoIP CONFIGURATIONS
bot_name varchar(255),
bot_port int,
bot_id int,
bot_system_language varchar(255),
bot_voice_gender varchar(255)
```

Figure 36: Database for virtual assistants

Considering the range of events that occur in this domain, its sequential and chronological order, from **START** to **HANGUP**, provides different interpretations. In circumstances where the virtual assistant was able to follow through with the intended task, it will register the events START, SUCCESS, and HANGUP in chronological order. Given the same example, in instances where there is a HANDOFF event instead of SUCCESS, it represents that the user either requested an operator in the middle of the conversation, or the bot failed to complete the task, thus recurring to human-mediated assistance. Only having the START and HANGUP, however, it is interpreted that a conversation was initiated with an assistant, but was abruptly terminated.

Upon developing a database solely for storing information regarding virtual assistants, the following step is dedicated to data processing: gathering all collected data, manipulating it, and producing meaningful information back to the user. In order to present valuable information, IPBRICK provided some guidelines during the process of data treatment:

- Daily statistics
  - ⇒ Today's calls by bot - Daily requested assistants, presented the total number of interactions and differentiated by preferred communication flow type;
  - ⇒ Today's call treatment - Daily number of successful calls, hand-off requests and abrupt exits during conversations, separated by communication type, per bot;
  - ⇒ Today's call treatment distribution - Presenting daily call treatment data, independently of communication type, percentage-wise.
- Statistics from the last 5 days
  - ⇒ Received bot conversations - Requested interactions per assistant in the last 5 days, presented the total number of interactions and differentiated by preferred communication flow type;
  - ⇒ Bot distribution - Total number of requested assistants in a 5 day window, independently of communication type;
  - ⇒ Call treatment - Number of successful calls, hand-off requests and abrupt exits during conversations, separated by communication type, given a 5-day time interval, per bot;
  - ⇒ Call treatment distribution - Call treatment data in the last 5 days, independently of communication type, percentage-wise.

- All time statistics
  - ⇒ Total calls by bot - Total number of interactions per assistant, presenting the total number of interactions, differentiated by preferred communication flow type;
  - ⇒ Total bot distribution - Total number of requested assistants, independently of communication type;
  - ⇒ Call distribution by time of day - Presenting the number of performed interactions, per assistant, in a 24 hour window;
  - ⇒ Total Voice/Chat bot distribution - Percentage-wise representation of communications involving virtual assistants, differentiated by communication flow type;
  - ⇒ Total call treatment - Total number of successful calls, hand-off requests and abrupt exits during conversations, separated by communication type, per bot;
  - ⇒ Total call treatment distribution - Percentage-wise call treatment data, independently of communication type.

Following these guidelines, a dedicated web page was implemented using KoolReport. Some extra information was displayed, such as the average call duration for both chat and voicebots, in order to best perceive if the duration of the implemented service is performing as expected. These statistics are displayed by performing SQL queries to the database, showcasing the requested tables, charts, and graphs, where some are presented in section 5.

5 Validation and Evaluation

5.1 Evaluated Metrics

Having devised specific implementations regarding both interfaces, suggesting some enhancements to the overall voicebot architecture, as well as constructing a chat-based component capable of functioning with two distinct communication protocols, these were adapted to comply with the proposed solution. Furthermore, apart from creating Rasa bots that report faults within the IPBRICK infrastructure, these once isolated platforms were merged into a single, virtual assistant-oriented ecosystem.
Some advancements concerning the assessment of bot learning capabilities are indicated, through the Rasa X interface, maintaining a constant CDD approach when reviewing past conversations, as well as using Google Cloud Platform to consult the performance of vital components that make up voice-based interactions, and analyze the number of performed service requests and their respective costs and latency.
However, when introducing this service in an environment, there is no practical way of determining if assistants are performing as expected. Despite being possible to review every single conversation, by carrying out an extensive process of manually reviewing conversations that resulted in either failure or success, using Rasa X alone does not account for possible errors that are interface-specific. Failing to address this issue, it compromises the assistant's learning process, as well as quality of service.
Thus, in order to practically determine the performance of any virtual assistant using both types of communication, the introduction of event registration is included, together with a dashboard interface, proposing a careful perspective when reviewing the performance of multiple assistants, in hopes to better identify its underlying issues, either Rasa or interface related, exposing all major occurred events.

To evaluate the efficiency of this solution, this system is introduced in a closed environment, conducting a simulated experiment that aims, not only to give a practical exhibition of its intended application but to demonstrate the assistant's gradual learning process, while being alert to information presented in the dashboard, contributing to better results.
This experiment was conducted for **5** days, performing a total of **200** conversations, **102** through voice and **98** using the chat interface, exploring an initial situation where the assistant consistently fails to submit IPTickets, gradually evolving to a state where there are noticeable improvements, successfully submitting IPTickets. Throughout this simulated experiment, the dashboard indicates areas of interest to where this system should be improved.
Given larger failure rates for a single interface, when compared to the opposite communication type, it suggests that the assistant is failing due to interface issues. However, if both interfaces indicate constant failed interactions, it is presumably due to a problem imposed with the Rasa assistant, affecting both interfaces in result, resorting to Rasa X for its correction.
The number of **HANDOFF** and **SUCCESS** events from each communication type are evaluated, expecting a gradual decrease in hand-off occurrences and an increase in the latter.
Additionally, a parallel analysis of the voicebot infrastructure is performed, examining results given by voice-based interactions, and observing the duration of each major component, while being attentive to the synthesized audio re-purposing mechanism.

5.2 Validation & Results

In this controlled environment, an initial scenario details a situation where the devised commercial support bots interact with a simulated target audience, IPBRICK clients and partners, in order to report system faults using IPTickets.
In this scenario, however, both virtual assistants are assumed to be poorly optimized, only having a few stories that follow the process of submitting a ticket. By not complying with a CDD approach, these assistants were not subjected to user testing before being inserted in a simulated ”live” environment, therefore lacking in vocabulary and common expressions that are unique to every language and individual. Without proper training, the virtual assistants fail to capture intents in a consistent manner, handing over the majority of conversations to an operator, recurring to human assistance in order to fulfill the original request.

Through the dashboard interface, IPBRICK users view the deployed virtual assistants, observing the outcome of implemented bots when placed in the target environment. In these circumstances, the daily performance of the English commercial support bot (EN - BOT) and the Portuguese commercial support bot (PT - BOT) are reviewed on the *Daily Statistics* segment on the dashboard. In an initial perspective, figure 37 denotes the number of calls that have occurred for each assistant, on the first day of this study, where a total of 49 conversations were performed.

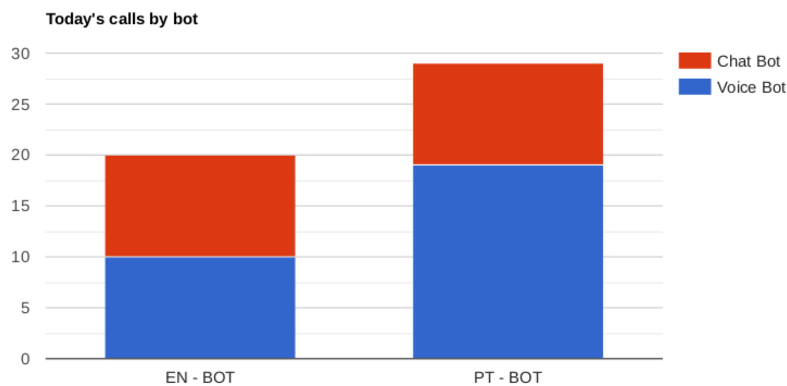

Figure 37: Number of calls for each assistant - day 1

Regarding the EN - BOT, 10 conversations were initiated for each communication flow type, whereas the PT - BOT performed 19 calls to the voicebot, and 10 calls using the chat component. As a result, 40.82% of calls were performed to EN - BOT, and the remaining 59.18% to PT - BOT. Depending on the demographic of the target audience, as well as the objective of each bot, this information is crucial when determining which interfaces and assistants should be prioritized when improving these services.
The daily outcome of each bot is further detailed in their respective column charts, showcased in figures 38 and 39, observing if assistants were able to fulfill their intended purpose.
Differentiated by communication flow type, these charts display the number of conversations where virtual assistants were able to comply through their objectives; the amount of events that resulted in abrupt exits, if these failed midway through the conversation; the number of hand-off occurrences, detecting when a human operator was requested, due to consecutive unsuccessful attempts when capturing user intent, or if this feature was simply requested by the user. Additionally, it also showcases the number of total calls performed to each bot, per communication flow.

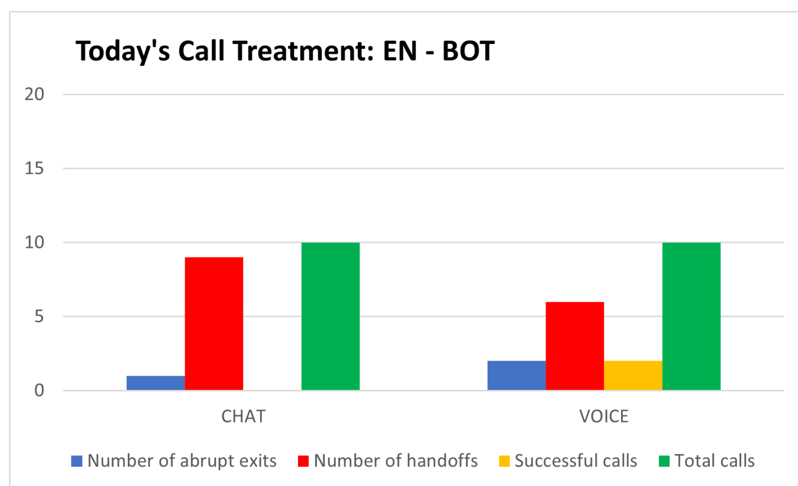

Figure 38: Call treatment for each communication flow, English assistant - day 1

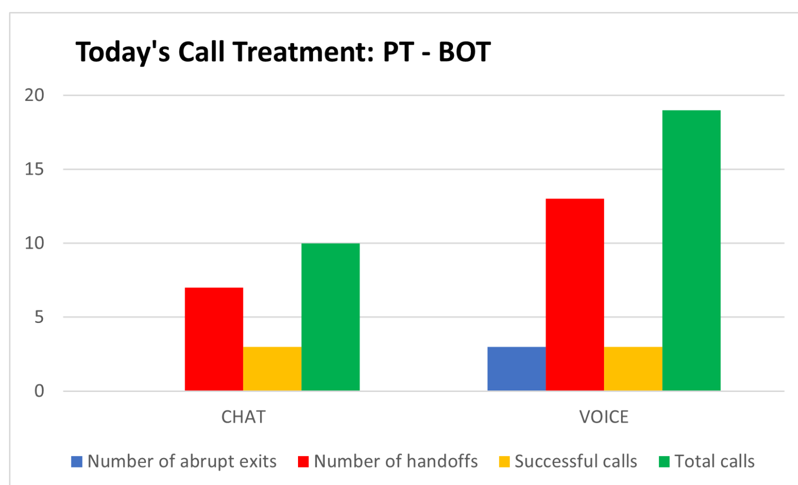

Figure 39: Call treatment for each communication flow, Portuguese assistant - day 1

Following the same scenario where deployed assistants did not possess a large vocabulary, the majority of captured utterances resulted in failure, thus being showcased a large quantity of hand-off occurrences, independently of each interface.

Concerning the English assistant, given the 10 conversations performed through the text-based interface, the browser window was closed a single time during a conversation, resulting in 1 abrupt exit event, and 9 interactions that resulted in hand-off requests.
For the same number of conversations, 2 calls were suddenly dropped in the voice-based English bot, 6 conversations were handed over to a designated operator, and 2 IPTickets were submitted with success. In total, 75% of calls account for hand-off events, interpreted as failure rate, 10% for successful attempts when submitting tickets, and 15% for the remaining occurrences.
As for the PT - BOT, using the chat component led to 7 failed attempts, and 3 user conversations that were able to successfully perform the intended task.
With respect to the voicebot, 3 calls were disconnected in the midst of a conversation, 13 required a human mediator and 3 effectively issued IPTickets. Correspondingly, 68.97% of calls constituted in failed attempts, 20.69% accounts for the assistant's success rate, and the remaining 10.34% for external events.

Having recognized the potential issue, the next phase involves performing a thorough analysis on past interactions using Rasa X, a Conversation-Driven Development tool. When reviewing every step of the conversation, developers are able to correct mistakes that were once performed by the assistant during an interaction, as well as expand its vocabulary by including user-portrayed expressions.
When used in tandem with Rasa X, the dashboard is expected to accompany the gradual improvement of deployed assistants, observing a reduction of conversations that resulted in hand-off events, while simultaneously increasing the number of calls that successfully issued tickets for both communication flow types.

The second, and final part of this scenario is set in a specific time frame, corresponding to the fifth day of results, where both assistants were constantly under review with Rasa X. From the dashboard interface, the viewer is presented with a column chart in figure 40 that describes a time-lapse, containing the number of received conversations for each assistant, throughout all five days.

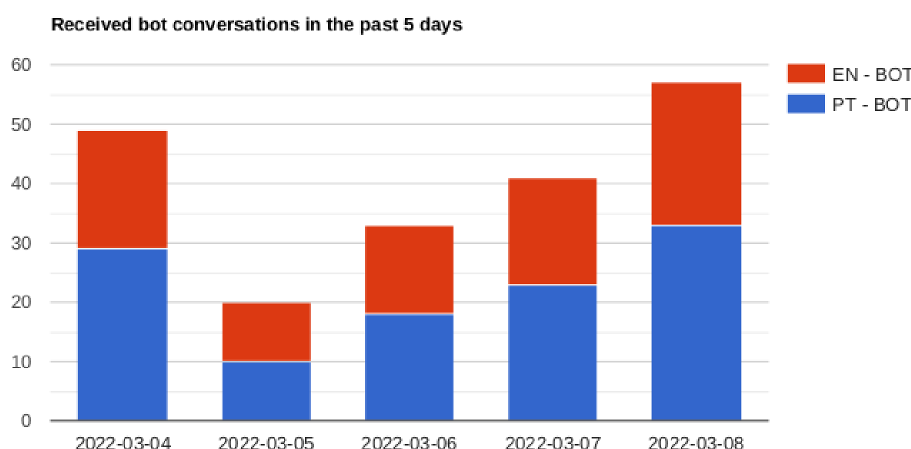

Figure 40: Number of calls to each assistant, per day

During this scenario, every bot was requested in all 5 days, performing a total of 200 conversations. Regarding the EN - BOT, the number of conversations for each day was as follows: 20, 10, 15, 18, and 24. With concern to the PT - BOT, 29 calls were performed on the first day, 10 for the second, with the remaining days having 18, 23, and 33, respectively. In total, 43.5% of calls were directed towards the English assistant, where the remaining 56.5% account for the number of conversations with the Portuguese assistant.

Similarly to figures 38 and 39, the daily outcome of all performed calls for the fifth day are presented in figures 41 and 42, noticing major improvements:

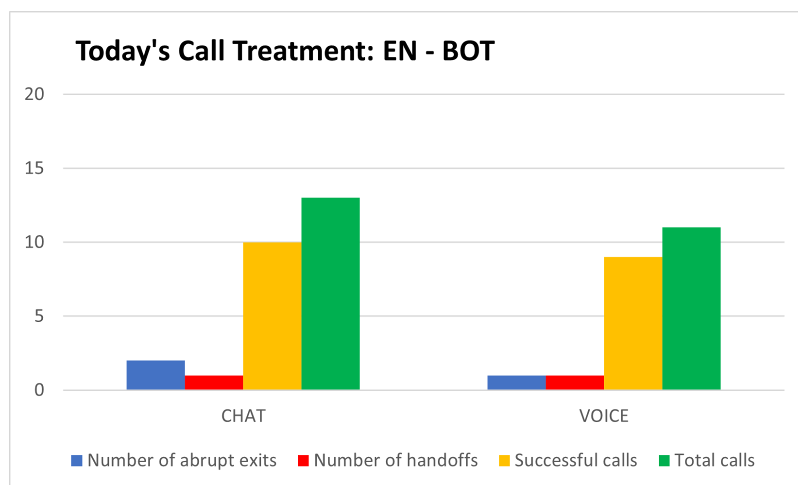

Figure 41: Call treatment for each communication flow, English assistant - day 5

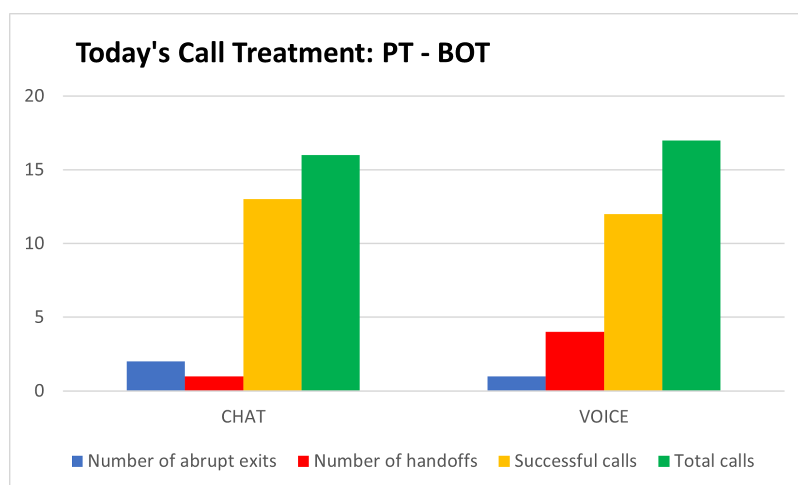

Figure 42: Call treatment for each communication flow, Portuguese assistant - day 5

Comparably to the first scenario, it is perceptible a drastic increase in the number of successful calls, regardless of interface. In the EN - BOT, 3 conversations resulted in sudden disconnections, 2 were handed over to an operator, and 19 calls successfully issued IPTickets, 10 using the chat interface, and the remainder with the voice interface. A total of 24 calls were performed to this assistant.
The same number of abrupt exits occurred in the PT - BOT, although the number of hand-off occurrences was higher, 4 using voice and 1 through chat. Nevertheless, 13 IPTickes were submitted with success in the chat component, and 12 used the voice interface. A total of 33 calls were performed to this assistant.
Regarding the total distribution for both bots, 77.2% accounted for successful calls, 12.3% for hand-off occurrences, and the remaining 10.5% for calls that resulted in sudden disconnections.
Throughout this monitoring process, the duration associated with every conversation was registered. Since the deployed virtual assistants were requested in two distinct communication flows, it becomes necessary to display the average call duration per communication type, observing if the bots performed as expected.
Regarding the EN - BOT, the average chatbot duration was 62.59 seconds, and 80.63 seconds for the voicebot. For PT - BOT, the average length of each conversation resulted in 66.45 seconds and 89.60 seconds, respectively.

Table 5 showcases the total call distribution by the time of day, identifying at what time each bot was requested. From these results, one can observe specific time periods where call traffic was at its highest and lowest. Instead of maintaining power and hardware usage to the maximum, developers can dynamically allocate energy and processing power, depending on previously collected traffic data, predicting instances where requests are supposed to be more frequent or scarce, thus running this system more efficiently.

Table 5: Call distribution by time of day, case study

Hours	**Total requests**	**PT - BOT**	**EN - BOT**
00 - 03h	1	1	0
09 - 12h	82	45	37
12 - 15h	23	12	11
15 - 18h	50	27	23
18 - 21h	27	15	12
21 - 00h	17	13	4

Ultimately, an analysis of the voicebot component was made for all 200 conversations, observing the performance of every major module throughout this experiment, discussing the obtained results. Since the voicebot was used along with the chat interface, only 102 conversations out of the total 200 went through the voicebot module, resulting in 650 registered interactions that happened during the 102 conversations. These interactions will serve as a reference point to the study of voice-based bot performance in Table 6 and figure 43.

Table 6: Voicebot component duration, case study

<table>
<tr><th colspan="6">Average Component Duration (seconds)</th></tr>
<tr><td colspan="2">Speech-to-Text</td><td colspan="2">Rasa</td><td colspan="2">Text-to-Speech</td></tr>
<tr><td colspan="2">1.684</td><td colspan="2">0.889</td><td colspan="2">0.195</td></tr>
<tr><th colspan="6">Minimum and Maximum Component Duration (seconds)</th></tr>
<tr><td colspan="2">Speech-to-Text</td><td colspan="2">Rasa</td><td colspan="2">Text-to-Speech</td></tr>
<tr><td>Min</td><td>Max</td><td>Min</td><td>Max</td><td>Min</td><td>Max</td></tr>
<tr><td>0.948</td><td>5.309</td><td>0.249</td><td>2.747</td><td>0.001</td><td>0.957</td></tr>
</table>

Name	Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Cloud Speech-to-Text API	774	0	915	2,666
Cloud Text-to-Speech API	308	0	170	467

Figure 43: Google Cloud API requests, case study

Regarding the STT component, prior to this analysis, it is worth taking into consideration that, from the conclusions gathered in 4.4.3, when comparing these voicebot components to the same 5 set of phrases (before and after the suggested solutions were implemented and presented in Tables 1 and 3, respectively), the duration of this component is directly correlated to the length of user utterances. When capturing audio from the microphone, background noise may compromise and extend the length of the recording file, which can be explained by having a maximum duration value of 5.31 seconds, which deviates most from the average STT component duration. Overall, the mean performance of this specific component stayed within a reasonable range of values.

When observing the number of requests from the STT component in figure 43, there is a noticeable discrepancy in values. In this study, for the reference amount of registered interactions **n = 650**, Google STT API appoints to **774** requests, which remarks 124 requests left unaccounted. Given that this Google account is not specific for this project, but shared across all IPBRICK projects that already use voice-based virtual assistants, one possible explanation could be those separate projects that were requested during the course of this study (5 days), resulting in this value being unexpectedly higher than the registered interactions, that act as a point of reference. Another possible explanation is given from Google Cloud documentation:

- "If no speech from the supplied audio could be recognized, then the returned results list will contain no items." [41].

In circumstances where a user failed to supply recognizable audio, or if no audio was captured from the microphone, the Google STT performs a request, increasing its value in the Google Cloud Platform, while simultaneously returning an empty list of values. If no text is transcribed, the custom program *system-bot.py* is unable to continue, exiting abruptly before forwarding any content to Rasa, and subsequently to the TTS component. Since component duration values are only inserted at the end of this program, it automatically fails to register these values, therefore not counting as a valid interaction.

Regarding the Rasa component, there was a noticeable increase in mean performance time, given that its story is comparably more complex to the modified standard implementation in 4.4.3. Moreover, the duration of this specific component is substantially higher, due to custom actions that involve external web services and database access. The first is responsible for user authentication and IPTicket submission, which are performed using a SOAP-based web service, which generally is a big factor in causing network traffic, high latency, and processing delay [42]. The latter involves inserting database entries, using a PostgreSQL connection, for event registration purposes that are showcased in the dashboard, naturally imposing an additional delay in the system. These types of custom actions are performed by the assistant throughout the conversation, so it is expected that the average duration of this component is relatively slower. A high discrepancy between the minimum and maximum component duration can be further explained in circumstances where the Rasa component depends, or not, on these external services to formulate a response.

As for the final voicebot constituent, the TTS component behaved accordingly, taking advantage of the audio re-purposing mechanism to cut down on execution time. However, some of the included Rasa responses were considered as dynamic, due to the fact that the virtual assistant would greet a user with the according company name when it performed a successful authentication; or when an IPTicket was submitted, it would convey its identification number to the user. As these responses were critical to an assistant's mode of operation and unique to every conversation, these corresponding audio responses were not stored locally, therefore recurring to Google Cloud services to perform synthesized audio of responses. However, given that there is a constant 1:1 ratio when using a voice-based bot without this re-purposing mechanism (as shown in figure 28), for a total of 650 registered interactions, only 308 TTS requests were performed, resulting in 52,62% of all Rasa responses selecting the appropriate audio file stored in the system, shown in figure 43 (assuming that there was no external intervention from other voice-based virtual assistants during this study).

6 Conclusion and Future Work

6.1 Accomplished Goals

The purpose of this dissertation was to incorporate a solution, focusing primarily on the subject of communication flow automation within the IPBRICK OS by enhancing its Artificial Intelligence subsystem. In regards to all performed developments, in the course of the intended implementation phase at IPBRICK, these were comprised into four major categories, to which this chapter is dedicated, carefully addressing the respective concluding remarks, specifying the extent up to which the goals of this dissertation were achieved.

In order to automate a task inherent to IPBRICK services, the first objective was dedicated to devising a Rasa assistant, with the purpose of reporting IPTickets. This genuine problem served as the baseline ambition for every aspect of this dissertation, where the architecture of a Rasa project is detailed, along with the recommended approach for building virtual assistants using Coversation-Driven Development. Given that Rasa bots are presented as a customizable service within the IPBRICK OS, it is expected that every different implementation will target very distinct and specialized segments of each market, for particular products that could be improved with automated services. Therefore, the devised assistants are served as a generalized model when providing niche solutions that require building a Rasa bot with access to external services, providing an opportunity for human-aided assistance in a last resort scenario. However, being constrained to a closed environment, the developed Rasa assistants faced little to no exposure to the intended target audience, IPBRICK clients and partners. Due to time and mobilization constraints, this project was not migrated to a live environment for real-world results, recurring to simulated events.

The second major objective was set to enhance the communication flow of voice-based bots. Being introduced to a preliminary implementation, an exposure of its underlying architecture was performed, proposing a targeted solution concerning all major elements that improved this system. Regarding the Text-to-Speech component, the focus was primarily directed towards voice and intent recognition, presenting a language-detection mechanism. In spite of being an elementary task, it ensures that all necessary information is extracted accurately, including the addition of domain-specific expressions, being able to detect language provided as an input, redirecting the conversation to an adequate recipient for response-generation.
Maintaining the same scope for voice-based assistants, the inclusion of an audio re-purposing mechanism in the Speech-to-Text component resulted in an approach that greatly affected performance by reducing execution time, effectively yielding results in a short span of time. This stands as a highly beneficial solution, reintroducing previously used elements back to the system under a new file-naming convention, reducing the number of requests performed to external services and consequently improving company revenue by diminishing monetary fees associated with these requests, while simultaneously promoting file organization and interpretation for both man and machine.

Enhancing IPBRICK's AI subsystem by developing a chat-based communication interface was the third major aim of this work. For this purpose, the concept of adaptability and modularity was set, arranging a solution that could be implemented on any website, without being too intrusive or disrupting the source code of target web pages, thus deciding on a simple pop-up interface to display a chatbox. Given the overall context of this solution, apart from communicating with virtual assistants, another essential feature was the support for human-to-human communication, enabling users to communicate with two distinct domains through an interface that transparently interchanges between recipients, naturally maintaining the flow of the conversation and quality of service. In addition, apart from being implemented as a major solution to this dissertation, all necessary arrangements were made to convert this specific module into a bundle-like solution, being presented to IPBRICK clients as a proof of concept project, demonstrating its feasibility and potential. This chatbot bundle was eventually implemented in Hospital

das Forças Armadas, deploying the chatbot interface along with their virtual assistant.

Creating a dashboard interface, along with the respective database dedicated to event registration was one of the goals required in this dissertation. This fourth major objective was demonstrated during the simulated experiment, proving to be an effective tool when interpreting information retrieved from data sources, playing an essential role in showcasing the obtained results through a clear and understandable interface. Being extremely useful in determining the performance of deployed bots in the system, it allows developers to identify possible focal points for future improvement, as well as granting the ability to accompany the gradual evolution of deployed AI solutions by observing the end results for each conversation. Moreover, it is also possible to predict call traffic by observing previous time frames where virtual assistants were more frequently requested, allowing to dynamically allocate energy and hardware resources, promoting efficiency in this system.

On an ending note, every goal proposed at the beginning of this dissertation addressed multiple modules that were unique from one another, thus requiring undivided attention when exposing their intricate structure, before figuring out the best course of action. Some existing solutions were refined, implementing new strategies that positively contributed to one's system, as well as developing interfaces that provided practical solutions for text-based communication, and supplied a clear view regarding data analysis. From initially being separate modules, all implementations ultimately converged into an interconnected subsystem that uses AI at its core, concluding that the main goal was accomplished to a very great extent.

6.2 Future Work

This section suggests some improvements concerning every major aspect of this work, discussing future approaches that could benefit every system that was addressed in this dissertation.

During the implementation phase, Rasa Open Source suffered a major upgrade by releasing its newest version, Rasa 3.0. Comparably to Rasa 2.X, the version that was used throughout the dissertation, this major upgrade adopted new syntax and semantics when creating virtual assistants. Drastically changing the way bots are built, it implies that migrating existing projects to 3.0 would cause dependency incompatibilities, as bigger systems that integrate several packages usually rely on tight specifications, preventing older projects to be updated since these only work under strict conditions. Thus, this new Rasa version should be explored in the future, since it provides new features that enrich virtual assistants. Another suggested approach is concerned with Rasa's method of entity extraction and intent recognition through its modular pipeline. In this work, little to no changes were performed to the pipeline, as a way to demonstrate a functional service with its default implementations, for the most part, as if one was being implemented to an end client. However, these intricate components can be adjusted to one's liking, and could directly affect Rasa's performance. Therefore, an additional study should be made, observing how finely tuning every component in the pipeline would affect the performance of a given assistant, finding the best approach for every use case.

Regarding voicebot improvements, one key structural hindrance detected was present in the Speech-to-Text component. For every spoken input, the sound captured with the microphone was stored into an audio file. Before being sent to Google Cloud Platform to perform text conversion, a big portion of the execution time was dedicated solely to reading the audio file, saving this stream of content into a variable, subsequently being sent to an external service for processing. Instead of relying on temporary audio files, a proposal is to perform a massive overhaul to the voicebot structure, introducing a live transcription mechanism that could perform audio-to-text conversion on the fly, instead of relying on audio files that must be read in order to send user utterances. Google Cloud Platform currently supports

a live transcription mechanism, and adopting this feature is certain to yield far better performance values to voice-based services.
In the Text-to-Speech component, the re-purposing mechanism proved to be a massive contribution to this system, effectively reducing execution time by re-using strict messages from the virtual assistant. However, this is not a silver-bullet solution, also presenting some restrictions. Given that this mechanism revolves around file nomenclature for file path mapping, it is worth mentioning that every operating system imposes a limit to a file's name, ranging from 255 to 260 characters. Despite being a reasonably healthy number, if a virtual assistant is configured to provide an extended response, the operating system is unable to save this file, therefore no audible response is presented to the user. In order to support longer messages, the mechanism responsible for normalizing bot utterances should be improved in a way that reduces the number of resulting characters, while still being identifiable by both man and machine.
Contrarily to the implemented chatbot solution, the current voicebot approach does not provide a conversational context in the act of performing hand-offs. Although this mechanism is done in the chat interface, by storing all interactions in browser memory, sending all interactions upon the detection of a hand-off intent, Rasa is able to track conversation history through its *Tracker* component. In a similar fashion to the chatbot's mode of operation during hand-offs, one suggestion could be sending a contextual text message to the designated human operator, with recourse to Rasa *Tracker*. This would imply providing ejabberd support within the voicebot solution, resulting in two communication streams working in parallel to improve quality of service to voice-based systems.

Regarding the chatbot component, this project could be further extended to support additional IPBRICK features. In order to provide quality-of-life improvements, one suggestion could be the implementation of *avatar-switching*. In the proposed solution, the chatbot component displays an avatar of IPBRICK's AI solution, Zuri. However, in the event of a possible hand-off, it could display the profile picture of the designated operator instead, clarifying that the user is engaging in interpersonal communication.
Moreover, in the same way that all necessary arrangements to the voicebot component are configured when devising a virtual assistant in the IPBRICK OS interface, a similar automatic procedure could be performed in the chatbot bundle, automatically configuring all necessary parameters that make chat-based conversations possible. For further customization, it is suggested that a separate web page within the IPBRICK OS is dedicated to the chatbot component, in order to easily perform additional configurations.

With respect to the dashboard interface, this implementation can also be improved, like any other system. When accessing the dashboard web page, all interactive components are rendered by performing strict SQL queries that are previously defined. A dynamic mechanism involving queries could be implemented, where the client uses a filter to access data that was obtained for a given date, or for a certain period of time, in order to better identify a gradual evolution of deployed virtual assistants. Similar to the chatbot module, this could also be included in a dedicated web page inside IPBRICK OS, making all necessary arrangements to automatically identify deployed virtual assistants and display their respective statistics.

On a final note, when implementing the commercial support bot, all IPBRICK-related web services were requested using SOAP. When requesting these web services in the Rasa custom actions, there was some difficulty in accessing these using Python, as multiple attempts with several external dependencies were required in order to make this possible. In the end, despite providing a functional service, the performance was found to be considerably slower. Transitioning to a RESTful service seems to be the best solution, as REST allows a greater variety of data formats, as well as using less bandwidth in general terms, resulting in faster service.

7 References

References

[1] P. Kulkarni, N. Sirsikar, A. Mahabaleshwarkar, K. Gadgil, and M. Kulkarni, "Conversational AI: An Overview of Methodologies, Applications & Future Scope," in *5th Int. Conf. On Computing Communication, Control And Automation (ICCUBEA)*, 2019, doi: 10.1109/ICCUBEA47591.2019.9129347.

[2] "Your Guide to Five Levels of AI Assistants in Enterprise — The Rasa Blog — Rasa." http://rasa.com/blog/conversational-ai-your-guide-to-five-levels-of-ai-assistants-in-enterprise/ (accessed Mar. 09, 2022).

[3] A. M. Turing, "Computing Machinery And Intelligence," in *Mind*, vol. 59, no. 236, pp. 433-460, Oct. 1950, doi:10.1093/mind/LIX.236.433.

[4] M. McTear, "Conversational AI: Dialogue Systems, Conversational Agents, And Chatbots," in *Synthesis Lectures on Human Language Technologies*, vol. 13, no. 3, pp. 1-251, 2020, doi:10.2200/S01060ED1V01Y202010HLT048.

[5] J. Jia, "The Study of the Application of a Keywords-based Chatbot System on the Teaching of Foreign Languages," Oct. 2003. [Online]. Available: http://arxiv.org/abs/cs/0310018

[6] R. Csaky, "Deep Learning Based Chatbot Models," Oct. 2017. [Online]. Available: http://arxiv.org/abs/1908.08835.

[7] J. Weizenbaum, "ELIZA — A Computer Program For the Study of Natural Language Communication Between Man And Machine," in *Commun. ACM*, vol. 26, no. 1, pp. 23–28, Jan. 1983, doi: 10.1145/357980.357991.

[8] J. Cahn, "CHATBOT: Architecture, Design, & Development," M.S. thesis, SEAS, UPenn, Philadelphia, United States of America, 2017. [Online]. Available: http://www.academia.edu/37082899/CHATBOT_Architecture_Design_and_Development

[9] E. Adamopoulou and L. Moussiades, "Chatbots: History, technology, and applications," in *Mach. Learn. with Appl.*, vol. 2, 2020, doi: 10.1016/j.mlwa.2020.100006.

[10] E. Adamopoulou and L. Moussiades, "An Overview of Chatbot Technology," in *IFIP Int. Conf. on Artificial Intelligence Applications and Innovations*, pp. 373-383, 2020, doi:10.1007/978-3-030-49186-4_31.

[11] M. Marietto et al., "Artificial Intelligence Markup Language: a Brief Tutorial," in *Int. J. Comp. Sci. Eng. Survey*, 2013. [Online]. Available: http://arxiv.org/abs/1307.3091

[12] Z. Yan et al., "DocChat: An information retrieval approach for chatbot engines using unstructured documents," in *Proc. 54th Annu. Meet. Assoc. Comput. Linguist. ACL 2016 - Long Pap.*, vol. 1, pp. 516–525, 2016, doi: 10.18653/v1/p16-1049.

[13] O. Vinyals and Q. Le, "A Neural Conversational Model," in *Proc. 31st Int. Conf. on Machine Learning*, 2015, doi: 10.48550/arXiv.1506.05869.

[14] A. Ritter, C. Cherry, and W. B. Dolan, "Data-driven response generation in social media," in *Proc. 2011 Conf. Empir. Methods Nat. Lang. Process.*, Jul. 2011, pp. 583–593. [Online]. Available: http://aclanthology.org/D11-1054/

[15] K. Cho, B. V. Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, ”Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation,” in *Proc. 2014 Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Oct. 2014, pp. 1724–1734, doi: `10.3115/v1/d14-1179`.

[16] M. McTear, Z. Callejas, and D. Griol, textitThe Conversational Interface, Springer, 2016.

[17] Z. Yu, Z. Xu, A. W. Black, and A. I. Rudnicky, “Strategy and Policy Learning for Non-Task-Oriented Conversational Systems,” in *Proc. 17th Annu. Meet. Spec. Interes. Gr. Discourse Dialogue*, Sep. 2016, pp. 404–412, doi: `10.18653/v1/w16-3649`.

[18] M. J. Pereira and L. Coheur, “Just.Chat - a platform for processing information to be used in chatbots,” 2013. [Online]. Available: `http://www.semanticscholar.org/paper/Just.Chat-a-platform-for-processing-information-to-Pereira/b5a9eb3d4b82936b556150165c961984b21fba09#paper-header/`

[19] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, “Rasa: Open Source Language Understanding and Dialogue Management,” in *NIPS Workshop on Conversational AI*, 2017. [Online]. Available: `http://arxiv.org/abs/1712.05181`

[20] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, “Bag of Tricks for Efficient Text Classification,” in *Proc. 15th Conf. Eur. Chapter Assoc. Comput. Linguist.: Volume 2*, Apr. 2017, pp. 427–431, doi: `10.18653/v1/e17-2068`.

[21] “Microsoft Bot Framework.” `http://dev.botframework.com/` (accessed Jul. 02, 2021).

[22] “Bot Framework SDK documentation.” Microsoft Docs. `http://docs.microsoft.com/en-us/azure/bot-service/index-bf-sdk?view=azure-bot-service-4.0` (accessed Jul. 02, 2021).

[23] “Dialogflow CX basics.” Google Cloud. `http://cloud.google.com/dialogflow/cx/docs/basics/` (accessed Jul. 02, 2021).

[24] “What it Botpress?.” Botpress. `http://botpress.com/docs/` (accessed Jul. 02, 2021).

[25] S. Williams, *Hands-On Chatbot Development with Alexa Skills and Amazon Lex*, 1st ed., Packt Publishing, 2018.

[26] “Página Inicial - IPBRICK.” `http://www.ipbrick.com/pt-pt/` (accessed Feb. 03, 2022).

[27] “Get Started Asterisk.” `http://www.asterisk.org/get-started/` (accessed Dec. 27, 2021).

[28] L. Madsen, J. Van Meggelen, and R. Bryant, *Asterisk: The Definitive Guide*, 3rd ed. United States of America: O’Reilly Media, Inc., 2011.

[29] “Google Cloud overview.” Google Cloud. `http://cloud.google.com/docs/overview/` (accessed Dec. 28, 2021).

[30] A. Gupta, P. Goswami, N. Chaudhary, and R. Bansal, ”Deploying an Application using Google Cloud Platform,” in *2020 2nd Int. Conf. Innovative Mechanisms for Industry Applications (ICIMIA)*, Mar. 2020, pp. 236-239, doi: `10.1109/ICIMIA48430.2020.9074911`.

[31] J. Sidna, B. Amine, N. Abdallah, and H. El Alami, “Analysis and evaluation of communication Protocols for IoT Applications,” in *SITA’20: Proc. 13th Int. Conf. Intelligent Systems: Theories and Applications*, Sep. 2020, doi: `10.1145/3419604.3419754`.

[32] J. Moffitt, *Professional XMPP Programming with JavaScript and jQuery*, 1st ed. Indianapolis: Wiley Publishing, Inc., 2010.

[33] J. Rosenberg et al., *SIP: Session Initiation Protocol*, 2002. [Online]. Available: http://www.hjp.at/doc/rfc/rfc3261.html#page_8

[34] "Introduction." ejabberd Docs. http://docs.ejabberd.im/admin/introduction/ (accessed Dec. 29, 2021).

[35] "ejabberd XMPP Server with MQTT Broker & SIP Service." ejabberd. http://www.ejabberd.im/ (accessed Dec. 29, 2021).

[36] "KoolReport Examples & Demonstration." KoolReport. http://www.koolreport.com/examples/ (accessed Dec. 29, 2021).

[37] "Rasa Architecture Overview." Rasa Docs. http://rasa.com/docs/rasa/2.x/arch-overview/ (accessed Jan. 24, 2022).

[38] A. Phillips and M. Davis, *Tags for Identifying Languages*, 2009. [Online]. Available: http://tools.ietf.org/search/bcp47

[39] "react-chat-popup." npm. http://www.npmjs.com/package/react-chat-popup/ (accessed Mar. 01, 2022).

[40] "GitHub - legastero/stanza: Modern XMPP, with a JSON API." GitHub. http://github.com/legastero/stanza/ (accessed Mar. 01, 2022).

[41] "Speech-to-Text basics — Cloud Speech-to-Text Documentation." Google Cloud. http://cloud.google.com/speech-to-text/docs/basics#responses/ (accessed Mar. 08, 2022).

[42] S. Mumbaikar and P. Padiya, "Web Services Based On SOAP and REST Principles," in *Int. J. Sci. Res. Publ.*, vol. 3, no. 5, May 2013. [Online]. Available: http://www.ijsrp.org/research-paper-0513/ijsrp-p17115.pdf