Application & Service Support Bots

Pedro Avilar Calado Barco

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Manuel da Costa Alves Marques

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. José Manuel da Costa Alves Marques
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

October 2019
Abstract

Using chat support services promises huge cost reductions in human resources and can facilitate communication and even foster customer engagement through the use of natural language. But beyond the promising advantages are also identified challenges: identifying the entity and action in natural language requests, identifying the need to redirect request to a human operator (handover) or system, their articulation with knowledge systems-based, are all obstacles that have to be overcome in order to ensure the correct adoption of this communication channel.

This thesis surpasses those challenges and applies an application and service support bot to a company use case, adding another support channel for service desk applications.

Keywords

Chatbot; Application and Service Support; Artificial Intelligence; Virtual Assistant; Cloud-native;
Resumo

A utilização em serviços de suporte de chatbots promete enormes reduções de custos em recursos humanos, podendo ainda facilitar a comunicação e até mesmo fomentar o comprometimento do cliente através da utilização de linguagem natural. Mas para além das promissoras vantagens estão igualmente identificados desafios: a identificação da entidade e acção nos pedidos em linguagem natural, a identificação de necessidade de redirecionar pedido para um operador humano (handover) ou sistema, a respectiva articulação com sistemas de knowledge-base, são todos obstáculos que têm de ser ultrapassados de forma a garantir a correcta adopção deste canal de comunicação.

Esta tese supera estes desafios e aplica um chatbot de suporte aplicacional e de serviço a um caso de uso de uma empresa, adicionando um outro canal de suporte a aplicações de service desk.

Palavras Chave

Chatbot; Suporte Aplicacional; Inteligência Artificial; Assitente Virtual; Cloud-native;
Contents

1 Introduction .................................................. 1
  1.1 Defining “Chatbot” ........................................... 3
  1.2 Motivation .................................................. 3
  1.3 Objectives .................................................. 4
  1.4 Context .................................................... 4
  1.5 Outline .................................................... 5

2 Background ................................................... 7
  2.1 Turing Test .................................................. 9
  2.2 History of Chatbots ......................................... 9
  2.3 Utterances, Intents, and Entities .......................... 10
  2.4 Edge Cases: Hijack and Handover ......................... 11
  2.5 Architecture Model ........................................ 13
  2.6 State of the Art ............................................ 15
    2.6.1 Chatbot Platforms .................................... 15
    2.6.2 Serverless Computing ................................ 16
  2.7 Integrations ............................................... 16
    2.7.1 Databases ............................................. 16
    2.7.2 Ticketing Systems ..................................... 17

3 Requirements .................................................. 21
  3.1 Problem Description ....................................... 23
  3.2 Goals ..................................................... 23
  3.3 Use Cases ................................................ 25

4 Methods ....................................................... 29
  4.1 Development Process ..................................... 31
  4.2 Development Environment ................................ 33
  4.3 Conversation Lifecycle .................................... 38
5 Results and Discussion
   5.1 Content Evaluation ................................................................. 43
   5.2 Functional Evaluation ............................................................. 45

6 Conclusion
   6.1 Conclusions .............................................................................. 51
   6.2 System Limitations and Future Work .......................................... 52
List of Figures

2.1 Conversation example: analyzing intents, entities and contexts .......................... 12
2.2 Bot to human handover ................................................................. 13
2.3 Chatbot architecture overview ....................................................... 14
3.1 Chatbot application providing content to different audiences ...................... 24
3.2 FAQs conversation flow .................................................................. 26
3.3 Submit Incident conversation flow .................................................... 27
3.4 Incident Status conversation flow ..................................................... 28
4.1 Project Architecture: a Google cloud-native approach ................................. 34
4.2 Edoclink’s chatbot widget ................................................................. 35
4.3 Edoclink’s operator backoffice with live conversations ............................... 37
5.1 Session flow based on 30 days of session data ........................................ 47
5.2 Middleware service request latency graph (in 7 days) ................................. 48
5.3 Middleware service cold start latency graph (in 7 days) ................................. 48
List of Tables

2.1 Comparison between popular chatbot platforms. ................................. 18
2.2 Comparison between SQL and NoSQL databases ............................... 19
2.3 Comparison between different NoSQL database models ....................... 19

3.1 List of considered use cases ................................................................. 25

5.1 Intents handled by the conversational agent ........................................ 44
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>NLU</td>
<td>Natural Language Understanding</td>
</tr>
<tr>
<td>NLG</td>
<td>Natural Language Generation</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>FAQ</td>
<td>Frequently Asked Question</td>
</tr>
<tr>
<td>GCP</td>
<td>Google Cloud Platform</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>JWT</td>
<td>JSON Web Token</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
</tbody>
</table>
Introduction

Contents

1.1 Defining "Chatbot" .................................................. 3
1.2 Motivation ............................................................ 3
1.3 Objectives ............................................................. 4
1.4 Context ................................................................. 4
1.5 Outline ................................................................. 5
This chapter starts with a simple definition of the term “chatbot”, presents a brief motivation regarding application and service support bots, introduces the target service in which this thesis is developed, proposes objectives, and summarizes the organization of the document.

1.1 Defining “Chatbot”

Oxford University Dictionary defines a chatbot as follows [1]:

**chatbot (n):** A computer program designed to simulate conversation with human users, especially over the Internet.

Also known as a “conversational agent”, or solely “agent”, a chatbot resembles a conversational turn-based game: each user query triggers a chatbot response (user - chatbot - user - chatbot ...), similarly to a chess game. This program is powered by Artificial Intelligence (AI) and Machine Learning (ML) tools which parse inputs, via text or audio, and converts tokens into information to reply accordingly.

1.2 Motivation

In traditional online experiences, websites are hard to navigate and customers are not getting answers to simple questions or basic details about a business need.

The growth of artificial intelligence is making progress towards consumers getting technological help in their everyday tasks. Internet users are accustomed to getting information by searching it online and this behaviour has changed the way services are expected to be delivered. A chatbot is an application that connects a customer to a variety of services, while being always available, reliable and easy to use.

In this recent growth, since 2016, chatbots are being implemented in almost every business: providing news, ordering food, advertising a product, supporting users, and other services. Ultimately, if a service is being delivered, it can be provided with a chatbot interface.

Application and service support is one of the themes regarding user experience in online services. This support includes application troubleshooting and ticket management often delivered by the help desk software. However, this solution may not offer the automation to further improve a business user experience. When users are resolving a complaint or problem, a chatbot may provide the automation needed to refine application and service support by filtering out the issues that must be created and those that can be answered through a trained knowledge base.

Although a chatbot may prove to be an advantageous complement to ticket management, the proper development of such a channel poses challenges. In this regard, building and developing a chatbot has
become a non-trivial implementation in which thorough testing and validation are necessary to deliver an application and service support channel that makes issue handling a simpler and easier experience.

### 1.3 Objectives

Similarly to microservice architecture, state-of-the-art chatbot technology is often being delivered as a small service that can be independently used with other services and may fit in various architectural models. Fomenting a cloud-native approach where data and functionality are exposed as a collection of loosely coupled services in a cloud environment, this thesis aims at building a cloud-native bot solution which leverages another support channel for service desk applications.

Businesses can benefit from a chatbot, as this solution is promised to be always available, faster and more cost-effective compared to a human operator, ultimately captivating its users for its efficiency. However, heavy testing should assess the benefits of having such a bot, being a well-behaved agent and performant between and throughout the different service integrations. Therefore, another aim of this research is to determine if this chatbot is ready for use in a professional environment.

### 1.4 Context

This thesis presents a practical case study of a chatbot solution for a company. Edoclink was the case study of choice given it is a Link Consulting product which has an application and service support software. The following paragraphs introduce the product, describe each component in detail, and explain the relevance of a conversational agent in this business.

The widespread computerization of companies and society has kept the promise of wiping out “paper” documents, but not only has it not ended, despite profound changes in their use, but it has also spawned many ways of creating and transmitting information in digital format. In organizations, documents are the way to record what is done. However, similarly to paper, governing virtual documents is difficult and requires an adequate methodology to prevent files from being lost or forgotten.

Edoclink is an application developed by Link Consulting that provides all document management functions, mainly focusing on documents and their life, besides saving and searching. This document and workflow management system enables organizations to store, manage, control, organize and track their electronic documents and processes. The information stored in the repository is available with quick and easy access. Edoclink has a modular structure that seeks to optimize the three main layers of a complete document management system.

**Digital Archive Module** is the core of the document repository functionality. It ensures all document
archiving support functions granting scalability, security, and high availability.

**Document Management Module** performs document management functions, also made available through an Application Programming Interface (API) that can be used by other applications.

**Document Management Portal Module** corresponds to the user interface providing the pages present in the modules described above.

This tool is linked to an external helpdesk service, namely Vision Helpdesk, which provides ticket management for a managed resolution of application problems; troubleshoot guides for common problems are available on the user's manual, inside the authenticated user's Vision Helpdesk portal [2].

To improve its user experience, Edoclink is looking for automating FAQs for troubleshooting problems and proposing a finer and friendlier User Interface (UI) regarding helpdesk ticket management. A chatbot was believed to be a solution, because it is always available to help users with their issues, while continually enhancing its knowledge base with training.

### 1.5 Outline

This thesis consists of 6 chapters, organized as follows. Chapter 2 briefly revisits the history of chatbots and explains background theory such as chatbot's architecture and conversational terms. Then, chapter 3 addresses the requirement assessment for this project by identifying the problem, goals, and use-cases. Chapter 4 explains the methodological approach and implementation per component and service used, followed by the test results and discussion in chapter 5. Finally, chapter 6 concludes this paper by reviewing and purposing future work.
2 Background

Contents

2.1 Turing Test .................................................. 9
2.2 History of Chatbots ........................................... 9
2.3 Utterances, Intents, and Entities ................................. 10
2.4 Edge Cases: Hijack and Handover .............................. 11
2.5 Architecture Model ........................................... 13
2.6 State of the Art ............................................... 15
2.7 Integrations .................................................. 16
Although it may seem the term “chatbot” has only been introduced recently, it has a much longer history [3]. This chapter aims at presenting a brief history of chatbots, explaining the fundamental concepts in a chatbot conversation, its architecture and which are the state-of-the-art technologies used in application and service support bots.

2.1 Turing Test

The Turing Test is a technique of investigation in AI to determine whether a computer is capable of responding like a human being. The test is named after Alan Turing, the pioneer of Artificial Intelligence. Turing launched the test in his 1950 article called “Computing Machinery and Intelligence” at the University of Manchester [3].

In his paper, Turing describes his test as a different version of “The Imitation Game”: a simple party game involving three players. Player A is a man, player B is a woman and player C is the interrogator. In the Imitation Game, player C is unable to see either player A or player B, and can only communicate with them through written notes. By asking questions of player A and player B, player C tries to determine which of the two is the woman. Player A’s role is to trick the interrogator into making the wrong decision, while player B attempts to assist the interrogator in making the right one. Turing changed the concept by introducing an AI as player A. This time, the interrogator’s role is to decide which player is the human.

As chatbots and digital assistants proliferate, one is often chatting with machines. This training and learning have an impact on the increasing amount of bots able to pass the Turing Test.

2.2 History of Chatbots

Chatbot’s history started with ELIZA in 1966, when Joseph Weizenbaum’s program fooled users into believing that they were talking to a real human. ELIZA’s operated by recognizing cue words or phrases in the input, giving, as an output, a template response that could move the conversation along. For example, the word “mother” would trigger the phrase “Tell me more about your family” [4]. Quoting ELIZA’s creator, “artificial intelligence ... machines are made to behave in wondrous ways, often sufficient to dazzle even the most experienced observer. But once a particular program is unmasked, once its inner workings are explained ... its magic crumbles away” [4].

In 1972, Parry was created at Stanford University [5]. Parry was a modified version of ELIZA but with an attitude, which meant it was getting close to be a human. During a Turing test, qualified psychiatrists were only able to distinguish 48% of responses from a genuine human and Parry.

While Parry and ELIZA were exclusive outputting text, a new bot, Jabberwacky, was released as the first bot to simulate human voice. Moreover, this chatbot was able to learn new responses and get context
based on real-time events, rather than replying statically.

In 1995, the most famous chatbot of 20th century was created, inspired by ELIZA. A.L.I.C.E (Artificial Linguistic Internet Computer Entity), also referred to as Alicebot or Alice, was the first chatbot to heuristically apply pattern matching rules to user’s input. Although it won the Loebner Prize (an artificial intelligence competition that awards prizes to programs considered by the judges to be the most human-like), this bot failed to pass the Turing test.

Later, every big company joined the chatbot movement. In 2006, IBM created Watson using natural language processing [6]. In 2010, Apple presented Siri, which answers user questions and performs web service requests. In 2012, Google Now was presented by Google: a mobile app developed to answer questions, make recommendations, and perform web requests just as Siri. In 2015, Alexa and Cortana joined. Alexa is an Amazon chatbot which uses Natural Language Processing (NLP) to recognize and respond to voice commands. Cortana is a Microsoft product which acts as a personal assistant using NLP and Bing services to answer user questions and set reminders. In 2016, the trend hits Facebook with Bots for Messenger. Facebook launched a platform to allow developers to create bots that can be deployed and delivered within Facebook’s chat interface to Facebook’s users [7, 8].

With 14 years of development, Mitsuku is considered to be the world’s best conversational chatbot. Created by Steve Worswick, this bot is a four-time Loebner Prize winner (in 2013, 2016, 2017, 2018) [9].

Today, chatbots use a similar technique which takes this notion of cue words, phrases, answers, personality together with machine learning.

2.3 Utterances, Intents, and Entities

Interacting with a chatbot is similar to a conversation: each user phrase has an appropriate response. Upon receiving a new message, the agent must be able to identify the user’s intention along with important parameters. This behavior is only achievable with AI and ML mechanisms and NLP tools. Such terms are explained in the following sections and illustrated in Figure 2.1.

Utterances

Utterances are user sentences.

Being an AI program, a chatbot is subject to training or testing utterances. These training phrases are a significant factor in determining a bot’s precision. Usually, more than 10 utterances should be used to train each conversational use case.
Intents

Intents are intentions. A chatbot program uses phrases, whether via text or audio, to get intentions. Intents can either represent “small talk” or company use cases and can be described by a developer or provided by the chatbot platform. If no intention is identified, the chatbot triggers a special intent called “Fallback Intent” that represents every user request without well-defined intention identified. The more utterances a chatbot uses as training, the better it is to correctly identify which intent the user has.

Entities

Entities are parameters. When programming, a function with several parameters could be created and defined by their type and requirement. Just like functions, intents can be described by its utterances and entities. When creating an intent, new entities may be defined as required; if so, that intent should only be triggered whenever all required parameters are present. There are typically two types of entities: system and custom. System entities are the default entities created and managed by the chatbot platform and its support and extensiveness vary between providers. Custom entities are created by developers and should be used for values that are not covered by system entities, or for objects or concepts that can have several different attributes (a developer composite containing other entities as aliases).

Contexts

Context gives control over what happens to the conversation. A chatbot conversation resembles a state machine where contexts are machine/conversation states. Each intention may have an input context, which is required to trigger that intent, and an output context, which is created when the intent is successfully triggered and exited.

2.4 Edge Cases: Hijack and Handover

Even if a chatbot is made to grant application and service support a specific and well defined goal, edge cases typically exist. Users need a human operator to handle delicate situations and to get complex problems properly solved, while the bot should be limited to situations that, with automation, can be handled. There are two scenarios in which a human operator may intervene: hijack and handover. Hijack occurs when a human operator proactively takes over a conversation or at the request of a user. This operation disables further chatbot replies and inserts the user into a conversation room, where it will be handled by a human.
An handover request occurs when a user requests for a conversation hijack. This protocol may be manually triggered when asking for help or automatically when, for example, there are several and consecutive fallback intents, intentions that can’t be recognized or handled by chatbots. The handover may notify a pool of operators or a specific one. From then on, operators are entitled to take over control of the conversation hijack (hijack).

Support for handover and hijack may vary, since these protocols are not scoped to chatbot development but rather what messaging platforms they use. Facebook is one of the most popular platforms to support...
2.5 Architecture Model

A chatbot application may be seen as a pipeline stage and is built upon a decoupled architecture with four components: NLU, Dialog Manager, Message Generator and Custom Integrations [11]. The following subsections describe in more detail each component illustrated in fig. 2.3.

NLU

This component is responsible for identifying intents and extracting entities.

Natural Language Understanding (NLU) is a subset of NLP which parses meaning and parameters from inputs while adapting to mispronunciations, misspelled words, abbreviations, synonyms, etc. NLP is the term used to describe the overall process of understanding what is being said, from retrieving an intention (NLU) to replying with an appropriate message.
Dialog Manager

The dialog manager is responsible for validating required parameters and executing actions and is built upon three sub-components: Slots, Tracker and Dialog Management.

Slots collect required parameters by using slot filling, a process in which parameters are filled based on type, name, and context, ensuring that all required intent parameters are present and the intent is ready to be fulfilled.

Tracker creates and manages conversational context and its lifespan: the number of interactions for which a context is alive. As a user interacts with a bot, the remaining lifespan decrements by 1 until it becomes inactive.

If all required parameters are present (Slots), Dialog Management executes the next action based on previous context (Tracker). This action may be a static response or a set of functions and procedures that dynamically generate the next response.

Without NLP and dialog management, machine learning training techniques are not applicable, and an otherwise intelligent chatbot becomes a computer program that performs repetitive tasks [12].

Message Generator

Natural Language Generation (NLG) is a subset of NLP which converts structured data into text using natural language. The Message Generator provides messages created by NLG or by additional
programmed logic, and is responsible for serving chatbot responses, using pre-defined templates with placeholders.

The message generator uses the executed action’s output to build a message appropriate to the use-case, thus resulting in a smart reply. The generated messages may also be randomly selected, making the bot response more human-like.

Dynamically adjusting a response engages customers, enriching their user experience.

**Custom Integrations**

Custom Integrations provide backend logic to chatbots and are often referred to as a fulfillment webhook. This logic is usually related to business use-cases, for example, requesting a service, reading and writing database records, creating a ticket, triggering a handover to a human operator, etc.

Integrations bridge the chatbot core experience with other services and take a fundamental part when building a chatbot, because they allow business context and actions to be used in a conversation, further engaging the customer and improving the user experience.

### 2.6 State of the Art

Rather than developers creating chatbots from scratch, current state-of-the-art allows the development of a chatbot to be based on the choice of the integral components, which are already supported by big companies. Evaluating the requirements and choosing the right alternative depends on what purpose the chatbot is built for. The following sections describe the recommended components when building an enterprise-grade conversational bot [13–15].

#### 2.6.1 Chatbot Platforms

To promote a human-like conversational experience, a chatbot must present a wide range of intents and well-defined entities. A chatbot platform offers a set of prebuilt intents and entities and out-of-the-box integration for messaging channels. Some platforms even offer a selection of domains which can be loaded into a fully working new agent although language support, pricing and limits may vary.

Table 2.1 compares different chatbot platforms based on big companies/players and references from related work [16–18]. It stems from the comparison the following points:

- Amazon Lex offers the largest number of combined prebuilt intents and entities.
- Dialogflow offers the largest number of prebuilt domains and channel integrations, including a web widget demo.
• Wit.ai offers the largest number of supported languages and is a free service with unlimited API calls, although analytics are not an out-of-the-box integration.

• Oracle Digital Assistant is the priciest platform with $0.0035 - $0.0053 per request and offers unlimited API calls.

• English is the only language supported by Amazon and Oracle;

• IBM Watson platform has two supported languages: English and Japanese.

• All platforms are supporting programming languages used in web development such as Python or JavaScript.

2.6.2 Serverless Computing

Serverless computing offers the attractive notion of a platform in the cloud where developers simply upload their code, and the platform executes it on their behalf, removing the need for server software and hardware management by the programmer. Applications are split up into individual functions, which can be invoked and scaled separately.

As discussed in the last section, chatbots often use a fulfillment webhook to add additional logic to a conversation. In this context, a function receives an object representing a “conversational turn” (intent matched, parameters, contexts, session identifier) and, upon validating parameters, managing conversation flow and integrating external services, outputs a coherent response.

Big companies are already supporting serverless computing in the form of functions: Google’s alternative is called Cloud Functions, Amazon offers AWS Lambda and Microsoft is providing Azure Functions [19]. There is also an open-source project, the “fn” project, which allows serverless computing over most cloud and on-premise infrastructures [20].

2.7 Integrations

As an application that later fits into the context in which it is developed, a support chatbot should integrate the components that allow it to help customers. While the previous section introduced the necessary components to build a chatbot, the following section describes the components scoped to an application and service support bot.

2.7.1 Databases

Accessing systems of record from within a dialog is one way of enriching a conversation. For example, a chatbot may use information from the authenticated user to refine entities and responses throughout
an interaction. Different chatbot use-cases may need to store different data and, therefore, a database should be chosen based on what information needs to be stored and how often it is accessed. When providing support services, a user may need to authenticate and an agent will need to store information that identifies a user as such. Moreover, in edge-cases, where operators are intervening, a chatbot must provide a detailed conversation log where users and operators can interact accordingly. Tables 2.2 and 2.3 give a top-down view of different database models with both on-premise and cloud solutions [21, 22].

### 2.7.2 Ticketing Systems

Usually, users encountering issues contact customer service by creating what is called a ticket. A ticket is a data structure similar to a bug report which contains a subject field, where the user summarizes the problem, and a description field. This structure has metadata fields such as id, author, priority, and category, which are used to filter and search for. When a ticket is created, customer service agents are expected to respond to it. However, operators may find that many tickets have a dedicated FAQ or troubleshoot guide which explains it. Moreover, this information is not easily available for everyone and may not be quickly read, since troubleshooting guides give exhausting documentation on various problems and solutions.

Helpdesk ticketing software providers such as Zoho and Zendesk are now offering chatbots to help businesses automate customer interactions. Features may include real-time help, CRM integrations, ticket management and more. However, since an agent is coupled to its helpdesk software, this alternative is not providing nor supporting other ticket management software.

A similar ticketing chatbot service was implemented using the Facebook NLP platform (Wit.ai) and serverless technologies [23]. While the latter project only focuses on implementing a ticketing service for the tourism and travel industry, this thesis is going to use a company case study to build and apply a chatbot to a known helpdesk provider, focusing on creating a knowledge base that helps distinguishing problems that already have an answer and issues that need to be created as tickets.
<table>
<thead>
<tr>
<th>Platform</th>
<th>Dialogflow (Google)</th>
<th>Lex</th>
<th>Watson</th>
<th>LUIS</th>
<th>Wit.ai</th>
<th>Digital Assistant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-built Domains</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-built Intents</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-built entities</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Key Features</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Speech Recognition</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Natural Language Processing</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Integration Capabilities</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Pricing</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
<tr>
<td>Limits for API calls</td>
<td>Facebook</td>
<td>Microsoft</td>
<td>IBM</td>
<td>Google</td>
<td>Amazon</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2.2: Comparison between SQL and NoSQL databases

<table>
<thead>
<tr>
<th></th>
<th>SQL</th>
<th>NoSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td>Relational</td>
<td>Non-Relational</td>
</tr>
<tr>
<td><strong>Data Format</strong></td>
<td>Table</td>
<td>JSON, Key/Value, Column stores, Graphs</td>
</tr>
<tr>
<td><strong>Schema</strong></td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td><strong>Transaction</strong></td>
<td>ACID properties</td>
<td>Varies per solution</td>
</tr>
<tr>
<td><strong>Consistency &amp; Availability</strong></td>
<td>Strong consistency enforced Consistency prioritized over availability and performance</td>
<td>Consistency, availability and performance can be traded to meet the needs of the application (CAP theorem)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Insert and update performance is dependent upon how fast a write is committed</td>
<td>Performance can be maximized by reducing consistency, if needed</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Vertical - more server resources</td>
<td>Horizontal - data partitioning</td>
</tr>
</tbody>
</table>

### Table 2.3: Comparison between different NoSQL database models

<table>
<thead>
<tr>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>USE CASE</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key/Value Based</td>
<td>Matches keys with values, similar to a dictionary, No structure nor relation.</td>
<td>Caching, queueing, distributing information/tasks, keeping live information</td>
<td>Redis, Memcached, Amazon DynamoDB</td>
</tr>
<tr>
<td>Column Based</td>
<td>Two dimensional arrays whereby each key has one or more key/value pairs attached to it. Very large and un-structured data.</td>
<td>Keeping unstructured, non-volatile information, scaling</td>
<td>Cassandra, Google BigTable</td>
</tr>
<tr>
<td>Document Based</td>
<td>Any complex and arbitrary structure can form a document. They allow much deeper nesting and complex structures to be achieved</td>
<td>Nested Information, JavaScript friendly (JSON)</td>
<td>MongoDB, Amazon DynamoDB, Cloud DataStore</td>
</tr>
<tr>
<td>Graph Based</td>
<td>Tree-like structures (e.g. graphs) with nodes and edges connecting each other through relations with clear boundaries for connections</td>
<td>Handling complex relational information, Modelling and handling classifications</td>
<td>Neo4j, Amazon Neptune</td>
</tr>
</tbody>
</table>
This chapter aims at contextualizing the project that was developed with Edoclink by describing the problem and identifying company requirements, goals, and use-cases.

3.1 Problem Description

Edoclink is a document management and process automation tool with three clear audiences:

- **Edoclink Users**, targeting functional use. This audience uses Edoclink on a daily basis and may use a chatbot for searching functional FAQs (how to use the product).
- **Vision Helpdesk Portal Users** for management and functional use. This group is a subset of the above and is allowed to create and manage tickets, in case the Frequently Asked Question (FAQ)s do not clarify their doubts.
- **Customers** for marketing. This set represents customers who are interested in acquiring the document management system.

Currently, Edoclink is approaching these audiences with different knowledge bases. FAQs are not stored in a single place, therefore, hindering its management. Moreover, this product often experiences duplicate or non-relevant issues which already have a solution; common issues are triggering ticket creation is being served within the Vision Helpdesk portal, without proper validation beforehand.

There is room for automation since a chatbot powered by a knowledge base could autonomously distinguish which issues already have a solution and which need to create a ticket, matching and searching FAQs, while still providing a single platform for managing different knowledge bases per audience.

3.2 Goals

Edoclink's goal is to deliver an application and service support bot for FAQs and ticket management while supporting handover and hijack features.

Regarding the conversation handover, Edoclink agreed on the development of an operator backoffice web application providing the necessary channel for monitoring conversations and communicating with users. As mentioned above, this chatbot is being deployed to three audiences in different websites. However, since users must be connected with the chatbot and Vision Helpdesk is not providing a custom chat widget, there is a need to develop an alternative chatbot interface which could be deployed to each audience website.

Concisely, the chatbot agent provides FAQs and ticket management and the backoffice provides customer support in near real time. The widget channel bridges the communication between users, backoffice (human) operators and Edoclink's agent.

Since the scope of this work is greater than the particular company use case, the following goals are
Figure 3.1: Chatbot application providing content to different audiences

tagged with “[B]” to identify business requirements and “[T]” for technical challenges. Therefore, the chatbot and widget should:

- [B] Support the Portuguese language
- [T] Integrate a single knowledge base
- [T] Be delivered once and deployed to all audiences
- [B] Distinguish each authenticated user
- [T] Handle handover and hijack protocols
- [B] Store conversation logs and metrics

And, when using the backoffice component, users should be able to:

- [B] List live conversations
• [B] Categorize conversations by metrics

• [T] Hijack a conversation

• [B] Send a conversation to a handover pool

• [B] Mark a conversation as resolved

### 3.3 Use Cases

A chatbot should be able to handle problems that are common enough to train and simple enough to resolve. The following table summarizes use cases that aim at delivering a single chatbot with three environments, one per audience.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Audience</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional FAQs</td>
<td>Edoclink Users</td>
<td>Edoclink Users should be able to search for a specific problem resolution or browse through functional FAQs</td>
</tr>
<tr>
<td>Marketing FAQs</td>
<td>Customers</td>
<td>Customers should be able to query any business and marketing related question and browse through marketing FAQs</td>
</tr>
<tr>
<td>Ticket Management</td>
<td>Vision Helpdesk Portal Users</td>
<td>Portal Users should be able to create a new ticket and query a ticket status</td>
</tr>
</tbody>
</table>

**Table 3.1: List of considered use cases**

By analyzing Table 3.1, it was possible to derive three main conversation flows. These flows were specified using the following methodology.

The design phases were divided by the different artifacts of a conversation.

In the first phase, the necessary parameters for each use-case were refined. Such parameters would be used to describe entities.

Subsequently, the different steps of the use-case were divided into intents that would facilitate the process.

Finally, the bot’s personality was traced by adding rich text messages and replies. In parallel, different interactions emerged with the *Edoclink* team to help drawing and approve the conversation flows.

The following diagrams provide the different intents, entities and responses to be used in such cases. Although this system is being applied to three audiences, this thesis is only focusing on implementing the use-cases scoped to *Edoclink* Users and Vision Helpdesk Portal Users.
EDOC CHATBOT - FAQs

Conversation Flow

Figure 3.2: FAQs conversation flow
**EDOC CHATBOT - Submit Incident**

**Conversation Flow**

**Figure 3.3:** Submit Incident conversation flow

**KEY**

#.

Standard bot messages

K.

User input keywords

{variable}

Variables within the messages

Blue message

User Input

Grey message

Bot Messages
Figure 3.4: Incident Status conversation flow
4

Methods

Contents

4.1 Development Process ................................................. 31
4.2 Development Environment ........................................... 33
4.3 Conversation Lifecycle .............................................. 38
The next chapter begins by describing the methodology used in project development. Then the architecture is presented with a description of each component and technologies used. Finally, the overall architecture flow is examined in detail. Given the greater complexity of a multi-cloud strategy and the timing and scope of the thesis, this project uses a cloud-native design restricted to a single provider, with the solution composed by three applications: the chatbot, a chat widget and a backoffice.

4.1 Development Process

The choice process and design method were primarily focused on choosing the platform on which the chatbot was built, since it is the main component of a conversational agent. In a single-provider cloud environment, the selection of other services would depend on this one. The platform of choice was Google’s Dialogflow. According to table 2.1, this is the platform that supports the largest number of prebuilt entities and intents within those that support the Portuguese language. Since Dialogflow is a Google service, the following components were going to be provided by Google Cloud Platform (GCP).

In order for a chatbot to handle more advanced actions, such as validating parsed parameters or requesting 3rd party services such as the ticket management API, chatbot platforms allow you to add logic to one via webhooks. This piece of code is often deployed into a stateless and serverless platform. In Dialogflow, this logic is called a fulfillment webhook and may be provided by an out-of-the-box integration with Google’s Cloud Functions.

For the widget to be loaded in different scenarios, one would need to version it, distinguishing different audiences. One alternative would be to have 3 instances of web applications behaving similarly. However, saving it to a file in storage and parameterizing for each audience would be a more economical and simple proposition. Therefore, the widget was stored in Cloud Storage. This online object storage was used to host a small file meant to be sourced in each environment.

For users to be helped, it would be necessary to access the history of the conversation. Also, in borderline situations such as handover and hijack, conversations would have to be updated in real time. Both live and recorded conversations had to be stored and displayed. Creating a backoffice needed a live notification engine to track chat sessions in almost real time. With Google’s Cloud Datastore in Firestore mode (also known as Cloud Firestore), an application can be updated in near real time using the “realtime listener” mechanism. In this document based database, each file represents a database reference which can be attached to an asynchronous listener; the listener is triggered once upon its creation and whenever data changes.

However, a piece of code to store conversation history and dynamically update chat rooms was lacking. This operation is not within the scope of chatbot’s operation and could therefore be asynchronously pro-
cessed by event management mechanisms. In a pub/sub protocol, messages are published (pub) to a topic, similar to an event bus, and subscribers (sub) trigger functions once a new message is consumed. Since this service is asynchronous, a new event could be triggered without being a bottleneck at the webhook execution level. Using the current Google-supplied Cloud Pub / Sub software, the chatbot webhook could be extended to publish new messages. Finally, the backoffice application would be deployed on Google's App Engine: a fully managed serverless application platform. Given the small size of the application, it was thought to use Cloud Storage. However, and preventing future scalability issues, App Engine ensures a more robust solution in an environment where instances can be customized to suit their compute resources and pricing.

In short, the solution is comprised of this Google ecosystem:

- Dialogflow
- Cloud Functions
- Cloud Storage
- App Engine
- Cloud Pub / Sub
- Cloud Firestore

Testing and functional validation were meant to follow three stages:

**Internal Testing**
At this stage, an Edoclink operator is selected to interact with the chatbot. After a brief presentation on the possible use-cases, this user is being evaluated with metrics such as goal completion time, messages or taps. Expected feedback may be new training phrases and possible tweaks to the conversation flow.

**Client Testing**
In client testing, an Edoclink customer company would be asked to test the bot. This company would select a few customers to test the application and fulfill a short user experience survey in the bot UI. Bugs may be reported in a dedicated Google Form.

**A/B Testing**
This stage is meant to be continuous, with no particular target, and metrics should be collected and monitored by Chatbase. Chatbase is a Google-made service focused on virtual agent analytics, providing metrics, session flows, rich filtering, and may help in identifying a problem, filtering and analyzing relevant conversation history.

Although this project consists of three applications (chatbot, widget and backoffice), only the chatbot and its behaviour would be exhaustively tested.
The next section presents the overall architecture and an in-depth description of each component present in fig. 4.1.

4.2 Development Environment

This project was developed using full stack Javascript technology. Back-end services were implemented in Node.js whereas front-end applications were created using the Vue.js framework (JavaScript, HTML and CSS) [24, 25]. The architectural style present in Figure 4.1 structures this project as a collection of the following services:

1. **Widget**
   
   This messaging channel connects users with the chatbot or a human operator. It features a small deployment script to include in a website and receives a parameter specifying if there should be an authentication screen prior to the chat view, useful when authenticating Vision Helpdesk Portal Users. Authenticated users must login using their Vision Helpdesk username (or email) and password. When a user logs in, a JSON Web Token (JWT) is received, stored and sent in subsequent requests, thus ensuring that requests come from a properly authenticated user [26].

   At the top, the chat widget has two buttons: the reload button, which resets the conversation, deleting all contexts and starting over; the star button, which toggles a small user experience form, where users can rate their conversation and make an observation.

2. **Cloud Storage**
   
   The chatbot widget is being hosted in an European public bucket in Cloud Storage. When a file is uploaded to a public bucket, a public URL is generated. This URL is then used in a script instruction to be place in any website to include the widget.

3. **Middleware**
   
   Dialogflow’s webhook has some limitations: responses must occur within 5 seconds, otherwise the request will time out; the response must be less than or equal to 64 KiB in size; fulfillment libraries may have bugs and are not supporting every Dialogflow API functionality. Since users may be importing screenshots when creating a ticket, the dialogflow webhook integration was not a viable option.

   This middleware component mimics a Dialogflow fulfillment webhook and connects the chatbot platform with the widget interface. This middleware responds to users queries by communicating with Dialogflow’s NLP and executing additional logic to each intent.

   “Webhook” logic is divided into slot-filling and fulfillment maps: the first validates and parses parameters when in slot-filling; the latter executes an action when an intent has every required parameter present. When a user triggers an intent that does not use the previously mentioned maps, Dialogflow responses are not subject to additional logic and this component sends the query result directly to
Figure 4.1: Project Architecture: a Google cloud-native approach
4. Dialogflow

Dialogflow is responsible for detecting intents, parsing entities, creating conversational context and generating static responses, being the core component of Edoclink’s chatbot.

To prevent intents from being wrongly triggered, two fallback intents were added to aid users into choosing what intent to trigger; the “Incident Fallback” is trained with generic user queries regarding tickets and suggests creating a ticket or listing available ones; the “Troubleshoot” intent is trained in a similar manner and helps in browsing the accessible FAQs by listing categories and questions for a category.

4.1 Knowledge Base

Regarding the knowledge base, Edoclink initially provided a set of functional FAQs to be imported to the Dialogflow Knowledge Base Connector. Since this feature is only available in English, the alternative was to map FAQs to intents. To identify the category and target audience of each intent,
every FAQ is considered to be an intent beginning with the tag “[FAQ]” and may have two additional tags: the second one is required and indicates the category in which the FAQ should be presented; the last tag can be optionally added to distinguish the target audience. In sum, a FAQ template indicating a category and special target should be “[FAQ] [CATEGORY A] [TARGET X] $FAQ$TITLE”. This mechanism is providing a custom and centralized knowledge base connector for every audience.

4.2 Ticket Management: routing and templating

Regarding ticket management, intents are divided into a Submit Ticket intent and a Ticket Status intent, allowing users to create a new ticket or list and review one. A routing engine was implemented to seamlessly generate dynamic responses according to the results of integrations with the ticketing system, since Dialogflow is not providing such features. Each dynamically routed intent has a hidden parameter representing a JSON string relating HTTP status codes with an array of appropriate responses. For example, when in ticket creation intent, one possible router may be “{201: [1], 500: [2]}”, where the first response block corresponds to a successfully created ticket (code 201) and the second response is used when there is a problem creating a ticket. Additionally, each response block may have template placeholders to be formatted with additional parameters in runtime. Following the previous example, a successfully ticket creation template may be defined as “Ticket $\{ticketId\}$ was created successfully” and generated in runtime with the appropriate ticket identifier.

5. Backoffice

This Vue app provides real-time conversation management for selected Edoclink managers. Every chat session represents a room, where metrics are collected along the conversation (average sentiment, duration, conversation loops, etc.). The conversations area lists all rooms, splitting into different ordered filters with one representing an handover pool. An operator may send an active room to the handover pool, hijack a conversation and mark a chat session as resolved. This implementation is based on the conversation lifecycle, later described in this section.

6. App Engine

This service is responsible for serving the backoffice app and retrieve live conversations updates.

7. Auth Server

Although this service is not being used and is not part of the project requirements, it was also thought to be part of the overall architecture. This function authenticates backoffice operators and proxies sensitive operations upon authentication. The login procedure and authentication protocol are provided by integrating with Firebase Authentication, supporting email and password based authentication, phone authentication and different OpenID providers such as Google or Microsoft [27].
Figure 4.3: Edoclink’s operator backoffice with live conversations
8. Firestore
Datastore in Firestore mode is the database being used to store chat history. Conversations are stored in a database collection named “rooms”, where each room has a collection of metrics to be displayed on the backoffice chat rooms screen. Each room stores previous messages in the “chat” sub-collection. Performance-wise, a backoffice operator would just query the relevant data in each screen: when listing rooms, metrics are retrieved; when clicking on a conversation, only the room chat would be queried.

9. Conversations Topic
This component is responsible for handling conversation updates. Upon receiving a new event, a custom JavaScript Object Notation (JSON) payload containing new messages, this topic triggers the store conversations and store metrics functionalities.

10. Store Conversations
This function stores new conversations in Firestore. Once it is triggered, this subscriber parses the payload, retrieves the session identifier, and attaches new messages to the end of the “chat” collection. Moreover, it collects metrics to be presented in the operator backoffice: document creation and last reply timestamps (the duration of a conversation), average sentiment (provided by Dialogflow’s sentiment analysis) and conversation loops (how many times an intent is consecutively triggered).

11. Store Metrics
This service sends new messages to Chatbase Analytics. Similarly to the previous subscriber, this function uses the session identifier and mutates the conversation object to a chatbase-like request, emulating a user interaction in the mentioned platform.

4.3 Conversation Lifecycle
In hijack and handover scenarios, Firestore takes a crucial role in providing the necessary room management. When a new session is created, a new room document is created and linked to that session with an “active” status. Upon requesting an handover request, the middleware is responsible for updating the room status to “handover”. In the backoffice application, rooms with this status are waiting for an operator and listed in a particular tab. At this point, operators may take possession of a conversation, triggering a room update to the “hijack” state. Additionally, operators may hijack a conversation directly (from “active” to “hijack” state) and archive or mark a conversation as resolved, updating its state to “closed”.

In sum, the conversation life cycle may be presented by the following states:

- If a user requests handover: active → handover → hijack → closed
- If an operator hijacks an active conversation: active → hijack → closed
• If neither the above scenarios occur: active → closed
Results and Discussion

Contents

5.1 Content Evaluation .................................................. 43
5.2 Functional Evaluation ................................................. 45
A chatbot must be tested to verify whether it meets the requirements for real use. However, in this context, there are different approaches for testing such a bot, depending on which goals to assess [28]. According to the previously mentioned advantages, a chatbot is a constantly available solution that effectively helps its users, given the supported use-cases. This application may be evaluated by its content, which corresponds to the correct behavior in the identification of intentions and response generation, or its performance, which corresponds to the delay between user queries and chatbot responses and the consolidation of custom integrations.

This chapter presents the metrics used in different assessment environments, justifying the results obtained with possible causes and solutions. The following data was the result of 18 user sessions, with an average of 11 queries per session.

5.1 Content Evaluation

By simulating a human conversation, a chatbot must be able to handle the intentions that are supported in its environment. Such conversations can occur in various ways, but only one goal should be achieved: fulfillment. Content is evaluated by the correct identification of intentions, the effective filtering of each entity, and the coherence between different flows and appropriate responses. Such tests can be done automatically or manually observed through conversation logs.

Chatbase provided table 5.1 and fig. 5.1, which describe intents statistics and relate conversation states with exit points: points in which the user exits the conversation. For example, it shows that the “Default Welcome Intent” was triggered 33 times and 15.15% of these ended the conversation session; interestingly, the 42.86% exit in “Submit Incident - Subject”, may reveal a bug regarding ticket creation, since this intent belongs to one of the intermediate ticket creation steps and should not have such a high exit rate.

By analyzing the previous results and using the Dialogflow conversation history, one may make the following observations:

- **FAQs are being wrongly triggered**
  
  Users are expecting a large knowledge base and, since these intents have similar utterances, a FAQ in the same category is often triggered. This behavior is a common problem in the first stage of training a chatbot and may be due to two different reasons:

  - The lack of training phrases in a FAQ may influence the weight to be given in the intent detection process: being an artificial intelligence application, a chatbot needs a significant amount of information to be trained with. Dialogflow recommends a minimum of 10 utterances per intent and intent similarity can be a factor into increasing this number. Adding utterances
Table 5.1: Intents handled by the conversational agent

<table>
<thead>
<tr>
<th>Intent</th>
<th>Sessions</th>
<th>Count</th>
<th>Exit %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Welcome Intent</td>
<td>18</td>
<td>33</td>
<td>15.15%</td>
</tr>
<tr>
<td>Check Incident - Details</td>
<td>7</td>
<td>30</td>
<td>13.33%</td>
</tr>
<tr>
<td>Submit Incident</td>
<td>6</td>
<td>20</td>
<td>5%</td>
</tr>
<tr>
<td>Incident Fallback</td>
<td>9</td>
<td>13</td>
<td>0%</td>
</tr>
<tr>
<td>Troubleshoot - Category</td>
<td>6</td>
<td>11</td>
<td>0%</td>
</tr>
<tr>
<td>Troubleshoot</td>
<td>8</td>
<td>11</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Distribuições] Criar Distribuição</td>
<td>2</td>
<td>9</td>
<td>0%</td>
</tr>
<tr>
<td>Submit Incident - Description</td>
<td>3</td>
<td>7</td>
<td>42.86%</td>
</tr>
<tr>
<td>[FAQ] [Pesquisas] Pesquisar Registo</td>
<td>2</td>
<td>6</td>
<td>0%</td>
</tr>
<tr>
<td>Submit Incident - Subject</td>
<td>3</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Pesquisas] Pesquisar Distribuição</td>
<td>3</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Acessos] Consultar Acessos</td>
<td>3</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>Submit Incident - Confirmation - yes</td>
<td>2</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>Check Incident</td>
<td>2</td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Pesquisas] Pesquisar Processo</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>[FAQ] [Processos] Criar Processo</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>Default Goodbye Intent</td>
<td>2</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>[FAQ] [Registos] Registar Documento</td>
<td>1</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Distribuições] Despachar Tarefa</td>
<td>1</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Configurações] Configurar Página Inicial</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Configurações] Criar Textos Predefinidos</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Configurações] Delegar/Aceitar Delegações</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>Handover Request</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>[FAQ] [Registos] Registar Documento Office</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>[FAQ] [Registos] Registar Mails</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
</tbody>
</table>

to Edoclink functional FAQs may require a superior product knowledge and, for that reason, should be taken into consideration for future work.

– On the other hand, the absence of an intent that maps a question may also be a factor. In this case, although the solution should be creating a new intent for this question, it may suffer from the previous point: with more similar intents, the training data may not be enough to distinguish them.

After consulting the conversation history, it became apparent that both symptoms applied to the
misidentification of intents. The bug reporting forms mentioned in chapter 4 have also shown to be a useful source for identifying such cases.

- **Ticket related entities need refining**
  Although the ticket creation (“Submit Incident - Subject”) exit percentage was not related with a bug, the problem resided in using the entity present in the “Check Incident” intent. When querying for a ticket, Vision Helpdesk users are using the ticket hash (ABCD-1234, XPTO-5678) instead of the ticket ID (123). After accessing the portal and its ticket organization, we found that it would not be easy to access the ticket ID as it could only been seen in the URL string or in the admin portal used throughout the chatbot development. The problem, however, would be to map a Dialogflow entity that was not a number (system entity) but a set of letters and numbers (custom entity), since regular expression entities were not supported by the platform at the time. However, in September 13, Dialogflow released a new version in which it is possible to define Regexp entities [29].

- **Custom fallback intents are helping the conversation flow**
  Testers were often prompting for generic content such as “FAQs” or “Tickets”. However, the fallback intents mentioned in chapter 4 and implemented in each use-case revealed to be aiding the user in following one of the possible conversation paths. For example, if a user was to ask for “tickets”, a chatbot would not be able to distinguish the user intention of creating a ticket or checking the status of an already created incident. In this scenario, a trained fallback intent captures this query and prompts for the possible alternatives.

### 5.2 Functional Evaluation

In a goal-oriented chatbot, such as an application and support one, functional validation is granted when user goals are fulfilled. For custom integrations, a goal is achieved once a request for fulfilment is completed and successfully executed. In this project, fulfillment testing should cover all ticket management integration points and data sources. Moreover, given the modular cloud architecture of the system, this testing section is also designed to ensure that there is a quick response (performance) between the user’s request and the corresponding chatbot action.

The following points were concluded after reviewing bug reporting responses (submitted to the previously mentioned Google Forms platform) and examining the middleware latency graphs (fig. 5.2 and 5.3) provided by GCP:

- **Dialogflow templates are being poorly rendered**
  The implemented template engine aims to define and edit all generated responses in Dialogflow, rather than hard-coding each response and not being able to edit one without a new deployment.
More accurately, if an action has multiple exit codes, responses are beforehand defined and chosen by the middleware component at run-time.

However, an update to the Dialogflow platform during the testing phase made the mechanism to render symbols while generating the final response [29]. Relying on such software tweaks may not be a viable alternative in an ever changing platform, specially when in a production environment.

- **Listing tickets may take long**
  It was thought that listing tickets would help users to quickly query one or more tickets. However, the more tickets a user has, the longer it takes to request the Vision Helpdesk API, even when limiting the number of tickets to be retrieved. This behavior was not detected since the Vision Helpdesk account used for development and internal testing had a limited amount of tickets. Reviewing the middleware logs and crossing the information provided by Dialogflow’s conversation history, one may conclude that some high latency calls in fig. 5.2 are, in fact, ticket listing requests (October 18th, around 6 PM). Therefore, it would be necessary to improve the conversation flow to reduce ticket listing operations, when, for example, the user already knows the ticket ID to retrieve. Changing the entity associated with the identifier may also help with the frequency of listing tickets. However, as far as latency is concerned, \( \approx 3 \) seconds in the 99th percentile establishes an acceptable value for a human-like conversation experience.

- **Cold start latency**
  In a serverless environment, a cold start latency is the worst-case time that a function execution may take [30]. This latency occurs whenever a container needs to be started due to previous inactivity. Although it is an economic initiative, it may compromise system performance, as in the graph presented in fig. 5.3 with a maximum loading latency of 5.78 seconds (October 18th, at 9:55 AM).

  This latency may not be so worrying in a production environment where the chatbot is constantly being queried. Nevertheless, a “container warmer” may be developed and deployed to prevent further latency variations. Since inactivity is a major factor in large loading latencies, a custom warmer service could regularly request the middleware API with a dummy payload, preventing the container service from shutting down. It is important to notice that, even when in constant load, serverless platforms periodically recycle containers, and, in this scenario, it is inevitable to have a cold start.
Figure 5.1: Session flow based on 30 days of session data
Figure 5.2: Middleware service request latency graph (in 7 days)

Figure 5.3: Middleware service cold start latency graph (in 7 days)
6 Conclusion

Contents

6.1 Conclusions ................................................................. 51
6.2 System Limitations and Future Work .............................. 52
This last chapter aims at summarizing the work described in previous chapters, identifying advantages and problems of building such a system, and concluding with evident system limitations and possible future developments.

6.1 Conclusions

The objective of this work was the application of chatbots to application support and support. The proposal was to use a cloud-native architecture using Google Cloud Platform's services in a decoupled architecture, developing three applications: a ticket management and FAQ's based chatbot, a backoffice in which service support operators could hijack conversations in real time and a web widget acting as a communication channel between users and both agents (bots) and/or (human) operators.

Given the familiarity with the technology, this project was developed using full stack JavaScript technologies, with no major implementation challenges. Backend services were developed using Node.js and Vue.js was used for the frontend applications. This project was built using a chatbot platform, a serverless functions platform to manage new user requests and bot responses, pub/sub events to trigger behaviors which stored conversation logs and metrics and a near real-time NoSQL cloud database to support live conversations. From the three previously mentioned applications, only the chatbot was subject to testing. It stems from the evaluation in chapter 5 that this chatbot is not yet ready to be tested in a different environment nor it should be deployed into such a stage while other applications still need to be properly validated and thoroughly tested.

Nevertheless, this project combined the three applications into one ecosystem with 11 loosely coupled services. Decoupled services revealed to be easy to build and maintain since its development is completely independent from the overall system.

The use of cloud technologies in a decoupled architecture benefited from a variety of Google services scoped to each use-case requirement. In particular, Cloud Firestore was a key component to use when planning the handover scenario and backoffice live conversation feed. The middleware component implements a substantial improvement to Dialogflow’s platform with dynamic templates and route handlers with negligible latency to the user experience. However, loading latencies may be a downside of using serverless platforms such as Cloud Functions or App Engine, even when latency reduction measures are applied, such as container warmers.

It was not expected that the third party integration for ticket management had to be hampered by the lack of support in Vision Helpdesk API and documentation. It was only possible to implement most use-cases by submitting tickets and scheduling conference calls with Vision Helpdesk developers or by inspecting the API calls in the live Helpdesk portal.

A major asset of this work was the implementation of handover and hijack protocols which ensure that
users can be helped if the chatbot fails to do so: crucial in an application and service support scenario.

6.2 System Limitations and Future Work

Despite the strong focus on developing these products, current limitations depend heavily on changes that may happen during the development phase. In particular, this project would benefit, for example, from a (currently supported) regex entity that was not supported by Dialogflow at the time. The platform lacks a router in which developers could match responses with status codes, therefore reducing additional logic to implement such a feature and replying with a more accurate response. Moreover, dynamic templates could be supported in order to match with external parameters often provided by custom integrations.

As regards testing, the different applications were subjected to individual and collective testing throughout their development. However, a functional approach would be needed to consolidate the tests done, especially when testing the backoffice application. Moreover, operationalization of chatbots in a production environment should be monitored, inspecting relevant conversations in order to update and add utterances. In this regard, the backoffice should have an area dedicated to this training process.

Prior to future work, one should consider refining what still remains to be done in the current project:

- Map new FAQs to new intents, tagging its audience and category, requiring functional domain context
- Continuously train the chatbot by reviewing the conversation history
- Redefine the ticket identifier entity to a regex based entity
- Schedule additional testing sessions with more users

For future developments, the following initiatives may be interesting:

- A FAQ management screen could be added to the backoffice application in which operators could strengthen the knowledge base with more utterances
- A training screen could be added to the backoffice application, where operators could validate conversation history and train the bot
- New ticket management use-cases could be added such as searching or filtering for a ticket
- Use other communication channels such as Facebook and integrate authentication between Facebook users and the help desk portal
- The English language could be supported to cover a wider customer base
In conclusion, the FAQs used in chatbot training were not sufficient to achieve filtering between FAQs and tickets. However, features that facilitate the training process and the enhancement of knowledge base will provide a practical interface for a process that would otherwise be time consuming. A chatbot can be a viable solution to emulate a conversation in a human-like manner, while being always available, performant and less expensive than a human agent. For now, however, application and service support bots should be designed to help users and not to replace the human role totally, since it is still a strong contender to the most accurate knowledge base.
Bibliography


