DM

A Generic Chatbot Framework Using a Knowledge Base

MASTER DISSERTATION

Rodrigo Severim Vieira
MASTER IN INFORMATICS ENGINEERING

A Nossa Universidade
www.uma.pt

September | 2023

A Generic Chatbot Framework Using a Knowledge Base

MASTER DISSERTATION

Rodrigo Severim Vieira
MASTER IN INFORMATICS ENGINEERING

SUPERVISION
Eduardo Leopoldo Fermé

CO-SUPERVISION
Jorge Dias Fernandes

FACULTY OF EXACT SCIENCES AND ENGINEERING

MASTER IN INFORMATICS ENGINEERING

A generic chatbot framework using a knowledge base

Rodrigo Severim Vieira

Guided by:

Eduardo Fermé

Jorge Fernandes

Thursday 14$^{\text{th}}$ December, 2023

Resumo

Os avanços tecnológicos permitiram criar e utilizar chatbots para tarefas mais complexas, levando a várias técnicas de desenvolvimento de chatbots tal como variados tipos de chatbot. Tal como pode ser observado pela utilização de bases de conhecimento e grafos de conhecimento, também houve um avanço em técnicas para armazenar informação. Apesar destes desenvolvimentos este modo de representação pode ser difícil de navegar quando existe imensa informação armazenada. Para lidar com este problema é proposto o projeto KBAI, sendo que procura criar uma aplicção inteligente para gerir conhecimento, esta dissertação faz parte deste projeto, focando-se numa das suas componentes com o proposito de facilitar as ações dos utilizadores.

Ao aprender novas ferramentas os utilizadores deparam-se com uma curva de aprendizagem, sendo que esta tambem se encontra presente ao aprender a navegação de novos sites e os seus variados *layouts*. Esta dissertação procura encontrar uma soluçao genérica a este problema de modo a que possa ser integrada em qualquer ferramenta ou aplicação e facilmente configurada para facilitar a navegação da mesma. A opção escolhida para resolver o problema foi a criação de uma *framework* genérica de chatbot. Sendo que com um chatbot o utilizador pode, ao longo de um diálogo navegar a base de conhecimentos e encontrar a informação desejada. Por outro lado o chatbot em si pode ser integrado em várias aplicações externas, identificando as intenções do utilizador e pedindo o diálogo apropriado a base de conhecimentos. Nesta dissertação pode ser observado a modelação e construção de um chatbot que utiliza uma base de conhecimentos para ir buscar os seus diálogos. Este chatbot permite aos clientes criarem um diálogo personalizado através do editor visual da base de conhecimentos, sendo que o chatbot depois executa este diálogo. O projeto engloba três casos de estudo: turismo, gestão de projetos e gestão de notícias, sendo que o caso de estudo do turismo será utilizado para demonstração do funcionamento.

O processo de desenvolvimento foi executado através de sprints semanais, refinando o protótipo inicial com cada sprint. Como resultado deste processo obtém-se uma *chatbot framework* versátil que permite aos clientes definir varias opções para o diálogo através do editor visual. Adicionalmente esta *framework* permite facilmente integrar *APIs*, como demonstrado no caso de estudo do turismo. Este resultado provem do projeto *Knowledge Base Artificial Intelligence* (KBAI), que tem como objetivo estabelecer uma plataforma de gestão de conhecimento, com o foco em três casos de estudo.

Keywords: Chatbots · Base de conhecimento · Gestão de conhecimento · Aprendizagem de máquinas · Processamento de linguagem natural

Abstract

As technology advances, the capability to create and utilize chatbots for intricate tasks grows, enabling the conceptualization and subsequent feasible deployment of various chatbot types. The capacity to employ more detailed techniques for information storage has also expanded, exemplified by the utilization of knowledge bases and knowledge graphs for knowledge representation. Nonetheless, this mode of expression can be challenging when dealing with substantial information. The KBAI project was proposed to handle the problems mentioned earlier, as it looks to create an intelligent application for knowledge management; this dissertation is part of the KBAI project, focusing on one of its components aimed at providing ease of use options to users.

When learning new tools, users face a learning curve, with this phenomenon being present when learning to navigate unfamiliar websites with their varying layouts. This dissertation looks to find a generic solution to the earlier problem, with this solution being capable of integrating into any tool or application and easily configured to streamline the application's navigation. The chosen option to solve this problem was the creation of a generic chatbot framework. With a chatbot, users can navigate a knowledge base through a dialogue and find the information they seek. On the other hand, a chatbot can be integrated into various applications, using its capability to find user intents and ask the knowledge base for the corresponding dialogue.

The development process involved iterative weekly sprints, refining the initial prototype. This culminated in a versatile chatbot framework that empowers users to define diverse dialogue flows through a visual editor. This generic chatbot framework can easily integrate APIs, as validated by the tourism case study's proof of concept. This effort stems from the Knowledge Base Artificial Intelligence (KBAI) research project, which aims to establish a knowledge management platform concentrating on specific case studies.

Keywords: Chatbots · Knowledge Base · Knowledge Management · Machine Learning · Natural Language Processing

Acknowledgements

First and foremost, I would like to extend my most profound appreciation to my family. Additionally I want to thank my advisors, Professor Eduardo Fermé and Eng. Jorge Dias Fernandes. They have consistently been available to provide assistance during development, review this dissertation, and offer valuable suggestions.

I also want to thank my colleagues and friends, Diogo Gouveia and Katherin Varela, for their support. In addition, I would also like to extend my deepest gratitude to my friends Ana Gonçalves and Daniel Barros, whose support was invaluable.

I want to acknowledge all my professors at the Faculty of Exact Sciences and Engineering, whose lessons have equipped me with the understanding and procedures to complete this thesis.

The project received funding under application no. M1420- 01-0247-FEDER-000073 in the Operational Program for the Autonomous Region of Madeira (Madeira 14—20).

Table of Contents

List of Figures viii
1 Introduction 1
1.1 Problem description 1
1.2 Proposed solution 2
2 KBAI Project 4
2.1 KBAI Project Description 4
2.2 Chatbot Component 5
3 State of the Art 6
3.1 The first chatbots 6
3.2 Artificial Intelligence in Chatbots 7
3.3 Chatbot components 7
3.3.1 Natural Language Processing Component 8
3.3.2 Natural Language Understanding Component 8
3.3.3 Sentiment Analysis Component 9
3.3.4 Human Computer Interaction Component 10
3.3.5 Machine Learning Component 10
3.3.6 Chatbots and Knowledge Bases 11
3.4 Chatbots evolution 11
3.5 Chatbots Design Techniques and Types 12
3.6 Conclusion 14
4 Tools used 15
4.1 Diagrams.net 15
4.2 Visual Studio Code 15
4.3 PHPStorm 15
4.4 Botman dependencies 15
4.5 Chatbot Frameworks 16
4.6 NLP Libraries 20
4.7 Conclusion 22
5 Development 23
5.1 Case Studies 24
5.1.1 Tourism case study 24
5.1.2 Project Management case study 24
5.1.3 News management case study 25
5.2 Chatbot requisites 25
5.3 Planning 26
5.3.1 Planning chatbot dialogues 26
5.4 Initial prototypes 29
5.5 Dynamization of the chatbot 32
5.6 Chatbot and Natural Language Processing 37
5.7 Connecting the chatbot to the Knowledge Base 37
5.8 Chatbot and APIs 39

5.9 Conclusion 40
6 Conclusions 42
6.1 Challenges 42
6.2 Next steps 43

References **45**
A Tools Used 47
A.1 Chatbot Frameworks 47
B Development 51
B.1 Planning 51
B.1.1 Chatbot Types Pros, Cons and Limitations 51
B.1.2 Chatbot Types Description, Inputs, Outputs and examples 54
B.2 Planning chatbot dialogues 57
B.2.1 Tourism case study dialogues 57
B.2.2 Project Management case study dialogues 57
B.2.3 News management case study dialogues 58
B.2.4 My Life case study dialogues 58
B.3 Botonic 60
B.3.1 Code snippet of a Botonic route 60
B.3.2 Code snippet of a Botonic action 62
B.4 Botman 63
B.4.1 Example of a Botman Conversation Class 63
B.4.2 Code snippet of Botman routes 64
B.5 Dynamic chatbot 66
B.5.1 Created JSON structure 66
B.5.2 Example of an interaction using the JSON structure 66
B.5.3 Code snippet of the method that runs chatbot dialogues 67
B.5.4 Code snippet of the method responsible for running an interaction with no buttons 68
B.5.5 Code Snippet of Botman buttons interaction method 69
B.5.6 Code snippet of the method responsible for checking if the interaction should be skipped 70
B.5.7 Code snippet of the method responsible for fetching an interaction to fill the missing requirements 70
B.5.8 Code snippet of the class responsible for input validations 71
B.6 Chatbot and NLP 72
B.6.1 Snippet of NLP corpus 72
B.6.2 NLP Middlware *getResponse* method 72
B.6.3 NLP Middlware *received* method 73
B.6.4 Snippet of routes file 73
B.7 Connecting chatbot to KB 75
B.7.1 KB template format example 75
B.7.2 Method to map templates do dialog 77
B.7.3 Method to get first interaction of the dialog 78
B.7.4 Chatbot API methods 78
B.7.5 Chatbot API example file 79

B.7.6 Chatbot API website view 79

B.8 API integration 80

B.8.1 Function to return mapped API response 80

B.8.2 Example of method that implements an API 80

B.8.3 Method that handles output of primitive functions 81

B.8.4 Example of using range Output Format in an interaction 81

List of Figures

1 Overview of KBAI project structure 5
2 Example of the AIML dialect [1] 7
3 Example of a Seq2Seq architecture [2] 14
4 Chatbot development cycle 23
5 Botonic actions 29
6 Example of a Botonic interaction 30
7 Example of Botman Interaction with no buttons 31
8 Example of Botman Interaction with buttons 31
9 Example of Botman Conversation classes 31
10 Code snippet of the method to run an interaction with buttons 34
11 Code snippet of the method to check if next interaction should be skipped 35
12 Code snippet of the method responsible for getting an interaction to fill missing requirements 35
13 Code snippet of the method responsible for validating user input 36
14 Chatbot API icon 38
15 Code snippet of the method to map API responses 40
16 Chatbot dataflow diagram 41
17 Example of Interactive Buttons 44

List of Acronyms

AI Artificial Intelligence

AIML Artificial Intelligence Markup Language

ALICE Artificial Linguistic Internet Computer Entity

ANN Artificial Neural Networks

API Application Programming Interface

GUI Graphical User Interface

HCI Human Computer Interaction

ID Identification

JS JavaScript

KB Knowledge Base

KG Knowledge Graph

ML Machine Learning

NLP Natural Language Processing

NLTK Natural Language Toolkit

NLU Natural Language Understanding

RL Reinforcement Learning

RNN Recurrent Neural Network

SA Sentiment Analysis

SL Sentiment Lexicon

Seq2Seq Sequence to Sequence

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

WAMP Windows Apache MySQL PHP

1 Introduction

This section will give a short introduction to the KBAI project, the problems it aims to solve with the chatbot component, and its goals. It will then provide a brief explanation of how this dissertation proposes to achieve these goals and finally give a quick overview of the structure of this dissertation.

This dissertation is part of the Knowledge-Based Artificial Intelligence (KBAI) project. The project consists of having a central Knowledge Base that is then utilized for communication between its various components. KBAI integrates a recommendation system based on preferences, a preference aggregation system, an online KB editor, and a generic chatbot framework. This dissertation aims to develop a generic chatbot framework that utilizes the KB to get its various dialogues based on user intent in real-time, so when a user edits or creates new dialogues through the KB editor, these are immediately reflected on the chatbot. The proposed chatbot framework uses the KB for easy integration of any API and communication with other components of the KBAI project.

Comprehensive research was done on chatbots, chatbot frameworks, their history, the components used to make up and enhance a chatbot, the various currently employed types of chatbots, and the techniques for developing these. This dissertation's state-of-the-art section results from the research, as mentioned earlier.

1.1 Problem description

The KBAI project has various goals, and among them is the need to provide simple and effective options for the user when it comes to getting what they want. As the number of users increases, businesses have a more challenging time providing quick and customized solutions to their users, making it harder to scale the business as human support can only help a limited number of customers at once. In addition it is challenging for new users to migrate to new tools or learn new website layouts that can make it difficult for a user who wants to make a simple hotel booking or flight purchase.

The developed product aims to resolve the previously mentioned problems by finding an adaptable solution that can be quickly changed by making use of a knowledge base to fit into any application, website or tool that may need this additional ease of use, as such the KBAI project this dissertation is a part of aims to achieve the following goals:

1. Using chatbots to assist users in their respective tasks (making reservations, managing projects, news management).
2. Having chatbot dialogues be created and edited in the knowledge base through its visual editor so the average client does not need programming skills to create a chatbot that can assist with booking rooms at their hotel.
3. Making use of natural language processing to identify user intents.
4. The chatbot must know the best dialogue to assist the user through the detected user intent.
5. Using the various tourism APIs to assist the user with making reservations and bookings as proof of concept.

6. Making the API implementation generic and straightforward for easy integration into the various case studies.
7. Allowing for easy integration with the rest of the KBAI project through the knowledge base.

In addition to the previously mentioned goals this dissertation looks to achieve the following:

1. The dialogues created on the knowledge base editor and their changes must be reflected in real-time (on a new dialogue start) on the chatbot.
2. Allow users to enter a second dialogue midway through the first dialogue and then resume the original dialogue at the exit point.
3. Create a maintainable code base that can be easily expanded upon with additional features.
4. Make the user experience as effortless and straightforward as possible.

1.2 Proposed solution

It could be feasible to go through the various case studies and overhaul the user interfaces to address the usability problem; however, this option would require tremendous work and would not be scalable. Another approach to the problem that is more generalized is allowing the users to search through the knowledge base for their desired goal and then offer a clickable button or icon; in this way, it would be scalable as the knowledge base would only need to incorporate the existing information. Regardless, incorporating everything into the knowledge base has some safety concerns, as clients might not want their existing system transitioned to a third party. The settled solution was to make use of a chatbot framework. This framework will ask the knowledge base for communication with the APIs needed for the various services. It will be easily adaptable as clients can create their dialogues in the knowledge base, having these reflected when a user talks to the chatbot in the website or application.

Chatbots can allow a business to have customized interactions for the needs of each customer; for example, a chatbot can enable a user to go online quickly, give the booking information to the bot, and let it make the booking for them, eliminating waiting times on the phone if the user was to try calling the hotel directly and possible mishaps and frustrations in navigating the hotel's website. Regardless, chatbots often are limited in their potential interactions, and it can be hard to change the dialogues for a client without programming knowledge.

Research was done into the different types of chatbots to implement a generic chatbot framework that interacts with the KBAI knowledge base to get its various dialogues, make necessary API requests, and see which would be best suited for the project requirements. Fig.B.1.1, Fig.B.1.2 represent the result of this research, outlining for each type of chatbot examples of their use, a short description, examples of possible inputs (necessary information) for the chatbot to work, some outputs (back-end actions and chatbot replies) and examples of these, advantages and disadvantages for each type of chatbot, limitations and areas where each type excels.

This dissertation will first explain the context of the KBAI project and what it is, along with the role of the chatbot component that the dissertation is focused on achieving. Afterward, the result of the research done regarding chatbots can be seen in the state-of-the-art chapter, followed by the various available tools for developing a chatbot and multiple NLP libraries to complement the chatbot. Then, this dissertation goes over the different development steps and explains these in

detail. The last chapter will review conclusions from the developed work, encountered challenges, how some were dealt with, and the following steps to make it a fully commercial product.

2 KBAI Project

This chapter details the KBAI project, giving an overview of its structure and outlining its primary focus and goals along with the various planned case studies. It will also cover the role this dissertation fills in the project, being the chatbot component, along with which case study was chosen to serve as proof of concept.

2.1 KBAI Project Description

The KBAI Smart Ed (KBAI-SE) project aims to create an intelligent knowledge management application that encompasses a knowledge editing tool bolstered by an automated reasoning engine.

The KBAI-SE project involves research and experimental development, integrating academic and scientific knowledge from research centers with the expertise and experience of the professional team. The resultant KBAI-SE product will be an interactive tool for a dynamic and collaborative knowledge base proficient in rationally processing amassed knowledge.

At the core of this system lies the integration of diverse reasoning mechanisms, facilitating automatic knowledge updates (such as accuracy verification) and generating new knowledge through learning and logical inference.

The project's primary focus is to delve into the theories and tools of Artificial Intelligence (AI), gauging their present maturity level and potential applications for the designated objectives. The research will pivot on two key aspects: (a) the exploration of mature Reasoning Algorithms and Knowledge Representation techniques for streamlined and efficient utilization, and (b) the discovery of user-friendly visual methods for consulting and editing knowledge, simplifying the process of inputting knowledge into the system and executing queries effectively. While these theories are well-developed academically, their practical application in business or personal contexts necessitates validation to ensure compatibility with the data structures chosen for KBAI-SE and to assess if processing requirements are efficient enough for timely user interactions or background processing [3].

An extensive survey of the field's current state will be conducted to assess maturity, relevance to the specified knowledge representation format, and the suitability of libraries for business scenarios. Within the framework of industrial research, an evaluation of existing data structures and the demands posed by each reasoning theory will be performed, followed by a comparison with available data repositories suitable for their integration (such as graph-oriented databases). From this evaluation, supplementary modules will be developed to integrate each reasoning theory into the final product seamlessly.

The KBAI project has three main case studies (CS):

- CS1 Project management - In this CS, the chatbot is meant to assist with navigating the tools used for project management, increasing accessibility for new team members, and facilitating usage by those accustomed to the tools.
- CS2 Tourism - This dissertation's primary CS focuses on the tourism booking system and its various APIs. Here, the chatbot will allow every client to have completely different dialogues and interactions depending on their configuration, and these will be used to help users quickly make reservations at hotels, book flights, restaurants, or activities, and cancel or change the dates for these if needed.

- CS3 News domain (Presspower) - For this CS, the chatbot allows the user to search for information such as articles of a specific author, topic, or name. Add personal notes to news, get trending articles either overall or based on user interests, edit user interests, and get articles recommended for the user through communication with the KB that contains the results of the recommendation system of the KBAI project.

2.2 Chatbot Component

The structure of the KBAI project, including its diverse components, is illustrated in Fig. 1. This dissertation centers on the chatbot component, indicated in red. This component aims to interact with users via the application interface; this means the chatbot component will be integrated into the respective software through its API. From its various interactions with the users, the chatbot will supplement the building of the user profile handled by the component outlined in yellow; this component also uses a different component, the preference aggregation outlined in blue. With the conjunction of these three components, the role of the chatbot becomes clear: it is meant to facilitate user interactions in various scenarios, such as making a hotel booking or using a project management tool, while also collecting valuable user information that can assist with building a user profile that can be used by not only the other project components but also the chatbot itself.

The tourism case study will be a demonstrative example of the chatbot component. The chatbot will engage in dynamic dialogues configured and generated via the KBAI editor within this case study. It will possess the capability to leverage necessary APIs. This implementation will ensure a straightforward and versatile integration of further APIs into the chatbot and various dialogues.

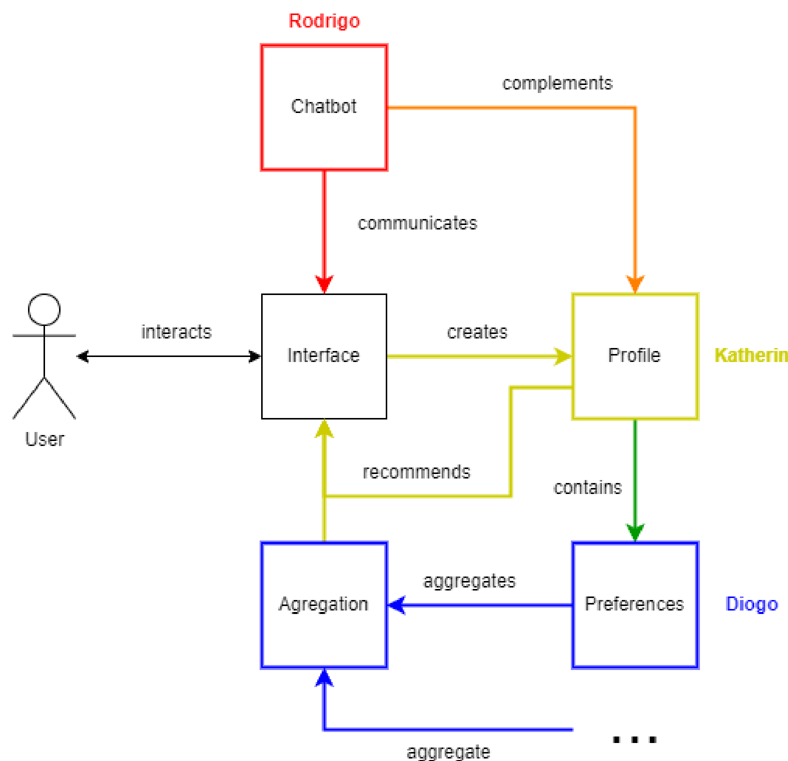


Fig. 1: Overview of KBAI project structure

3 State of the Art

This section will present the current state of the art for chatbots, starting with going over the first chatbots and their uses, then transitioning into the various components that can be included in a chatbot to enhance it. After going over how chatbots were first created and used, along with the elements involved with their creation, this section presents an overview of how chatbot technology evolved by incorporating various fields of AI. Additionally, it goes over the currently used chatbot design techniques and the main chatbot types.

The notion of a machine emulating human behavior was first introduced by Alan Turing in 1950. However, it took little time to develop automated systems, known as bots, that simulate human conversation [4] [5]. Chatbots enable users to interact with machines using natural language or through a graphical user interface with buttons and prompts [6] [4]. Chatbots, comprising distinct components, generally function in three steps: comprehending the user's natural language input, generating pertinent automated responses, and crafting realistic, "human-like" answers. However, chatbots' failure to grasp user intents can result in inadequate responses [4]. While chatbot types vary, they typically process user text inputs, extract keywords, employ rules, or leverage natural language processing to comprehend user intent, facilitating informative responses [6].

Alan Turing also created the Turing test to test whether a machine could think like a human. The test consists of a game with three players: player A (machine), player B (human), and player C. Player C cannot see the other players and only communicates with them through written text. They must then decide if player A or player B is the machine through the written responses [7]. A machine passes this test when player C cannot tell which of the two is the machine, meaning when the machine's responses are human-like to the point of being indistinguishable.

3.1 The first chatbots

Some of the first bots that could simulate human conversation were ALICE (Artificial Linguistic Internet Computer Entity) in 1995 and ELIZA in 1996. Each of these two chatbots utilized different techniques to achieve their goal. ELIZA used scripted responses, and ALICE employed NLP and artificial intelligence markup language (AIML) [4] [1]. These initial chatbots were characterized as social chatbots designed to engage in human-like conversations. Initially created for psychological therapy, they continue to serve this purpose, with ALICE progressing toward becoming an authentic conversational agent [2].

The concept of ELIZA is relatively simple. It is a rule-based chatbot [2] that relies on a pattern-matching algorithm and sentence reconstruction. Surprisingly, ELIZA achieved significant success in capturing user attention, generating heightened interest that considerably impacted the evolution of various other chatbots [5]. Despite these results, ELIZA had a straightforward implementation that did not consider context, leading to it failing the Turing Test [1].

ALICE is another of the first bots developed after the concept suggested by Alan Turing. It functioned differently from ELIZA, as it did not rely simply on a pattern-matching algorithm and sentence reconstruction. Instead, it utilized its custom markup language, AIML. AIML is an extension of the XML dialect; an example of this can be seen in Fig.2. This markup language is accessible and provides a standard way to encode chatbot behavior, it allows for the defined behavior to be exchanged between different chatbots if they also utilize AIML [5]. ALICE utilizes AIML to obtain matching patterns for which it also uses wildcard symbols, intending to improve

flexibility and the results [8]. These patterns are used to give meaning and context to user inputs through a depth-first search algorithm with backtracking, and when a match between the user input and a pattern in the template of the KB is found, it is used to generate an output answer [5] [1].

```
<aiml version="1.0.1">
<topic name="The topic name"
    <category>
    <pattern>User Input</pattern>
    <that>Last response</that>
    <template>Chatbot answer</template>
    </category>
    <category>
    <pattern>User input * </pattern>
    <template><srai>User Input</srai></template>
    </category>
    ....
</topic>
</aiml>
```

Fig. 2: Example of the AIML dialect [1]

3.2 Artificial Intelligence in Chatbots

An example of a chatbot utilizing AI is the chatbot made to help prioritize user queries in a college management system using NLP and its Sentiment Analysis component to prioritize user queries [9]. This chatbot can detect the user's "tone" meaning negativity, positivity, or neutrality, and prioritize the question accordingly. This system works by checking the user question against the knowledge database, meaning it has to be able to detect when users ask the same question differently. Some might ask the question more frustratingly, while others may ask it more calmly. If the question is found in the knowledge database, then the chatbot returns the answer corresponding to that question; otherwise, the question is answered by an admin, or in other words, a person, and then added to the knowledge database in conjunction with the question that leads to it [9].

3.3 Chatbot components

Chatbots can range from simple ones like ELIZA, which does not consider context, to more complex chatbots that combine artificial intelligence and machine learning to help the bot better understand the user by considering the context, user emotions, relations between words, and more factors. Chatbots have various components with social or "chatting" bots consisting mainly of a knowledge base, a chat engine, and an interpreter program [5] that help them understand user inputs.

The knowledge base is where the chatbot extracts the keywords, scripts, or the dataset to train its machine learning model if it uses one. When a knowledge base is more comprehensive, the chatbot has a broader range of answers for different topics and becomes less limited. Knowledge

bases can take many forms, with typical implementations for simple bots that utilize keywords being XML, AIML (e.g. ALICE), databases, and others [10] [5].

The generator allows the chatbot to create a coherent sentence for the user. However, it is only able to accomplish that task if the knowledge base has the required information (for example, the keywords match) and if the interpreter can correctly understand the meaning behind the user's words, utilizing various techniques like pattern-matching and subdividing sentences [5].

As referenced above, chatbots can vary from simple to more complex versions. Various technologies are involved for chatbots that utilize scripts or keyword recognition, such as artificial intelligence; this allows chatbots to have Natural Language Processing (NLP) and Natural Language Understanding (NLU). These help the chatbot understand the context and the relationships between words, with NLP also including Sentiment Analysis, which allows the chatbot to understand a user's "emotions". Artificial intelligence can be powerful and valuable when coupled with Human-Computer Interaction techniques. They allow chatbots to appear more human, leading to various benefits while giving answers more suited to the user's emotional state. The most advanced chatbots use machine learning coupled with the various artificial intelligence techniques mentioned, allowing them to learn from their interactions, remember previous interactions with specific users, and predict user needs. Additionally, instead of predefined solutions, they can generate their answer based on what the users tell them.

The rest of this subsection will examine the various chatbot components commonly used to improve chatbots and their capabilities to service the user. It must, however, be noted that a chatbot does not need to incorporate every single component. Often, only some of these components are integrated depending on the functions the chatbot must perform.

3.3.1 Natural Language Processing Component

Natural language processing is a chatbot component that incorporates AI [9]. It allows the bot to process user input into an understandable form. The processing of the user input ranges from simple tokenization (splitting a sentence into multiple parts) to understanding the various synonyms of a word, with the process of implementing NLP being facilitated by varied libraries with the necessary functions, an example being the Natural Language Toolkit (NLTK) [11] [10].

This component is also integrated with natural language understanding and sentiment analysis. NLU uses the techniques and processing available from natural language processing to understand the meaning behind user inputs. In contrast, sentiment analysis allows NLP to understand the tone of the user, classifying it into three options: positive, neutral, and negative.

3.3.2 Natural Language Understanding Component

While Natural language processing can process the user input, natural language understanding focuses on understanding its meaning using ML and NLP techniques. Due to the difficulty involved with developing an NLU from the ground up, there are a few widely available NLU systems that can be integrated into chatbots; among these are IBM Watson, Google DialogFlow, Rasa, and Microsoft LUIS [12]. Since there are multiple available options for NLUs that can be integrated into chatbot development, it is relevant to know the tasks where each NLU excels and the ones where they have poor results.

To help achieve this understanding, the article comparing multiple NLU systems [12] can be used. The paper tests and compares the results of the four natural language understanding systems. It found that while their performance varies from task to task, IBM Watson, for example, is the best at extracting unique entities (something that only appears once in its dataset, meaning the ML model has no pre-training on it) and also at intent classification. However, it differs from the one with the highest confidence score (the certainty of its result), with the best in this aspect being Rasa. These differences can be relevant depending on the priorities of the chatbot [12].

However, some techniques can help enhance these NLU systems. An example is increasing the dataset used to train the ML model, as it can be better prepared with more samples. This option requires the collection or access to a large amount of data, making it a time-consuming process, utilizing frameworks to help with identifying entities when these are expressed in new unknown forms while also changing the query structure (changing the position of words for example) to increase its adaptability to different contexts [12].

It should be noted that NLU and NLP have the downside of requiring a robust corpus; otherwise, the results might be skewed despite the engine having a high confidence score. Creating a robust corpus is a continuous task that requires frequent updates with additions to possible mentions and their respective intents.

3.3.3 Sentiment Analysis Component

Sentiment Analysis is a subset of NLP that essentially detects the emotional part of the user input, it can understand if the user is speaking in a positive, neutral, or negative tone (e.g. a frustrated user is likely speaking negatively). Understanding this can be helpful for the chatbot to know when to change their approach. One approach to SA is using a Lexicon [13], essentially a "vocabulary" made through a corpus. However, it has some problems as emotions are only sometimes transparent in text or voice and are more subtly displayed; hence, the chatbot would have difficulty distinguishing these.

A possible solution would be using ML with supervision. However, this means the data must be manually labeled, as detecting emotions can be difficult for a machine. Manually tagging the data would leave it prone to human error while making it even more time-consuming, in addition to training an ML model. There are multiple ways to create a Sentiment Lexicon with satisfactory accuracy in a somewhat automated way: taking an existing SL and adapting its domain to the desired one, utilizing graph-based label propagation algorithms on the web - this technique has the advantage of picking up some slang words and not just formal speech. Using graph-based label propagation algorithms on the web is done by using Matrix Factorization to completely automate the creation of a sentiment dictionary by using a social media corpus, allowing it to also pick up on slang (informal speech) and some word acronyms, using polar phrase extraction as an unsupervised approach, essentially assuming that "neighboring" sentences have similar emotional polarity (whether they are positive, negative or neutral). Social media is charged with emotions, making it a great resource to grab a corpus to create a SL, as it is where people go to express their moods and feelings via text or hashtags and emoticons.

Finally, there is the use of conjunction of rules on adjectives to automatically know the "meaning" of an adjective (if it is positive, neutral, or negative). This conjunction uses an algorithm to pair adjectives "joined" by connecting words such as and, or, but, either-or, and neither-nor, as when these words join adjectives, they follow detailed linguistic rules.

The downside of creating a complex SL for sentiment analysis is the biases that can be introduced through social media and the web. Words in social media have varying emotions depending on the context used and who is using them, as slang and words seen with a negative connotation can be appropriated and have their negative connotation become a positive one.

3.3.4 Human Computer Interaction Component

One thing all chatbots have in common is the interaction with the user; therefore, the importance of Human-Computer Interaction in this field should be acknowledged. Despite being more meaningful for some chatbots like client support or social chatbots, it still has various helpful techniques to help diminish user frustrations in case something goes wrong [14].

Chatbots can help augment the learning experience of students in class, specifically language courses, as they can keep steady conversations with the student, allowing them to practice while also providing higher accessibility to a training partner as chatbots are easily accessible anywhere, including on the phone. This accessibility also contains some customization in terms of the interaction, whether by voice or text [15].

Despite these advantages, there are some challenges for chatbots to be seen as viable conversation "partners" in social contexts. These challenges can be addressed by ensuring the chatbot has some desired social traits; a chatbot with a correct and rich vocabulary promoting a positive mood is preferred over a robotic chatbot [16]. Chatbots who communicate clearly, including using emojis, and appear to have human traits such as a personality a name and appear to have emotional or social intelligence such as empathy or giving suggestions as a way to "add" to the conversation [16] [14] [17] are also preferred.

3.3.5 Machine Learning Component

Chatbots can also incorporate ML like the bank chatbot in [11]. The main goal of this chatbot was to help provide customer support in an "automatic" way. To create this chatbot, they first needed to compile a dataset with frequent customer questions, doubts, and their respective answers. This dataset went through the usual pre-processing steps that help achieve better ML results. New additions from user interactions also go through this pre-processing with the help of a library for NLP. The pre-processing consists of tokenization and lemmatization (knowing when different words have the same meaning), additionally removing punctuation marks and extra spaces to diminish ambiguity. To further improve the results, this data is also turned into a different format [11] [4].

As the dataset grows through user interactions, it helps the chatbot have a broader understanding of user queries but creates the problem of increasing the time it takes to find the answers. There are, however, various solutions, one of them being Classification. Classification essentially consists of grouping answers into "groups", and when the machine learning model is asked a question, it tries to classify the query into one of those groups, limiting its search to a specific group as opposed to the entire dataset [11].

The major hurdle in using a ML component in a chatbot is related to the difficulty of amassing an extensive and complete enough dataset that needs to be labeled correctly to be helpful to the chatbot; this process is particularly time-consuming and only sometimes feasible. In the long run, the chatbot becomes self-sustaining by building on its existing dataset from its various interactions.

3.3.6 Chatbots and Knowledge Bases

Although the various components chatbots can use, as mentioned above, are relevant to know and understand for their development, it is also essential to have a good knowledge base for the chatbot. Regardless, building a knowledge base is not as heavily focused upon due to the available knowledge graphs and knowledge bases that can be utilized in chatbots [1]. A tree is one of the most common ways to organize a knowledge base for use in a chatbot, but it can represent and utilize knowledge in different ways. Each node could represent a chatbot response or action to the user input. The knowledge base can also be split into multiple knowledge units, each representing a response to a single query. Using knowledge units would create a requirement for the chatbot to use machine learning to help navigate and make use of these knowledge units [1].

Yet, despite knowledge bases and graphs being convenient ways to represent knowledge, it can also be helpful to transform a knowledge base into natural text to create a corpus and vice-versa, as this can automate building a knowledge base. However, these processes are challenging, as natural text lacks the same grouping and relationships. Some tools were developed and are available to aid in the process of transforming a knowledge graph or knowledge base into natural text [18] such as TeKGEN, a model to help translate a knowledge graph into text whose result can be further improved through the use of an existing corpus and different methods of transforming data into text [18]. There are also tools such as the DSTLR to transform natural text, such as documents or articles, into a knowledge graph that can later be integrated into a knowledge base to complement its ability to answer user queries. The DSTLR process consists of taking a document store and using NLP to identify and extract entities, mentions, and relationships between them from the documents, finalizing with using an external knowledge graph to help further complete and complement them [19].

As previously shown, a chatbot knowledge base does not always contain all the information needed, and it can be helpful to extend it through external sources with accurate and relevant information, such as Wikipedia or Wikidata; these can be used when the chatbot knowledge base lacks the means to answer a user query. In [20], through an API, this extension of the chatbot knowledge base can be facilitated by simply passing the question to the API, essentially asking the external knowledge base the same query.

The difficulty with using knowledge bases comes from their complexity, mainly in building the knowledge base itself and establishing various ontologies for the necessary communication. Using a knowledge base implies that the chatbot would need to have a specific ontology that it can use to communicate with the knowledge base. The ontology must be well planned out, as changing it once the components are connected and working can be a tricky task that introduces various problems, such as the chatbot or the KB having faulty communication.

3.4 Chatbots evolution

Initially, chatbots started using simple pattern-matching techniques, as seen by ELIZA and ALICE. Over time, as technology evolved, new chatbots started using different languages for this approach, starting with AIML, which then transitioned to Rivescript and Chatscript [21]. When it became possible to implement theoretical machine learning models, chatbots began to make use of these in their implementations, leading to a significant evolution in chatbot development; this permitted chatbots to make use of NLP and NLU, meaning they started to understand conversa-

tions as a whole in contrast to chatbots that used pattern matching as those were only capable of understanding the current interaction [21].

As technological advancements and advancements in some areas that chatbots use, such as ML, AI, NLP, and, to some extent, HCI, are made, they benefit the development of chatbots. Notably, there are now various accessible Natural Language Understanding (NLU) platforms and chatbot services that readily integrate the NLU component into chatbots [12] [22]. Additionally, HCI advancements pertinent to chatbots acknowledge their potential to fulfill social needs beyond being mere bots. Achieving this involves integrating human-like social characteristics into chatbot conversations, which enhances user satisfaction and recognizes chatbots' capacity to serve as social companions [23].

Machine Learning can enhance specific chatbots by creating more natural interactions, reducing the sense of conversing with a machine. For instance, the ALICE chatbot was augmented with ML to automate the construction of AIML using an existing language corpus [24]. This addition of ML significantly expedites and automates adapting a corpus into AIML format, which chatbots utilize. The increased accessibility to creating chatbots and the improved understanding of enhancing interactions have led to their integration into everyday tasks. These tasks encompass functions such as personal assistants and purchasing aides on online platforms. These chatbots engage in seamless conversations, aiding users in accomplishing their objectives [2].

3.5 Chatbots Design Techniques and Types

There are various ways to label chatbots based on their specific tasks and areas where they excel, from menu-based chatbots that use decision trees for their process being more focused on helping the user navigate a website [22] to Reinforcement Learning (RL) chatbots that can learn over time from their interactions with the users having the ability to remember a user and their preferences making it capable of offering them suggestions based on these [8], this also means chatbots can either have predetermined answers such as public transportation schedules or dynamic answers making them more flexible as they generate their response based on the interaction [25].

The following is a list of the various chatbot types, along with an explanation of their design techniques:

Template Based Chatbots employ predetermined responses chosen by comparing user queries to predefined patterns. Often, these chatbots utilize AIML, which offers a standardized means to encode chatbot behavior via template and pattern recognition. A notable instance of an AIML-based chatbot is ALICE, which employs a depth-first search with backtracking to match patterns to templates in the knowledge base, prioritizing the most comprehensive and accurate template for user responses [8] [5] [1]. These chatbots are common due to their straightforward design and the ease with which output can be modified by adjusting AIML content [8].

Menu Based Chatbots are very common and easy to use; they essentially present the user with buttons that have text on them, and the user clicks the appropriate one, the chatbot then proceeds along the decision tree, updating its interface and, if necessary, giving some feedback in the form of text or answering the user [22]. These chatbots have the disadvantage of being relatively slow and require the user to often go through the decision tree compared to simply asking the question in text.

Rule Based Chatbots function with predefined rules by the developers; these rules dictate the conversation flow and direction. Some examples of rule-based chatbots are ELIZA and PARRY, used in psychotherapy. The simplicity of rule-based chatbots makes the creation of these accessible when it comes to simple tasks like helping a user navigate a website or book a flight. Despite this, when it comes to more complex tasks, there is also the requirement of having more rules and creating rules can be time-consuming. These chatbots are not helpful when the user differs from the defined conversation flow [2].

Keyword Based Chatbots employ natural language processing alongside keywords to comprehend user queries in context and meaning. Using keywords involves a personalized list that guides the chatbot in interpreting questions through an algorithm. However, limitations arise when distinct questions share identical keywords, causing these chatbots to falter [22].

Reinforcement Learning Chatbots chatbots possess the capability to learn from their interactions. Through engagement with users, these chatbots can discern positive or negative outcomes, progressively improving with each interaction, essentially making it so just being used enhances them.

Initially, reinforcement learning was conceived using Markov Decision Processes and a Speech Language Understanding component. With advancements in technology and the utilization of artificial neural networks, reinforcement learning has transitioned to utilizing ANNs and a natural language generation component [2]. Currently, reinforcement learning chatbots are predominantly retrieval-based, considering the context for accurate responses. While they offer advantages over corpus and template-based counterparts, these chatbots face the challenge of acquiring vast conversational datasets for training [8].

Hybrid Chatbots each approach to constructing a chatbot presents its own set of strengths and weaknesses. Efforts have been made to combine different techniques, resulting in hybrid chatbots. For instance, one hybrid approach involves combining a seq2seq model with a reinforcement learning chatbot. This strategy capitalizes on the meaning representation of seq2seq models and combines it with the learning capabilities of a reinforcement learning model, thereby enhancing chatbot performance [2]. Another hybrid method involves merging a dialogue system with a corpus-based chatbot, and this entails supplying the chatbot with retrieval outcomes from the dialogue system, with additional optimization achieved through word clustering and topic matching [8]. However, it is important to note that hybrid chatbots also inherit the drawbacks associated with both approaches. Yet, these drawbacks are often mitigated or addressed through the amalgamation of techniques.

Sequence to Sequence Chatbots when chatbots employ Sequence to Sequence (Seq2Seq) learning, they utilize a specialized and intricate neural network called Recurrent Neural Network (RNN). Seq2Seq learning was initially proposed for chatbots due to its encouraging performance in phrase-to-phrase machine translation tasks. These models can map input and output sequences of varying sizes [2].

Seq2seq models possess both an encoder and a decoder, using an encode-decode architecture as depicted in Fig. 3. The encoder transforms the input into a vector, effectively capturing its meaning. Subsequently, the decoder employs this vector to generate an anticipated output based on its training data.

It is important to note that these chatbots are not necessarily confined to matching user queries with predetermined query-response pairs. Instead, they directly generate responses based on their training datasets. Consequently, they excel at autonomously generating responses rather than retrieving them, making them well-suited for roles such as social chatbots [8] [2].

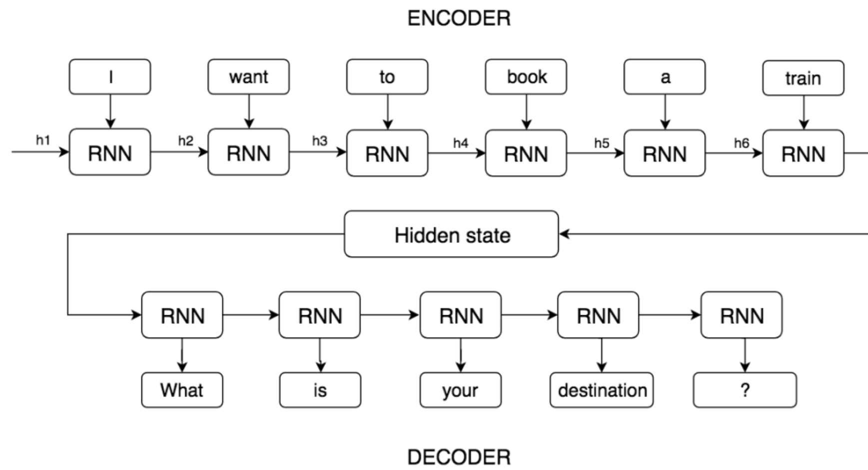


Fig. 3: Example of a Seq2Seq architecture [2]

3.6 Conclusion

Throughout this chapter the literature related to chatbots was covered, starting from the first introductory concepts to understanding chatbots such as the Turing test and how the concept of chatbots was initially introduced. Then the first chatbots mainly ELIZA and ALICE were covered as these were important in their time, with these core concepts covered the next part covers the various components that can be integrated into chatbots followed by the various types of chatbots that can be developed alongside for design techniques for each.

4 Tools used

This chapter will review the various tools used during this dissertation's development and why these tools were picked over their alternatives. In addition, it will also go over the multiple options found when researching for tools to assist with the goals of this dissertation.

4.1 Diagrams.net

Diagrams.net was used to create the necessary diagrams for the planning phase of development. It can be installed locally on the device and the browser. Diagrams.net is a graph drawing software with a simple interface that can be used to create flowcharts, wireframes, use cases, UMLs, and others. The various diagrams seen throughout this dissertation that were used for the development of the chatbot were created using diagrams.net.

4.2 Visual Studio Code

The development of this project initially started with using Visual Studio Code because of its flexibility with working on various languages. Visual Studio Code is a source-code editor developed by Microsoft to work on multiple operating systems. It offers crucial features for development, such as debugging, syntax highlighting, code completion, refactoring, and integrated version control. It allows users to install various extensions to support more languages or extend upon its existing features. This project used these to add support for React, JavaScript, and PHP.

4.3 PHPStorm

Once Botman was chosen as the framework, development switched to using Jetbrains *PHPStorm*. PHPStorm specializes in PHP development, including working with standard tools in this area, among these Laravel, Symfony, and WordPress. This tool offers intelligent suggestions during coding, including following naming conventions, quick refactoring, error prevention, lifting members up, and others. It also has integrated version control, integrated tools to use command line commands, and the use of Docker, Composer, and others. These integrations are relevant as Botman uses both Composer and Laravel for its development.

4.4 Botman dependencies

Additionally, since Botman runs on a PHP environment, Windows Apache MySQL PHP (WAMP) was used to install and run this environment. Botman offers Botman Studio an environment for easier development to test the code while reflecting real-time changes. It makes use of Laravel, an open-source PHP framework based on Symphony. With Laravel, Botman can dump variables to the console and give a debugging log in case of any errors or unexpected behaviors, and this helps speed up development as Botman does not crash but defaults to not replying in case of a mistake and then resetting. Finally, to manage its various dependencies, Botman requires Composer, an application-level dependency manager for PHP, which provides a standard format to manage dependencies for PHP software and required libraries.

4.5 Chatbot Frameworks

Multiple chatbot frameworks were analyzed, with their upsides and downsides, to find the correct framework for the KBAI project. This framework has to allow full control over the chatbot and NLP engine, be hosted locally, and need no external cloud services. The results of the analysis are as follows:

Microsoft Bot Framework is Microsoft's solution to creating chatbots; it offers software development kits for working in C#, JS, Python, and Java, allowing developers to pick their preferred language. It also provides the feature of converting speech-to-text through machine learning, allowing it to translate speech and give voice to the chatbot through text-to-speech. This framework facilitates the integration of other various Microsoft services. While aimed at developers with its code-driven approach, it offers multiple templates for practical examples of its distinct functions.

This framework's downside is its dependency on using other Microsoft services to augment its features. The framework has no built-in NLP or NLU depending on Microsoft's LUIS for these tasks, with the same being said for the speech-to-text as mentioned earlier and text-to-speech features as these use the Unified Speech Services. Although this framework is open-source, as its code is freely available on GitHub, when it comes to hosting the chatbot, it heavily encourages the developer to use another Microsoft service, Azure Bot Service, as it is both language and software development kit independent while also being one of the easiest to implement. On top of the previously mentioned costs that come with using the various Microsoft services as they have subscription plans, using the chatbot created with the framework will also have an additional price as the free plan is limited to only ten thousand messages each month, making it unable to scale with the project without incurring additional fees.

Rasa is an open-source framework oriented towards contextual chatbots. To train the chatbots built with this framework, Rasa allows the developer to create user stories initially. These user stories are essentially a set of user intents and correspondent chatbot actions. With the initial training from the user stories, the chatbot will then learn through their interactions with various users using machine learning, meaning the bot can store user data, which is beneficial to the KBAI project. Unlike Microsoft's Bot Framework, Rasa has a built-in NLP engine, eliminating the need to send data to a third-party API for processing, centralizing the chatbot process. Lastly, Rasa offers various services to assist with chatbot development, from the Rasa Pro infrastructure to a more visual approach to development through Rasa X, including various other tools to improve the chatbot and update it over time.

For its downsides, Rasa uses spaCy to process the various user inputs, which consumes considerable memory. It also does not offer complete control over the processing of the dialogues, and this control is desired for the KBAI project. Lastly, Rasa is aimed towards developers with previous experience in using NLP and creating chatbots; despite not necessarily being a disadvantage, this makes the framework harder to use correctly for beginners, somewhat slowing down development.

Wit.ai is an open-source chatbot framework belonging to Meta; it has good documentation for its API, making it accessible. Wit.ai is focused on conversational chatbots and understands over 132 languages through its NLP engine. Wit.ai is free, although it has the restriction of rate limits and all the data (intents, entities, traits, utterances processed) being accessible

to the community. This framework offers software development kits for various programming languages: Python, Node.js, Ruby, and iOS. One of its most substantial points is the NLP engine, a strong contender with both IBM Watson and Microsoft's options for NLP and NLU. Wit.ai is meant to be easy to integrate into channels, offering out-of-the-box interactions with some websites and mobile apps; this easy integration into third-party applications also provides a text-to-speech and speech-to-text feature.

Despite its strong NLP engine and free accessibility, Wit.ai is limited in the control offered to the developer as it is built through an API, and the chatbot is both trained and hosted in Meta's cloud service, imposing a rate limit and making a lot of the data used for the NLP and training public.

BotPress is a chatbot framework powered by OpenAI, the creators of chatGPT; it is free, open-source, and has good documentation. It uses a modular architecture with a pre-built NLP engine, a chat emulator, and a debugger to assist development. With its visual flow editor, this framework offers a visually oriented approach to easily create conversations that feel more human-like. It offers various templates to start developmental prototypes that can be augmented through the visual flow editor. There is built-in chatbot analytics for further refinement of the chatbot. It also allows the user to define intents and entities through its NLU modules that offer over 100 languages.

However, the framework is limited regarding the types of chatbots that can be implemented, as it focuses more on rule-based chatbots that follow a set of rules and interaction paths. Alongside this, it is a cloud-based framework imposing a one thousand messages per month limit on its free plan, making it not a viable alternative to a scalable solution without incurring additional fees. It being cloud-based is also a limitation as for the KBAI project, the chatbot must be hosted locally on the KBAI servers.

IBM Watson is IBM's open-source alternative for chatbot development; it offers chatbots that have already been trained by vast amounts of data, including many Wikipedia entries, and then adapted to be communicative to users. With this training, it can use machine learning to understand user text inputs in various languages and classify them accordingly. IBM Watson has a high focus on assuring security and privacy; when it comes to user data, this is achieved by storing the information in a private cloud. However, it can be a risk, as in the case of the KBAI project, this data is meant to be stored on the KBAI servers. On top of this privacy focus, the framework also offers analytics on the various interactions, a valuable addition as it allows developers to identify better where changes are needed.

Despite offering multiple chatbots as documentation examples, the framework is considered difficult to use for unfamiliar developers. However, it can be deployed onto websites, but maintaining the caveat of the chatbot being cloud-based and hosted on the provider's side, having restrictions when using the free plan, leading to scalability concerns without added fees. Lastly, the framework focuses more on chatbots with high security and privacy measures for enterprises. Although these are valid concerns, they are not the focus of the KBAI project.

BotKit is an open-source conversational AI that offers developer tools for managing chatbot content and analytics for its various interactions. This framework is supplemented with frequent community support and extensive documentation on its workings. This documentation is beneficial as it is code-oriented despite offering a visual conversation builder. Besides its community

support and extensive documentation, this framework was built with the focus of being easy to work with for developers, which its visual conversation builder supplements along with a system meant to handle the various dialogues. This framework allows developers to pick from various NLP engines as it has no NLP engine, using Microsoft's LUIS by default. Concerning integration with third-party applications, it offers some integration with popular ones like Slack, Facebook Messenger, and Microsoft Teams.

For its limitations, development is limited to *Node.js* and has no built-in NLU or NLP. As previously mentioned, it uses an external product for this purpose, Microsoft's LUIS. This framework has also been fully integrated into Microsoft Bot Framework, gaining some of its advantages and disadvantages from this integration.

OpenDialog is a visually oriented chatbot framework; it uses a visual editor to handle the chatbot prototyping and design, giving it the capacity of being a mostly code-free approach. This framework offers full support to integrate external NLP/NLU engines to handle and assist in processing user inputs, and it is possible to deploy locally through Docker.

As development for this framework is mainly focused on its visual editor, it offers less control to the developer than a code-based approach. This editor can help prototype small chatbot interactions to understand how they would work and make small, controlled user tests.

Botonic is an open-source React chatbot framework focused on building chatbots for conversational apps. This framework offers various templates to start development and gives practical examples of its possibilities. The framework has built-in support for various services like Google Analytics to provide developers with analytics for their chatbots and some NLP/NLU engines for better processing of user inputs. The development is code-oriented, with the various conversations and their flows defined in code and optional menu buttons. Chatbots created with Botonic can be hosted locally, eliminating the dependency on third-party cloud services and their restrictions. It even offers support to platforms such as browsers and mobile.

However, This framework focuses on simpler chatbots like menu and keyword-based chatbots, lacking the machine learning components necessary for more complex iterations. The framework being built in React can be disadvantageous, requiring developers to know both JS and React to use its features in their entirety.

Claudia Bot Builder is a JS bot builder to make the chatbot creation process quick and easy, with the main idea being to have all the advanced features more complex frameworks offer while maintaining simplicity. This bot builder allows for creating and configuring a chatbot that integrates into various platforms from the start, as the bot builder already creates webhooks for platforms like Slack, Telegram, and Facebook Messenger. It also offers different types of UI auxiliaries, like functions to create custom buttons.

Although the bot builder offers a quick solution for making a chatbot, it does not provide high-level control as it creates the various webhooks automatically and handles all the back-end processing of the messages from connected platforms. Since it comes with its webhooks and is mainly meant to be run along with Amazon web services to handle the scalability, it makes it somewhat difficult to host the created chatbot locally as Claudia bot builder uses Amazon web services API gateway.

Tock is The Open Conversation Kit, an open-source platform to create chatbots fully independent from any third-party API, but it does offer the possibility of integrating external APIs. Tock offers programming support, mainly in Kotlin, with the option of using Node.js and Python. It can connect to various text and voice channels from other applications like Messenger, Alexa, Twitter, and Microsoft Teams, among others, through its connectors. It does not offer an NLP engine of its own but instead chooses to use existing NLP and NLU engines by allowing easy integration of these, such as CoreNLP and Rasa. This framework offers a UI to assist with development. This UI includes the ability to check user analytics and create decision trees alongside conversation stories that can be used to train the chatbot without coding. It also offers the possibility of locally hosting the chatbot through Docker supporting containers, providing scalability.

Botman is a chatbot framework, an agnostic PHP library meant to assist with creating chatbots and integrating them into various platforms like Slack, Telegram, Messenger, and Microsoft Bot Framework, among others. Botman gives the ability to integrate various frameworks into itself to enrich its features further. This framework offers good documentation for new developers and Botman Studio, which uses Laravel to provide additional tools for development. Botman Studio allows users to test and see the changes to their code in real-time. Botman allows developers to save conversation states and return to these later without the user needing to go through all the steps again. For NLP, Botman allows developers to integrate external NLP engines through its middleware system, coming with compatibility for both Wit.ai and DialogFlow out of the box.

This framework offers the use of buttons in its interactions, allowing users to implement more complex buttons like those from Telegram or Facebook Messenger by implementing a connection to their APIs. However, integrating external APIs means every message will be passed to them. Aside from incorporating external APIs to extend its features, Botman allows developers to define custom functionality and create new features.

Bottender is an open-source framework to build "conversational user interfaces", e.g. chatbots; this framework is focused on alleviating the necessity to build the UI, allowing developers to designate actions corresponding to the current user input and state of the conversation. This framework uses automatic request batching, making multiple API calls in a single request to enhance performance. It will also use graceful degradation and progressive enhancement to support various platforms.

Bottender offers both documentation to assist developers and community support through Discord. This framework is built on top of various messaging APIs, providing easy configuration when working with multiple communication channels (Slack, Telegram). However, this makes it less ideal for developing an independent chatbot that can run by itself on the KBAI servers as required.

DeepPavlov is an open-source chatbot framework that gives developers the tools to create more advanced chatbots that fully use machine learning to adapt to their tasks over time. To start development, DeepPavlov offers various deep learning models already trained, always allowing the developer to re-train these models with new data should they need it. It features integration into REST, Socket, and Amazon web services and allows for the implementation of third-party APIs for easier deployment and development of a chatbot. This framework is made to be easily

deployable, allowing developers to use Docker and its containers to even host individual trained machine learning models. One of its limitations comes from using various PyPI models, which require a CUDA-capable graphics processing unit to run.

After reviewing various frameworks A.1, Botman and Botonic were chosen as the primary framework candidates as they can be hosted locally, offering high control and allowing for the easy integration of NLP while maintaining high control over the NLP engine, they accomplish this by facilitating the integration of a third-party NLP engine. Furthermore, the data to train a more advanced machine learning chatbot model like DeepPavlov was unavailable. The other frameworks raised concerns with scalability or control, as they either required subscription services as they are hosted through cloud services or did not allow complete control over which NLP engine to use and its corpus.

4.6 NLP Libraries

Alongside a chatbot framework an NLP library that can be easily integrated into the chatbot is needed, preferably through an API. The library has to allow hosting locally and give complete control over the corpus. The research identified various NLP libraries developed in different languages, with Python and JavaScript being the languages with the most libraries.

Catalyst is an NLP library built in C#, supporting .NET standard 2.0 and .NET core. It is open-source, having its source code freely available on GitHub. Catalyst offers tokenization, named entity recognition and comes with models trained on the Universal Dependencies project (a framework for consistent annotation of grammar in over one hundred languages) and models to understand and learn various abbreviations. It offers support for training with FastText, a library to aid in text classification, including language detection and representation learning, alongside StarSpace, a neural model for entity embedding and other tasks to assist the NLP engine. Finally, it uses lemmatization to identify words that share the same meaning and comes with various models trained in entity recognition. This NLP engine is relatively simple to install as it comes in the form of a NuGet Package with no additional dependencies.

OpenNlp is an open-source NLP library, providing the developer with various tools to assist with NLP in C#. OpenNlp provides the following features: sentence splitting, tokenization, a part-of-speech tagger, chunker, coreference (e.g. Mary is coreferent to she), named entity recognition, and tree parsing. Despite being a NuGet Package, making installation simple, this library only supports the most basic tools and tasks needed for NLP.

Spacy is an NLP library built in Python and Cython; it is modular with the goal of always being updated with new research. Spacy offers pre-trained pipelines with tokenization and training in over 70 languages, making extensive use of machine learning models and offers various NLP related tasks such as tagging and parsing text, sentence segmentation, lemmatization, morphological analysis, entity linking, named entity recognition and classifying text. This library can be extended by custom-made components, allowing a developer to introduce new features.

Polyglot is a free natural language pipeline developed in Python, providing support for various languages in its features. Polyglot is capable of tokenization, detecting the language for a text, named entity recognition, part of speech tagging, doing sentiment analysis on text, word embedding, doing a morphological analysis, and automatic translation to different languages. It provides documentation with practical examples of use cases for its features.

PyNLPI is a Python library for natural language processing; it offers various models, each separately focused on a different NLP task. This modularity allows developers only to use the necessary modules for their tasks and later add modules as their needs expand. For its modules PyNLPI offers the following features: a module to provide extra datatypes like patterns, a module for tasks related to evaluation, parameter search and progressive sampling, a module to handle parsing of the Corpus Gesproken Nederlands, a module to handle working with documents in Format for Linguistic Annotation (FoLiA) along with a library for FoLiA Query Language, a parser for Corpus Query Language, a module containing various search algorithms, a module to handle text processing containing tokenization and a module to use a pre-built language model that is responsible for analyzing text and providing word predictions.

Natural offers various operations for natural language processing developed in JavaScript using Node.js. Natural offers tokenizing and treebank tokenizers, stemming (to get the root form of words), text classification, a simple sentiment analysis algorithm, phonetic matching, Term Frequency–Inverse Document Frequency to determine the importance level of words in a corpus, integration with WordNet to look up various word definitions, and identification of similar words like synonyms.

Wink.js is a natural language processing library developed in JavaScript to be performant, accurate, and intuitive. This library has no external dependencies and supports various NLP features such as tokenization, sentence boundary detection, handling negation of tokens, sentiment analysis, named entity recognition, part of speech tagging, and custom entity recognition for entities defined by the developer.

Sentiment is a module for Node.js that performs sentiment analysis on text inputs pre-trained with the AFINN-165 wordlist and Emoji Sentiment Ranking. The focus of Sentiment is to be highly performant while allowing developers to extend upon it by adding additional data to train its sentiment analysis in new languages and letting developers define their approaches for sentiment analysis in each language.

NLP.js is a natural language processing library developed in JavaScript for Node.js. NLP.js supports most generally used functions such as language guessing, searching substrings of strings, using stemmers and tokenizers, sentiment analysis for sentences with support for negations (when a user uses "not"), recognition of similar strings, named entity recognition and named entity management, a classifier to categorize utterances into their respective intents which can be defined in the corpus and a tool to manage usage and training of its various features in multiple languages called NLP manager. This library also offers out-of-the-box support for 40 languages, with this number increasing to 104 if Bidirectional Encoder Representations from Transformers (BERT, a family of language models) is integrated; on top of these languages, NLP.js also supports any language, including imaginary ones through intelligent use of tokenization.

Due to being rich in features *(NLP.js* became a strong candidate as it can detect language, supports various languages, and has entity recognition and sentiment analysis. Importantly, it is also hosted locally and allows total control over the corpus, meaning the developer can make their custom corpus with their custom entities and associations.

In addition, *NLP.js* and the other libraries share similar features, and despite not all being fully used for the planned initial development, one of the goals is to allow future expansion easily.

The other libraries share similar features with *NLP.js*, and none have any features that make them stand out, as the main differences are the language they are developed in alongside the number of languages supported. With quite a few of the NLP libraries not supporting the same amount of features as *NLP.js* does, however, the vast majority has support for significant features such as entity recognition. However, as previously mentioned, one goal is to allow the easy addition of new features in the future. For these reasons, the chosen NLP library is *NLP.js*, as it also allows for the easy creation of an API that can be integrated into the chatbot.

4.7 Conclusion

This chapter went over the various tools used during the development process. It also covers the various alternatives found during research of frameworks for developing a chatbot and for choosing an NLP engine; after covering the various alternatives, the chosen option is highlighted, and an explanation is given on why it stood out from its peers.

5 Development

This chapter will review this project's case studies and various development stages for the generic chatbot framework. As proof of concept, the tourism case study was utilized, with the booking API implemented. The chatbot also implemented an NLP API, with the chosen NLP engine being *NLP.js*. This generic chatbot framework allows any client to customize the various chatbot dialogues they intend to have available to their users. These dialogues can have buttons for the user to click or ask questions, possibly employing API operations such as listing available services in a city or the tariffs for a chosen service.

The development was broken down into various steps, beginning with taking account of the various requisites the chatbot had to fulfill. A development plan was created, and two promising frameworks were tested to choose the most appropriate. The implementation started with creating hard-coded examples of chatbot functionality that were then dynamized over time to take the dialogues from a *JSON* file with a custom-built structure. Furthermore, the chatbot transitioned to utilizing NLP to understand user intents and infer what dialogue to load from the intent. The chatbot fetches the dialog from an API containing the various dialogue templates created in an online editor by the client.

Fig.4 outlines the development process used for this project The following subsections go over each development phase of the chatbot along with their challenges in further detail.

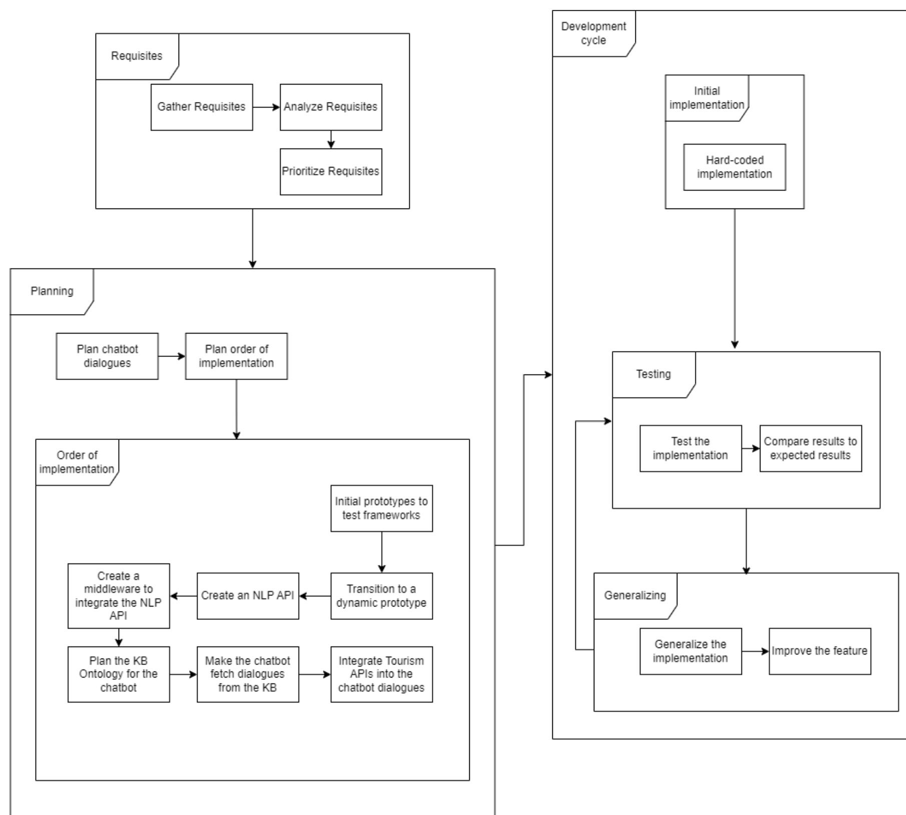


Fig. 4: Chatbot development cycle

5.1 Case Studies

The KBAI Project has three primary planned case studies: tourism, project management, and news management. From these, tourism is meant to serve as a proof of concept for the chatbot implementation; nonetheless, the chatbot must remain flexible as it is meant to adapt to the various case studies through simple changes.

5.1.1 Tourism case study

In regards to the tourism case study, the chatbot is meant to assist users in their various tasks for traveling; this involves not only helping a user plan out their trips by suggesting activities, restaurants, hotels, and landmarks but also making these processes easier for the user, as instead of having to navigate various websites that can often be confusing and tiring the user can provide the information to the bot, which will then list out their various options and automate the process. In addition, the bot will also have the capacity to serve as a personal secretary to the user, letting them quickly change their bookings and manage their schedules. The personal assistant feature will be augmented by using the other APIs of the KBAI project to make user recommendations based on their profiles and interactions. Besides making the process for the user more accessible, the chatbot is meant to assist the client's business by allowing complete control over the dialogues without any need for coding, as the dialogues will be edited through the knowledge base editor. This ease of access can allow the product to be used by various clients with entirely different dialogues, including different languages depending on the client's localization, as the dialogues are all created and configured by the clients.

For this case study, the chatbot will use a natural language processing API so the user can have a more natural interaction with the bot through text inputs; this will complement the feeling of the chatbot being a personal assistant. Beyond making the conversations with the chatbot feel natural, there is also the need to make the various dialogues easily editable by clients; this will be achieved by connecting the bot to the knowledge base so the dialogues are located on the knowledge base and fetched by the chatbot depending on the user intent that the NLP API identifies on the first user input. Nevertheless, clients can add buttons to their websites that automatically begin their respective dialogues and open the chatbot window. For the chatbot to know the availability of the various places users can book and the overall information it will need to present to users, it will make extensive use of various APIs, connecting to them through post requests, making the entire process hidden and automated in the eyes of the user, and will then display the information to them. This connection must be generic, so every time a new API has to be added, there is no need to make extensive changes to the code-base, but instead, call the method that handles the API communication and then process the information.

5.1.2 Project Management case study

For project management, the chatbot aims to assist users unfamiliar with the tools, achieving this by allowing users to do all the operations that can be realized with the tools through the chatbot by using natural language. Beyond helping new users unfamiliar with the tools used in project management, the chatbot will also serve the other spectrum, users who are familiar with the tools, by speeding up the process through chatting. With these two combined, the chatbot seeks to make project management faster and easier for all members involved, eliminate the need to learn a new tool for project management, and make the tool's usage faster for those familiar with it.

As project management often differs with each company and team, the chatbot will allow clients to make their dialogues. In this way, clients can create a dialogue for every feature their project management tool has or only for the features they consider critical, problematic to learn for new users, or tiring to repeat frequently for more experienced users. A core project management task is managing the various tasks; hence, the chatbot will allow users to ask for a list of their tasks. On each task listed, there can be options to interact with the task, and the client can configure these. This feature of using buttons with options is optional; however, it can help manage a project's tasks and assign various team members, as the dialogues can give different options for a project manager or team member. These dialogues will be configured through the knowledge base editor, and the chatbot will use NLP to identify user intents and know which dialogue to run depending on their inputs. Additionally, the bot will use various APIs to complete the tasks in project management that will be requested through the dialogues.

5.1.3 News management case study

Concerning news management, the chatbot's purpose is to assist users of Presspower with navigating the large amount of articles and news available quickly. It is meant to augment the existing functionality Presspower offers its users with the ease of access of a bot. Additionally, the chatbot will allow users to execute the same functionalities as the website while assisting with building the user profile through its interactions.

Users can ask the chatbot for articles related to specific topics or authors and quickly get a list of titles to which they can add notes or choose to read. Beyond navigating the available news articles, the chatbot will assist the user with building their profile, and this profile is used to know what articles the user is interested in and what recommendations and news recaps should be sent. The chatbot will assist with the profile building by remembering user choices and requests from its interactions and giving these to the knowledge base, with this information then being used by the respective APIs to build the user profile. In addition, the bot can also offer questionnaires to users to further complement their profile, along with asking for feedback at the end of a recommendation.

5.2 Chatbot requisites

The KBAI Project links its various parts through a KB, from which the chatbot is meant to communicate with the recommendation component and get its configurations. The chatbot is also meant to be hosted locally, avoiding dealing with possible difficulties regarding scalability and control that can arise when using cloud services such as Microsoft's Bot Framework. Cloud services limit the number of monthly messages through various subscription services and often do not allow maximum control of the NLP. From the knowledge base, the NLP corpus will be fetched and updated over time, enhancing the chatbot's ability to understand the user.

When adding the chatbot to a website, the process has to be simple and allow the chatbot to use newly created knowledge base dialogues without any changes. In addition to being dynamic, as the project involves multiple case studies, the bot must be flexible in its implementation of APIs to allow for the easy integration of the various APIs from each case study. A clean and maintainable code base will augment its flexibility.

5.3 Planning

Research had been previously done on the various types of chatbots; this research was compiled into an Excel document with two pages, with the first B.1.1 outlining the various types of chatbots previously mentioned along with short examples of how user interaction with each chatbot would go, some advantages and disadvantages to each type along with their limitations and areas where each excels. The second page B.1.2 gives a short description of how each chatbot type works along with some example inputs (data) that can be required for their development while also containing some example outputs (chatbot replies or back-end actions) of what would happen during interactions with the user.

While menu-based chatbots provide simple and intuitive navigation for the user, they can become convoluted to maintain as their decision tree expands with additional dialogues over time. This complex decision tree can lead to interaction delays as the chatbot iterates through the tree to find appropriate responses. Additionally, these chatbots can be frustrating for a user if they choose the wrong option at one point and need to restart the entire process to correct their mistake, as they do not allow the user to search for what they desire.

On the other hand, keyword recognition chatbots use NLP to recognize the user intent, allowing the user to interact with the chatbot more naturally through text inputs instead of navigating through menus in a GUI. Despite this advantage, this type of chatbot is limited when it does not recognize the user intent as it will not be able to answer the questions prompted correctly and can give erroneous answers in case the question has keywords similar to a separate question, leading to the chatbot confusing the two user intents from both questions.

From the previous analysis B.1.1, B.1.2 the chosen chatbot type for this project was a Hybrid Chatbot merging aspects from both a menu-based chatbot and one using natural language processing. This decision was made based on the planned case studies for the project; as for the leading case study tourism, having the ability to let the client choose between creating a chatbot that uses buttons, natural language, or both together is considered beneficial as often a user will want to quickly book a hotel or a house to stay in for their trip and a GUI can allow this with a few clicks. Other users will prefer to investigate their options first or directly ask the chatbot to make a reservation for a specific hotel on a specified date. The NLP will allow the chatbot to identify the user's intent to book a hotel and start the appropriate dialogue. The proposed chatbot will allow users to avoid the frustrating part of needing to go back through various menus to get their desired dialogue for a menu-based chatbot by using text that is identified through the NLP component and sets the user on their desired path while also negating possible failures in the natural language processing if the user intent is not correctly detected, as the chatbot has the fallback of allowing the user to navigate through menus.

5.3.1 Planning chatbot dialogues

Before trying to implement a hybrid chatbot, it was necessary to plan out initial dialogues for every case study to determine a clear path for development, as the first step would be implementing a hard-coded implementation of these dialogues that would then be generalized.

The planned dialogues for the tourism case study can be seen in Fig.B.2.1. Four prominent use cases exist: Personal Assistant, Bookings, Recommendations, and Information. The personal assistant use case focuses on having the chatbot serve as an interim secretary for a user on a

trip. The initially planned functionalities are to allow the user to check the schedule for a specific appointment (check activity date) and check their schedule on a chosen day (check appointments on specific date), which at the end gives the option of either changing or canceling their appointment while also allowing them to message the entity responsible for one of their activities on that day. This use case has its four dialogues interconnected; as in the case of a human secretary, it makes sense for them to follow up by checking the schedule and making necessary changes.

The second use case is related to making any booking, ranging from flights and housing to *levadas* and sightseeing. However, this use case has to be generalized not to necessitate a completely new use case for every possible type of booking. The chosen approach was to check the similarities with most bookings. Every booking will need the type of service, date and time, the number of participants, meaning the number of people the booking is for, and the specific place the user wants to book. Once all this information is gathered, the chatbot will show the user a list of valid options and allow them to pick from these based on their preferences.

The third use case uses the recommendation API developed for the KBAI project. It will allow users to ask the chatbot for recommendations on specific topics for a specific date, and in the case the user does not have an existing profile, the chatbot will try to aid with building a profile through its interactions with the user, the concrete steps for building a user profile through the chatbot (beyond making use of the interactions) were not clearly defined as it was considered a future step if the project continues development.

The final use case for this case study is related to having the chatbot serve as a way for the user to search for information on any chosen topic, like, for example, asking the chatbot for vegan restaurants in the area, to which the chatbot would list various vegan restaurants, the user would then have the option of seeing additional information on a chosen restaurant and if desired, book a table there. The previously described booking use case would handle the booking process.

Project management is another case study planned for the KBAI project. This case study identified and planned three primary use cases for the chatbot, as shown in Fig.B.2.2. These use cases are as follows: adding a note to a project task, managing tasks through the chatbot, and the final use case is related to checking project-related information.

The simplest use case for project management is adding a note to a task, following the process of first checking if the chatbot knows which task it should add a note to, asking the user in case it does not, and then requesting the user to input the note for that ask along with user confirmation and finally adding the note to the chosen task.

A crucial aspect of project management is handling the various tasks and assigning the correct priorities and team members. For this purpose, the chatbot will provide the user with various dialogues related to task management. Among these, the following were planned out: task assignment, recommending team members for tasks, changing task status, changing task priority, creating new tasks, and deleting existing tasks.

To assign a task to a team member will first implicate knowing which task the user wants to assign, then giving the user the option of getting a recommendation on which team member to assign or directly assigning someone. Once the chatbot has the necessary information (which task to assign and to whom), the user will be asked to confirm the task assignment, which, upon being granted, will let the user have the option of assigning a different task or going back to the start. As previously mentioned, during task assignment, the user can ask the chatbot for recommendations

on whom to assign a task. This recommendation connects to the recommendation part of the task management use case, which will show a list of recommended team members for the specified task that the user can pick.

Aside from assigning tasks to team members, the chatbot can also allow the user to quickly edit existing tasks, changing their priorities (low, medium, high) or status (open, closed, backlogged), upon which the user can choose to assign a team member to the task if none are assigned.

Furthermore, creating and deleting tasks to keep the workflow updated is vital to good project management. The chatbot allows quickly creating a new task, only needing to ask the user for the task's name, estimated completion time, and priority. When the user fills in the task details, it will be created, and the user will have the option of immediately assigning someone to the task if they so choose. The reverse is also possible, where the user can quickly delete a task through the chatbot. Despite task management being pivotal to good project management, there is also a need to quickly ascertain information on the project, which is the focus of the check information use case. This use case allows users to check their list of tasks, view all currently unassigned tasks for the project or currently ongoing tasks, and manage or add notes to any of them. Moreover, the check information use case includes the option to view information on the assorted team members, such as their current roles and tasks on the project.

For news management, the chatbot intends to assist the user with navigating vast amounts of daily news to find their desired information while also using the recommendation API to assist the user. The planned dialogues with the initial use cases for this case study can be seen in Fig.B.2.3 and involve the following use cases: add notes, get recommendations, search for information, add user interests, and get trending news.

The first use case for news management is adding a note to an existing article, with the chatbot first needing to know which article to add a note to and asking the user for the note. The second use case for this case study connects to the recommendation API, also being developed as part of the KBAI project, allowing the user to ask for recommendations on either a specific topic or based on their existing profile. After showing the recommendations to the user, this use case connects to the previously mentioned add-note use case, giving the option to add a note to one of the recommended articles.

As this case study is focused on news management, one of the use cases is to allow the user to search for articles by either author name to get every article written by a specific author or by article name. As previously mentioned, the user can ask the chatbot for recommended articles based on their profile. One of the use cases is related to the chatbot assisting users with enhancing these recommendations by allowing them to select attractive topics. The final use case for this case study is regarding having the user ask the chatbot to check for current trending news, supporting the choice between a specific topic like innovating medical treatments or getting trending news that matches their interests.

During the planning phase of the project, a new case study was suggested, this case study relates to using the chatbot to manage personal information. This case study is the My Life case study and its planned dialogues can be seen in Fig.B.2.4, it has three central use cases: adding notes to existing information, managing information such as creating, editing, or deleting nodes or reminders and searching for specific nodes or reminders.

The first use case is similar to the previous add-note use cases from the news management and project management case studies, allowing users to add notes to existing information (nodes) on the app. About managing information, this use case is focused on allowing the user to add nodes to the app, for example, taking notes from a conversation the user had. As this use case is focused on managing information, the chatbot must also be able to assist the user with deleting or editing existing nodes to change their information or add notes by making use of the previously discussed use case. Besides, the user must also be able to create reminders through the chatbot by inputting a custom name and date. From extended use of the app, the information available to the user through their various additions and reminders can become challenging to navigate. To alleviate the problem of navigating ample information, the final use case is related to using the chatbot to assist with information searching, having the option to search for both reminders and overall information, continuously presenting the choice for the user to manage the information presented through the use of the previously mentioned use case.

5.4 Initial prototypes

After researching and analyzing various frameworks for building a chatbot, an initial hard-coded prototype was created for two of the most promising candidates, *Botonic* and *Botman*. The first prototype was created utilizing Botonic. This framework used React and relied on setting up routes, child routes, and the actions corresponding to each. Botonic Routes are how to map user inputs to actions, and child routes are the possible routes taken after the first one. An example of a route can be seen in B.3.1. The actions referenced in the various routes of B.3.1 point towards files in the project, each containing an action as shown in Fig. 5.

Botonic actions are React components that run JavaScript (JS) code, meaning they can fetch data from APIs, fetch the previous user input, validate user inputs, and utilize the results from their operations to return components that can reply to the user through a supported messaging channel. For example, in B.3.2, the Text component replies to the user while the Reply component presents them with menu prompts, as seen in Fig. 6.

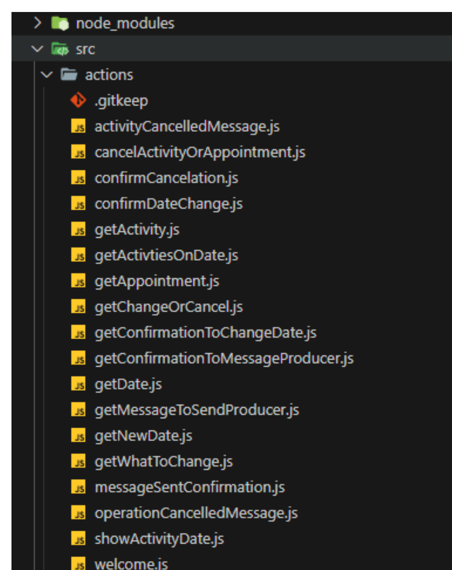


Fig. 5: Botonic actions

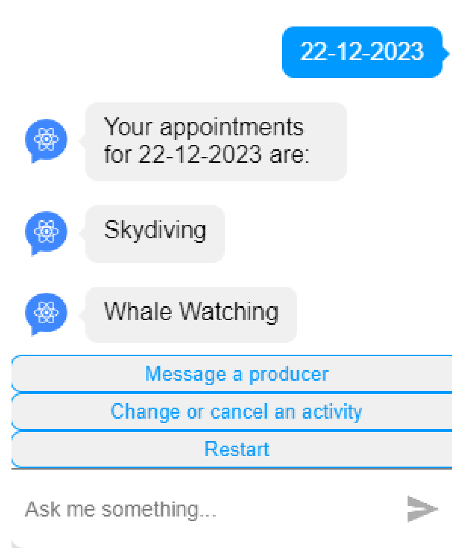


Fig. 6: Example of a Botonic interaction

The second prototype utilized the Botman framework. Botman was made in PHP and has a few requirements: Laravel, Composer, and a PHP installation of 7.1.3 or higher. Composer is used to manage PHP dependencies that Botman relies on, while Laravel allows for the usage of Botman Studio to facilitate the development of the chatbot. This studio gives access to logs for when an error or a crash occurs and allows for testing the code while automating some of the initial Botman setup.

This framework also relies on routes to map user inputs to chatbot actions, such as starting a conversation, accessing an API, and giving a simple reply. In B.4.2, there are various routes that each start a different conversation depending on the user input. These conversations are each defined in their class, as seen in Fig. 9. These conversations can consist of various actions and operations or simple chatbot text replies or button prompts to the user, as shown in Fig. 7 and Fig. 8.

The advantage of these conversation classes over Botonic action classes is that they allow a set of actions instead of a single action. The conversation in B.4.1 happens when the user asks the bot for the appointments on a specific date and consists of asking the user for the date they wish to check, presenting them with their schedule for that day, and suggesting additional actions like changing or canceling activities on the selected date.

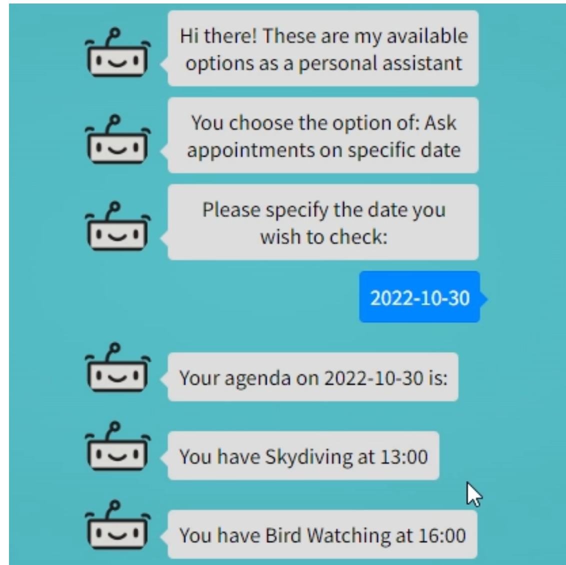


Fig. 7: Example of Botman Interaction with no buttons

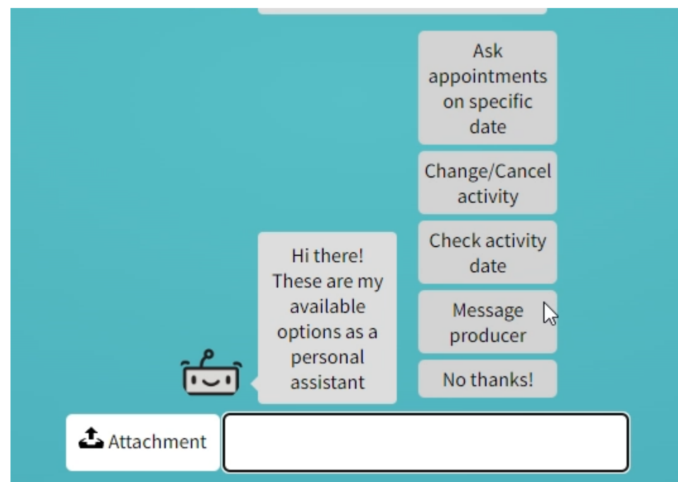


Fig. 8: Example of Botman Interaction with buttons

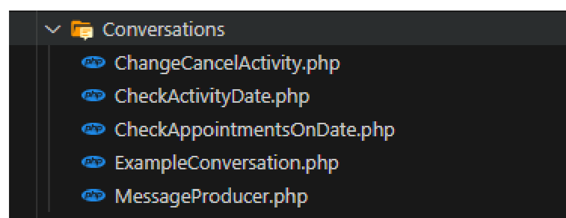


Fig. 9: Example of Botman Conversation classes

As seen in Fig. 5, Botonic requires the creation of various files, each corresponding to an action. Although this can be simplified by having one action file reference other action files, this structure makes the code for the chatbot splintered. In both prototypes, the same conversations and interactions were created; however, Botman's code is much more organized in the routes file and in each separate conversation class. As shown in Fig. 9, the number of files required to have the same conversations is much lower in Botman. For both prototypes, some research was also done into the NLP side of the chatbot. Botman allows for easy integration into any NLP of choice by creating a Middleware class that handles the communication with the NLP API.

Botonic has its own built-in NLP and allows for some NLP integration through pre-built plugins and custom plugins. Regarding documentation, both frameworks have their pages with documentation introducing various concepts and examples of how to utilize their advantages. One final detail discovered during the development of the prototypes was that Botonic does not allow for the saving and usage of previous conversation states. At the same time, Botman can save user inputs and variables into storage that can be retrieved from different conversations and conversation stages.

There was also the consideration of previous experience with PHP, as due to time restraints, learning to utilize JS and its React framework on top of the Botonic framework built with React knowledge in mind could cause significant delays and challenges. After careful consideration and the various reasons mentioned, Botman was the chosen framework for creating a chatbot.

5.5 Dynamization of the chatbot

Regarding the chatbot development cycle, the chosen approach was to create an initial hard-coded prototype with rudimentary features, meaning it had no NLP and did not use the knowledge base. This prototype was iterated upon, being tested and reviewed weekly. New features were implemented statically and tested, then an approach to dynamize them was discussed, planned, and implemented. After dynamically implementing the features, these were tested again, and improvements were discussed.

From the static chatbot conversation implemented previously, an analysis was made of which aspects of the dialog could be customized by the user. From these, a structure for the chatbot dialogues was created. This structure is a dialog composed of various interactions. Each interaction has a name and contains various parameters, as seen in B.5.1. The parameters and their purposes are as follows:

1. Requirements - The necessary values needed to start the interaction.
2. Title - What the chatbot will say to the user when the interaction starts (e.g. in Fig. 7, the title would be "Please specify the date you wish to check:").
3. Buttons - A list of lists containing the text for each button the chatbot will generate in the first position of each list, and the following strings are alternative answers that lead to that button (e.g. in Fig. 8 the parameter would be as shown in B.5.2).
4. Actions - The names of the primitive functions to execute this can include showing the results of an API call.
5. Output - The names for the variables the chatbot will save from the user inputs of this interaction (e.g. in Fig. 7, this parameter would be "data").

6. Output Format - The expected format for the output variable (e.g. in Fig. 7 this parameter would be "date").

7. Next Steps - The names for the subsequent possible interactions, for interactions with buttons like in Fig. 8 this parameter would contain the interaction that matches each button as shown in B.5.2.

After having the structure of each dialog and its interactions well defined, the code was altered since the conversations of Botman would all be handled by a single class as opposed to in Fig.9 where each possible conversation has its class. This new class is called *DynamicConversation*. This class, via the initial user input to the chatbot, gets the JSON file's name from which to fetch the dialogue. This development phase was still hard-coded; if the user input "template1" the chatbot would start the Personal Assistant dialogue. If the user inputs "template2" the chatbot will start the Booking dialogue. The chatbot would default to its fallback response for all other user inputs. Once the chatbot knows which dialogue to start, it reads the JSON file corresponding to the given name and saves its contents as an associative array of arrays. From this array, the chatbot starts the first interaction it finds, meaning the one at index 0, and follows the path set by the Next Steps parameter of each interaction until it reaches an interaction without any Next Steps, where it exits the conversation and goes back to awaiting a user input to start a new conversation.

With the dialogues' format and interactions being well defined, the next step was taking this information from a JSON file and using it to run dialogues dynamically. To run the dialogues, the method in B.5.3 runs through the various parameters of the interaction starting with the *Actions* parameter, executing any present actions, it starts with the action parameter as an action might contain information from an API call that is necessary to run the rest of the interaction. After running the necessary actions, if any are present, and in case the interaction does not need any API information from the action, the next step is checking if the interaction has any buttons and presenting them to the user if there are any. In case there are no buttons for the interaction the chatbot will then run a regular interaction through the method in B.5.4, replying to the user with the *Title* parameter of the interaction and awaiting the user input if there are any outputs, in this case the chatbot will also only proceed to the next interaction once the user input is in a valid format. For interactions with no buttons and no title parameter, as with interactions that exit the dialogue, there is a final check to stop the chatbot from replying with an empty text box to the user and instead exiting the dialogue.

Initially, the chatbot lacked NLP and could not recognize user text inputs when it gave the option of interactive buttons. This constraint meant the user either clicked a button or the chatbot exited the conversation, ultimately leading to a complete restart of the entire process up to that point. To address the limitation, the function responsible for handling the interactions with buttons was expanded upon; a snippet of this function can be seen in Fig.10, B.5.5. This functionality implied a slight change in the JSON template. The buttons parameter became an array of arrays, with the index 0 of each array being the button text displayed to the user. The following positions contain possible mentions that would be compared to the user input, and the array with the most similar mention would have its button chosen.

An additional improvement was adding the ability for the chatbot to accept and utilize multiple requirements and outputs, allowing various requirements to increase control over when each interaction can start. On the other hand, allowing various outputs is a quality of life change. This

way, the user can, for example, introduce both the start and end date of their stay at a hotel in a single interaction, separated by a ";" as opposed to needing to wait for the bot to request the start date insert it, then wait for the chatbot request for the end date and insert it. The previously mentioned feature of taking various requirements and outputs also necessitated a change in the JSON template, with both requirements and output now being arrays.

Following improvements made to the requirements and output fields of the chatbot, further quality-of-life improvements were brought up, this time related to allowing the user to skip individual interactions if they were unnecessary. This new method shown in Fig.11, B.5.6 will first check if the next interaction has buttons or actions as these interactions might not have outputs and give choices to the user. It then looks over the outputs of the next interaction and checks if these were already fulfilled. In contrast, the chatbot also gained the ability to go over the requirements parameter of the next interaction and fetch an interaction from the dialogue to fill in any missing requirements. This new method shown in Fig.12, B.5.7 goes through every interaction in the current dialogue to find an interaction whose output matches any of the missing requirements for the interaction the chatbot tried to run. Once the method finds an interaction matching this criteria, it will run that interaction and then try to run the interaction whose requirements were initially missing again. If it still has missing requirements, the chatbot will follow the same process until it can fill every requirement. This method assumes that the dialogue is configured correctly and that for every requirement of an interaction, there will be another interaction with that requirement as an output.

```php
$buttonsArray = [];
foreach ($buttonsInteraction['buttons'] as $id => $button)
{
    $newButton = Button::create($button[0])->value($id);
    $buttonsArray[] = $newButton;
}
$question = Question::create( text: '')
    ->addButtons($buttonsArray);
$this->ask($question, function($answer)
{
    $currentInteraction = $this->bot->userStorage()->get('currentInteraction');
    $currentDialog = $this->bot->userStorage()->get('currentDialog');
   if($answer->isInteractiveMessageReply())
   {
        $this->bot->userStorage()->save([
            'chosenButton' => $answer->getValue()
        ]);
       $this->StartNextInteraction($currentDialog,$currentInteraction, emptyTittle: false, hasButtons: true);
   }
   else
   {
       $textAnswer = $answer->getText();
       $mostSimilarOption = $this->GetMatchingButton($currentInteraction, $textAnswer);
       if($mostSimilarOption > count($currentInteraction['buttons']) - 1) //if the option (index) is higher than the number of buttons
       {
           $this->say( message: 'This does not look like a valid option. Please click on a button or type the desired button input');
           $this->repeat();
       }
       else
       {
            $this->bot->userStorage()->save(['chosenButton' => $mostSimilarOption]);
            $this->StartNextInteraction($currentDialog,$currentInteraction, emptyTittle: false, hasButtons: true);
       }
   }
});
```

Fig. 10: Code snippet of the method to run an interaction with buttons

```
{
    //check if the output for nextInteraction has already been filled
    if(count($nextInteraction['buttons']) == 0 && count($nextInteraction['actions']) == 0) // if it has no buttons and no actions
    {
        $nextOutputs = $nextInteraction['output'];
        if(count($nextOutputs) > 0) //if it has outputs
        {
            $numOfFilledOutputs = 0;
            foreach ($nextOutputs as $nextOutput)
            {
                $savedOutput = $this->bot->userStorage()->get($nextOutput);
                if(empty($savedOutput) == false) //if we already have this output
                {
                    $numOfFilledOutputs++;
                }
            }
            if($numOfFilledOutputs == count($nextOutputs)) //if every output has already been saved
            {
                return true;
            }
        }
    }
    return false;
}
```

Fig. 11: Code snippet of the method to check if next interaction should be skipped

```
{
    //iterate through currentDialog to find interaction with matching output for requirement
    $result = [];
    $nextDialog = $currentDialog;
    $nextFunctionRequirements = $currentDialog[$nextInteractionName]['requirements'];
    foreach ($currentDialog as $interaction)
    {
        $interactionOutput = $interaction['output'];
        $currentInteractionOutput = $currentInteraction['output'];
        if(count($interactionOutput) > 0)
        {
            for ($j=0; $j<count($interactionOutput); $j++) // iterate through possible outputs for interaction
            {
                //if the current interaction output is one of the requirements AND is not the same output as the interaction we just had
                if(in_array($interactionOutput[$j], $nextFunctionRequirements, strict: false) && !in_array($interactionOutput[$j], $currentInteractionOutput, strict: false))
                {
                    //$this->say('DEBUGGING LOG CURRENT INTERACTION OUTPUT:' . $interactionOutput[$j]);
                    //if the bot does not have the output already saved in cache (prevents a loop of asking the same question)
                    if(empty($this->bot->userStorage()->get($interactionOutput[$j])))
                    {
                        //change that function nextSteps to be current next step
                        $newNextInteraction = $interaction;
                        $newNextInteraction['nextSteps'] = [];
                        $newNextInteraction['nextSteps'][] = $nextInteractionName;
                        //$this->say('DEBUGGING LOG NEXT INTERACTION NAME: ' .$nextFunctionConfigName);
                        array_push( &array: $result, $nextDialog, $newNextInteraction);
                        return $result;
                    }
                }
            }
        }
    }
    return $result;
}
```

Fig. 12: Code snippet of the method responsible for getting an interaction to fill missing requirements

A new class was created containing various methods to aid in the validation of user inputs along the dialogue; one of these methods can be seen in Fig.13 B.5.8. This method enables the chatbot to accept a new parameter called *outputFormat* in its interactions. This parameter is responsible for checking whether the user input (output in the interaction) is in the expected format, supporting the following types: integer (int), string (string), and date in the YYYY-MM-DD format. This validation can smoothly be extended with new cases if the project has a continuation. When the user input does not match the expected format, the chatbot warns the user that their input is in the incorrect form and repeats the question, following this cycle until the user input is in the expected format, upon which the chatbot will move on to the next interaction.

```
{
    $trimmedInput = trim($input);
    switch ($expectedFormat)
    {
        case 'date':
            return $this->isDate($trimmedInput);
            break;
        case 'int':
            return $this->isInteger($trimmedInput);
            break;
        case 'string':
            return true;
            break;
        case 'array':
            return $this->isArray($trimmedInput);
            break;
        case 'range':
            return $this->inRange($trimmedInput, $range);
            break;
        default:
            return false;
            break;
    }
}
```

Fig. 13: Code snippet of the method responsible for validating user input

To test the dialogue resulting from the reading of the JSON file, the result of this reading was dumped to the console and compared to the expected dialogue; this was done with two different dialogues from the conversation flow of the tourism case study presented in Fig.B.2.1. Once bugs were corrected and the chatbot correctly converted the JSON file into a dialogue for both test dialogues, the chatbot was tested using Botman Studio. These tests involved talking to the chatbot like a user and checking if it returned the expected answers. This testing method explored every option for the two dialogues, making sure they connected correctly during testing.

The identical process of talking to the chatbot and dumping variables to the console to compare them with the expected result was used to test the various features implemented to make sure the chatbot skipped interactions for which it already had the output. Interactions with repeated outputs that came one after the other, sharing different titles, were created for this testing. During the chatbot dialogue, it was confirmed that it was correctly skipping these interactions after some bugs were squashed.

To ensure the chatbot found interactions to fill the missing parameters for the interaction it wanted to run some interactions were skipped. The nextSteps parameter was changed to two

interactions ahead to ensure this search. This change in the dialogue allowed confirmation that the chatbot searched the entire dialogue to find an interaction that could fulfill the requirements to run the desired interaction.

For testing that the chatbot correctly validated user inputs and repeated the interaction if the input was invalid, the wrong inputs were purposely introduced during interaction with the chatbot, and this was done for every type of outputFormat supported.

5.6 Chatbot and Natural Language Processing

Building on research into NLP libraries, an NLP API was built utilizing *NLP.js*. The goal was to connect the chatbot to this API to tell the chatbot which dialogue to start and what is the first interaction of that dialogue, allowing the chatbot to start different dialogues depending on user intents dynamically. These intents are set in the NLP corpus, with the corpus having a defined reply for each intent as shown in B.6.1. The NLP API feeds the utterances to a Neural Network capable of understanding variations of these as having the same intent (e.g. the user inputs: "I want to book something" the NLP knows the intent is booking).

Botman offers the option of using middleware to establish a connection from the chatbot to the NLP API. From the given examples that come with the framework, a middleware for Google's Dialogflow and one for Meta's Wit.ai, a custom middleware was built for NLP.js. The main goal of this middleware is to send every user input to the API and use the generated answers; as such, most of the functions from the middleware examples were simplified as they were not needed for the current goal. This custom middleware has two main functions. Firstly, *getResponse* shown in B.6.2 is responsible for sending the user input to the API through a post request. Then, *received* shown in B.6.3 to process and save the output of the API, this method goes over every parameter in the response and either saves them or creates an empty equivalent if they are null. A property is set in the file containing the various routes before defining the first route, as shown in B.6.4 to ensure the chatbot sends every input to the NLP API. Following this addition, a route was defined for any user input. This route always uses the output from the NLP API, and if the output is valid, it starts the conversation. Nevertheless, if the chatbot does not get a valid response, it replies with the fallback message defined in the JSON configuration file.

To test if the chatbot was correctly sending post requests to the NLP API, a debugging file with the output from the processing was added on the side of the API. This debugging file was then compared to the chatbot's result from the post request, which was dumped into the console. Once confirmed that the requests were being correctly sent and processed, it was necessary to test whether the chatbot correctly accessed and saved the information it received from the API. Through Botman Studio, the integration with the API was tested in the following way: during the first user input, all the information received from the API was used as a reply to the user, and these replies were compared with the file created by the NLP API itself and what was dumped to the console, with necessary changes being made until all the values matched up consistently over various tests.

5.7 Connecting the chatbot to the Knowledge Base

Upon completion of adding NLP to the chatbot, the next step was to have the chatbot retrieve the dialogue from the knowledge base. As an additional feature, the output the chatbot used from the

NLP was also simplified to use the dialogue name or node ID, not requiring the name of the first interaction as the first interaction would now be the one containing *firstinteraction* in its name. In the case of the booking dialogue, the first interaction is named *firstInteractionBooking*; this change was necessary as knowledge base nodes are unique.

An ontology was developed to store the chatbot dialogues in the KB. This ontology can be applied equally to all case studies for this dissertation. However, due to a confidentiality agreement with the KBAI company, the ontology and its accompanying description cannot be provided.

It was necessary to map the fetched dialogue to the same format as the chatbot's dialogues, similar to B.5.1, as the knowledge base dialogues came in a completely different format seen in B.7.1. Mapping the templates to chatbot dialogues required a new class containing methods to map the dialog, its various interactions, their nodes, and arches. The chatbot instantiates this class at the beginning of a conversation, and its mapping methods are called before starting the dialogue. These are B.7.2 responsible for mapping the template to the dialogue format the chatbot utilizes, shown in B.5.1 and B.7.3 responsible for getting the first interaction to start the dialogue.

At this stage, the chatbot got its dialogues from the knowledge base and knew which dialogue to fetch using NLP. A prototype of the chatbot API was implemented into a website, hosted using *WampServer64*, as until now, it had been developed in Botman studio. This API adds an icon to the website as seen in Fig. 14; this icon can be customized and opens a chat window for the user to input information for the chatbot to handle directly. The API has some available methods to help better integrate into the website shown in B.7.4. An example of the file used to integrate the API on a website for testing can be seen in B.7.5 and its appearance in Fig.B.7.6. This integration makes use of some API methods. If the user clicks the *Booking* button, the chatbot will start the booking dialogue as if the user had input "Booking" in the chat window the same applies to the *Personal Assistant* button with its respective dialog, opening the chat window will greet the user with a message as depicted in Fig.B.7.6 and clicking *Close Widget* closes the chat window.

Fig. 14: Chatbot API icon

The returned templates were analyzed to ensure they had all the information in the proper format to test the KB integration. The resulting object from the methods responsible for mapping the templates was written to a JSON file to test that the chatbot could map the templates to the dialogue format it used. As the dialogues in the templates were previously used to test the chatbot during the dynamization step of development, these were used as a comparison. Once the methods responsible for mapping the templates were outputting a JSON file with a dialogue identical to the previously used ones, these dialogues mapped from the knowledge base template were then used

to run the chatbot through Botman Studio; this process was done for both available templates to make sure the generic implementation to map the templates was working correctly.

Once it was confirmed that the chatbot correctly mapped the knowledge base templates to dialogues and was able to run them, the next step was using the NLP API to identify which template to fetch based on the user intent. As only two templates were available for testing, the two possible intents are *personal assistant* and *booking*. The NLP corpus had various utterances added for each intent along with the answer for each intent, and this answer corresponded to the node ID of the template. Upon receiving the output from the NLP API, the chatbot would try to append the node ID to the link for the template API, as the ID is the missing component to get a dialogue. The chatbot would start the respective dialogue after validating both the answer parameter of the NLP API as an integer and the final URL for the template. When starting, the chatbot maps the template into a dialogue and then runs the dialogue starting with the interaction whose name contains *firstInteraction*. This process was tested through Botman Studio to confirm that the correct dialogue was fetched for the user intent, was correctly mapped, and that the chatbot ran the dialogue accurately, starting with the proper interaction.

5.8 Chatbot and APIs

As proof of concept that the chatbot allows for the simple implementation of various APIs, the APIs responsible for booking were implemented into the chatbot. This implementation had two stages: a hard-coded implementation, with each API operation having its function to map the output into a format the chatbot understood, and the transition into a dynamic implementation, having a single function that maps any API operation output into a format the chatbot understands.

The method in Fig.15, B.8.1 was created to simplify adding various APIs into the chatbot in a straightforward mode. This method takes the URL for the API and the parameters for the post-operation; it then makes the post-operation using a different method to obfuscate the token individual to each client. The result of the API operation is returned as a JSON object that is then translated into an associative array object. However, if, for some reason, the API returns an empty object, the method returns an associative array with the *errorOcurred* set to *true*.

Since constantly making posts and calls to the various APIs can add latency and further cost, the option to allow caching the current output from the API was added. As previously shown, the chatbot, at the start of an interaction, executes the primitive functions outlined in the *Actions* parameter. At the end of each action, it handles the output from the action as shown in B.8.3. This method is responsible for caching the output from the API if the primitive function returns $saveOutput = true$ in its parameter list. It also sets the *range* value in case the interaction has it as an *OutputFormat*, this is mainly used to validate user input when presenting a list of options to the user as shown in Fig.B.8.4, it only accepts a value higher or equal to minimum range and lesser than maximum range.

Finally, the method in B.8.3 is also responsible for printing out the various messages for the user. In the case of Fig.B.8.4, these are the various locations returned from the API call. It also checks if any error occurred through the *errorOcurred* parameter, and for any messages containing *ERROR* as in the case of errors, the chatbot returns the user to the previous interaction.

Creating Actions that implemented these APIs was necessary to test their generic implementation. These Actions were then integrated into chatbot interactions to form a dialogue to test the

```
{
    $postHeaders = ['Content-Type: application/json; charset=utf-8'];
    $responseToMap = $this->PostToAPI($apiURL,$postParameters, $postHeaders);

    if (isset($responseToMap) == false || isset($responseToMap['d']) == false)
    {
        return [
            "errorOcurred" => true
        ];
    }
    return $responseToMap['d'];
}
```

Fig. 15: Code snippet of the method to map API responses

use case of creating a booking. The chatbot replies were compared with the results generated by the API directly to ensure it was correctly showing the results for each API call. As if the same information is fed, each API can run directly on its website; these are compared to the chatbot replies to ensure the correct integration of the APIs.

5.9 Conclusion

The results of the development process can be seen in the diagram of Fig.16; it should also be noted that throughout the various development phases, no user tests were done, as the chatbot was only tested during development and at the weekly meetings, where needed enhancements were pointed out and then implemented from these short tests.

The dataflow diagram in Fig.16 outlines the various back-end chatbot processes during user interaction. The user will communicate with the chatbot through the application interface, allowing the chatbot to read the user inputs and send them to be processed by the NLP API. The output from the NLP is then sent back to the chatbot, processed, and saved. The NLP output is how the chatbot knows the user intent, sentiment and which dialogue to request from the KB. If no user intent is found, the chatbot will reply with a fallback message, and the user will be prompted to give their input again.

Once the chatbot knows which dialogue to start, it sends a post request to the KB to obtain it in the KB template format, which the chatbot then transforms into its dialogue format and looks for the interaction whose name contains *firstInteraction* as this is the first interaction of the dialogue. The chatbot first tries to run the actions (API related operations) if any are present and then replies to the user and awaits their input. The user input is then sent to be validated by various chatbot methods. In the case of being invalid, the chatbot replies to the user explaining that their information had an invalid format and awaits a valid user input. When the user input is correct, the chatbot saves it, ends the interaction, and searches for the next interaction. This process is repeated until the chatbot runs an interaction with no specified next interaction at which point, it exits the dialogue and goes back to awaiting a user input to repeat the outlined process with the respective dialogue corresponding to the new user intent.

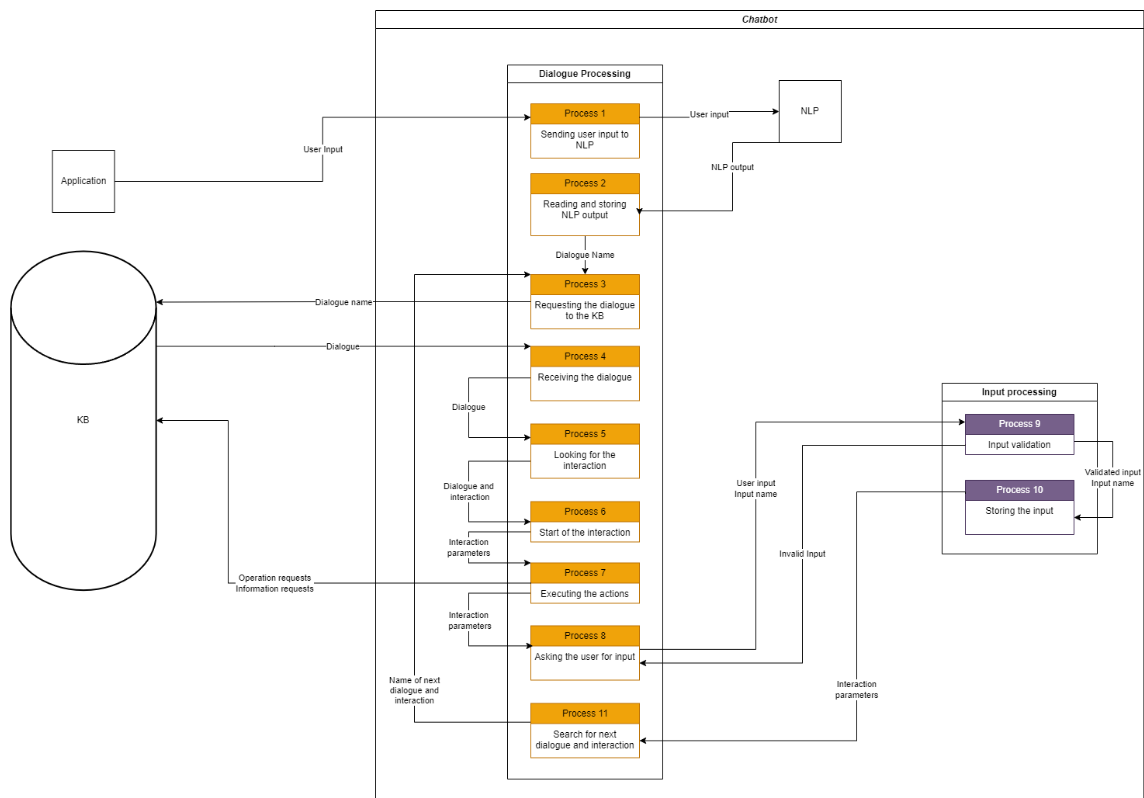


Fig. 16: Chatbot dataflow diagram

6 Conclusions

At the end of the development, the final product is a generic chatbot with NLP that allows for straightforward creation and configuration of various interactions with the user, having the ability to join multiple interactions into a coherent dialogue with an end purpose in mind. The chatbot is connected to an editor also produced in the KBAI project that allows for a more visual approach to editing and creating the dialogues instead of editing a *JSON* file. The process of connecting the chatbot to the editor had the goal of allowing the chatbot to fetch dialogues from the editor dynamically. For dialogues to have more flexibility, the chatbot supports the implementation of any API in a standardized manner, allowing the interactions to call the required API operations through the name of a primitive function that implements it.

The development of this project was a learning opportunity to get more familiar with PHP, existing chatbot types, various frameworks utilized to develop chatbots, and the diverse available NLP and NLU frameworks currently used. The project demanded research into different chatbot types and existing frameworks to choose the most appropriate one. The investigation revealed that although there are various quality frameworks, most are either cloud-based or have their own NLP/NLU, allowing for little control over that component. Seeing as the chatbot had to be dynamic, allow for easy integration of various APIs and easy integration of the chatbot itself into any website, this required planning how to structure the files that will be read by the chatbot to generate the dialogue, always making sure the code was dynamic either reading from files or fetching directly from an API. Due to a lack of experience utilizing APIs, the project was an opportunity to gain some knowledge into that part of software development as the chatbot implements a booking API to serve as proof of concept and is also linked to an NLP API to know which dialogue it should start based on user inputs and the knowledge base API to fetch the dialogue corresponding to the user intents.

6.1 Challenges

During development, various challenges were encountered; the first challenge was defining the objectives of the chatbot and its requisites. For this step, research was done to compile documents identifying the various chatbot types with their pros, cons, necessary inputs, outputs, examples of functionality, limitations, observations, and areas where they excel. This document helped ascertain the possibilities for developing a chatbot and knowing which would be better for the planned case studies.

The main challenge was making the entire framework dynamic, allowing users without coding knowledge to create and edit dialogues using the online KBAI editor. The chatbot was developed iteratively, starting with a hard-coded implementation that transitioned into a dynamic one. Once the chatbot got its dialogues from a local *JSON* file, the following challenges were incorporating NLP and getting the dialogues through the KBAI API. To integrate NLP, an NLP API was created to which the chatbot sent every user input and then utilized the API output to determine which dialogue it should fetch.

There were some struggles with incorporating the KBAI knowledge base into the chatbot, as the KBAI NLP corpus was only built to work with entities. In contrast, the chatbot had utilized the *answer* parameter of the NLP output up to that point. The alternative of adding additional data to the corpus was tested, but in the end, the chatbot needed some changes in how it handled

the output from the NLP. However, since the editor was not finished, these changes could not be thoroughly tested beyond the local NLP API.

Finally, the last challenge came with implementing a generic method that allows any API to be easily implemented. This challenge resulted from some complications with the API token so the API implementations on the chatbot side could be tested. The implemented method takes the API link to its parameters and returns the API output into a format the chatbot can understand and save. Since constantly making API requests can be expensive and slow down the interactions, as the user will have to wait for the API to process and return the request to the chatbot to obtain an answer, the chatbot supports saving the API output for future actions.

With the challenges mentioned earlier, this project provided ample opportunity to explore previously unfamiliar areas of software development and get some practical experience in other areas. As the project was developed using sprints and iterative prototyping, it was necessary to learn how to organize the various phases, split each stage into numerous tasks, and estimate the time each task would take. This hands-on experience was a valuable opportunity to become more familiar with using sprints and iterative prototyping for development cycles. In addition to learning how to use sprints, using an API and creating a project that can work as an API to integrate into other projects was a new experience beyond just working with various APIs; the developed product required a generic implementation of its API integration methods to allow for easy future additions.

6.2 Next steps

Although the developed generic framework has all the necessary bases to be expanded upon, only some things to make it commercially viable were implemented due to time constraints. The framework implemented an API as proof of concept and two distinct dialogues selected based on the NLP's recognized intent of the user input.

For this project to be commercially viable, it must be exhaustively tested with the KBAI editor once the editor is completed. The project would also need a way to have its actions defined in the editor in the same way, the dialogues are created, which would be challenging as the actions are small PHP methods, although these methods mostly call the generic API function to access the chosen API, giving it the parameters for their desired result and then choosing which messages to show the user. Additionally, there is the need to conduct user tests as that stage of the project was not realized due to unforeseen delays; user testing is necessary as it will likely point out various problem areas and areas for improvement in the project.

Once commercially viable, the ensuing steps for this project would involve implementing different types of buttons into the framework, such as the interactive telegram buttons as seen in Fig.17. Having interactive buttons without any external dependencies like Telegram, Facebook, Microsoft Bot Framework, or Slack would allow further extensibility of the framework as this would also allow for higher quality of life improvements in the user experience. Once the extra button functionality is implemented into the framework, the dialogue and interaction structure can have new fields to define the text and action of each button interaction. A proposed implementation of these new parameters:

1. ButtonInteractions - This parameter would be an array of arrays, with each array having the text for the various interactive buttons that each button from the Buttons parameter would

possess. In the case of Fig.17, this parameter would be as follows: ButtonInteractions: [["Mark completed", "Delete"]].

2. ButtonInteractionsActions - This parameter would be an array of arrays, containing the same number of arrays and elements of each array as ButtonInteractions; each element of these arrays would correspond to the action for when the user clicks the text on Button Interaction. Using Fig.17 as an example this parameter would be: ButtonInteractionsActions : [["MarkTaskAsCompleted","DeleteTask"]]

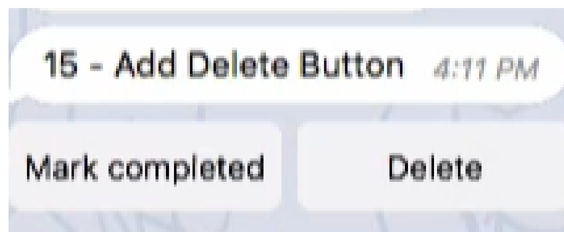


Fig. 17: Example of Interactive Buttons

References

[1] X. Sánchez Díaz, G. Ayala-Bastidas, P. Fonseca, and L. Garrido, "A knowledge-based methodology for building a conversational chatbot as an intelligent tutor," 09 2018.

[2] M. Mnasri, "Recent advances in conversational nlp: Towards the standardization of chatbot building," *arXiv preprint arXiv:1903.09025*, 2019.

[3] KBAI, "KBAI Research," 2023, [Online; accessed 29-November-2023].

[4] P. Suta, X. Lan, B. Wu, P. Mongkolnam, and J. Chan, "An overview of machine learning in chatbots," *Int J Mech Engineer Robotics Res*, vol. 9, no. 4, pp. 502–510, 2020.

[5] S. Reshmi and K. Balakrishnan, "Implementation of an inquisitive chatbot for database supported knowledge bases," *sādhanā*, vol. 41, no. 10, pp. 1173–1178, 2016.

[6] N. A. Ahmad, M. H. Che, A. Zainal, M. F. Abd Rauf, and Z. Adnan, "Review of chatbots design techniques," *International Journal of Computer Applications*, vol. 181, no. 8, pp. 7–10, 2018.

[7] Wikipedia, "Computing Machinery and Intelligence — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Computing%20Machinery%20and%20Intelligence&oldid=1171942679, 2023, [Online; accessed 29-August-2023].

[8] B. Luo, R. Y. Lau, C. Li, and Y.-W. Si, "A critical review of state-of-the-art chatbot designs and applications," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 12, no. 1, p. e1434, 2022.

[9] K. Bala, M. Kumar, S. Hulawale, and S. Pandita, "Chat-bot for college management system using ai," *International Research Journal of Engineering and Technology*, vol. 4, no. 11, pp. 2030–2033, 2017.

[10] S. A. Abdul-Kader and J. C. Woods, "Survey on chatbot design techniques in speech conversation systems," *International Journal of Advanced Computer Science and Applications*, vol. 6, no. 7, 2015.

[11] C. S. Kulkarni, A. U. Bhavsar, S. R. Pingale, and S. S. Kumbhar, "Bank chat bot–an intelligent assistant system using nlp and machine learning," *International Research Journal of Engineering and Technology*, vol. 4, no. 5, pp. 2374–2377, 2017.

[12] A. Abdellatif, K. Badran, D. E. Costa, and E. Shihab, "A comparison of natural language understanding platforms for chatbots in software engineering," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 3087–3102, 2021.

[13] M. Darwich, S. A. Mohd Noah, N. Omar, and N. Osman, "Corpus-based techniques for sentiment lexicon generation: A review," *Journal of Digital Information Management*, vol. 17, p. 296, 10 2019.

[14] A. Rapp, L. Curti, and A. Boldi, "The human side of human-chatbot interaction: A systematic literature review of ten years of research on text-based chatbots," *International Journal of Human-Computer Studies*, vol. 151, p. 102630, 2021.

[15] W. Huang, K. F. Hew, and L. K. Fryer, "Chatbots for language learning—are they really useful? a systematic review of chatbot-supported language learning," *Journal of Computer Assisted Learning*, vol. 38, no. 1, pp. 237–257, 2022.

[16] A. P. Chaves and M. A. Gerosa, "How should my chatbot interact? a survey on social characteristics in human–chatbot interaction design," *International Journal of Human–Computer Interaction*, vol. 37, no. 8, pp. 729–758, 2021.

[17] M.-C. Jenkins, R. Churchill, S. Cox, and D. Smith, "Analysis of user interaction with service oriented chatbot systems," in *International conference on human-computer interaction*. Springer, 2007, pp. 76–83.

[18] O. Agarwal, H. Ge, S. Shakeri, and R. Al-Rfou, "Knowledge graph based synthetic corpus generation for knowledge-enhanced language model pre-training," *arXiv preprint arXiv:2010.12688*, 2020.

[19] R. Clancy, I. F. Ilyas, and J. Lin, "Scalable knowledge graph construction from text collections," in *Proceedings of the Second Workshop on Fact Extraction and VERification (FEVER)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 39–46. [Online]. Available: https://aclanthology.org/D19-6607

[20] S. Hussain and G. Athula, "Extending a conventional chatbot knowledge base to external knowledge source and introducing user based sessions for diabetes education," in *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE, 2018, pp. 698–703.

[21] E. Adamopoulou and L. Moussiades, "Chatbots: History, technology, and applications," *Machine Learning with Applications*, vol. 2, p. 100006, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666827020300062

[22] A. Gupta, D. Hathwar, and A. Vijayakumar, "Introduction to ai chatbots," *International Journal of Engineering Research and Technology*, vol. 9, no. 7, pp. 255–258, 2020.

[23] J. Rhim, M. Kwak, Y. Gong, and G. Gweon, "Application of humanization to survey chatbots: Change in chatbot perception, interaction experience, and survey data quality," *Computers in Human Behavior*, vol. 126, p. 107034, 2022.

[24] B. AbuShawar and E. Atwell, "Automatic extraction of chatbot training data from natural dialogue corpora," in *RE-WOCHAT: Workshop on Collecting and Generating Resources for Chatbots and Conversational Agents-Development and Evaluation*, 2016, pp. 29–38.

[25] A. Rahman, A. Al Mamun, and A. Islam, "Programming challenges of chatbot: Current and future prospective," in *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*. IEEE, 2017, pp. 75–78.

A Tools Used

A.1 Chatbot Frameworks

Framework	Pros	Cons
Microsoft Bot Framework	SDKs for .NET and Node.js Speech to Text ML Offers cognitive services Integration with other microsoft services Has ready to use templates to help understand it better and give examples	Free plan has a message limit per month (10k messages) Full feature support is limited to .NET and Node.js Does not handle NLP/NLU alone, requires LUIS for it Needs various other microsoft services
Rasa	Allows for the storage of user data, can be beneficial for other parts of KBAI Allows for the creation of bots that learn as they communicate with the user (they can be initially trained through built user stories) Has a ML core that decides how to respond to the user query The data used doesnt need to be run through a third party API Offers Rasa X as a set of tools to improve the chatbot	Uses spaCy to process user inputs and this can consume a lot of memory Does not offer complete control over dialogue processing Is meant for developers with NLP and chatbot experience
Wit.ai	SDK available for Python, Node.js , Ruby, iOS Meant to be easily integrated into websites and apps Better NLP engine than IBM and Microsoft bot Allows for voice based chatbots	Seems focused on conversational chatbots and not chatbots meant to navigate a KB/DB Uses an API, however the bot is trained and created in their cloud and this can limit control Limited slots and parameters for training the bot
BotPress	Highly customizable (Modular architecture) Comes with pre-installed NLU Engine, chat emulator/debugger and visual flow editor Meant to be easy to deploy to chosen platform Gives chatbot analytics	Has some limitations as to the chatbots it can implement, its better for rule and action based chatbots Can be difficult to utilize at first Is Cloud Based
IBM Watson	Good ML engine (has been vastly trained) Offers analytics Has good security and privacy options for user data Works on websites	Can be difficult to use Seems more focused on offering good security and privacy measures Limited free plan Is Cloud Based
BotKit	Offers tools for analytics Meant to be easy to develop with Offers a visual conversation builder Offers a system to handle scripted dialogues and transactional questions Allows for the addition of plugins if necessary Allows for the choice of what NLP engine to utilize	Has no built in NLU/NLP Limited to Node.js Is being integrated into Microsoft bot framework

Framework	Pros	Cons
OpenDialog	Has a visual editor to design and prototype conversations (code-free could be useful for small tests) Can be deployed via Docker (locally) Can work as a server unit Supports integration with NLUs	The development is all based on an editor, meaning no coding is done (less control); however there is full acess to the github repository
Botonic	Can be deployed locally Offers various templates Has various plugins available for easy integration of services (Google analytics, various NLUs) Supports various platforms (browser, mobile, etc) Allows for the definition of the conversation flow via React code and the addition of menu buttons with options	Appears limited to simpler chatbots (Menu, Rule, Keyword based) Made in React
Claudia Bot Builder	Supports various chatbots Can be deployed to AWS (Amazon Web Services) Greatly simplifies the coding of a bot Has commands to assist with webhooks for various services Automatically converts responses to the right template Supports asynchronous replies	Hard to run locally as it uses an API gateway

Framework	Pros	Cons
Tock	Independent from external APIs Supports Domain Specific Languages (DSL) in Kotlin, Node.js, Python and REST Can connect to various text and voice channels from apps Allows for the building of stories and analytics through a UI Compatible with various NLPs (Stanford, OpenNLP, Rasa and others) Offers web/mobile integration with React and Flutter Can be deployed locally (via Docker) or in the cloud	No support for third party APIs
Botman	Allows for the connection into various messaging services Works with various frameworks Can be added to existing codebases and can be deployed locally Focused on business logic Offers good documentation Has botman Studio, uses Laravel a PHP framework to provide various tools to facilitate development Can save conversation states Can use NLP platforms like Wit.ai and NLP.js	
Bottender	Has both functional and declarative approaches to build a conversational UI Allows for the designation of actions based on states and events Allows for the use of progressive enhancement or graceful degradation Optimized for real-world use cases, has automatic request batching Easy configuration to work with various channels Has a Discord community to help with doubts on utilization	Seems focused strictly on conversational Uis Built on top of various messaging APIs
DeepPavlov	Allows for the creation of dialogue systems Offers various tools to create flexible conversational assistants Offers pre-trained deep learning models for various NLP tasks Allows for integration via APIs Allows for the training of their pre-trained models on new data Models can be made available via socket servers/Docker	Requires CUDA capable devices

B Development

B.1 Planning

B.1.1 Chatbot Types Pros, Cons and Limitations

Chatbot Type	Example	Advantages	Disadvantages	Limitations	Areas where it excels
Menu Based Chatbot	The user logs into a company support website and the chatbot asks: What department they wish to contact and presents them with three buttons: IT, Marketing, Customer Service	Easy to use and intuitive Good enough for standard Q&A The user will never be in a position where the chatbot doesn't understand (It gives us a higher control of the conversation) Easy to implement	It can be slow and tedious to go through the decision tree Limited in how it allows the user to interact (They cant just ask what they want to know)	It can be frustrating if the user has to go back through all of the options just to get the answer to one of his questions	Customer Support (Helping users navigate a company website)
Rule Based Chatbot	Turism Chatbot: The user says the arrival and departure airports The chatbot then shows the filtered information And asks the user to confirm and then asks if they would like to book a hotel etc	High control over the converstion flow as the chatbot answers and flow are predefined The chatbot doesn't need any ML training making it faster to implement Relatively simple implementation	If the user has a typo in their question the chatbot might not understand their question They do not learn by themselves, improvements need to be manually made The interaction with this chatbot doesn't feel natural	These chatbots are only useful If the conversation direction can be predicted	When the conversation flow can be predicted
Keyword Recognition Chatbots	The Keywords and their match: If the user has menu and restaurant X in a sentence then the chatbot will tell the user the menu for that restaurant	Allows the user to ask in natural text Users can ask more advanced questions as the chatbot can understand some of the relations between the words (keywords)	If the user asks a question that contains no known keywords then the chatbot is unable to answer/understand	These chatbots can give redundant answers if the users ask similar questions due to redundant keywords	Areas where the domain is well defined, this way the user questions will always contain some keyword the bot can recognize

Chatbot Type	Example	Advantages	Disadvantages	Limitations	Areas where it excels
Hybrid between Keyword Recognition and Menu Based	The user logs into a company support website and the chatbot asks: What department they wish to contact and presents them with three buttons: IT, Marketing, Customer Service. The chatbot also presents the user with a text input box where the user can type what they wish and if there are recognizable keywords the bot will direct them correctly	The user can eitheir navigate through the use of menu buttons or ask the question straight away If the chatbot doesnt understand the user question then the user can simply utilize the menu buttons instead	Requires the implementation of both Keyword recognition and Menu Based Chatbot	If the menu buttons arent enough to satisfy the user requirements and the user asks questions with keywords the chatbot cant understand (different domain for example) then the chatbot cant help the user	Customer Support Helping users navigate websites Helping users navigate large databases/information stores
Seq2Seq Chatbots (RNN)		Since the chatbot generates their own answers they can "adapt" in a way to the user questions The chatbot can learn on its own	Requires ML training, meaning it requires extra time and a good amount of data	The efficiency/accuracy of the chatbot answers can vary greatly depending on how big the dataset used to train the model is and also the quality of the data	Extracting information from a corpus Answering simple questions Taking on a personality from its dataset for example a TV character
Contextual Chatbots/Reinforcement Learning Chatbots (ML chatbots)	If the user previously searched for ML papers then the chabot will initially prompt the user with "Would you like to see papers related to ML?"	The chatbot can give suggestions to the user based on their interaction history/ user profile The chatbot will remember the user and if its a new user then they will add their information to the KB to help build a user profile The chatbot has the ability to learn from its interactions	Requires ML training, meaning it requires extra time and a good amount of data	The efficiency/accuracy of the chatbot answers can vary greatly depending on how big the dataset used to train the model is and also the quality of the data	Helping the user navigate large information Predicting and understanding user intentions
Voice Based Chatbots	The Keywords and their match: If the user says set an alarm for 4pm The chatbot will see the keywords alarm and 4pm and know to set an alarm for that time	High acessibility to people with visual impairement Very easy to use as you just speak to it	Text to Speech (TTS) can be tricky as the bot may misunderstand the user because of their accent this can lead to frustration Can be harder to implement as it involves Text to Speech and Speech to Text components		Personal assistant style chatbot

B.1.2 Chatbot Types Description, Inputs, Outputs and examples

Chatbot Type	Description	Input	Input Example	Output	Output Example
Menu Based Chatbot	Works through a GUI (user presses buttons) This chatbot functions through a decision tree	Decision Tree - From KB User input - button presses User input - text (only if it requires the user to input their information)	A decision tree from KB: Node A has two children -> B and C these have their own children and so on until it reaches a child that tells the chatbot to do an action	Advance the decision tree Change the menu options according to the decision tree If required save the user information	Change the button options Show a contact box for the user to write his information (contact info)
Rule Based Chatbot	They follow a bunch of defined rules (if this then do that), things like word order, synonyms Things like helping the user make travel plans	Chatbot Rules from KB User input - text	Rules could be something like: IF text == specified then .. IF wordOrder == specified then ..	Reply to user with pre determined replies based on the rules Save user information to KB if necessary Transform user question into KB query	If the user asked to book a restaurant then ask which restaurant OR show the list of restaurants the chatbot can book If the user asked to book a plane trip then ask the arrival and departure airports The list of airports/restaurants the chatbot can show could be accessed from the KB?
Keyword Recognition Chatbots	These utilize keywords and NLP, their weakness is similar questions as there can be keyword redundancies/overlap	Keywords from KB User input - text	Keywords: restaurant, book, menu, trip, plane	Save user information to KB Use Keywords and NLP to understand the user intent and formulate an answer If necessary make KB queries to get the information the user asked for	User asks: What are the best hotels in Lisbon? Chatbot detects: keywords BEST and HOTELS and LISBON and gives the user a list of the top rated hotels in Lisbon It gets this list from the KB?
Hybrid between Keyword Recognition and Menu Based	Combines the ideas of both, giving the user the option to ask a question directly via text if the Menu Based part is not satisfying their demands	Decision Tree - From KB User input - button presses/text Keywords from KB	A decision tree from KB: Node A has two children -> B and C these have their own children and so on until it reaches a child that tells the chatbot to do an action Keywords: restaurant, book, menu, trip, plane	Change Menu options Advance the decision tree Use Keywords and NLP to understand the user intent and formulate an answer If necessary make KB queries to get the information the user asked for	Change the button options If the user writes in the input box the chatbot: Detects the Keywords and gives an appropriate answer using a KB query if needed

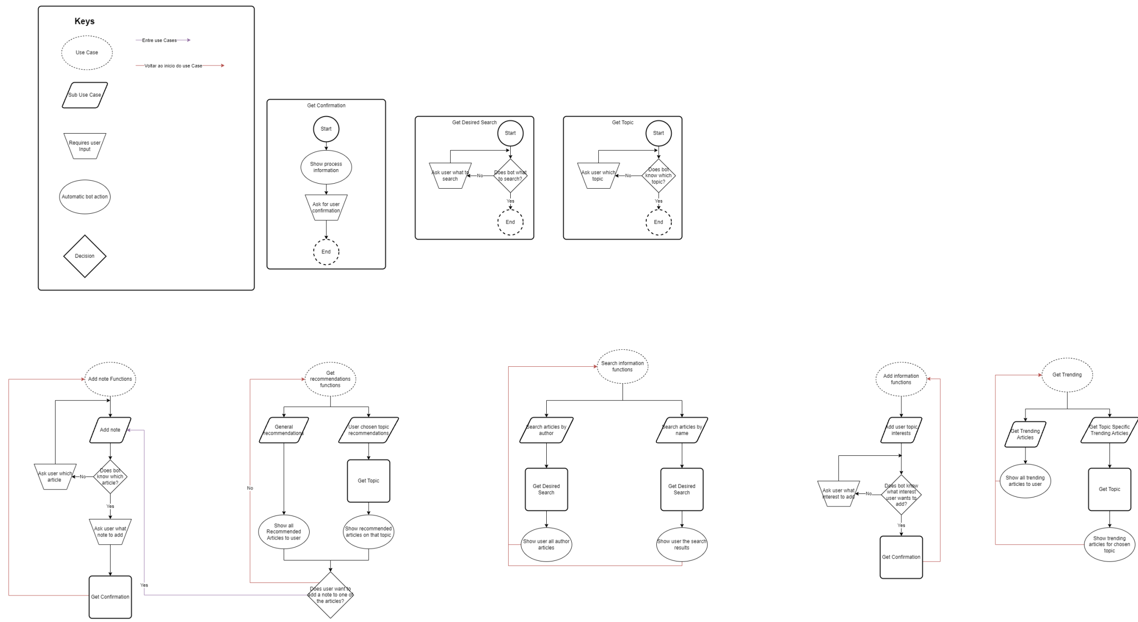
Chatbot Type	Description	Input	Input Example	Output	Output Example
Seq2Seq Chatbots (RNN)	These use ML to train with datasets of conversations to allow them to generate their own answers These chatbots are good for chat oriented applications	Conversation data from KB or a dataset to train the ML model User input	A .csv file with the conversations dataset to train the model	Store the conversation data (user preferences, user information, etc) into KB Generated response based on what the user said and the Seq2Seq model training	Extract user preferences from their answers/questions Return the generated answer from the ML model to the user
Contextual Chatbots/Reinforcement Learning Chatbots (ML chatbots)	These chatbots remember specific user interactions and can learn from them, they are also contextually aware	Conversation data from KB or a dataset to train the ML model User input Information from the current user from the KB if present to try prediction their wishes	A .csv file with the dataset (multiple conversations and the user profile for each) to train the ML model	Store the conversation data (user preferences, user information, etc) into KB If the user had a history of conversation then the chatbot will try to predict the user needs If there is no previous interaction with the user then the chatbot will try to understand the context and intent behind the user questions and generate appropriate replies through its ML model, NLP and KB queries	
Voice Based Chatbots	Speaking to these via voice like Siri or Alexa	User input Keywords from KB	Keywords: alarm, 4pm	Display the user request in Text to confirm if its correct Do the requested task (set an alarm, make a calendar appointment etc) If the user asked a question then convert it into a KB query and give the results	The chatbot would set an alarm if the user asked for an alarm for X hour Display the user speech in text If the user asked a question then convert it into a KB query and give the results

B.2 Planning chatbot dialogues

B.2.1 Tourism case study dialogues

B.2.2 Project Management case study dialogues

B.2.3 News management case study dialogues



B.2.4 My Life case study dialogues

Keys

Use Case

Sub Use Case

Requires user input

Automatic bot action

Decision

Get Node

Get Confirmation

Get Search

Add note Functions

Manage Information Functions

Information Search Functions

B.3 Botonic

B.3.1 Code snippet of a Botonic route

```
export const routes =
[
    {
        path: 'welcome',
        payload: 'welcome',
        text: /hi|hello/,
        action: Welcome,
    },
    //Personal Assistant Use Cases
    {
        path: 'check-appointments-on-specific-date',
        payload: 'check-appointment-on-date',
        action: GetDate,
        childRoutes: [
            {
                path: 'GetActivtiesOnDate',
                text: /.*/,
                action: GetActivtiesOnDate,
            },
        ],
    },
    {
        path: 'change-cancel-appointments',
        payload: 'change-cancel-appointment',
        action: GetWhatToChange,
        childRoutes: [
            {
                path: 'change-or-cancel',
                text: /.*/,
                action: GetChangeOrCancel,
                childRoutes: [
                    {
                        path: 'changeActivityDate',
                        payload: 'change',
                        action: GetNewDate,
                        childRoutes: [
                            {
                                path: 'GetConfirmationToChangeDate',
                                text: /.*/,
                                action: GetConfirmationToChangeDate,
                                childRoutes: [
                                    {
                                        path: 'Deny',
```

```
                                        payload: 'Deny',
                                        action: OperationCancelledMessage,
                                    },
                                    {
                                        path: 'confirm',
                                        payload: 'Confirm',
                                        action: ConfirmDateChange,
                                    },
                                ],
                            },
                        ],
                    },
                    {
                        path: 'cancelActivity',
                        payload: 'cancel',
                        action: ConfirmCancelation,
                        childRoutes: [

                            {
                                path: 'No',
                                payload: 'No',
                                action: OperationCancelledMessage,
                            },
                            {
                                path: 'Yes',
                                payload: 'Yes',
                                action: CancelActivityOrAppointment,
                            },


                        ],
                    },

                ],
            },
        ],
    },
```

B.3.2 Code snippet of a Botonic action

```
render() {
    return (
      <>
        <Text>Your appointments for {this.context.session.date} are:</Text>
        <Text> Skydiving</Text>
        <Text> Whale Watching</Text>
        <Reply payload='message-activity-producer'>Message a producer</Reply>
        <Reply payload='change-cancel-appointment'>Change or
        cancel an activity</Reply>
        <Reply payload='welcome'>Restart</Reply>
      </>
    )
  }
```

B.4 Botman

B.4.1 Example of a Botman Conversation Class

```
class CheckAppointmentsOnDate extends Conversation
{
    protected $dateToCheck;
    protected function AskDate()
    {
        $this->ask('Please specify the date you wish to check: ',function($answer)
        {
            $this->dateToCheck = $answer->getText();
            $this->say('You have Skydiving on at 13:00');
            $this->say('You have Whale Watching on at 16:00');

            $this->AskToMessageProducerOrChangeActivities();
        });
    }

    protected function AskToMessageProducerOrChangeActivities()
    {
        $question = Question::create('Do you want to message an
        activity producer or change/cancel an activity?')
            ->addButtons
            ([
            Button::create('Message Producer')>value('MessageProducer'),
                Button::create('Change/Cancel Activity')->value('Change/CancelActivity')
                Button::create('No')->value('Deny'),
            ]);
        $this->ask($question, function ($answer) {
            if ($answer->isInteractiveMessageReply())
            {
                if ($answer->getValue() === 'MessageProducer')
                {
                    return $this->bot->startConversation(new MessageProducer());
                }
                else if ($answer->getValue() === 'Change/CancelActivity')
                {
                    return $this->bot->startConversation(new ChangeCancelActivity());
                }
                else if ($answer->getValue() === 'Deny')
                {
                    $this->say('Thank you for using our services!');
                }
            }
            else
```

```
            {
                $this->repeat();
            }
        });
    }
    public function run()
    {
        $this->say('You choose to check your appointments on a date.');
        $this->AskDate();
    }
}
```

B.4.2 Code snippet of Botman routes

```
$botman->hears('Hi', function ($bot) {
$bot->reply('Hello!');
});

$botman->fallback(function ($bot) {
    $bot->reply('I did not understand you,
    please use the Help command to see my available functions.');
});

$botman->hears('Check appointments on date', function ($bot) {
    $bot->startConversation(new CheckAppointmentsOnDate);
});

$botman->hears('Change or Cancel', function ($bot) {
    $bot->startConversation(new ChangeCancelActivity);
});

$botman->hears('Check activity date', function ($bot) {
    $bot->startConversation(new CheckActivityDate);
});

$botman->hears('Message Producer', function ($bot) {
    $bot->startConversation(new MessageProducer);
});

$botman->hears('Help', function ($bot) {
    $bot->reply('These are my functions:');
    $bot->reply('Check appointments on a date');
    $bot->reply('Change/Cancel Appointments');
    $bot->reply('Check the date of an activity');
    $bot->reply('Message an activity producer');
})->skipsConversation();
```

```
$botman->hears('Stop', function ($bot) {
    $bot->reply('You have exited the conversation.');
})->stopsConversation();
```

B.5 Dynamic chatbot

B.5.1 Created JSON structure

```
{
    "firstInteraction": {
        "requirements": [],
        "title": "Please specify your prefered language",
        "buttons": [],
        "actions": [],
        "output": ["language"],
        "outputFormat": ["string"],
        "nextSteps": [
            "showLocations"
        ]
    },
    "showLocations": {
        "requirements": [
            "language"
        ],
        "title": "Input the option number you want",
        "buttons": [],
        "actions": ["ShowBookingLocations"],
        "output": ["location"],
        "outputFormat": ["range"],
        "nextSteps": [
            "askActivity"
        ]
    }
}
```

B.5.2 Example of an interaction using the JSON structure

```
"buttonsConfigPersonalAssistant": {
    "requirements": [],
    "title": "Hi there! These are my available options as a personal assistant",
    "buttons": [
        ["Ask appointments on specific date", "ask"],
        ["Change\/Cancel activity", "change activity", "cancel activity"],
        ["Check activity date", "activity date", "get date"],
        ["Message producer", "message", "talk to producer"],
        ["No thanks!"]
    ],
    "actions": [],
    "output": [],
    "outputFormat": [],
    "nextSteps": [
```

```
        "askDateOrShowAgendaConfig",
        "changeCancelActivityConfig",
        "askActivityToCheckDateConfig",
        "askActivityConfig",
        "endConversationConfig"
    ],
    "nextDialog": "personalAssistantDialog"
}
```

B.5.3 Code snippet of the method that runs chatbot dialogues

```
    if(count($currentInteraction['actions']) > 0)
    {
        $nextConfig = $this->ExecuteActions($interaction);
        if($this->bot->userStorage()->get('range') == "1-2")
        {
            $this->bot->userStorage()->save([$nextConfig["output"][0] => 1]);
            if(count($nextConfig['buttons']) == 0)
            {
                $this->bot->userStorage()->save(['range' => null]);
                $this->say('------------------------------------------');
                $this->StartNextInteraction($currentDialog, $nextConfig,true,false);
            }
            return;
        }
        else
        {
            $this->RunDialog($currentDialog, $nextConfig);
        }
        return;
    }
    if(count($currentInteraction['buttons']) > 0)
    {
        $this->RunButtonInteraction($currentInteraction,$currentDialog);
        return;
    }
    else
    {
        if(empty($interaction['title'] == false))
        {
            $this->RunInteraction($interaction,$dialog);
        }
        else //Stops printing of empty space if it has no title
        {
            $this->StartNextInteraction($dialog,$interaction,true, false);
```

```
        }
    }
```

B.5.4 Code snippet of the method responsible for running an interaction with no buttons

```
    {
        if(empty($interaction['output']) == false)
        {
            $this->ask($interaction['title'], function ($answer)
            {
                $userInput = $answer->getText();
                $functionConfig = $this->bot->userStorage()->get('config');
                $dialog = $this->bot->userStorage()->get('dialog');
                if (empty($functionConfig['output']) == false ||
                count($functionConfig['output']) > 0)
                {
                    if(count($functionConfig['outputFormat']) !=
                    count($functionConfig['output']))
                    {
                        $this->say('ERROR You must specify
                        a format for each output');
                        return;
                    }
                    else
                    {
                        $wasOutputSaved =
                        $this->SavedOutput($functionConfig,$userInput);
                        if($wasOutputSaved == false)
                        {
                            $this->repeat('Your input is incorrect,
                            please introduce it correctly');
                            $this->say($functionConfig['title']);
                            return;
                        }
                    }
                }
                $this->StartNextInteraction($dialog,$functionConfig,false, false);
                return;
            });
        }
        else
        {
            $this->say($interaction['title']);
            $this->StartNextInteraction($dialog,$interaction, false, false);
            return;
```

```
        }
    }
```

B.5.5 Code Snippet of Botman buttons interaction method

```
$buttonsArray = [];
    foreach ($buttonsInteraction['buttons'] as $id => $button)
    {
        $newButton = Button::create($button[0])->value($id);
        $buttonsArray[] = $newButton;
    }
    $question = Question::create('')
        ->addButtons($buttonsArray);
    $this->ask($question, function($answer)
    {
        $currentInteraction = $this->bot->userStorage()->get('currentInteraction');
        $currentDialog = $this->bot->userStorage()->get('currentDialog');
       if($answer->isInteractiveMessageReply())
       {
            $this->bot->userStorage()->save([
                'chosenButton' => $answer->getValue()
            ]);
           $this
           ->StartNextInteraction($currentDialog,$currentInteraction,false,true);
       }
       else
       {
           $textAnswer = $answer->getText();
           $mostSimilarOption =
           $this->GetMatchingButton($currentInteraction, $textAnswer);
           if($mostSimilarOption > count($currentInteraction['buttons']) - 1)
           {
               $this->say('This does not look like a valid option.
               Please click on a button or type the desired button input');
               $this->repeat();
           }
           else
           {
                $this->bot->userStorage()->save([
                'chosenButton' => $mostSimilarOption
                ]);
                $this
               ->StartNextInteraction($currentDialog,$currentInteraction,false,true);
           }
       }
```

```
});
```

B.5.6 Code snippet of the method responsible for checking if the interaction should be skipped

```
{
    if(count($nextInteraction['buttons']) == 0 &&
    count($nextInteraction['actions']) == 0)
    {
        $nextOutputs = $nextInteraction['output'];
        if(count($nextOutputs) > 0) //if it has outputs
        {
            $numOfFilledOutputs = 0;
            foreach ($nextOutputs as $nextOutput)
            {
                $savedOutput = $this->bot->userStorage()->get($nextOutput);
                if(empty($savedOutput) == false)
                {
                    $numOfFilledOutputs++;
                }
            }
            if($numOfFilledOutputs == count($nextOutputs))
            {
                return true;
            }
        }
    }
    return false;
}
```

B.5.7 Code snippet of the method responsible for fetching an interaction to fill the missing requirements

```
$result = [];
$nextDialog = $currentDialog;
$nextFunctionRequirements =
$currentDialog[$nextInteractionName]['requirements'];
foreach ($currentDialog as $interaction)
{
    $interactionOutput = $interaction['output'];
    $currentInteractionOutput = $currentInteraction['output'];
    if(count($interactionOutput) > 0)
    {
```

```
            for ($j=0; $j<count($interactionOutput); $j++)
            {
                if(in_array($interactionOutput[$j], $nextFunctionRequirements, false)
                &&
                !in_array($interactionOutput[$j], $currentInteractionOutput, false))
                {
                    if(empty($this->bot->userStorage()->get($interactionOutput[$j])))
                    {
                        $newNextInteraction = $interaction;
                        $newNextInteraction['nextSteps'] = [];
                        $newNextInteraction['nextSteps'][] = $nextInteractionName;
                        array_push($result, $nextDialog, $newNextInteraction);
                        return $result;
                    }
                }
            }
        }
    }
    return $result;
```

B.5.8 Code snippet of the class responsible for input validations

```
    $trimmedInput = trim($input);
    switch ($expectedFormat)
    {
        case 'date':
            return $this->isDate($trimmedInput);
            break;
        case 'int':
            return $this->isInteger($trimmedInput);
            break;
        case 'string':
            return true;
            break;
        case 'array':
            return $this->isArray($trimmedInput);
            break;
        case 'range':
            return $this->inRange($trimmedInput, $range);
            break;
        default:
            return false;
            break;
    }
```

B.6 Chatbot and NLP

B.6.1 Snippet of NLP corpus

```
{
    "intent": "Personal Assistant",
    "utterances": [
      "Personal Assistant",
      "Assistant",
      "Make appointment",
      "Change appointment",
      "Cancel appointment",
      "Message Producer",
      "Check agenda",
      "Check date of activity"
    ],
    "answers": [
      "78"
    ]
  },
  {
    "intent": "Booking",
    "utterances": [
      "Book appointment",
      "Book hotel",
      "Book recreation",
      "Book something",
      "Book activity",
      "Book room",
      "Book flight",
      "Book anything",
      "I want to make a booking?",
      "How do I make a booking?",
      "Can I book something?",
      "Please help me with the booking process"
    ],
    "answers": [
      "65"
    ]
  },
```

B.6.2 NLP Middlware *getResponse* method

```
{
    $response = $this->http->post($this->apiUrl, [], [
        'USER_INPUT' => $message->getText(),
```

```
    ], [
        'Content-Type: application/json; charset=utf-8',
    ], true);

    $this->response = json_decode($response->getContent());
    return $this->response;
```

B.6.3 NLP Middlware *received* method

```
$response = $this->getResponse($message);

$userInput = $response->utterance ?? '';
$language = $response->language ?? '';
$classifications =
isset($response->classifications) ? (array) $response->classifications : [];
$intent = $response->intent ?? '';
$intentScore = $response->score ?? '';
$entities = isset($response->entities) ? (array) $response->entities : [];
$sentiment = isset($response->sentiment) ? (array) $response->sentiment : [];
$actions = isset($response->actions) ? (array) $response->actions : [];
$possibleAnswers = isset($response->answers) ? (array) $response->answers : [];
$generatedAnswer = $response->answer ?? '';

$message->addExtras('userInput', $userInput);
$message->addExtras('language', $language);
$message->addExtras('classifications', $classifications);
$message->addExtras('intent', $intent);
$message->addExtras('intentScore', $intentScore);
$message->addExtras('entities', $entities);
$message->addExtras('sentiment', $sentiment);
$message->addExtras('actions', $actions);
$message->addExtras('possibleAnswers', $possibleAnswers);
$message->addExtras('generatedAnswer', $generatedAnswer);

return $next($message);
```

B.6.4 Snippet of routes file

```
$nlpJS = NLPjs::create($setUp->GetNlpServerPort('NLPConfig.json'),'en');
$botman->middleware->received($nlpJS);

$botman->hears('(.*)' , function($bot) {

    $helper = new HelperValidateInput();
```

```
        $setUp = new SetUp();
        $defaultCommands = $setUp->GetCommandsAndReplies('ChatbotDefaultMessages.js

        $intent = $bot->getMessage()->getExtras('intent');
        $userInput = $bot->getMessage()->getExtras('userInput');
        $generatedAnswer = $bot->getMessage()->getExtras('generatedAnswer');

        $dialogTemplateLink = '...' . $generatedAnswer;

        if($helper->isInteger($generatedAnswer) &&
        filter_var($dialogTemplateLink, FILTER_VALIDATE_URL))
        {
            $bot->startConversation(new DynamicConversation($dialogTemplateLink));
        }
        else
        {
            $bot->reply($defaultCommands['FallbackMessage']);
        }
    });
```

B.7 Connecting chatbot to KB

B.7.1 KB template format example

```
  {
"nodes": [
  {
    "existingNodeId": ...,
    "type": "Dialog",
    "typeId": ...,
    "properties": {
      "reference": "bookingDialog"
    },
    "nodes": [
      {
        "existingNodeId": ...,
        "type": "Interaction",
        "typeId": ...,
        "properties": {
          "reference": "askBookingService",
          "Title": "Hi there! To book something please
          specify the type of booking, example: hotel, flight, etc "
        },
        "arches": [
          {
            "type": "Writes",
            "typeId": ...,
            "properties": {},
            "node": {
              "existingNodeId": ...,
              "type": "Variable",
              "typeId": ...,
              "properties": {
                "reference": "bookingService",
                "Format": "string"
              }
            }
          },
          {
            "type": "Starts",
            "typeId": ...,
            "properties": {},
            "node": {
              "existingNodeId": ...,
              "type": "Interaction",
              "typeId": ...,
              "properties": {
```

```
                "reference": "showBookingOptions",
                "Title": ""
              }
            }
          }
        ],
        "nodes": []
      },
      {
        "existingNodeId": ...,
        "type": "Interaction",
        "typeId": ...,
        "properties": {
          "reference": "askBookingDates",
          "Title": "Please specify the booking dates "
        },
        "arches": [
          {
            "type": "Reads",
            "typeId": ...,
            "properties": {},
            "node": {
              "existingNodeId": ...,
              "type": "Variable",
              "typeId": ...,
              "properties": {
                "reference": "bookingService",
                "Format": "string"
              }
            }
          },
          {
            "type": "Writes",
            "typeId": ...,
            "properties": {},
            "node": {
              "existingNodeId": ...,
              "type": "Variable",
              "typeId": ...,
              "properties": {
                "reference": "bookingStartDate",
                "Format": "date"
              }
            }
          },
```

```
            {
              "type": "Writes",
              "typeId": ...,
              "properties": {},
              "node": {
                "existingNodeId": ...,
                "type": "Variable",
                "typeId": ...,
                "properties": {
                  "reference": "bookingEndDate",
                  "Format": "date"
                }
              }
            },
            {
              "type": "Starts",
              "typeId": ...,
              "properties": {},
              "node": {
                "existingNodeId": ...,
                "type": "Interaction",
                "typeId": ...,
                "properties": {
                  "reference": "askNumberOfParticipants",
                  "Title": "Please specify the number of participants "
                }
              }
            }
          ],
          "nodes": []
        },
    ...
  ]
}
```

B.7.2 Method to map templates do dialog

```
$currentDialog = [];
$decodedTemplate = $this->DecodeTemplate($template);
if(isset($decodedTemplate['nodes']) == false)
{
    return [];
}
$dialogs = $decodedTemplate['nodes'];
foreach ($dialogs as $dialog)
{
```

```
        $dialogProperties = $dialog['properties'];
        $dialogName = $dialogProperties['reference'];
        $dialogNodes = $dialog['nodes'];
        $currentDialog = $this->MapTemplateInteractions($dialogNodes);
    }
    $this->WriteTemplateToJson($currentDialog);
    return $currentDialog;
```

B.7.3 Method to get first interaction of the dialog

```
$firstInteractionName = '';
foreach($dialog as $interactionName => $interaction)
{
    $lowerCaseInteractionName = strtolower($interactionName);
    if(str_contains($lowerCaseInteractionName, 'firstinteraction'))
    {
        $firstInteractionName = $interactionName;
    }
}

return $firstInteractionName;
```

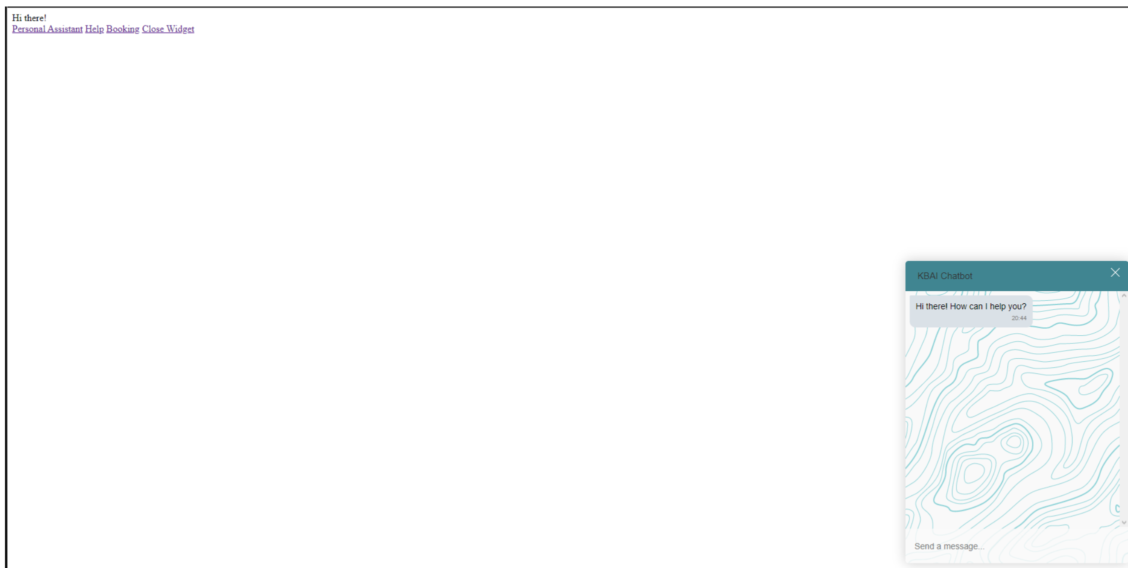
B.7.4 Chatbot API methods

METHOD	DESCRIPTION
botmanChatWidget.open()	Open the chat widget.
botmanChatWidget.close()	Close the chat widget.
botmanChatWidget.toggle()	Toggle the chat widget.
botmanChatWidget.say(text)	Say something on behalf of the user. The given text is visible in the widget.
botmanChatWidget.whisper(text)	Similar to say, with the difference that the text is not visible.
botmanChatWidget.sayAsBot(text)	Say something on behalf of the chatbot. The text is visible in the widget.

B.7.5 Chatbot API example file

```
<body>
    Hi there!
    <div class="links">
        <a href="#" onclick="botmanChatWidget.say
        ('I want to use the Personal Assistant');">Personal Assistant</a>
        <a href="#" onclick="botmanChatWidget.sayAsBot
        ('Hello, how can I help you?');">Help</a>
        <a href="#" onclick="botmanChatWidget.whisper('Booking');">Booking</a>
        <a href="#" onclick="botmanChatWidget.close();">Close Widget</a>
    </div>
    <script>
        var botmanWidget = {
            frameEndpoint: '...',
            title: 'KBAI Chatbot',
            introMessage: 'Hi there! How can I help you?',
            aboutLink: '',
            aboutText: ''
        };
    </script>
    <script src='...'></script>
</body>
```

B.7.6 Chatbot API website view



B.8 API integration

B.8.1 Function to return mapped API response

```
{
    $postHeaders = ['Content-Type: application/json; charset=utf-8'];
    $responseToMap = $this->PostToAPI($apiURL,$postParameters, $postHeaders);

    if (isset($responseToMap) == false || isset($responseToMap['d']) == false)
    {
        return [
            "errorOcurred" => true
        ];
    }
    return $responseToMap['d'];
}
```

B.8.2 Example of method that implements an API

```
public function ShowBookingLocations()
{
    $language = $this->userInputs['language'];
    $postParameters = ['Language' => $language];
    $mappedArray = $this->apiResults
    ->MapAPIToArray
    ('https://pickabuy.gestoolsasp.com/ws_qual/catalog.asmx/ListTypeByLocation'
    $postParameters);
    if(empty($mappedArray['errorOcurred']) == false)
    {
        return $mappedArray;
    }
    $messages = [];
    $i = 1;
    foreach ($mappedArray as $location)
    {
        $message = $i . ': ' . $location['name'];
        $messages[] = $message;
        $i++;
    }
    return [
        'range' => '1-' . $i,
        'saveOutput' => true,
        'outputToSave' => $mappedArray,
        'messages' => $messages];
}
```

B.8.3 Method that handles output of primitive functions

```
if(empty($actionOutput['deleteCache']) == false)
{
    $this->bot->userStorage()->delete();
    return;
}
if(empty($actionOutput['errorOcurred']) == false)
{
    $this->bot->userStorage()->save(['errorOcurred' => true]);
    return;
}
if(empty($actionOutput['range']) == false)
{
    $this->bot->userStorage()->save(['range' => $actionOutput['range']]);
}
if(empty($actionOutput['saveOutput']) == false)
{
    if($actionOutput['saveOutput'])
    {
        $this->bot->userStorage()
        ->save(['savedActionOutput' => $actionOutput['outputToSave']]);
    }
}
foreach ($actionOutput['messages'] as $message)
{
    if(str_contains($message, 'ERROR'))
    {
        $this->bot->userStorage()->save(['errorOcurred' => true]);
    }
    $this->say($message);
}
return;
```

B.8.4 Example of using range Output Format in an interaction

Please specify your prefered language
PT
1: Lisbon
2: Madeira
3: Calheta
4: Camara de Lobos
5: Canico
6: Fajã dos Padres
7: Funchal
8: Lugar de Baixo
9: Ponta Delgada
10: Ponta do Sol
11: Porto Moniz
12: Sao vicente
Input the option number you want
Attachment