Bachelor Thesis

November 24, 2021

GitHub Repository Search Bot

Design of a GitHub Repository Search Chatbot

Michael Brülisauer

of St. Gallen, Switzerland (17-714-379)

supervised by

Prof. Harald C. Gall
Dr. Pasquale Salza
Marco Edoardo Palma

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

GitHub Repository Search Bot

Design of a GitHub Repository Search Chatbot

Michael Brülisauer

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Michael Brülisauer, michael.bruelisauer@uzh.ch

URL: https://github.com/mibruel/GitHubSearchBot

Project period: 24.05.2021 - 24.11.2021

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Dr. Pasquale Salza from the Software Evolution and Architecture Lab research group (s.e.a.l) at University of Zurich for supervising my thesis. In frequent meetings, he provided valuable inputs, suggestions and tips to guide me in my thesis.

Additionally, I would like to thank Marco Edoardo Palma from the s.e.a.l. group for assisting me with valuable inputs and feedback to complete my thesis.

Finally, I would like to thank Prof. Dr. Gall for allowing me to write my thesis at the research group s.e.a.l at the department of Informatics at the University of Zurich.

Abstract

Searching code is a daily task for every software engineer. With the growing amount of data available on the internet, software engineers are actively researching new advanced techniques to find certain publicly available code for reusage. This thesis further contributes to this active research by developing a new conversational-based approach for software engineers to find publicly available software. With the development of a conversational agent (chatbot), this thesis describes the design and implementation of a new approach that is built on the growing demand of conversational agents to fulfil a specific task. The chatbot is able to return a repository that best matches a project description provided by a user throughout a natural language conversation. The chatbot is capable of asking the user questions about the repository to search for and remembers past answers from the user. This chatbot offers an easy-to-use interface for software engineers to retrieve a repository with certain specifications. The implementation presented in this thesis is further expandable in future work by increasing the knowledge domain of the chatbot.

Zusammenfassung

Die Suche nach Code ist eine alltägliche Aufgabe für jeden Softwareentwickler. Mit der wachsenden Menge an Daten, die im Internet verfügbar sind, erforschen Softwareingenieure aktiv an neuen fortschrittlichen Techniken, um bestimmten, öffentlich verfügbaren Code für die Wiederverwendung zu finden. Diese These leistet einen weiteren Beitrag zu dieser aktiven Forschung, indem ein neuer konversationsbasierter Ansatz entwickelt wird, mit dem Softwareingenieure öffentlich verfügbare Software finden können. Mit der Entwicklung eines konversationellen Agenten (Chatbot) beschreibt diese Arbeit das Design und die Implementierung eines neuen Ansatzes, der auf der wachsenden Nachfrage nach konversationellen Agenten zur Erfüllung einer bestimmten Aufgabe aufbaut. Der Chatbot ist in der Lage, eine Repository zurückzuliefern, welche am meisten mit einer Projektbeschreibung übereinstimmt, die von einem Benutzer im Rahmen einer Konversation in natürlicher Sprache eingegeben wurde. Der Chatbot ist in der Lage, dem Nutzer Fragen zum Repository zu stellen und sich an frühere Antworten des Nutzers zu erinnern. Dieser Chatbot bietet eine einfach zu bedienende Schnittstelle für Softwareingenieure, um ein Repository mit bestimmten Spezifikationen abzurufen. Die in dieser Arbeit vorgestellte Implementierung kann in Zukunft durch die Erweiterung des Wissensbereichs des Chatbots weiter ausgebaut werden.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Since the birth of the Software Engineering field and still to this day, software engineers are confronted with the challenge to create software systems in a reliable, cost-effective way [1]. Due to this prevalent goal of achieving large and reliable software systems in a quicker and cheaper way, reusing existing software has been of interest in this field for a long time and has been researched for years [1]. More precisely, a study by Singer et. al. in 1997 has displayed that "the most frequent developer activity was code search" [2], as cited by Sadowski et al. [3]. Furthermore, the study by Sadowski et al. [3] indicated that a programmer on average performs 12 queries on Google Code Search per workday to search for code. It is apparent that software developers have a high interest in code search tools and are therefore actively researching advanced code search techniques. As a result, more recent approaches have started integrating machine learning techniques for learning to retrieve code from a natural language search query [4], [5], [6].

This thesis intends to further contribute to this ongoing research in the field of software reusage with a new approach. By developing a conversational agent (chatbot) that assists a software engineer in finding reusable software, this thesis presents a new dialog-based approach to find publicly available, reusable software. The approach builds on the growing demand of conversational agents to carry out tasks [7] and deviates from a usual single-time search query due to the ability to converse with a user.

Chatbots are known as communication programs that can recognize a user's input in natural language and are able to respond intelligently to it, similar to how humans communicate [8]. What makes a chatbot particularly useful in carrying out a task is its continuous form of interaction whereby the user is able to follow up and refine a task by conversing with the chatbot. With the ease of use and continuity provided by the conversational-based interaction, the chatbot developed in this thesis is aiming to provide an efficient solution to retrieve software artefacts that are of interest to the user.

Contribution. This thesis presents the development process of designing and implementig a new chatbot to solve the problem of finding software artefacts based on the interest of a user. In particular, this chatbot is able to retrieve a repository from GitHub, the largest development platform world-wide [9], that best matches a project description provided by the user. The chatbot is capable of asking the user a set of questions about the repository in order to narrow down a single repository that best matches the user's answers. Furthermore, the chatbot is capable of understanding natural language answers by the user, thereby providing an easy-to-use interface. Lastly, the chatbot is capable of leading the search conversation and handling nonsensical responses from the user.

Thesis organization. To demonstrate the development process of this chatbot, this thesis focuses on four main contributing parts. In chapter 3, the state-of-the-art techniques used for chatbots in recent literature are reviewed. Chapter 4 describes the designing process for the architecture of the chatbot by establishing certain requirements and fulfilling these requirements with the appliance of a state-of-the-art chatbot approach. In chapter 5, the chatbot is implemented as a proof-of-concept with the designed architecture, focusing on the infrastructure and the implemented components used to deploy the chatbot in production. Chapter 6 demonstrates a quick sample conversation with the deployed chatbot to prove that the chatbot design can successfully be implemented for this use case. Lastly, the development is concluded and some core challenges as well as limitations that have come up during the development process are presented. Possible contributions are raised where future work can further improve and extend the features of this chatbot.

Chapter 2

Related Work

Relating to chatbots that perform information retrieval tasks, different approaches have been developed to successfully carry out a specified task.

Bhavika R. Ranoliya et al. [10] have built a chatbot that is able to answer frequently asked questions by a user. They have used a rule-based approach, where each user input is matched to a predefined knowledge base that includes patterns (questions) and templates (answers). If the user input matches a predefined pattern, the chatbot returns the template answer stored for that pattern. This chatbot uses an Artifical Intelligence Markup Language (AIML) script to store the questions and answers as a knowledge base which can be matched with the user input. The AIML-approach used by Bhavika R. Ranoliya et al. is a straightforward approach without the need of complex developments. The disadvantage to an AIML approach is that each rule has to be handwritten, a large knowledge base entails a lot of handcrafting.

In the paper by Jhonny Cerezo et al. [11] a chatbot was developed that can recommend experts of open projects for a certain field or topic, based on what the user is interested in. For the user input analysis, this chatbot makes use of two NLP techniques, one to categorize the sentence into predefined categories and one to extract source code artifact names such as method, class or package names out of the sentence. To create a knowledge base with expert profiles, Jhonny Cerezo et al. have used an algorithm that makes use of source code mining to define who has expertise in which category based on the authors of the source code. The expert who has most expertise in the extracted category and source code artifact is then recommended to the user. The approach used by Jhonny Cerezo et al., compared to the AIML approach by Ranoliya et al., requires less handcrafting since the artifact names are extracted by an NLP algorithm and matched to the expertise of experts. However, this approach is dependent on more complex NLP techniques than an AIML-based approach.

Ashay Argal et al. [12] have developed an intelligent chatbot that is able to recommend traveling choices based on a conversation with the user. The user provides multiple inputs of information about his traveling desires regarding flight, car or hotel and the chatbot returns a recommendation based on the input as well as the user preference, which is either predefined or learned with the usage of the chatbot. For this chatbot, the communication was speech-based with speech recognition and the user input was analysed with an NLP approach. The queries were gathered to create a query to retrieve the best option from the database. They have developed two approaches for the recommendation part; one where the search query is first matched to the user preferences and scored based on the best match. Afterwards, the best match from the data set, including the user preference score, is found. The best match is then returned as a recommendation to the user. The second approach makes use of a machine learning model that is based on a neural network to predict the best recommendation. Regarding the response generation, the questions

are predefined while the last response with the recommendation is generated. The advantage of the approach developed by Ashay Argal et al. is that this approach can perform queries with incorporation of an additional (predicted) scoring factor, such as the predicted user rating in their developed chatbot. A disadvantage to this approach is its dependency on a large set of data to train the model for better accuracy.

The approach used by Tsung-Hsien Wen et al. [13] introduced in 2016 was a novelty in task-oriented chatbots. They implemented an end-to-end trainable dialog system that is able to conversate with the user and recommend restaurants based on the user's input. The dialog system consists of an intent network, a belief tracker, a policy network, a generation network and a database operator. The user inputs are converted into "a distributed representation generated by an intent network and a probability distribution over slot-value pairs called the belief state generated by a set of belief trackers" [13]. The value with the highest probability is then used for a search query that is performed on the database. The generation network of this dialog system generates a response based on a vector created by the policy network. This vector is a combination based on the result from the query, the intent representation and the belief state. The generated response, including the result from the database query, are structured to a response that is return to the user. The model was trained with 680 dialogs that are specific to the domain of searching restaurants. The advantage of this end-to-end dialog system is that no information about the underlying task is needed, with a certain training corpus the dialog system can task-specifically be used. Same as for the approach by Argal et al., a disadvantage to the approach by Tsung-Hsien is its dependency on enough training data to train the end-to-end dialog system.

Chapter 3

Background

3.1 Chatbots

A chatbot is known as a communication program that can recognize a user's input and is able to respond intelligently to it, similar to how humans communicate [8]. Chatbots use two forms of conversation types, which can either be speech and text. Chatbots with an integrated text technology are able to understand user input in form of natural language text and respond back in natural language text [8]. Chatbots which are able to understand audio inputs from a user and respond with a speech-based response use the same foundational techniques as text-based chatbots. However, a speech-based chatbot additionally uses speech analysis and speech generation techniques to translate audio to text and the other way around [14].

Knowledge Domain. Chatbots function on a certain knowledge domain. The term knowledge domain in the context of chatbots is used to describe the set of user inputs that the chatbot is able to understand. The knowledge domain of chatbot is typically categorized into one of two types. The first type of knowledge domains are called open-domain. Open-domain chatbots are able to understand any input from a user about any topic, without a restricting set of knowledge [15]. The second type of domains are closed-domain chatbots or domain-specific chatbots. Chatbots with closed domains do only understand a specific set of user inputs and can act and response based on these inputs [16], while user inputs outside of this domain cannot be handled by the chatbot.

Task-Oriented Chatbots. The technology and concepts for each chatbot may differ with the use cases it has been built for. Almansor et al. [17] differentiate chatbots by their purpose into a task-oriented and non-task-oriented category. Task-oriented chatbots, as the name suggests, are chatbots that serve for a certain task. Typical examples are online booking of a flight ticket, place an order in an online service or checking the status of an order in an online service. As Hussain et al. [18] state, task-oriented chatbots operate within a closed domain, meaning that these chatbots cannot perform small talk, but are rather suited for specific user inputs relating a chatbot specific task.

Non-task-oriented Chatbots. Hussain et al. [18] describe non-task-oriented chatbots as chatbots that are designed with the goal to perform a human-like conversation. Non-task-oriented chatbots typically have an entertainment value, the focus is on the conversation itself rather than a task to solve. They operate on an open domain, therefore should be able to respond intelligently to any user input given. Machine learning based techniques in open domain chatbots generate a response based on the user input rather than responding with a predefined response [15].

3.2 Chatbot Approaches

Within the following sections, the best-practice chatbot approaches that have arisen since the beginning of chatbot development are declared. The state-of-the-art approaches with their underlying techniques are presented and their applicability for certain use cases are investigated.

3.2.1 Rule-Based Conversation

The first ever developed chatbot named ELIZA was built on the rule-based approach [19], thereby introducing the earliest chatbot approach created. Rule-based chatbots generate a response by a predefined rule [20]. The rules in chatbots are based on a matching premise where a response is selected when a user input was matched to an entry in a knowledge base.

Pattern Matching

Pattern Matching is the core technique for rule-based chatbots, with a various number of forms and complexity developed over the history of chatbots [21]. Pattern Matching is described as the comparison of two patterns to check whether the patterns are the same, in which case they match [22]. When used as a chatbot technique to generate a response, this commonly refers to the matching of a natural language input by the user to a predefined list of patterns in a knowledge base [23]. Due to the variety of implementations, two well known chatbots called ELIZA and ALICE that applied the Pattern Matching technique are presented.

ELIZA. The first basic approach of Pattern Matching was introduced by the first chatbot known as ELIZA [19]. To be more precise, ELIZA uses a parsing technique and applies a Pattern Matching technique afterwards. Bradesko et al. [21] describe parsing as a technique to define the grammatical structure of an input and convert it into a simplified set of words. Furthermore, the converted set of words can then for example be matched to a pattern, making it possible to match a pattern to multiple sentences with the same grammatical structure, as stated by Bradesko et al. [21]. ELIZA uses the parsing technique to extract each word of a sentence. If a word is declared as a keyword, this keyword will then be matched to a pattern in the knowledge base [19], thereby selecting the response that is linked to the matched pattern.

Artificial Intelligence Markup Language (AIML). Another technique that is based on pattern matching is the AIML technique. AIML was introduced in 1995 with the widely known ALICE chatbot [23]. ALICE has won the Loebner prize three times, which is an annual competition to decide the best performing chatbot of the year [21]. As described by Abushawar et al. [23], ALICE makes use of AIML files to understand natural language and relies on the pattern matching technique. AIML files consist of categories which are optionally structured in topics. A category consists of a pattern to match with the user input and a template which is the output response when this pattern was matched. Furthermore, Abushawar et al. [23] state that the matching of the user input with the patterns is done by matching each word of the patterns with the input and selecting the longest pattern match as the best match. The output response of the chatbot is the template of the category with the best matching pattern.

3.2.2 Machine Learning-Based Conversation

Machine learning (ML) approaches are a more modern approach for chatbots. ML-approaches are different to rule-based approaches as they do not rely on handcrafted rules to generate a response, but rather use ML models that are trained with data sets to understand user inputs and generate responses [20]. As further described by Adamopoulou and Moussiades [20], ML-based chatbot approaches use Natural Language Processing (NLP) to extract information of the user input to understand user input in natural language. To generate or retrieve a response, ML-based chatbot approaches use a trained ML model based on the extracted information.

Natural Language Processing

The goal of natural language processing (NLP) in chatbots is to convert unstructured user input into a structure that can be further used by the chatbot to create an appropriate response [24]. NLP techniques are used to extract meaning and information out of natural language, most of them using machine learning [20].

A popular NLP technique used in chatbots is the intent classification. It makes use of machine learning models to classify a sentence into predefined intent categories [25]. An intent describes the purpose of the sentence [20]. For example, an input of "Hello!" can be classified into an intent "Greeting Intent". According to Adamopoulou and Moussiades [20], by classifying the input into an intent, the chatbot can then further handle the response based on this intent. Intents are domain-specific and have to be defined beforehand for classification. The model that identifies intents needs to be trained with training phrases that represent this intent.

Apart from the intent of a user input, NLP also uses techniques to extract information out of the user's input. For example, a user input "How is the weather at 5 pm?" could have an intent "weather prognose", but the information "5 pm" would need to be extracted from the input so the chatbot can retrieve the weather at that specific time. According to McTear [26], as cited by Cahn [24], extraction techniques tokenize words, numbers, punctuation marks etc. and analyse the generated tokens afterwards. Historically, the models used to extract information are handcrafted, but modern approaches, such as deep learning, can also extract information based on data-driven and statistical models. A well known type of information extraction is Named Entity Recognition, where a named key-word is detected in a sentence and classified into a class of named entities which can be anything defined, such as books, places, dates and so on [27].

Retrieval-Based Response

As Zhao Yan [28], states, the retrieval-based response generation does, similar to the rule-based approach, select a predefined response, with the difference that this technique uses ML techniques to first analyse the user input. In retrieval-based response generation, the user input is matched with a training corpus, thereafter the matches are ranked by a score and the highest scored response is returned to the user. The scoring of the matches is often handled by a neural network model. There are multiple types of neural networks and algorithms that are appliable for the match scoring (e.g. sequence-to-sequence model that is shortly described in the next subsection) [28]. Yu Wu et al. [29] have defined a three-step process for retrieval-based responses: Firstly, the user input is pre-processed. Secondly, all response candidates are selected from the input-response pair index, thereby removing all other response candidates that are not indexed as a pair with the input. The response candidates and the input are then matched for their similarity and scored based on how similar they are. Lastly, the response candidates are ranked by a pretrained model and the response with the highest rank is chosen. Yu Wu et al. [29] report that, in comparison to a state-of-the-art sequence-to-sequence generative-based model, their developed retrievel-based model has chosen better responses during their experiment.

Generative-Based Response

Generative Response Models are a relatively modern approach, this technique was first introduced by Ritter et al. in 2011 [30]. Generative based response models allow a chatbot to generate a response based on the input and does not depend on predefined responses. The model is trained with training data that consists of real dialogues and makes use of statistical analysis to "translate" an input to a response [30]. A widely used technique is the sequence-to-sequence (seq2seq) model, a successor from Cho et al. in 2014 [31], to the first generative model by Ritter. As described by Cho et al. [31], it makes use of two RNN models, one to encode the user input, and the second to decode the input and thereby generate a response. More precisely, they state that the encoder converts the input over multiple steps, so called hidden states, into a vector, the decoder afterwards predicts the response based on the results of the encoding. The seq2seq model can generate a response or can also be used as a scoring technique in retrieval-based models. Using statistical machine translation to generate responses is the popular approach for open domain chatbots, since it does not depend on specific handcrafted rules or responses.

3.2.3 Dialog Management

After a response has been generated or selected, there are additional techniques that were defined for chatbots to simulate a human-to-human conversation. This section proposes three common approaches used to manage a dialog.

Conversation Handling. Many chatbots use communication strategies to reduce errors, which are situations when no response could be selected or generated, during the conversation with the user. One simple approach from McTear [26], as cited in Cahn [24], is to make the chatbot partially lead the conversation within the knowledge domain of the chatbot, as this reduces errors of not recognizing the user input. Another popular communication strategy approach is to confirm with the user what the chatbot understood. This can be achieved by repeating back the understanding of the chatbot, e.g. "Is it correct that you want to...?" [20]. This can be especially useful for chatbots to learn what user inputs were not understood and to adapt the chatbot with this information.

Language Tricks. Language tricks are a common approach used to make a dialog appear more human like. A typical use case for a language trick is when no response could be matched with a high certainty. In this case, Yu et al. [32] as cited in Cahn [24], propose some response strategies such as switching the topic, telling a joke, asking open ended questions or let the user provide more information to keep the conversation up. Additionally, some chatbots produce canned responses, which are hard-coded responses for a certain input [21].

Human-Imitation. One way to make a conversation more human-like, is to give the chatbot an own personality and give it human-like features. The approach of giving the chatbot a personal story and identity [21] is used to make the user feel like he is conversating with a human, rather than a machine. In a smaller concept, a human could also be imitated by purposely including typos in the response [21]. Human-imitation is more prevalent in non-task-oriented chatbots that usually focus on the entertainment value of a conversation.

3.2.4 Backend

Task-oriented chatbots often interact with other systems to perform their task, and therefore have implemented a connection to a backend or directly to a database. The connection to a backend allows the chatbot to perform tasks including an external system, which allows the chatbot to access external information or induce specific logic into the conversation.

For rule-based chatbots, Adamopoulou and Moussiades state that the chatbot is dependent on a database or data storage that stores a knowledge base. The knowledge base consists of the handcrafted rules used to match the user input to a pattern and select the response of the pattern with the best match. [20] Furthermore, for chatbots that keep track of the context of a conversation, a database is used to store information about previous or the current conversation. This implementation further improves the conversation handling of a chatbot since the responses can be selected based on the context, and are thereby contextually more correct and precise [20].

Chapter 4

Chatbot Design

This thesis dissects the development of a chatbot that retrieves a GitHub repository which matches a user's description. This entails the design of the architecture for the intended chatbot and thereafter the implementation of said architecture. This chapter focuses on the designing process of the chatbot architecture by first defining a description of what the chatbot is intended to do, and create a list of user requirements which the chatbot must fulfil. Afterwards, the components of the chatbot and the design choices that are applied to meet the listed user requirements are explained in detail.

4.1 Description

This chatbot can be described as an intelligent agent that is able to create a conversation with a user, with the goal to collaboratively search for a GitHub repository that best matches a project description provided by the user throughout the conversation. During the conversation, the chatbot continuously asks the user for information about a GitHub repository that the user is looking for, narrowing down the number of repositories that match the description of the user and eventually return a repository that best matches the answers of the user. What differentiates this chatbot to an instant single-time search function, which exists as such on the GitHub website already, is the ability to narrow down the best repository matched with the user's answers during the conversation and assist the user while doing so. The chatbot assists the user by asking questions about the repository they are looking for, being able to understand answers in complete sentences and imposing logic into the conversation to create a better user experience.

Since this task has a specific task to carry out, this chatbot is considered a task-oriented chatbot with a certain knowledge domain. The knowledge domain of this chatbot is the set of answers to the questions asked by the chatbot.

4.1.1 Requirements

To define what this chatbot must be able to do in detail, a list of user requirements is defined. Table 4.1 shows all the requirements for this chatbot to be able to carry out its specified task.

For a recommendation of a GitHub repository, the chatbot needs to know what key-information is of interest to the user when searching for a repository. In general, the user must to be able to have a conversation with a chatbot by typing into a text field and receiving a response in a text field. A software engineer typically wants the repository to be for a specific use case or field. Furthermore, a software engineer is typically interested in repositories from a certain date range to exclude old approaches that are outdated. Additionally, the following details are also defined

ID	Requirement Description
1	As a user, I want to be able to communicate to the chatbot by typing into a textfield.
2	As a user, I want to be able to start a conversation with the chatbot where the chatbot guides me to search for a GitHub repository.
3	As a user, I want the chatbot to ask me about topics (keywords) of the repository I am looking for.
4	As a user, I want the chatbot to ask me about the programming language of the repository I am looking for.
5	As a user, I want the chatbot to ask me about a time span of when the repository that I am looking for was created.
6	As a user, I want the chatbot to ask me about an earliest date of when the repository I am looking for was last updated.
7	As a user, I want the chatbot to ask me about the number of stars of the repository I am looking for.
8	As a user, I want the chatbot to ask me whether I am interested in a repository that is licensed and therefore open-sourced and available for software reusage or not.
9	As a user, I want the chatbot to ask me questions about the repository in an intelligent matter, thereby narrowing down the results of repositories to a single best match with my answers.
10	As a user, I want the chatbot to ask me a question again if it did not understand my response.
11	As a user, I want the chatbot to tell me the amount of repositories that match my answers so far.
12	As a user, I want the chatbot to return me the repository that best matches all my answers.

Table 4.1: User Requirements

as key-information when searching a repository: The programming language used is a key information for a software engineer to exclude repositories in languages that are not familiar to a software engineer. The number of stars from GitHub repositories define the popularity of a repository and informs the engineer whether the repository has been of interest for other engineers in the past. Furthermore, the number of authors informs the engineer whether the repository was an implementation from a single developer or whether it is a large project from multiple authors. This is relevant information when looking for software artefacts, as developments from a single developer, for example, might not have been reviewed by another developer. The license of a GitHub repository is relevant for a software engineer to confirm that the repository is open-sourced and therefore usable for software reusage or not, as unlicensed repositories fall under the default copyright law and therefore are not reusable [33].

The chatbot must ask the user about their interest considering all these specific attributes of a repository. The chatbot should also ask a question repeatedly if the answer from the user was not understood. Lastly, the chatbot should keep the user updated about the number of repositories that match their answers and eventually return the repository that best matches all the user's answers.

4.2 Design

After the definition of what the chatbot must be able to do, including the requirements one has as a user, the architecture must be designed with consideration of the state-of-the-art chatbot approaches that were presented. Each component that this chatbot consists of is defined by the chosen approach and the chosen underlying techniques.

4.2.1 Dialog Handler

The dialog handler component, as defined for this architecture, handles the interaction with the user and is the central part of a chatbot. The dialog handler, more generally, is responsible for understanding the user's input and selecting or creating an appropriate response.

For this chatbot, it was decided to use one of the big publicly available chatbot platforms to provide a dialog handler component, due to the time constraint for this thesis. Chatbot platforms typically make use of modern approaches and allow a faster and easier process to build a dialog handler by using their provided techniques and models, whereas the development of an own agent and own models would be more time-consuming. The following paragraph describes which platform was chosen for this chatbot and what approaches are usable with this platform. Afterwards, the composition of the dialog handler is described in detail and how the used platform fulfils the requirements for the dialog handler.

Google DialogFlow

For this thesis, it was decided to use the chatbot platform provided by Google known as DialogFlow [34] to build the dialog handler. DialogFlow uses an NLP technique to extract information of a user input. To be more precise, for each user input, the intent and entity of the user's input are recognized and extracted. Afterwards, the extracted intent is matched to the intents that were predefined for the chatbot. The response is chosen based on the highest scored matching intent. Dialogflow does so by applying two algorithms; a rule-based grammar matching algorithm and a Machine Learning matching algorithm [34]. Unfortunately, the specific algorithm used is not disclosed by Google, therefore the algorithms used for the user input analysis themselves are a blackbox within our chatbot.

Two other platforms were examined for their usability, called Wit.ai [35] and Amazon Lex [36]. Both platforms also use a NLP approach, same as Google Dialogflow, to extract intent and entity out of a user input. Since there is no core difference in the approach these platforms take, using Dialogflow for the dialog handler is simply a personal preference and could be interchanged with the other two platforms.

User Input Analysis. The dialog handler must be able to understand the natural language input by the user in order to create an appropriate response. However, for this chatbot, the key-information needs to be extracted out of the user input as a text. The reason for this requirement is that the chatbot needs to perform a search query on a database to retrieve a GitHub repository. The key-information must be extracted from the user input to convert it to a query for the database, since a fulltext search of the whole user input would not return accurate results. See figure 4.1 for an example where the key-information of the user input is extracted into an intent and an entity. The user intends to define the programming language of the repository that they are looking for. When the user types in "I want the programming language to be Python", the extracted intent could be "to define programming language" and the extracted entity would be

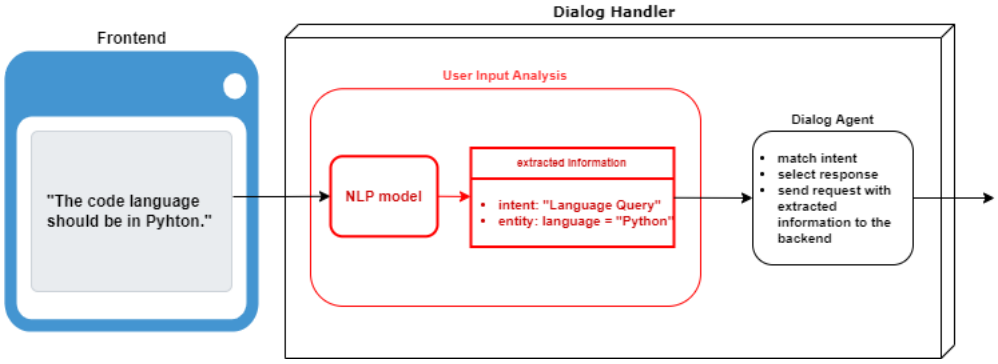

Figure 4.1: An exemplary information extraction of a user input

"Python". Eventually, the extracted entity can be transmitted to a backend where it is converted to a search query that selects all repositories which have defined the programming language as Python.

For the user input handling, the NLP technique to extract entities from the user input that is provided by Dialogflow fulfils the requirement. By predefining the entities (e.g. programming language, numbers, dates), the key-information can be extracted from a user input to perform a query.

Response Generation. Considering the requirements of this chatbot, it shows that all the chatbot responses could be predefined. First, the dialog handler must ask the user all the predefined questions about the repositories. Second, after each question, the number of results that match the user's description so far must be returned. Third, the dialog handler must ask a question again if it did not understand the response of the user. Lastly, the chatbot must return the best matching repository. Therefore, this chatbot must only define a few responses that could be predefined and are to be retrieved depending on the user's input. To be more precise; one response starts the conversation, there is one response predefined per user input intent, one response as a fallback when no intent was recognized and one response when the conversation ends.

Regarding these requirements for the response generation, a generative-based response model is not a practical approach for this use case as this chatbot is not open domain. Therefore, there are two approaches possible. The first possible approach is a rule-based model that matches the extracted intents to patterns, which are predefined intents. The second approach is a retrieval-based response model that selects a predefined response based on the highest scoring match between the input intent and a training corpus.

The retrieval-based response generation provided by Dialogflow [34] is capable of scoring intents and performing actions, as well as returning the predefined response set for the highest scoring intent. Furthermore, Dialogflow provides the usage of context-parameters. Depending on the current state of an interaction, the chance of matching the user's intent can be increased by setting a context-parameter. For example, when the chatbot asks "What language do you want the repository to be in?", the chatbot should already expect an intent "to define programming language" for the next user input. By setting a context-parameter to the intent "to define programming language" after the question has been asked, Dialogflow will more likely recognize this intent for the next user input. To summarize, Dialogflow provides the necessary NLP techniques to handle the user input analysis, as well as the response generation within our requirements.

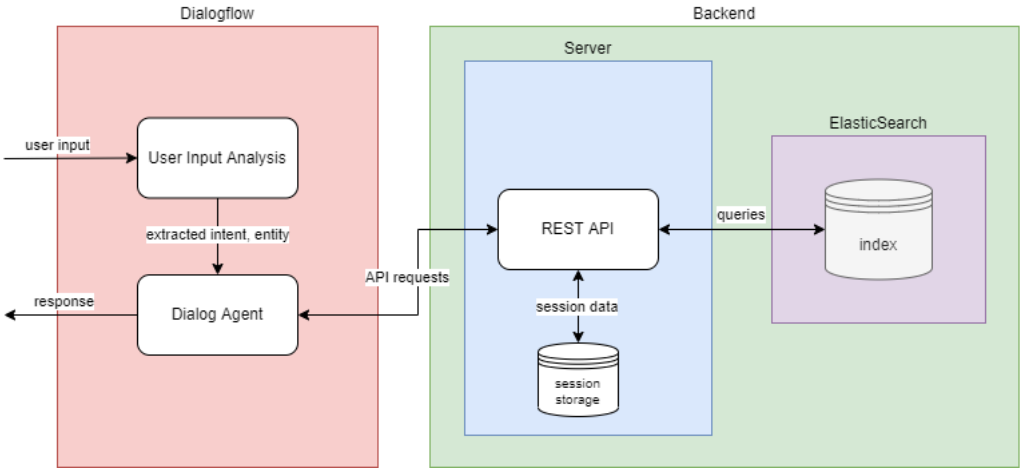

Figure 4.2: A schematic overview of the chatbot architecture

4.2.2 Backend

The second component, the counterpart to the dialog handler, is the backend of this chatbot application. The application's backend part is responsible for imposing logic and data to our chatbot. This subsection is divided into two core components, which is the knowledge base for our search and retrieve functionality, as well as the server to handle the interaction with the knowledge base.

Knowledge Base

Since it is the core functionality of the chatbot to search and retrieve GitHub repositories, performing queries on a data set with GitHub repositories is a given requirement. To be able to perform queries on a data set, the data must be stored in a database and available for information retrieval. Due to the large size of the data available, the database system must be able to efficiently handle queries on large amounts of data.

ElasticSearch. For our chatbot, it was decided to use a state-of-the-art database management system (DBMS) available. To store the data set of GitHub repositories and perform queries on this data set, it was decided to make use of the free and open DBMS called ElasticSearch (ES) [37].
ES "is a distributed, RESTful search and analytics engine capable of addressing a growing number of use cases" [37]. It allows very fast searches on a data set by making queries using a REST API or by installing it on a machine. ES makes use of the Apache Lucene library which is considered a very advanced search-engine library to perform fulltext search queries. The main reason for choosing ES as a DBMS is the architecture of ES that provides very good scalability due to its architecture; ES stores data as documents in indexes, which are comparable to a database table. Each index can be split into multiple shards that can all retrieve data simultaneously once an index is too large to perform well [37].

Server

As a last component of this chatbot, it was decided to run a server that functions as a backend for the chatbot. By providing a REST API, the backend must be open for the dialog handler component to interact with it. One of the core advantages of chatbots is their ability to remember past user inputs and therefore the context of the interaction. This functionality is handled in the backend of our chatbot. This is achieved by creating, updating and deleting data about the conversation in a session storage throughout the conversation. The backend keeps track of the state of the conversation by storing all the questions the chatbot has asked the user, as well as the ones it is still going to ask the user, the search queries that were performed due to the past answers by the user, and the current result of the search query with all the user inputs so far. Apart from the session storage, a core function of the backend is to provide an interface for the dialog handler to trigger search queries on the knowledge base with the information it receives from the dialog handler. Since certain types of knowledge bases expect a certain query structure, the backend must convert the information that were extracted from the dialog handler into a query that is understandable by the knowledge base.

All the requirements for the server so far could be handled directly by the dialog handler as well. The requirement that led to the decision of using a server is that the chatbot is supposed to ask the user intelligent questions. A server between the dialog handler and the knowledge base allows more sophisticated handling of the questions that are asked to the user, by, for example, integrating a trained model that selects a best question. Although this requirement could not be met for this thesis, the usage of a server is still desired for a complete implementation of this chatbot and therefore established as a component of the chatbot.

Figure 4.2 displays a schematic overview of the components interacting with each other.

Chapter 5

Implementation

After the architecture for the chatbot has been designed, this chapter describes the implementation process to deploy the chatbot as a proof of concept in a productive environment. This chapter is split into the backend implementation, the dialog handler implementation, as well as the deployment infrastructure used to host the chatbot.

5.1 Backend Implementation

As defined by the architecture, the backend consists of a knowledge base and a server. This section focuses on the implementation process of providing the data in a knowledge base, as well as the hosting of a server.

5.1.1 Knowledge Base

The knowledge base consists of the data of interest for the chatbot. The implementation was done in two steps; the preparation of the data by downloading, editing and restructuring the data files, and the upload to ElasticSearch afterwards.

Data Preparation

For an information retrieval chatbot, the first step is to define what information is of relevance for the user. Once the information of relevance is defined, the information data needs to be stored so that it is available for retrieval queries by the chatbot.

For this chatbot, there is an interest in retrieving information about GitHub repositories. The interest is mainly on the metadata of GitHub repositories, since the user must be able to search for a GitHub repository based on the metadata that GitHub stores for a repository, while the events and actions (such as commits) are not particularly relevant to find a repository.

To prepare a database, three sources of public GitHub data exports were investigated. Firstly, GitHub has worked together with Google to release a data set of public GitHub repositories on Google BigQuery [38]. The bigquery table is updated regularly and has 2,4 Terabyte of repository code stored [39]. Apart from a large amount of code, it only provides little metadata about the repository such as the size, programming language and the license used for the GitHub repositories.

The second data set that was investigated is the GitHub data from https://www.gharchive.org. This data set is an archive of GitHub event recordings. It records over 20 event types from GitHub repositories and updates the archive hourly [40].

Finally, the open data from libraries.io was considered, which have tracked data from over 30 million GitHub repositories. They provide a set of repository data with more than 30 attributes that represent metadata attributes such as the programming language, a repository description, the number of stars given by users, and a lot more [41]. The drawback to this set is that the last release was from January 2020, therefore there are no recent GitHub repositories stored in this data set.

Eventually, it was decided to use the data set from libraries.io, as they provide the most metadata in a structured CSV in their export, compared to the other two sources investigated. Although this data is not up-to-date, it is possible to enrich the data set with additional repository data after 01.01.2020. The two other sources lacked some minimal fields such as the creation date of the repository or a description field that at least were required. While the amount of code on Google Bigquery is very useful to analyse code itself, it does only provide little value to our repository search chatbot. The data set was therefore not used due to the reason that it lacks structured metadata about repositories. Same goes for the data set from www.gharchive.org, as there is an interest in metadata of repositories, while actions (events) on GitHub are not of interest.

Before the data set could be loaded into our knowledge base, it was pre-processed to make it better usable for our search queries. The data export from libraries.io contains repositories that are not on GitHub, these repositories were removed and only GitHub entries from the data set were kept. Moreover, all attributes that were not of interest to us considering the requirements, as well as not maintained columns were removed from the data set. This was done by first defining which columns to delete and deleting them by their index with the following script 5.1:

```
def rewriter(f_in, f_out):
    inc_f = open(f_in, 'r', encoding="utf8")
    csv_r = csv.reader(inc_f)
    out_f = open(f_out, 'w', encoding="utf8")
    csv_w = csv.writer(out_f, delimiter=',', lineterminator='\n')
    for row in csv_r:
        new_row = [row[1], row[2], row[3], row[4], row[5], row[7],
                   row[9], row[10], row[11], row[12], row[13], row[14],
                   row[15], row[17], row[19], row[20], row[22], row[23],
                   row[31], row[33], row[38]]
        if new_row[0] == 'GitHub' or new_row[0] == 'Host Type':
            csv_w.writerow(new_row)

    inc_f.close()
    out_f.close()
```

Listing 5.1: Data Set Edit

Data Hosting

For our chatbot, it was decided to use a state-of-the-art database management system (DBMS) available. To store the data set and perform queries on this data set, it was decided to use of the free and open DBMS called ElasticSearch (ES). The choice for using ES for our chatbot is mainly the scalability it provides due to the rather large size of our data set. ES also allows a large variety of search queries with the API it provides, which essentially is helpful for the interaction with the dialog handler that is described later in this chapter.

The data set was indexed in an ES index by using a script that reads in the CSV file, converts it to a dictionary, and performs bulk uploads to a locally created index on the current machine that ES runs on. To make sure that the data is correctly uploaded to the index without overusing the RAM of the personal computer where the implementation was developed, the data set was split into multiple CSV files and uploaded one by one to ElasticSearch. See listing 5.2 for the code. Essentially, an index with 36'567'566 data entries was created, each representing a repository.

```
index_name = 'github_repos'

file = "home/ubuntu/GitHub_data/final_new_data_1_edited.csv"

df = pd.read_csv(file, low_memory=False)
df = df.replace({numpy.nan: ""})

documents = [{k: v for k, v in m.items()} for m in df.to_dict(orient='records')]

print("Indexing Start fr File: " + file)

helpers.bulk(es, documents, index=index_name, raise_on_error=True,
         request_timeout=60 * 100)

print("Index finished:" + index_name)
```

Listing 5.2: Data Upload to ES

5.1.2 Server

For the backend server, it was decided to use a Python framework called FastAPI to provide the REST API that interacts with the dialog handler and triggers queries on the ES index. FastAPI is a modern framework to create REST APIs run on Python. It was decided to use FastAPI mainly due to its ease of use and fast development process. The framework is intuitive to use and many big applications such as Netflix or Uber use FastAPI productively to provide their APIs today. [42] The REST API is hosted on a uvicorn server, as recommended on the FastAPI documentation. Uvicorn is an Asynchronous Server Gateway Interface (ASGI) server implementation to pass the requests to the application backend and therefore make the API available for requests [43].

REST API. The REST API provides 4 REST APIs to interact with the application backend:

- POST /sessions: This method creates a new empty session storage with a unique ID
- POST /sessions/id/query: This method sends a query to the backend
- GET /sessions /id/next_question: This method returns the next question
- DELETE /session/token: This method deletes a session storage

Session Storage. For this proof-of-concept implementation, the session storage was not optimized and simply created as a dictionary within our backend. Optimally, a session storage could be integrated as a database or another ES index. The session storage keeps track of information and state of the search interaction between user and chatbot. Each user interaction is stored with a unique session ID. Additionally, the session storage stores the questions that were asked to the

user and are going to be to ask the user, the queries created out of the user answers and the state of the interaction.

Question Handling. The predefined questions that are asked to the user are stored in the session storage at the beginning of the interaction with the user. Each time the dialog handler requests a new question, the backend returns a question and deletes the question from the list of possible questions to ask, to make sure the same question is not asked more than once.

In the requirements for this chatbot, it is defined that the chatbot should ask questions intelligently to narrow down the repository that is of most interest for the user. In the implementation for this thesis, this requirement could not be fulfilled due to a lack of time and skills. More about this can be read in conclusion and future work.

Query Handling. The backend is able to carry out requests on the ElasticSearch index which are triggered by the API requests from the dialog handler. The backend is responsible for converting the information from the user input into a search query that is compliable with ElasticSearch. Each user input is pre-processed by the dialog manager, as described in the next section, and the extracted entity is sent to the backend server. The extracted entity, as well as the data field that the entity refers to, are converted to the structure of an ElasticSearch query and saved to the session storage. After each question, all queries stored so far are executed as one whole ElasticSearch query to retrieve the number of results that match the answers of the user given so far.

5.2 Dialog Handler Implementation

For this chatbot, it was decided to use Google Dialogflow as a chatbot platform since it can fulfil the requirements of this chatbot with the techniques it provides. Chatbot platforms typically make use of modern approaches and allow a faster and easier process to build a dialog handler by using their provided techniques and models, rather than to develop an own agent and own models.

5.2.1 User Input Handling

To extract information out of a user input, each intent needs to be defined and trained with training phrases. For each training phrase that contains an entity, the entity must be annotated in the training phrase for the model to be able to extract these in a user input. Each entity must be defined beforehand as well. After the model has been trained, the model detects intents and entities in the user input with a certain confidence threshold and selects the response of the intent with the highest confidence score. Since the model is not open source, it is not documented how DialogFlow exactly trains their model to recognize intents and entities. A total of 12 intents were defined for our chatbot. The intents are important for the chatbot to understand the meaning of the user input. For each intent, at least 40 training phrases were handcrafted for the DialogFlow training model. Additionally, DialogFlow automatically adds similar training phrases to the handcrafted ones.

Entities are defined for inputs where a specific text or value must be extracted from the user input so it can be converted into an ES search query. For each entity, all possible texts/values must be listed for the chatbot to recognize them in a user input.

Intent	Creator	Intent description
Welcome Intent	Predefined	Response to greetings
Fallback Intent	Predefined	Response when no intent was detected
Start Search Intent	Self defined	Response when the user wants to start a search interaction
FullText Query Intent	Self defined	Response when the user wants to add a fulltext query to the search (to search for keywords).
Language Query Intent	Self defined	Response when the user answers a question about the programming language.
Date Query Intent	Self defined	Response when the user answers a question about the creation date.
Date Query Intent	Self defined	Response when the user answers a question about a last update.
Stars Count Query Intent	Self defined	Response when the user answers a question with a number and the context is set to the "Stars Count" intent.
Contributors Count Query Intent	Self defined	Response when the user answers a question with a number and the context is set to the "Contributors Count" intent.
Yes Query Intent	Self defined	Response when the user answers with a “Yes” to a question
No Query Intent	Self defined	Response when the user answers with a “No” to a question
Irrelevant Intent	Self defined	Response when the user answers with a “don’t matter” intent to a question.

Table 5.1: Dialog Intents

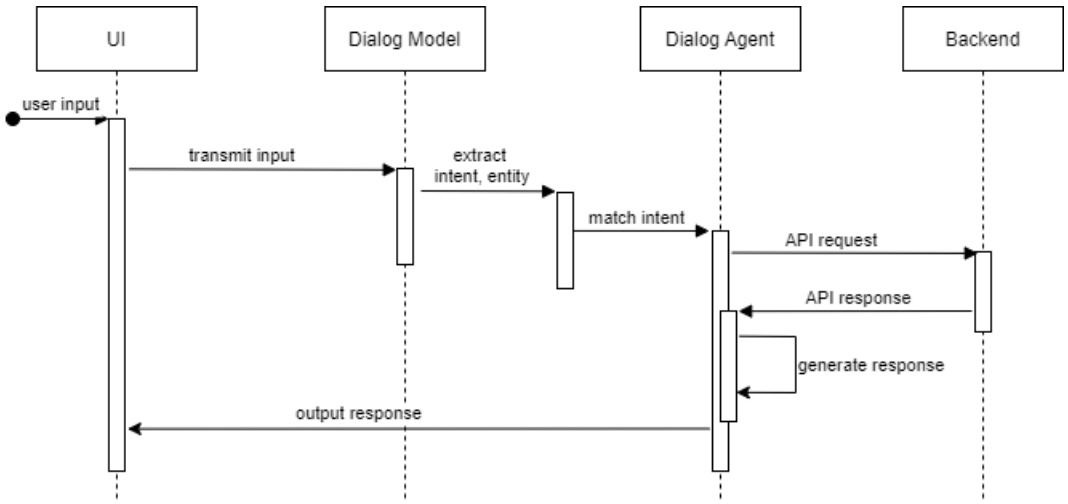

Figure 5.1: The sequence of receiving input and generating a response

5.2.2 Dialog Management

The DialogFlow fulfillment is established with a custom code. With this custom code in Node, a custom DialogFlow agent is provided that handles the actions and responses when an intent is matched. The DialogFlow agent interacts with the backend by making API requests to the backend where information and state about the interaction is stored. To be more precise, once an interaction to search for a repository has begun, the agent requests a question from the backend and returns it to the user. The answer of the user is then analysed by the dialog handler, and the agent receives the extracted intent and entity. Depending on the best matching intent, the agent sends a request to the backend with the extracted entity to create and store the query. The agent then requests the next question from the backend. This goes on until there is one result left that matches the user description or until the user wants to stop. Figure 5.1 shows the interaction flow of how the conversation is handled. For our implementations, due to the fairly low knowledge domain (13 intents), a conversation strategy was used where the dialog agent directs the conversation, so it can expect the input of the user and handle it properly in the backend.

5.3 Deployment Infrastructure

5.3.1 Virtual Machine

The backend, including the ElasticSearch engine and the server, are hosted on a Virtual Machine (VM) on ScienceCloud, a cloud platform provided by University Of Zurich (UZH). Since these components do not depend on much computational power, a single VM with the following specifications was configured:

VM, Image on Ubuntu 20.4, 8vCPUs, 32 GB RAM

ElasticSearch is installed locally on the VM and made available locally only, since only the server is interacting with the ElasticSearch index directly. For the backend server, the API must be configured to be publicly available so the DialogFlow agent can interact with the backend. To do so, the VM is hosted on a private IP address, which is linked to a router that connects it to the public network. Furthermore, the private IP is associated with a public floating IP. With the floating IP, one can connect to the VM from external sources. In our application, only a single port was opened to the public for the API of the backend server, as no other ressources should be exposed. Additionally, the VM can only be accessed with SSH key authentication to secure the access to the VM.

5.3.2 Google Cloud

The dialog handler, which was implemented with Google DialogFlow, is deployed on the Google cloud. It is very easily deployed on their platform with just a single click. The interaction with the user is achieved by integrating the dialog handler to a chatting platform. DialogFlow provides a list of chatting platforms that allow a very easy integration into a chatting platform available for the user. This integration was used to make the chatbot available for a user on TeleGram by simply creating a Bot on Telegram, and then inserting the bot token on the DialogFlow platform to allow the exchange of requests and responses between TeleGram and DialogFlow.

Chapter 6

Preliminary Results

To varify the chatbot's workability, the implemented chatbot was tested with some sample conversations. For every few conversations, the same answers are formulated differently to check if the chatbot consistently understands the user input. The chatbot is not evaluated on its performance due to its incompleteness as a proof-of-concept implementation. If an evaluation was needed, this thesis suggests that one would focus on the user experience and the error rates for a task-oriented chatbot like the one present.

Figure 6.1 displays that the conversation with the chatbot was a success and a repository could be retrieved. The returned result appears to match the description that was given. As long as the answers are expected from the chatbot, the process works sucessfully. Apart from that, the chatbot does often not recognize a user input if formulated in an unusual way. This is a matter of training the chatbot with training phrases, which are all handcrafted and therefore relatively small with around 40 training phrases per intent.

Regarding the user requirements that were established, the chatbot is able to fulfil the formulated requirements, except for the one requirement which states that the questions have to be asked in an intelligent matter. In the current state of the implemented chatbot, this is not handled. The questions are asked randomly, due to complexity and time constraint to establish an intelligent question handling.

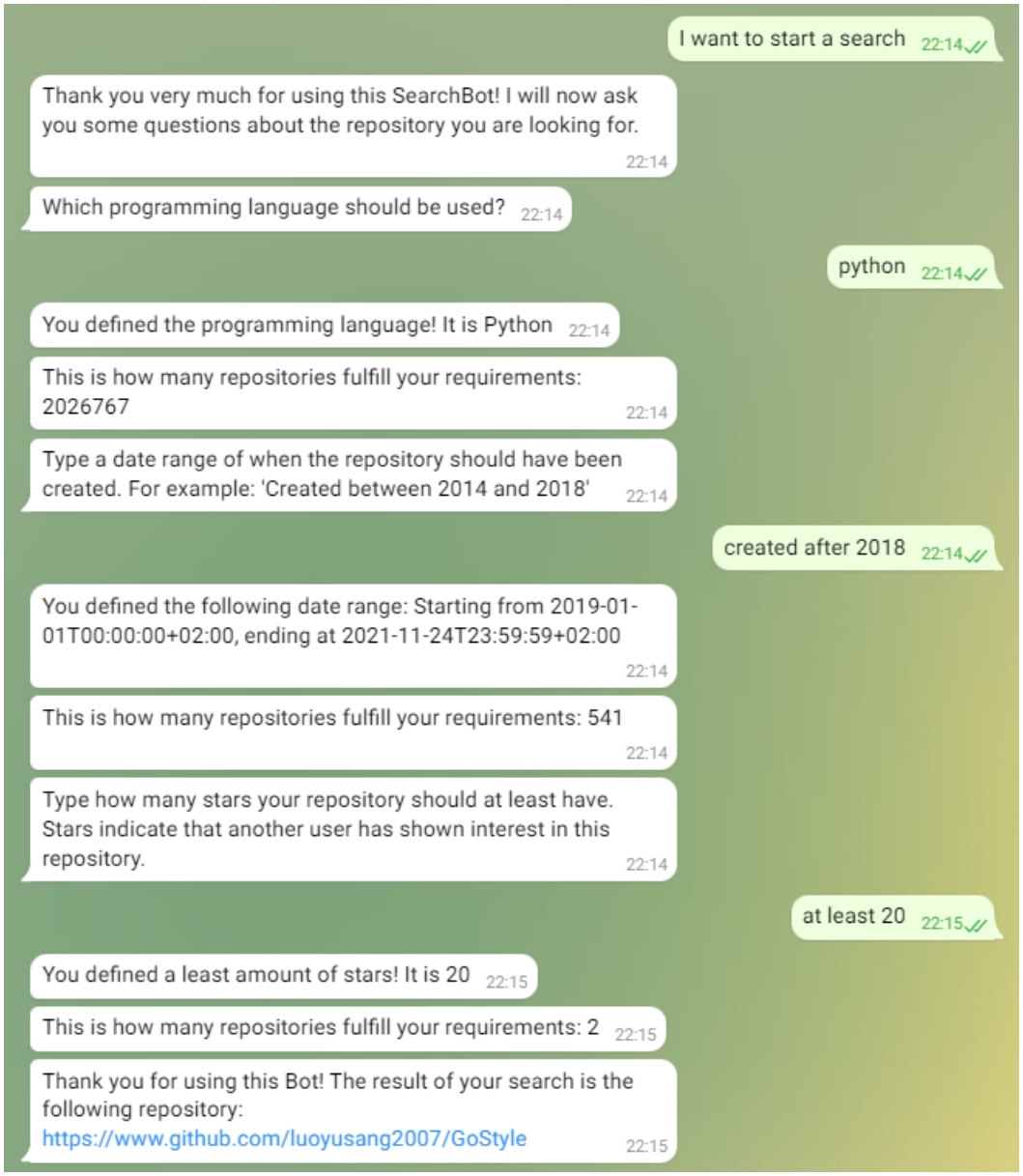

Figure 6.1: A sample search conversation

Chapter 7

Conclusion and Future Work

In this thesis, we have developed a chatbot that is able to create a conversation with a user, with the goal to collaboratively search for a GitHub repository that best matches a project description provided by the user throughout the conversation. The chatbot asks the user predefined questions about metadata from a GitHub repository such as the programming language, the creation date, and so on. The user inputs are analysed by the ML algorithms provided by Google DialogFlow to extract intent and entity of a user input. The entity is sent to a backend API that converts the entity to a search query and stores it in the session storage. This process continues until the number of results for the combined search query with all previous user inputs is fairly low. Finally, some defining questions are asked to the user to retrieve the repository that best matches the user inputs from the ElasticSearch index.

This chatbot provides an easy-to-use interface for a software engineer to find a GitHub repository for software reusage. The chatbot provides a more detailed search and more specific queries on GitHub repositories than currently available. The drawback to this implementation is the lack of up-to-date data available. Recent repositories are not available for retrieval, therefore projects with very novel concepts and artefacts cannot be retrieved by this chatbot. Additionally, the small knowledge domain due to time constraints causes the chatbot to get stuck during a conversation.

Preliminary results have shown that the chatbot is able to conversate with the user in natural language, and can successfully retrieve a GitHub repository that best matches all the user inputs given throughout the conversation. One requirement of asking a user intelligent questions could not be met, since this feature requires a trained model to predict a best question. Developing such a model was not feasible for this thesis due to its complexity and the time available. As this implementation is a proof-of-concept, the chatbot was not further evaluated on its performance. Future work could evaluate the work-ability of the chatbot based on error rates of users using the chatbot or the probability of the chatbot not recognising a user's input.

At last, the implemented solution is not a complete application yet and therefore leaves some room for further work.

Future work. Although the architecture design has shown to be applicable for the implementation of this chatbot, some steps have shown to be challenging to overcome throughout the development and have led to a reduction of the features due to time constraints or high complexity of the implementation. This chapter describes the challenges that have come up during the designing and the implementation process and where future work can further improve this chatbot.

For the designing process of this chatbot, future work could more thoroughly investigate an end-to-end solution for task-oriented chatbots and whether such an approach would be applicable for this type of chatbot. Tsung-Hsien Wen et al. [13] have introduced a new approach for task-oriented chatbots that is end-to-end trainable by integrating the database queries into the

dialog system and generating responses including the retrieved data from the database. This approach might have some advantages compared to our implementation, as less handcrafting is needed for the training phrases with the possibility of using reinforcement learning. Additionally, the answers generated by such a chatbot with generated responses are less static than in our implemented approach where predefined responses were used.

For the implementation process, there are some improvements to be made in future work. A first big challenge that has arisen for the implementation is the data used for the knowledge base. An information retrieval chatbot is only as capable as the information available. Different sources, as described in section 4.1.1, provide GitHub data exports, but the number of exports with good structured data and a large number of repositories is fairly low. Eventually, the libraries.io data set was chosen as the knowledge base, knowing the drawback that there are no repositories created or updated after January 2020 in this data set. In future work, this data set can be extended with more recent GitHub repositories to retrieve more up-to-date projects. One approach to do this would be to scrape repository data from the GitHub API, obviously within the guidelines of GitHub. Another approach to extend the data set would be to merge the data set with file contents stored in the Google BigQuery table provided by GitHub.

The greatest challenge of this chatbot that was not implementable for this thesis is the ability to ask the user "intelligent" questions. One of the core advantages of using a chatbot is the ability to ask questions intelligently to retrieve a single last result rather than zero or multiple results by the end of the conversation with the user. For this thesis, there was not have the time to implement this feature. A possible approach to implement this feature was to calculate which question would maximize the entropy, meaning that a question is chosen that maximizes the chance of the answer filtering out the most repositories.

One general improvement to the chatbot that can be done in future work is to expand the knowledge domain of the chatbot. Our chatbot currently has a small knowledge domain of asking about metadata from GitHub repositories and perform a search query with the inputs. This can possibly be expanded to questions about the code of the repositories as well. This would entail that, apart from the knowledge base that needs to be extended with code, the chatbot must be expanded on the intents and the entities that can be extracted in the dialog handler, as well as the query handling in the backend of the chatbot and the questions to ask the user. Another feature that might be of interest for a software developer would be to directly ask the chatbot questions about the returned repository. This feature could be implemented differently than the task that was implemented so far, since this information is directly retrievable from the GitHub API with the repository name, therefore it need not necessarily be stored in a knowledge base.

Bibliography

[1] Charles W. Krueger. "Software Reuse". In: *ACM Comput. Surv.* 24.2 (1992), pp. 131–183. ISSN: 0360-0300. URL: https://doi.org/10.1145/130844.130856.

[2] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. "An examination of software engineering work practices". In: *CASCON First Decade High Impact Papers*. 2010, pp. 174–188.

[3] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. "How developers search for code: a case study". In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 191–201.

[4] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. "When deep learning met code search". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 964–974.

[5] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep code search". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 933–944.

[6] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. "Retrieval on source code: a neural code search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2018, pp. 31–41.

[7] Ewa Luger and Abigail Sellen. ""Like Having a Really Bad PA": The Gulf between User Expectation and Experience of Conversational Agents". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. CHI '16. San Jose, California, USA: Association for Computing Machinery, 2016, pp. 5286–5297. ISBN: 9781450333627. URL: https://doi.org/10.1145/2858036.2858288.

[8] Menal Dahiya. "A tool of conversation: Chatbot". In: *International Journal of Computer Sciences and Engineering* 5.5 (2017), pp. 158–161.

[9] *GitHub*. URL: https://www.github.com/ (visited on 11/20/2021).

[10] Bhavika R Ranoliya, Nidhi Raghuwanshi, and Sanjay Singh. "Chatbot for university related FAQs". In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE. 2017, pp. 1525–1530.

[11] Jhonny Cerezo, Juraj Kubelka, Romain Robbes, and Alexandre Bergel. "Building an expert recommender chatbot". In: *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*. IEEE. 2019, pp. 59–63.

[12] Ashay Argal, Siddharth Gupta, Ajay Modi, Pratik Pandey, Simon Shim, and Chang Choo. "Intelligent travel chatbot for predictive recommendation in echo platform". In: *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE. 2018, pp. 176–183.

[13] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Lina Maria Rojas-Barahona, Pei-Hao Su, Stefan Ultes, David Vandyke, and Steve J. Young. "A Network-based End-to-End Trainable Task-oriented Dialogue System". In: *CoRR* abs/1604.04562 (2016). arXiv: 1604.04562. URL: http://arxiv.org/abs/1604.04562.

[14] Sameera A Abdul-Kader and John C Woods. "Survey on chatbot design techniques in speech conversation systems". In: *International Journal of Advanced Computer Science and Applications* 6.7 (2015).

[15] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. "Towards a Human-like Open-Domain Chatbot". In: *CoRR* abs/2001.09977 (2020). arXiv: 2001.09977. URL: https://arxiv.org/abs/2001.09977.

[16] Jizhou Huang, Ming Zhou, and Dan Yang. "Extracting Chatbot Knowledge from Online Discussion Forums." In: *IJCAI*. Vol. 7. 2007, pp. 423–428.

[17] Ebtesam H Almansor and Farookh Khadeer Hussain. "Survey on intelligent chatbots: State-of-the-art and future research directions". In: *Conference on Complex, Intelligent, and Software Intensive Systems*. Springer. 2019, pp. 534–543.

[18] Shafquat Hussain, Omid Ameri Sianaki, and Nedal Ababneh. "A survey on conversational agents/chatbots classification and design techniques". In: *Workshops of the International Conference on Advanced Information Networking and Applications*. Springer. 2019, pp. 946–956.

[19] Joseph Weizenbaum. "ELIZA—a Computer Program for the Study of Natural Language Communication between Man and Machine". In: *Commun. ACM* 9.1 (1966), pp. 36–45. ISSN: 0001-0782. URL: https://doi.org/10.1145/365153.365168.

[20] Eleni Adamopoulou and Lefteris Moussiades. "Chatbots: History, technology, and applications". In: *Machine Learning with Applications* 2 (2020), p. 100006. ISSN: 2666-8270. URL: https://www.sciencedirect.com/science/article/pii/S2666827020300062.

[21] Luka Bradeško and Dunja Mladenić. "A survey of chatbot systems through a loebner prize competition". In: *Proceedings of Slovenian language technologies society eighth conference of language technologies*. Institut Jožef Stefan Ljubljana, Slovenia. 2012, pp. 34–37.

[22] Tony Hak and Jan Dul. "Pattern Matching". In: *Erasmus Research Institute of Management (ERIM), ERIM is the joint research institute of the Rotterdam School of Management, Erasmus University and the Erasmus School of Economics (ESE) at Erasmus Uni, Research Paper* (2009).

[23] Bayan AbuShawar and Eric Atwell. "ALICE Chatbot: Trials and Outputs". In: *Computación y Sistemas* 19.4 (2015), pp. 625–632.

[24] Jack Cahn. "CHATBOT: Architecture, design & development". In: *University of Pennsylvania School of Engineering and Applied Science Department of Computer and Information Science* (2017).

[25] Sangkeun Jung. "Semantic vector learning for natural language understanding". In: *Computer Speech & Language* 56 (2019), pp. 130–145. ISSN: 0885-2308. URL: https://www.sciencedirect.com/science/article/pii/S0885230817303595.

[26] Michael Frederick McTear, Zoraida Callejas, and David Griol. *The conversational interface*. Vol. 6. 94. Springer, 2016.

[27] Jiafeng Guo, Gu Xu, Xueqi Cheng, and Hang Li. "Named entity recognition in query". In: *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*. 2009, pp. 267–274.

[28] Zhao Yan, Nan Duan, Junwei Bao, Peng Chen, Ming Zhou, Zhoujun Li, and Jianshe Zhou. "Docchat: An information retrieval approach for chatbot engines using unstructured documents". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 516–525.

[29] Yu Wu, Zhoujun Li, Wei Wu, and Ming Zhou. "Response selection with topic clues for retrieval-based chatbots". In: *Neurocomputing* 316 (2018), pp. 251–261. ISSN: 0925-2312. URL: https://www.sciencedirect.com/science/article/pii/S0925231218309093.

[30] Alan Ritter, Colin Cherry, and William B Dolan. "Data-driven response generation in social media". In: *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*. 2011, pp. 583–593.

[31] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[32] Zhou Yu, Ziyu Xu, Alan W Black, and Alexander Rudnicky. "Strategy and policy learning for non-task-oriented conversational systems". In: *Proceedings of the 17th annual meeting of the special interest group on discourse and dialogue*. 2016, pp. 404–412.

[33] *GitHub: Licensing a repository*. URL: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository#choosing-the-right-license (visited on 11/20/2021).

[34] *Google Dialogflow Training*. URL: https://cloud.google.com/dialogflow/es/docs/training (visited on 11/20/2021).

[35] *Wit.ai*. URL: https://https://wit.ai/ (visited on 11/20/2021).

[36] *Amazon Lex*. URL: https://aws.amazon.com/de/lex/ (visited on 11/20/2021).

[37] *elastic*. URL: https://www.elastic.co/ (visited on 11/20/2021).

[38] Stephanie Wills. *GitHub data, ready for you to explore with BigQuery*. URL: https://github.blog/2017-01-19-github-data-ready-for-you-to-explore-with-bigquery/ (visited on 11/20/2021).

[39] *BigQuery*. URL: console.cloud.google.com/bigquery (visited on 11/20/2021).

[40] *GHArchive*. URL: https://www.gharchive.org/ (visited on 11/20/2021).

[41] *libraries.io*. URL: https://libraries.io/data (visited on 11/20/2021).

[42] *FastAPI*. URL: https://fastapi.tiangolo.com/ (visited on 11/20/2021).

[43] *uvicorn*. URL: https://www.uvicorn.org/ (visited on 11/20/2021).