

Master Thesis

October 26, 2021

CodexBot

A conversational agent suggesting code examples

Seungwoo Han

of Daejeon, South Korea (14-333-363)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza



University of
Zurich^{UZH}



Master Thesis

CodexBot

A conversational agent suggesting code examples

Seungwoo Han



University of
Zurich^{UZH}



Master Thesis

Author: Seungwoo Han, seungwoo.han@uzh.ch

URL: N.A.

Project period: 01.May.2021 - 31.October.2021

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

To my parents.

Abstract

Finding the right code example fast and easy can help developers to have better performance. However, the current way of searching code examples such as web search has several drawbacks. First, the massive amount of information about the source code on the web requires a considerable amount of time to preprocess them to use. Furthermore, the repeated search activity will hinder their performance since it breaks the flow of work. In order to resolve these downsides, this master thesis proposes a chatbot, a conversational agent, for searching the right code example.

By using a chatbot, rather than searching via the web, developers can have several advantages. First of all, they can directly check the relevant code example without any accessory information. This will help them to stay focused on their work. Second, it removes the time required to preprocess the information when they use the web search. Lastly, the communication component of the chatbot can guide developers to make a good query in a step-by-step manner, since it is not easy to make a good query in one attempt. Thus, they can have a more relevant code example in the end.

Implementation of our chatbot is based on three major components: Elasticsearch, FastAPI, and DialogFlow. Elasticsearch is a very powerful search engine; FastAPI is a modern way of creating API in Python; DialogFlow is a tool for creating chatbot frames, pre-trained by Google, with very easy integration with other popular chat applications.

Even though it is a preliminary result, the performance of our chatbot is promising in retrieving correct search output in a single-round conversation. In order to evaluate this, we create mock queries from the docstring in our dataset. The result shows that over 60% search accuracy for the tf-idf based mock queries and over 30% search accuracy for the mock queries created from the randomly selected words. Last but not least, we also propose the possible experiment design to test our chatbot in how well it can guide the developers to end up finding the right code example in the multiple-round conversations.

Zusammenfassung

Das schnelle und einfache Finden des richtigen Codebeispiels kann die Leistung der Entwickler verbessern. Die derzeitige Methode zum Durchsuchen von Codebeispielen wie die Websuche hat jedoch mehrere Nachteile. Erstens erfordert die enorme Menge an Informationen über den Quellcode im Web eine beträchtliche Zeit, um sie für die Verwendung aufzubereiten. Darüber hinaus wird die wiederholte Suchaktivität ihre Leistung beeinträchtigen, da sie den Arbeitsfluss unterbricht. Um diese Nachteile zu beheben, schlägt diese Masterarbeit einen Chatbot, einen Conversational Agent vor, um das richtige Codebeispiel zu suchen.

Die Implementierung eines Chatbots anstelle einer Websuche stellt für den Entwickler mehrere Vorteile. Zunächst können sie das jeweilige Codebeispiel ohne unnötigen Zusatzinformationen direkt überprüfen und die Entwickler verlieren dabei nicht den Hauptfokus der Arbeit. Zweitens entfällt die Zeit, die für die Vorverarbeitung der Informationen erforderlich ist, wenn sie die Websuche verwenden. Schließlich kann die Kommunikationskomponente des Chatbots Entwickler Schritt für Schritt dabei unterstützen, eine gute Anfrage zu stellen, da es schwierig ist eine gute Anfrage beim ersten Versuch zu erstellen. Somit kann mit der Verwendung des Chatbots am Ende ein relevanteres Codebeispiel gefunden werden.

Die Implementierung unseres Chatbots basiert auf drei Hauptkomponenten: Elasticsearch, FastAPI und DialogFlow. Elasticsearch ist eine sehr leistungsfähige Suchmaschine; FastAPI ist eine moderne Methode zum Erstellen von APIs in Python; DialogFlow ist ein von Google bereitgestelltes Tool zum Erstellen von Chatbot-Frames mit sehr einfacher Integration in andere beliebte Chat-Anwendungen.

Obwohl es sich um ein vorläufiges Ergebnis handelt, ist die Leistung unseres Chatbots vielversprechend, um in einer einzigen Gesprächsrunde die richtige Suchausgabe abzurufen. Um dies auszuwerten, erstellen wir Mock-Queries aus dem Docstring in unserem Dataset. Das Ergebnis zeigt eine Suchgenauigkeit von über 60% für die tf-idf-basierten Mock-Queries und eine Suchgenauigkeit von über 30% für die Mock-Queries, die aus den zufällig ausgewählten Wörtern erstellt wurden. Zu guter Letzt schlagen wir auch das mögliche Experimentdesign vor, um unseren Chatbot zu testen, wie gut der chatbot die Entwickler beim Finden des richtigen Codebeispiels in mehreren Gesprächsrunden lenken kann.

Contents

1	Introduction	1
2	Background and Requirements	3
2.1	Chatbot	3
2.1.1	Requirements of our chatbot	3
2.2	Elasticsearch	4
2.2.1	Reason to choose Elasticsearch	5
2.2.2	Score in Elasticsearch	5
2.3	FastAPI	6
2.3.1	Reason to choose FastAPI	6
2.4	DialogFlow	6
2.4.1	Reason to choose DialogFlow	7
2.5	Finite state machine	7
3	Related Work	9
3.1	Code search	9
3.2	Chatbots for information retrieval	10
4	Chatbot Design and Implementation	11
4.1	How it works	11
4.1.1	Start the chat	11
4.1.2	Run initial search	11
4.1.3	Go to filter	12
4.1.4	Go to extend	12
4.1.5	Get keywords	13
4.1.6	Get final result	14
4.2	Detailed implementation	15
4.2.1	Elasticsearch	15
4.2.2	DialogFlow	19
4.2.3	Database connection	24
4.2.4	Threshold	25
5	Preliminary Results	27
5.1	Single-round conversation	27
5.1.1	Evaluation results	28
5.2	Multiple-round conversation	29
5.2.1	Evaluation design	29

5.2.2	Preliminary results	29
5.3	Future experiment for the proper evaluation	30
6	Conclusion and Future Work	33
6.1	Future work	33

List of Figures

4.1	Diagram showing how three main components are connected	12
4.2	Finite state diagram showing how our chatbot behaves	13
4.3	Start the chat	14
4.4	Ask initial query to the user and get a response	14
4.5	Ask the user about this answer is a right one	15
4.6	Ask user more information	15
4.7	Return the search result from filtering	16
4.8	Ask the user a new query to extend the query	16
4.9	User gets candidate key words	17
4.10	User found the right code example	17
4.11	Schema for user and query history	18
4.12	Schema for chat session information	18
4.13	List of intents of our chatbot	19
4.14	Training phrases for particular intents	20
4.15	Enabling a fulfillment	20
4.16	Fulfillment request from a DialogFlow	21

List of Tables

5.1	Evaluation result of Sachdev et al.	28
5.2	Evaluation result of our chatbot.	28
5.3	Percentage of asking more information from the tf-idf query that chatbot did not give correct answer.	30

List of Listings

2.1	The standard score function for Elasticsearch	5
4.1	Function for the initial search	15
4.2	Function for the filtering	17
4.3	Back-end programming when intent is give answer or extend	20
4.4	Back-end programming when intent is filtering	22
4.5	Back-end programming when intent is satisfy	23
4.6	Asking users if they like to have key words suggestion	23
4.7	Creating schemas for database	24
4.8	Connecting to database by SQL alchemy	25
4.9	Updating a threshold based on the user response	25

Introduction

Programming is an essential part of many areas [1]. Particularly, recent success in machine learning and artificial intelligence (AI) leads to increasing programming activities in many fields to enhance their productivity. While programming, developers search code examples many times [2]. There are two main reasons why developers do code searches. First, writing codes requires many memories in syntax. It is simply not possible to remember all syntax for every programming language. Second, having relevant code examples from the beginning can make developers complete their code much easier and faster than starting to write codes from the scratch.

Finding relevant code examples is not an effortless task [3, 4]. A good search result requires a good query, and this needs experience [5, 6]. Furthermore, if developers use web search for the code example, then it gives the number of search results a lot more than they need [7, 8]. Additionally, search outputs contain both source codes and other comments that can result in noise. Thus, developers have to go through the whole output list to filter out the noise.

In order to help this search activity, researchers explored the use of AI [9–13]. Specifically, by using advanced techniques in the natural language understanding [14–17], researchers try to find a way to retrieve the target source code examples based on the natural language query. And these endeavors have been tried with various forms of AI to assist the developers to find the right code examples in an efficient way [18].

Firstly, researchers used an AI technique without any mediating agent to find the most relevant code example based on the natural language query [9, 11, 19, 20]. The main motivation for this approach is that when developers give not-very-precise but still informative-enough query then AI can find relevant code examples. Thus, it can make developers less restricted from writing a very precise query which can be time-consuming.

However, this way still has several problems. The first problem is it shows noisy search results. Google search is a good example. It is very good at understanding the intent from the natural language query. Thus, it will return what we are looking for when it receives the query which is not written very precisely. But the number of search outputs is immense, thus we need to do filtering. Furthermore, Google shows the search result whenever it receives the query. In this case, if the input query is too general then it will return results that are not what the user is looking for.

By using the chatbot as a mediator for the code search task [21–25], we can resolve the above drawbacks while keeping the advantages. A chatbot is a conversational agent or bot which is able to perform some kind of task or give answer to the query from the user. Using a chatbot for code search tasks is particularly beneficial for the following reasons:

- a chatbot can return the search result without the noise since it only shows the most relevant answer directly to the developer. For instance, the chatbot returns the top five code examples which are the most relevant based on its evaluation metric. Moreover, outputs do not contain any unnecessary comments and the number of outputs is also manageable;

- when developers do programming it is very important to have as least distraction as possible. By providing only the necessary information the chatbot can offer a more focused and continuous environment for doing programming;
- the conversation component of a chatbot can give a step-by-step process for searching the most relevant code example. That means when the developer gives a very uninformative query at the beginning, the chatbot can ask the developer for more information to make the query more informative. In a web search case, it will return everything whenever it receives any query regardless of the quality. This is not desirable since it is very unlikely that the output list contains the result users want. In this case, it is better to ask for more information instead of showing irrelevant results.

This thesis is aimed at designing and implementing a code search chatbot that is able to offer the benefits mentioned above. The thesis is structured as follows. Chapter 2 discusses the necessary requirements of our chatbot and background knowledge to understand the concepts we use. In Chapter 3, we talk about the related works for this thesis. Then, in Chapter 4, we explain the implementation part of our chatbot. Here, not only do we give a high-level overview to explain how our chatbot works, but also we give detailed implementation including code snippets. In Chapter 5, we show the performance of our chatbot and compare it with a previous approach from other research. We finalize this thesis with the conclusion and mention the possible future work.

Background and Requirements

In this chapter, first, we explain more about what a chatbot is, then discuss the main requirements of our chatbot. This gives the answers to the questions about why we decide to use Elasticsearch, FastAPI, and DialogFlow. Also, we discuss each of their background information and terminologies.

2.1 Chatbot

A chatbot is a computer program, which responds like a smart entity when conversed with through text or voice and understands one or more human languages by Natural Language Processing (NLP) [26]. We can categorize chatbot by two based on how it is built. First is rule based chatbot and second is deep learning based one.

Rule based chatbot. The rule based chatbot is a bot which behaves based on the predefined rules [27]. A rule based chatbot cannot be used for the general purpose since the number of required rules is almost infinite. Thus we usually make a rule based chatbot for a specific purpose. For example, a chatbot for making a reservation in a hotel or hair salon can perform almost like a human. And customers even cannot recognize that they had a chat with the bot. Thus we will focus on creating a chatbot in the specific use case, code search, in this thesis.

Deep learning based chatbot. The fundamental technology behind the deep learning based chatbot is natural language processing which allows machines to understand human language by representing it into mathematical space called embedding space [28–31]. By using this technique, the machine not only can represent natural language sentences in a machine-understandable way but also can capture the synonyms and antonyms by measuring the distance in the embedding space. With help of the most advanced technology in deep learning, now deep learning based chatbots can produce human-like sentences even on open topics. For example, GPT-3 developed by OpenAI has billions of parameters to understand and produce human-like sentences. [32,33]. But the main downside of this type of bot is that it requires a gigantic amount of data to train the model.

2.1.1 Requirements of our chatbot

The main purpose of our chatbot is to find a relevant code example for the given natural language query. Recalling the discussion from the introduction, we aim our chatbot to assist developers in

two ways. First, it frees the developers from the time-consuming noise filtering process. Second, it communicates with developers to build a better query, thus returning better search results. For example, it asks developers for more information when it receives a query with bad quality, instead of returning irrelevant results. This will guide developers to end up with a high-quality query so the chance our chatbot returns the right answer increases. Additionally, our chatbot must be able to perform a task quickly. Otherwise, users cannot have an experience of real conversion like they do with humans. In sum, here are the two most critical requirements for our chatbot.

Developers (users of our chatbot) should be able to get a target code example fast without noise. First of all, our chatbot should be able to find the code examples the users want. And the speed of retrieving the search results from the data to the user is very critical in our case since we do not want to break the workflow of developers when they use our chatbot. If developers feel that our chatbot response time is slower than a real conversation with humans then they can be distracted more easily. Furthermore, the output from our chatbot should not contain any unnecessary noises. Thus, our users do not need to do a time-consuming filtering process to find the answer.

Chatbot should be able to find the most relevant code example in a communicating way. In order to get the most relevant code examples, we need to give a chatbot a good query. A good query is one that contains precise and rich information about what the user wants. For many developers, writing a good query in one attempt can be demanding. In order to help this, our chatbot should be able to guide users by asking for more information if it is necessary to find better search results.

2.2 Elasticsearch

Elasticsearch is a real time search and analytic engine [34]. The data in the Elasticsearch is stored in something called Node which is a single server storing data and indexing data injected. Each Node is linked with other Nodes, creating so called Cluster. This is a collection of Nodes that together hold the entire data. Here are some terminology used in Elasticsearch.

Index. An index indicates where a collection of documents with similar characteristics are, and is identified by its own name. This name is used to refer to the index while performing indexing, search, updating, and deleting. So the index is not storing documents which keeps tracking where the documents are stored.

Type. A type is a category or partitioning of an index. It is defined for the documents that have a set of common fields. We can define more than one type for the index.

Documents. A document is a basic unit of information which can be indexed. This information is stored in JSON format which is widely used in API and web applications.

Shards. A shard is where the data is stored. Elasticsearch provides the ability to subdivide the index into the multiple pieces called shards. Sharding is a key component for speed in Elasticsearch since we can run a search in multiple shard at the same time in parallel.

Replica. A Replica is simply a identical copy of shards, our information or data. It also improve the performance of the search. As the number of search query is increased, Elasticsearch can also use replica on the top of shard for search.

2.2.1 Reason to choose Elasticsearch

Elasticsearch gives a search result fast. Because of its unique architecture, Elasticsearch offers very fast and scalable search performance. The most interesting and unique point of Elasticsearch is that it supports full-text search, which means, it does not use any schemas or tables for the data search. Instead, it uses only documents to search. This gives the main advantage of Elasticsearch, a search speed. It gives almost real-time search performance once we inject the data as JSON format into Elasticsearch.

Elasticsearch offers a filtering functionality. As we discussed in the requirements of our chatbot, it should be able to communicate with users in multiple rounds until it receives the necessary information to find a correct search result. Elasticsearch is a perfect candidate to use to fulfill this requirement since it has the functionality of filtering. What filtering does is that it allows us to do an additional search on top of the initial search results. Thus, it filters search results so that the final output becomes more relevant to the given query. When we use filtering will be determined based on the score given by Elasticsearch. A more detailed explanation for the score will be discussed in the threshold section.

2.2.2 Score in Elasticsearch

In order to show the most relevant search result to the user, Elasticsearch ranks the results from the query based on its internal score and shows top n results. The main base of the scoring function is a tf-idf [35], term frequency, and inverse document frequency. This is a very popular method in a basic approach in natural language processing to figure out how much one specific word is relevant to a particular topic we want to search. Since these keywords are something that occurs very frequently only in that particular documents and less frequent in another document with a different topic, inverse document frequency will go up for such keywords. The below function shows the whole look of the score function which takes two inputs, user query, and target documents [36].

```
score(query, document)=
    queryNorm(query) *
    coord(query, document) *
    SUM(
        tf(term in document),
        idf(term^2),
        term.getBoost(),
        norm(term, document)
    )(term in query)
```

Listing 2.1: The standard score function for Elasticsearch

Here is a brief explanation for each component of the score function. Please note that all these measures are for tuning and adjusting the fact that relevance is subjective. That means, how much the search result is relevant to the query depends on the query itself. And this score is used as the default score in Elasticsearch.

queryNorm. This is a query normalization to compare the relevancy across the different queries. Typically normalization is done by the sum of squared weight for each term in a query.

coord. coord is a coordination factor that counts the number of terms in the query that appear in the document. Thus a document that contains more terms from the query will have a higher score.

norm. This is an inverse of the square root of the total number of terms in the document. So this can be viewed as the document length normalization.

getBoost(). This is a boost that is applied to the query. Boost makes particular parts in a query more important by giving them more weight.

2.3 FastAPI

FastAPI is a modern high performed way to produce an API in Python 3.6+. As its name implies, FastAPI allows developers to create API very fast. The special thing about FastAPI in terms of coding is it requires a type of variables in the code, so the program automatically raises an error when the input type does not match with the required type. In this way, developers can code 200 to 300% faster if they use FastAPI [37].

FastAPI uses a special way to create JSON format by using pydantic model. Using pydantic model, FastAPI offers the BaseModel class which we can inherit. By simply stating the type and variable names, we can create any type of JSON format. Since JSON format is used in web applications dominantly, using FastAPI can be a great tool for API development.

FastAPI offers a very convenient tool for documentation for API. It automatically creates API documents for created methods in real-time. Furthermore, the document also offers the functionality of testing these methods. In the document, a developer can check not only the required input and output type but also the relevant error message.

2.3.1 Reason to choose FastAPI

Fast and efficient multiple operations handling. FastAPI offers asynchronous programming, async for short. The benefit of using the async function is that the program can handle multiple operations much more efficiently than a normal function. With the async function FastAPI makes the function can handle many I/O operations at once without any waiting, thus the program can run much faster.

2.4 DialogFlow

Developed by Google, DialogFlow is a very convenient tool for creating a chatbot [38]. DialogFlow has two key features which are used for our chatbot.

Intents. Intents in DialogFlow can be seen as states in the finite state machine. A chatbot can capture the intent of the user and give a corresponding action. This can be done by additional training of a chatbot in DialogFlow. In the beginning, the chatbot in DialogFlow is only trained to understand a general expression. But it does not understand the intent of the user. Thus,

by additionally training our chatbot for specific phrases, the chatbot becomes to understand the particular intents.

For example, let us say we want to create a “ask query” intent. And we know this intent is started by asking something like “How to”. Then we can train our bot for that phrase to let the bot know that this phrase is for the ask query intent. Then bot can do the following response such as giving an answer to the query. To set response for particular intent we can simply add response through DialogFlow or can use back-end development for this. In this thesis, we used both ways.

Fulfillment. In order to do back-end development, we need to use a fulfillment feature. It is just a link to your URL to use the webhook functionality offered by DialogFlow. And for each intent which we want to use a back-end development, we need to enable a fulfillment option. Then we can check the output from the back-end in DialogFlow directly or other integrated systems such as Telegram in our case. Once we enable the fulfillment option we can check a fulfillment request with JSON format. This contains all relevant information about the conversation between user and bot, including query history, session id, response id, intent category, and more. So it is very convenient to get all necessary information from this JSON file and we can store it in our database. And we can use the stored data for future use if we want.

2.4.1 Reason to choose DialogFlow

Already pre-trained. Based on the large dataset from Google, DialogFlow is already pre-trained for the basic natural language understanding part. Thus, we do not need to worry about how to get trainable data with correct word embedding.

Easy integration with other applications. DialogFlow has many integration options such as Viber, Telegram, Twitter, and many more. In this thesis Telegram is the one integrated with DialogFlow. Thus, users of our chatbot can use it through their smartphones.

2.5 Finite state machine

The finite state machine is an abstract mathematical model, which can be applied when we have a discrete number of states. And we can be in one state at any given time [39–43]. It is typically described by a graph with nodes representing a state and edges representing a transition between states. In our use case, a state or event can be the intent of our chatbot created from a DialogFlow. And transition happens based on the user’s response from the result given by the chatbot.

The main advantage of using a finite state machine is it simplifies a situation we need to handle. For example, suppose we want to make a program handling a situation that varies a lot. If we do hard code to cover every possible single event then we have to make countless if-else statements to cover them. Thus program becomes very error-prone. However, if we use a finite state machine then we can put similar events into the same state by creating a virtual class. In this way we do not need to handle a single event separately but can group them so that we end up with much smaller states we need to handle.

Within a software development process, representing a finite state machine is usually the first high-level step to make a blueprint to solve the problem. This representation is called a finite state diagram. Once we build this diagram, it can be directly converted to the code in a much less error-prone way.

Related Work

In the following, the related work for this thesis is presented. It is divided into two main aspects: code search problem, and chatbots for information retrieval. The code search problem is about searching a target source code by using machine learning without any mediating agent (chatbot). And chatbot for information retrieval part is about using a chatbot as a mediating agent to extract the target information by using a rule-based approach.

3.1 Code search

Sachdev et al. [19] tested a neural code search based on the natural language processing. In order to capture a semantic of source code, they used a word vector representation, word embedding, of each code fragment. Given a natural language query from the user they mapped it to the same embedding space of source code. And they compared the distance between the two to pick the most relevant source code for that particular query.

Sachdev et al. [19] invented a subset words retrieval metric. This metric works as follow. First, they extracted 20% of words from the source code, and use it as a query input. This can be done in a randomized way or based on the tf-idf score. Then, they checked the percentage of correct source code returned from the query. They used two ways to compute the percentage. One is the percentage of correct source code returned in the first place. Another is to use source code returned from the top 9 scores. Since this metric only requires the source code to evaluate the bot, we decide to use this to evaluate our chatbot performance for the single-round conversation.

Husain et al. [44] created CodeSearchNet to provide the data of source code and corresponding natural language query in code search activity. This dataset contains 6 different programming languages and about 2 million functions. The main purpose of this dataset is to provide the performance evaluation of code search. They also proposed the baseline model with a joint vector representation of source code and natural language. We used the data offered by CodeSearchNet. That means, we injected this data into Elasticsearch.

Salza et al. [13] used transfer learning to solve a code search problem. They used a pre-trained BERT, a deep learning framework, to perform a code search. Their model is pre-trained on the combination of natural language and source code, and this pre-training gives an advantage of transfer learning. Their research showed that model which is pre-trained with more data outperformed when it is used in a specific information retrieval task such as code search. And they also showed that transfer learning can be more effective when there is a lack of data for tuning the pre-trained model. Since this paper demonstrates the code search by using transfer learning, it shows the possible future modification to our current rule-based chatbot.

We also can use deep learning-based search technique, but Patil et al. [45] showed that the performances among machine learning, artificial intelligence, and Elasticsearch showed that Elastic-

search demonstrates sufficient performance in a search task in the real estate domain. And using Elasticsearch reduces the amount of work in training compared with deep learning. Therefore, we decide to use Elasticsearch for this thesis.

3.2 Chatbots for information retrieval

Shawar et al. [46] used a chatbot by retraining ALICE, a chatbot system inspired by ELIZA, in the school website for FAQ. They used a dataset of FAQ from their website and they extracted the most important keywords from the FAQ. Then they performed rule-based pattern matching to find a corresponding answer to that FAQ from the dataset. Matching can be done as a whole sentence or two most significant keywords. The result of this paper showed that more people are able to find the correct answer when they used a chatbot than when they used Google. This result supports our decision to use a chatbot as a mediating agent in a code example search.

Yi et al. [47] used a finite state machine in the chatbot. By using a finite state machine instead of deep learning-based methods they can make a chatbot very light and efficient. In order to use a finite state machine they first grouped the topics in similar ones, thus creating a state. The performance showed that the users satisfy with their bot as much as the bot based on deep learning. We also used a finite state machine by grouping the intents of our users. The only difference is that our target information is source code examples. Additionally, the evaluation method from Yi et al. [47] gives us the possible experiment design set up for testing our chatbot in the multi-round conversation. A more detailed discussion is in the future work chapter.

Chatbot Design and Implementation

In this chapter, we discuss how we designed and implemented our chatbot. We give a high-level overview and diagram to show how our chatbot works. Also, we show some details of our implementation.

4.1 How it works

In this section, we describe a high-level overview of how our chatbot works. In general, our chatbot consists of three main components which are FastAPI, Elasticsearch, and bot (DialogFlow). Before diving into the explanation, figure 4.1 shows the big frameworks about how we connect the three main components. Based on this workflow, we used a finite state machine to design how our API uses Elasticsearch as a backend. Figure 4.2 shows how our chatbot behaves depending on which state it belongs to. Here is how our chatbot is working based on the states it faces.

4.1.1 Start the chat

Chat is started by the user saying “Hi” to the bot (Figure 4.3). Then the bot will give two options, one is to ask a query and another is just to end the chat right away. These are given as buttons so users can just click what they want. Of course, users can start a chat right away by writing a query.

4.1.2 Run initial search

Once the user clicked the option of asking a query our chatbot asks the user to input an initial query by starting with “How to” (Figure 4.4). This is important to let our chatbot and DialogFlow know that this chat belongs to the state (intent in DialogFlow) for asking an initial query. So our chatbot can perform a corresponding action via FastAPI.

After running an initial search via Elasticsearch if the score from this search is higher than the threshold, then our chatbot returns the output to the user. How we set this threshold will be discussed in the section threshold. Then the bot asks the follow-up question to check whether this answer is what the user is looking for (Figure 4.5).

If the user chooses “Yes”, then chat is over, and the user can ask another query if he has. If they choose “No”, then we move to another state, extend the query. This part is not shown in our diagram above to avoid making our diagram too complicated.

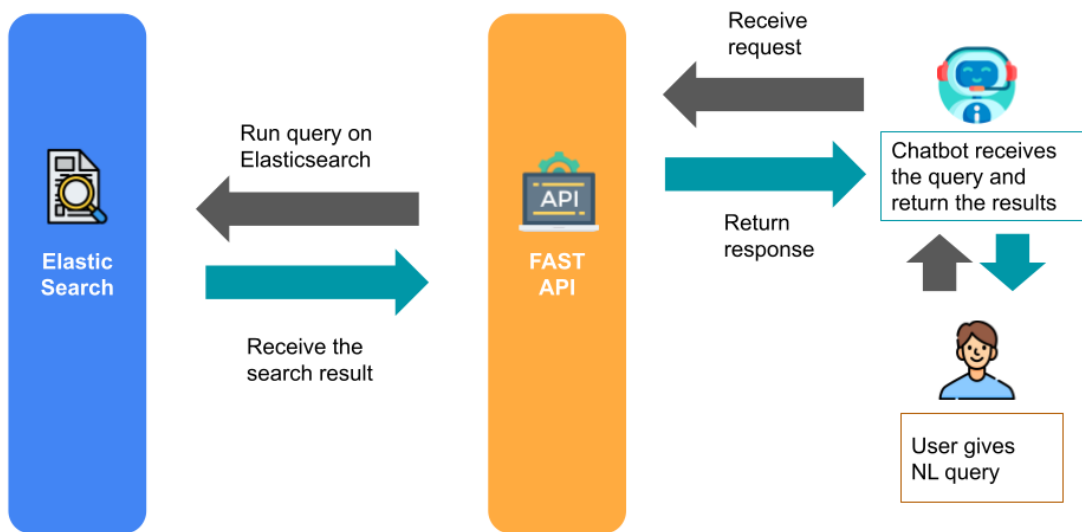


Figure 4.1: Diagram showing how three main components are connected

4.1.3 Go to filter

The score from the first search can be below the threshold when the user gives a rough query. In this case, our bot will ask users for more information instead of returning irrelevant search results (Figure 4.6).

In this case, the user's initial query returns too many results since it is too broad. And a score of outputs is also below the threshold. Then the user needs to provide more information starting with "Add: " to the bot. This is again important to let our bot know that this belongs to the filtering intent (or state). Thus, FastAPI can run the correct filtering code for this. The optimal value of the threshold is very important, since if it is too high then users will not see any result from their query but keep providing more information until the score exceeds the threshold.

Once the score after adding new information becomes higher than the threshold, our bot will return the search result (Figure 4.7). And bot will ask users that this is what they are looking for. From there it is the same as the initial query case.

4.1.4 Go to extend

It is possible that the developers are not able to make a very informative and precise query from the beginning. Thus, it would be nice to offer a way to rephrase their query if the initial query is too off from what they are really looking for.

When the user does not like the search result returned by our chatbot from the filtering, our chatbot moves to the extend state (Figure 4.8). If the score is below the threshold then our bot will ask for more information to filter the search result and it will return an output if found.

Here, again adding "New: " in front of the query is a key to let our bot and DialogFlow know that now it is time to move extend state or intent. By using DialogFlow it is very easy to manipulate all states and transitions based on the natural language response from the users.

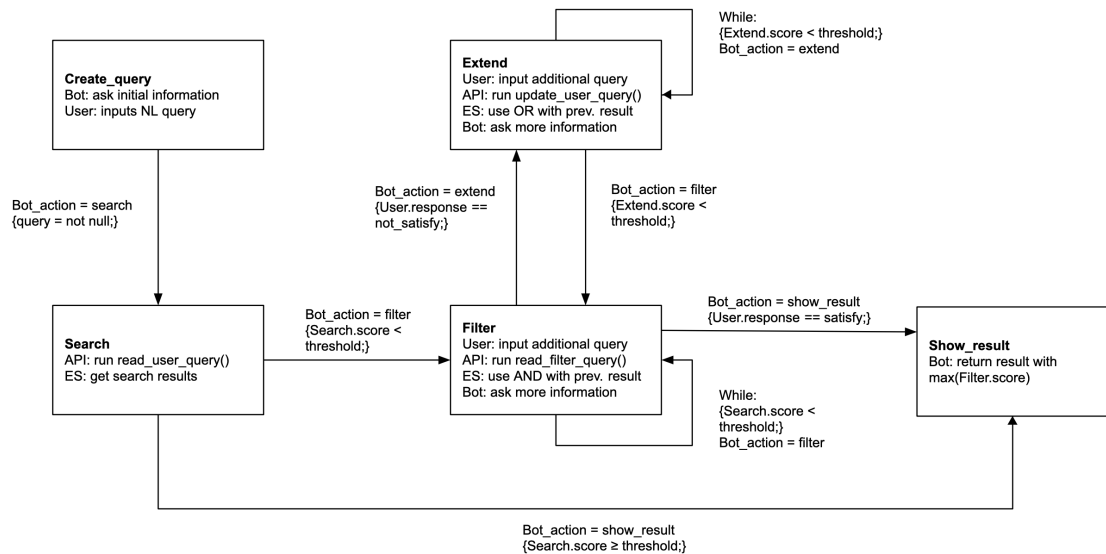


Figure 4.2: Finite state diagram showing how our chatbot behaves

All responses from users regarding whether they liked the returned search result will be stored in our database and will be used later to adjust our threshold. More things about this will be discussed in the threshold section.

4.1.5 Get keywords

Sometimes users may not have a clear idea or query in their mind. In such a case, it would be helpful if our chatbot can help the users to come up with a good query. That means, given an initial query, our chatbot is able to suggest some candidates of keywords that may make their query more informative. Thus, our chatbot can give a better answer to the users.

For that our chatbot will run an initial query and check all search results within the top 10 Elasticsearch scores. As it is shown in Sachdev et al. [19] the chance correct answer will be contained in the top 9 search results is very high, more than 90%, and this is also the same for our case (we will show this in the preliminary result chapter). Then our chatbot will suggest the possible keywords that users may like to use. In order to suggest meaningful keywords, we removed stop words from the data.

Figure 4.9 shows the case we aim to have. Suppose the user wants to ask “how to upload the file to Google cloud storage”, but he cannot come up with a precise query. To demonstrate that, first we searched with “upload”, then add “google” to the initial query. After that, we can see that the correct query appeared at the end of our keywords list.

Of course, this is a made-up example for the demonstration purpose. It would be very nice if we can test this keywords suggestion functionality indeed helps users to make a better query. Therefore, our users can save their time, thus stayed focus. Details about how to test this will be discussed in preliminary results and future work chapters.



Figure 4.3: Start the chat



Figure 4.4: Ask initial query to the user and get a response

4.1.6 Get final result

If the user thinks he found the correct answer, then he can click “Yes” button to the question “Like this answer?” (Figure 4.10). Then the selected code example is returned to the user and the chat is terminated. Also, all information from the chat is stored in our database.

There are two data schemas we store in the database. One schema is for the user query history with two columns, user and query. And another is for all information from the chat session including session id, response id, the score of search, and user satisfaction columns.

In the schema for user and query history, everything the user has typed in the chat is stored including the query and other responses. This is shown in figure 4.11. The important thing about the second schema, figure 4.12, for session information is the intent column. Our bot remember which answers the user likes or does not. Then it correctly updates this column. This will be used later to update our threshold score based on user response.

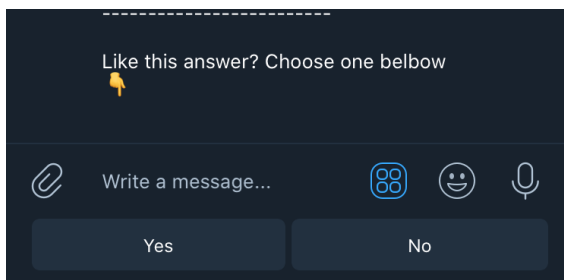


Figure 4.5: Ask the user about this answer is a right one

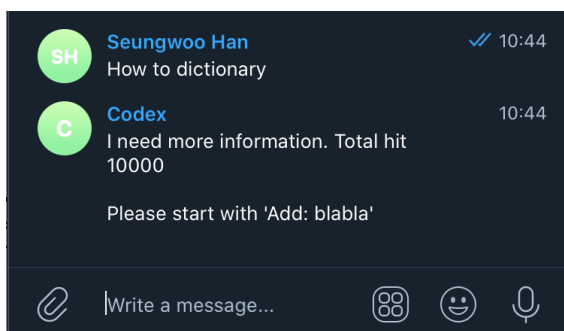


Figure 4.6: Ask user more information

4.2 Detailed implementation

In this section, some of the implementations of the main components of our chatbot are discussed. Python is chosen to be used to implement. Implementation of our chatbot requires the following packages in Python > 3.6.

- elasticsearch = 7.13.1
- fastapi = 0.66.0
- pydantic = 1.8.2
- uvicorn = 0.14.0
- SQLAlchemy = 1.4.23

4.2.1 Elasticsearch

In the Elasticsearch part, two main functions are implemented. One is for the initial search based on the natural language query (Listing 4.1). Another is for the filtering state to narrow down the search results, so we can find a more relevant code example with a higher score (Listing 4.2). For extend state, we use the same function for the initial search.

```
def read_user_query(query):
    res_search = es.search(index="test-index",
                           body={"size": 1000,
```

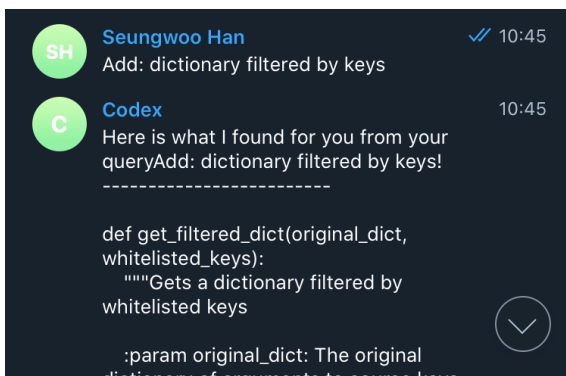


Figure 4.7: Return the search result from filtering

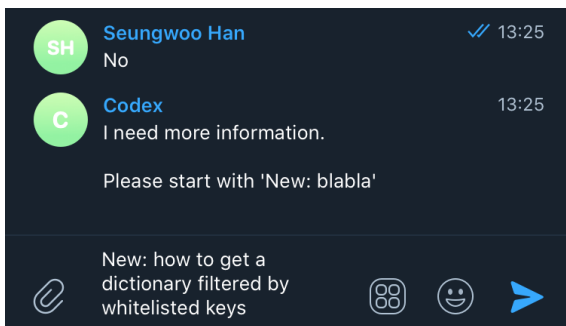


Figure 4.8: Ask the user a new query to extend the query

```

        "query": {
            "bool": {
                "must": [
                    {"match": {"docstring": query }},
                ]
            }
        },
        "stored_fields": []
    }
)

```

```

total_hit = res_search['hits']['total']['value']
response = []
scores = []
for hit in res_search['hits']['hits']:
    ids = hit["_id"]
    response.append(ids)
    score = hit["_score"]
    scores.append(score)

```

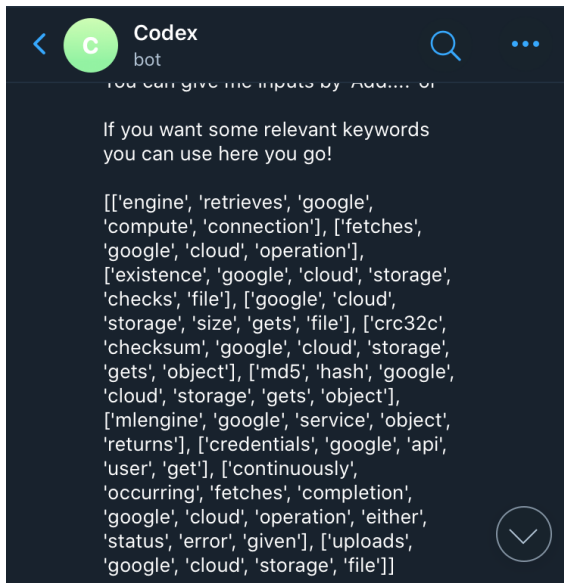


Figure 4.9: User gets candidate key words

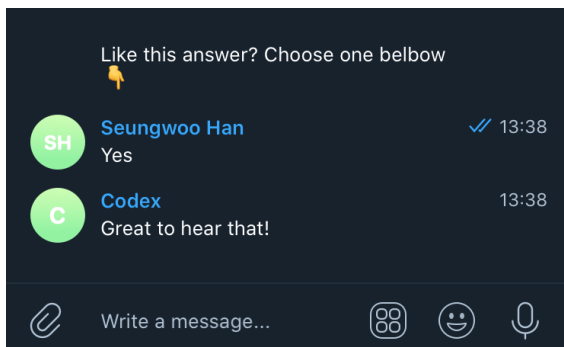


Figure 4.10: User found the right code example

```
id_selected[:] = response
result = es.get(index="test-index", id=id_selected[0])
code_example = result['_source']['code']
score_highest = scores[0]

return code_example, score_highest, id_selected, total_hit
```

Listing 4.1: Function for the initial search

This function receives the input of the natural language query then searches the output based on the id from our dataset stored in the Elasticsearch. Once we found the search results then our chatbot returns the one with the highest score from Elasticsearch. Also, it returns the id selected from this initial search so that we can use it for the filtering.

```
def read_filter_query(query, id_selected):
```

q_id	user_id	q
1	NULL	how to get list
2	NULL	No
3	NULL	new: how to get sum of list
4	NULL	No
5	NULL	new: how to get sum of last n elemen...
6	NULL	Yes
7	NULL	how to get sum of list

Figure 4.11: Schema for user and query history

id	session_id	user_id	response_id	score	intent
1	projects/codex-n9aw/agent/s...	NULL	d5536dfc-d023...	10.506203	not satisfy
2	projects/codex-n9aw/agent/s...	NULL	a9f58d15-e0da...	14.491508	not satisfy
3	projects/codex-n9aw/agent/s...	NULL	43f4c5e6-70c3...	17.790575	satisfy

Figure 4.12: Schema for chat session information

```

res_search = es.search(index="test-index",
    body={"size" : 1000,
        "query": {
            "bool": {
                "must": [
                    {"match": {"docstring": query }}],
            },
            "filter": {
                "ids": {"values": id_selected}
            }
        },
        "stored_fields": []
    })

```

```

total_hit = res_search['hits']['total']['value']
response = []
scores = []
for hit in res_search['hits']['hits']:
    ids = hit["_id"]
    response.append(ids)
    score = hit["_score"]
    scores.append(score)

id_selected[:] = response

```



```

result = es.get(index="test-index", id=id_selected[0])
code_example = result['_source']['code']
score_highest = scores[0]

return code_example, score_highest, id_selected, total_hit

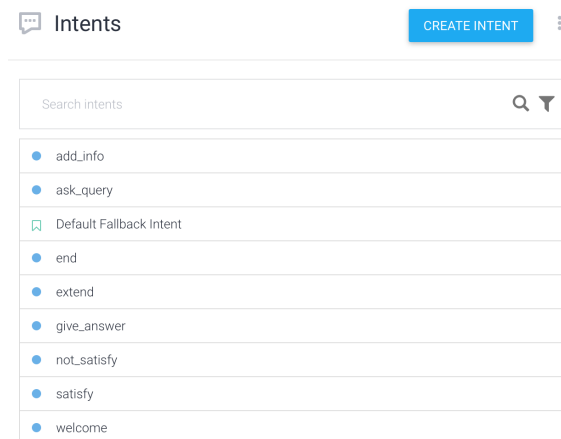
```

Listing 4.2: Function for the filtering

For the filtering function, please note that we give two inputs, an additional query and the previously selected id from the former query result. Then Elasticsearch will filter from the previous result so the number of outputs will be equal to or less than that of previous outputs. Again, we return the code example with the highest Elasticsearch score and update the id selected from this time search. Thus, we can use this for the next filtering if it is necessary.

4.2.2 DialogFlow




This part is the main part of our chatbot implementation. With FastAPI and DialogFlow we can integrate our Telegram to the Python back-end via fulfillment functionality. Figure 4.13 shows what kinds of intents our chatbot has. Again intent here can be thought of as the state in the finite states machine.

**Figure 4.13:** List of intents of our chatbot

Chat starts with welcome intent and ends with end intent. Some of the intents from this list are offered by DialogFlow as a default. For other intents, we need to do additional training by offering them specific training phrases which will be used for that intent. Hence, it is important to give DialogFlow the right set of training phrases that specify the right intents. If our chatbot does not understand which intent it is, then it falls into a Default Fallback Intent. Figure 4.14 shows the list of additional training phrases used to the “give answer” intents for the initial search.

Once all the intents are set, we need to distinguish which intents can be used directly with DialogFlow and which intents need to be connected to our back-end pipeline. For this, we need to enable additional functionality called fulfillment.

As we can see from the figure 4.15, we need to use a webhook for fulfillment. For this, we used ngrok, a cross-platform application that enables developers to expose a local development

Training phrases  Search training ph  

When a user says something similar to a training phrase, Dialogflow matches it to the intent. You don't have to create an exhaustive list. Dialogflow will fill out the list with similar expressions. To extract parameter values, use [annotations](#) with available [system](#) or [custom](#) entity types.


” Add user expression


” what

” how can

” how

Figure 4.14: Training phrases for particular intents

 Fulfillment


Webhook ENABLED 

Your web service will receive a POST request from Dialogflow in the form of the response to a user query matched by intents with webhook enabled. Be sure that your web service meets all the [webhook requirements](#) specific to the API version enabled in this agent.

URL* <https://747c-192-41-132-243.ngrok.io>

BASIC AUTH Enter username Enter password

HEADERS Enter key Enter value

 Add header


SMALL TALK Disable webhook for Smalltalk 

Figure 4.15: Enabling a fulfillment

server to the Internet. Once we connect to ngrok then we will receive the HTTPS URL. Then we simply copy this address and paste it in DialogFlow

Now basic part of DialogFlow is ready. Listing 4.3 shows our back-end development which governs the action-flow of our chatbot based on the intents it faces.

```
# Initial search
code_example, score, id_selected, total_hit = read_user_query(query_from_user)
if intent == "give_answer" or intent == "extend":
    if score >= 0.5*threshold:
        # add response_id, session_id, score to the database UserSession
        new_session = models.UserSession(response_id=response_id,
        session_id=session_id,
        score=score, intent="not satisfy")
        db.add(new_session)
        db.commit()
        db.refresh(new_session)

    return {
```

```

# "fulfillmentText": text,
"fulfillmentMessages": [
  {
    "quickReplies": {
      "title": f"Here is what I found for you from your
query{query_from_user}!"
      f"\n-----"
      f" \n \n{code_example}. "
      f"\n \n score {score}."
      f"\n \n Total hit {total_hit}."
      f"\n \n threshold {threshold}."
      f"\n \n Like this answer? Choose one below",
    "quickReplies": [
      "Yes",
      "No"
    ]
  },
  "platform": "TELEGRAM"
}
]
}

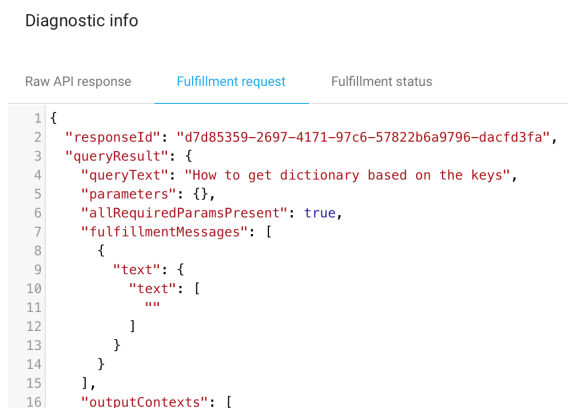
```

Listing 4.3: Back-end programming when intent is give answer or extend

First, we run the initial search based on a given query. If the score is higher than half of the threshold we first add the session information to the database. The reason why we use half of the threshold is to show the results user may like even if it is below the threshold.

More about database connection will be discussed in the next section. Then, we need to return the necessary output to the user through a DialogFlow. To communicate with DialogFlow via the back-end program we need to give the right JSON format that DialogFlow can use.

This format can be checked in the diagnostic info in the DialogFlow (Figure 4.16). There we can check the fulfillment request. This is a fixed format given by DialogFlow, so we cannot change this. But on the back-end side, we have some flexibility to manipulate this format so that we can only return the keys we want to show.

**Figure 4.16:** Fulfillment request from a DialogFlow

In order to integrate our chatbot to Telegram, we need to specify this in “platform” key. Also, DialogFlow offers a very convenient tool for quick replies which allows us to ask the user the additional question right after showing the result. We use this to check how much the user likes the initial answer.

Listing 4.4 shows the implementation for the case when we have the intent of filtering. The basic process is the same as the initial search case. And the only difference is we use the filtering function.

```
# add_info intent case: new query is automatically updated
if intent == "add_info":
    query_from_user = body.queryResult.queryText
    query_id = body.responseId
    session_id = body.session

    code_example, score, id_selected, total_hit = read_filter_query(
        query_from_user, id_selected)

    if score >= 0.5*threshold:
        # add response_id, session_id, score to the database userSession
        new_session = models.UserSession(response_id=response_id,
            session_id=session_id, score=score, intent="not satisfy")
        db.add(new_session)
        db.commit()
        db.refresh(new_session)

    return {
        "fulfillmentMessages": [
            {
                "quickReplies": {
                    "title": f"Here is what I found for you from your query
                    {query_from_user}!"
                    f"\n-----"
                    f" \n \n{code_example}. "
                    f"\n \nscore {score}."
                    f"\n \nTotal hit {total_hit}."
                    f"\n \nthreshold {threshold}."
                    f"\n \nLike this answer? Choose one belbow",
                    "quickReplies": [
                        "Yes",
                        "No"
                    ]
                },
                "platform": "TELEGRAM"
            }
        ]
    }
```

Listing 4.4: Back-end programming when intent is filtering

Again, we finish our intent by asking the user whether he found the answer he is satisfied with. Listing 4.5 shows the implementation for the case when the user says he likes this answer.

```

if intent == "satisfy":
    #change right above intent to satisfy
    # select last row in db
    last_row = db.query(models.UserSession).order_by
        (models.UserSession.id.desc()).first()

    # set intent to satisfy
    last_row.intent = "satisfy"
    db.commit()

return {
    "fulfillmentMessages": [
        {
            "quickReplies": {
                "title": "Great to hear that!",
                # "quickReplies": [
                # "Add more",
                # "End"
                # ]
            },
            "platform": "TELEGRAM"
        }
    ]
}

```

Listing 4.5: Back-end programming when intent is satisfy

The special thing about this intent is when the user says he likes this answer. In this case, we set this answer as “satisfy” and store it in our database. Therefore, we can use this information for updating a threshold score based on the user response.

As is explained in the previous section our chatbot is able to help users by suggesting possible keywords they may use to make their query better. When the search result is below the threshold, our bot will run the following code (Listing 4.6) to get the list of candidate keywords from our data. Since it has a hash table data structure, the search will be done in a constant time.

```

while score < 0.5*threshold:
    id_selected = id_selected_filter
    # add response_id, session_id, score to the database userSession
    new_session = models.UserSession(response_id=response_id,
        session_id=session_id, score=score,
        intent="not satisfy")

    db.add(new_session)
    db.commit()
    db.refresh(new_session)

    id_list_keyword = get_keywords(query_from_user)
    # id is string so change it to int
    id_list_keyword = [int(id) for id in id_list_keyword]

    key_word_list = [select_word_tf_no_stop[id] for id in id_list_keyword]

```

```

return {
    "fulfillmentMessages": [
        {
            "quickReplies": {
                "title": f"I need more information. \n \nTotal hit {total_hit}."
                f"\n \ngiven query {query_from_user}"
                f"\n \nYou can give me inputs by 'Add:...' or "
                f"\n \nIf you want some relevant keywords you can use
                here you go!"
                # f"\n \n{id_selected_filter[:10]}"
                # f"\n \n{id_selected[:10]}"
                f"\n \n{key_word_list}",
                # "quickReplies": [
                # # "Add input directly",
                # "Yes help me!"
                # ]
            },
            "platform": "TELEGRAM"
        }
    ]
}

```

Listing 4.6: Asking users if they like to have key words suggestion

First, we get the ids of the top 10 search results from the user query. Then, we get the keywords list of those search results from the hash table. This will be a nested list showing the keywords list for each of the 10 search results. Keywords are ordered by tf-idf scores, and this will be explained in more detail in the preliminary result chapter.

4.2.3 Database connection

It is important to store the data especially query history and satisfying answers from the users. If multiple users use our chatbot then this becomes more important since our bot is able to be optimized to each user based on their history. FastAPI offers an easy way to connect our system to the database by using SQL alchemy. In order to do a database connection first, we need to create the schemas for the data (Listing 4.7). This can be done easily by using class inheritance.

```

class UserQuery(Base):
    __tablename__ = "user query"
    q_id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer)
    q = Column(String)

class UserSession(Base):
    __tablename__ = "user session"
    id = Column(Integer, primary_key=True, index=True)
    session_id = Column(String)
    user_id = Column(Integer)
    response_id = Column(String)
    score = Column(Float)

```

```
intent = Column(String)
```

Listing 4.7: Creating schemas for database

We can create as many schemas as we want, but we created two main schemas at the moment. One is the `userQuery` schema which contains all the queries from the user. Another is the `userSession` schemas which have the information regarding the chat session, score of answers, and response from the user (satisfied or not satisfied). Once we created the schemas we need, then we simply create the engine for the SQL alchemy to connect to the database by using the code described in the listing 4.8.

```
SQLALCHEMY_DATABASE_URL = "sqlite:///./codex.db"
engine = create_engine(
    SQLALCHEMY_DATABASE_URL, connect_args={"check_same_thread": False}
)

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative_base()
```

Listing 4.8: Connecting to database by SQL alchemy

4.2.4 Threshold

As it is described in the finite state diagram (Figure 4.2), we use a threshold to decide which search result will be returned. However, setting a correct threshold is very tricky especially if we use a constant value for it. Having a fixed valued threshold has several problems in our use case.

Different users have a different threshold. This is the obvious problem from the fixed threshold value. Some users think that a slightly less relevant answer is still fine for them since they can figure out from that search result to solve their task. For such users, our bot does not need to keep asking for additional information and delaying showing the search result. On the other hand, the person who wants to see a highly relevant answer will need a higher threshold.

Score from Elasticsearch is useful to rank the result from the same query. This is a not so obvious problem and also partially stemmed from the score of Elasticsearch. Elasticsearch gives a different range of scores depending on which query is fetched. Thus, if we simply use one fixed threshold for all the queries then our bot will not be able to compare the scores from the different queries in a meaningful way.

Our solution to these problems. In order to fix these problems, our bot has the functionality of tuning or updating its threshold (Listing 4.9). The threshold value is set by some initial value. As the chat progresses, our bot stores all the responses from the user. Therefore, we can check the list of answers which satisfy the user. From this list, we calculate the average score which satisfies the user. And we use this average value as an updated threshold.

Of course, this is not a perfect way to have a correct threshold. Maybe having a threshold is not a good idea. Or there can be a better design choice for different thresholds. And this will be discussed in the future work section.

```
@app.post("/")
async def create_user_query(body: schemas.Fulfillment,
```

```
db: Session = Depends(get_db)):

# update the threshold from database with the average scores users satisfy
if db.query(models.UserSession.score).filter(
models.UserSession.intent == "satisfy").count() > 0:
    avg_score = db.query(func.avg(models.UserSession.score)).filter(
models.UserSession.intent == "satisfy").first()
else:
    avg_score = []

# set threshold according to user satisfaction
# we need to increase threshold not to fast
initial_val_threshold = 20
threshold = initial_val_threshold if not avg_score else avg_score[0]
```

Listing 4.9: Updating a threshold based on the user response

Preliminary Results

In this chapter, we will mainly discuss the performance of our chatbot. Evaluating the effectiveness of a chatbot requires many resources, involving a user study with humans. However, in this thesis, we present a preliminary study under limited conditions for evaluation. This makes our evaluation results preliminary, but still, it shows potential benefits from using a chatbot in code search.

Firstly, we will discuss which evaluation metric we chose and why. Then, we compare the result of our chatbot with the performance done in Sachdev et al [19] for the single-round conversation. Next, we show the preliminary results for the multiple-round conversation. This is to check the ability of our chatbot in guiding users to get the right answer at the end.

For the multiple-round part, evaluation cannot be done properly without the separate experiment design with many participants. Hence, we partially test our bot for this part.

5.1 Single-round conversation

Here we evaluate how many times our chatbot returns the correct search results in one round conversation. That means, no filtering and extending are used. Thus it can be viewed as the evaluation of Elasticsearch. But it is still necessary to show since it proves that the basic functioning of our chatbot is fine. The methodology we used is subset words retrieval metric which is proposed by Sachdev et al. [19].

The main reason why we chose this metric is that this does not require any human resources to conduct. This can be done in an automatic way, so it is the right choice for us. Furthermore, from the process for this evaluation, we can get a very useful intermediate output that can be used for the keywords suggestion property of our chatbot. The subset words retrieval metric works as follows:

- create a mock query by picking 20% of words from the source code example. For example, if the source code has 10 words, then pick 2 words from there. Thus, the minimum length of the source code should be larger than 5 words. There are two ways to pick these words.
- the first way is to pick words randomly.
- The second way is to pick words based on the tf-idf score of words.
- then check how many percent the bot returns the correct code example from the mock query.

In the original paper, they performed an evaluation in two ways. Firstly, they checked whether the correct search output is returned in the first place. Secondly, they checked the correct search

output is included in the top 9 search results. We also perform those two ways of evaluation for our chatbot.

Here is the one difference between Sachdev et al. [19] and ours. When making a mock query, Sachdev et al. [19] used all words from the source codes. That means, their mock query is built solely based on the source code. It may increase the performance since the words which consist of mock queries are basically the same as the search target. However, in reality, users will use their own natural language expression to search the source code, and the structures of those two are quite different.

Thus, we use only docstrings from our dataset to build mock queries, excluding the source code. Docstring is the comments explaining the function in the source code. This will make our evaluation more challenging than the original paper for the following reasons.

- docstring contains a much smaller amount of words than source code in many cases. Hence, the mock queries created from the docstrings will have less information.
- the words and expressions used in docstrings can be very different from the ones used in the source code. This is a well-known issue in the source code search problem.

5.1.1 Evaluation results

Based on the evaluation metrics described above, we checked and compared the performance for the single-round case. Of course, since we used the different datasets from Sachdev et al. [19] it is not a precise comparison, but this may give us some potentials of our chatbot for future use.

Sachdev et al. [19] proposed the way for code search based on the natural language processing, mainly word-embedding and tf-idf. Therefore, they transformed the natural language query and source code into the embedding space, then computed the vector distance between those two.

They used over 700,000 methods in the 1,000 Android repositories from GitHub. As it is described in the table 5.1, the percentage to return the correct answer from the randomly generated query is slightly lower than 50%. When they used the query generated by high tf-idf scored words the percentage goes up to about 70%. When they expand the output to the top 9 search results, then the percentage is increased very much.

Sachdev et al. [19]	Top 1	Top 9
Random	46.1%	74.4%
TF-IDF	68.9%	94.6%

Table 5.1: Evaluation result of Sachdev et al.

We also performed the evaluation for the top 1 and top 9 results. The dataset we used is the same one we used for Elasticsearch. We used 18,248 docstrings to evaluate our performance. Again, this evaluation is for Elasticsearch rather than our chatbot since we only performed a single-round conversation here. And our evaluation condition is harsher than that of Sachdev et al.

Codex bot	Top 1	Top 9
Random	30.78%	50.47%
TF-IDF	61.65%	86.85%

Table 5.2: Evaluation result of our chatbot.

Table 5.2 shows that the performance in the single-round case of our chatbot is slightly lower than that of Sachdev et al. [19]. However, considering we only used the docstrings for our queries, we can say that the performance of our chatbot is very promising.

5.2 Multiple-round conversation

The previous result of the single-round case is based on the initial search only. That means, this neglects the power of the conversational component of a chatbot, including the filtering, keywords suggestion, and expanding properties of our chatbot.

For example, suppose the right answer to the initial query is contained in the top 9 search outputs but not the top 1 search result. Thus, our chatbot shows the wrong search results. In this case, expand functionality enables our chatbot to ask the user for more information if the user says that this is not what they are looking for.

Regarding an evaluation of chatbot, Stent et al. [48] and Liu et al. [49] already showed that the automatic metrics is not a good choice to evaluate the dialogue response system because it requires human judgment.

Because of the limited resources, we put this evaluation part as our future work. However, we can conduct the beginning part of this. That means, testing whether our chatbot is able to ask for more information rather than give a wrong answer to users.

5.2.1 Evaluation design

The accuracy of our chatbot is low in certain cases in the single-round setting (Table 5.2). What we want from our chatbot is that it asks for more information in such cases. Hence, our chatbot initiates a series of conversations with users to get more information. Here we tested our chatbot for that. This test is conducted by one of the authors of this thesis and the detailed procedure is described as follow:

1. From the single-round evaluation, we saved the tf-idf based mock queries that our chatbot failed to give the correct answers.
2. We give those queries to our chatbot in the multiple-round conversation setting.
3. Check how many times our chatbot asks users for more information.
4. Try this with multiple thresholds.

5.2.2 Preliminary results

Please note that we used the mock queries that our chatbot returns in the single-round evaluation. That means the search results from these mock queries are already above the threshold used for the single-round test, 10 namely.

As we can see from the table 5.3 the percentage of asking more information when our chatbot received the query that it failed to give a correct answer is increasing as we increase the threshold. The meaningful threshold required to ask for more information is about 50 or above.

This is a very high threshold to use in real practice. This is mainly because the query is directly from the data. And when Elasticsearch catches the same keywords from the query as its data, it gives a very high score. Hence, we need a higher threshold to reject this score.

Threshold	Total number of tf-idf query	Percentage of asking more information
10	6998	0%
20	6998	5%
30	6998	29.97%
40	6998	61.39%
50	6998	79.05%
60	6998	85%
70	6998	87.08%
80	6998	88.01%
90	6998	88.71%

Table 5.3: Percentage of asking more information from the tf-idf query that chatbot did not give correct answer.

In real cases, the query from the users is probably somewhat different from the data in Elasticsearch. This will make the score decrease a lot. In my own experience, a threshold of around 15 or above can give a relevant answer to my query.

Of course, this is a subjective area, and that is why we added the tuning property for the threshold based on the individual response. Unfortunately, it is infeasible to test how much our chatbot works well in real cases by one person. Instead, we present the possible experiment design to test our bot for the real use case.

5.3 Future experiment for the proper evaluation

Yi et al. [47] evaluated their chatbot which is built based on the finite state machine with 17 participants, and this gives an idea for how we can evaluate our chatbot in the future. Firstly, they created a Facebook page where participants can use their bot via text messages. And each participant will answer four questions regarding the different aspects of the performance of their bot.

1. How coherent was the conversation?
2. How engaging was the bot?
3. Whether it avoided inappropriate responses?
4. Overall experience.

Then, participants gave the scores to each question from 1 (very bad) to 5 (very good). Based on these questions we can propose the following questions to evaluate the performance of our chatbot.

1. How many times you find the answer you like at the first query?
2. How many times chatbot asked you more information?
3. How many time you find the answer you like at the end of conversation?
4. How much the key words suggested by chatbot is useful?
5. How engaging was the bot?

6. Whether it avoided inappropriate responses?
7. Overall experience

The first three questions are for the quantitative evaluation, and the rests are for the qualitative ones. By asking the quantitative questions we can get the statistical values that show how much our chatbot is able to guide users to end up with satisfactory search results. Also, we can get an overall impression of our chatbot from the qualitative questions. Of course, there can be more questions we may add in the future to make this experiment more reliable and precise. More about this will be discussed in the future work section.

Conclusion and Future Work

In this thesis, we created a conversational agent which can provide a code example (functions mainly) based on the user's natural language query. The main objective of our chatbot is to help the developers in an efficient manner by not only offering an actual code example without noise but also communicating with them to search for the more relevant code example.

Therefore, our chatbot can offer a continuous workflow of programming. The developers will get only the relevant code examples without any other massive information. Additionally, since our chatbot is very fast in action, our users will have much less distraction than when they use the online search. Lastly, our chatbot can be accessed via Telegram, so the users can ask their queries whenever they want.

6.1 Future work

As it is verified in our evaluation section our chatbot can be a useful tool for the code example search. However, there are several things which can be done in future work.

Replace Elasticsearch with deep learning based approaches. As Salza et al. [13] and Sachdev et al. [19] showed, using deep learning can be a good method for the code search. Since Elasticsearch uses text-based search, it does not capture the contextual meaning of the natural language query. However, the deep learning approach can capture it. Therefore, as long as it does not hamper the search speed, applying a deep learning technique can make our chatbot more contextual.

Evaluate the effectiveness of our chatbot with multiple users. This is to check the threshold tuning functionality of our chatbot for each user. At the current stage, our bot is evaluated by only one user, so it cannot be tested in multiple user settings. Once our chatbot is deployed, this part can be tested with the experiment design mentioned in the preliminary results chapter.

Change the way to formulate a threshold. The threshold of our chatbot is tuned based on the user response. Now we use the average value of the scores from the search results user satisfied. However, different queries may impact users to have a different threshold. Therefore, having multiple ways to update the threshold depending on the topic of queries can make our chatbot have a more precise threshold.

Check whether our chatbot can guide users to search for the right answer in the multiple-round setting. We can conduct the further evaluation described in the preliminary result chapter with students from various classes. Since there will be a good mix of beginner and non-

beginner programmers in the class, we can divide our participants is two groups, beginner and non-beginner. In this way, we can check how each group thinks about how much our chatbot is useful for them.

Bibliography

- [1] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, “Popularity, interoperability, and impact of programming languages in 100,000 open source projects,” in *2013 IEEE 37th annual computer software and applications conference*, pp. 303–312, IEEE, 2013.
- [2] C. Sadowski, K. T. Stolee, and S. Elbaum, “How developers search for code: A case study,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, (New York, NY, USA), p. 191–201, Association for Computing Machinery, 2015.
- [3] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, “An examination of software engineering work practices,” in *CASCON First Decade High Impact Papers*, pp. 174–188, 2010.
- [4] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, “Exemplar: Executable examples archive,” in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, pp. 259–262, IEEE, 2010.
- [5] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, “Query expansion via wordnet for effective code search,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 545–549, IEEE, 2015.
- [6] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, “Automatic query reformulations for text retrieval in software engineering,” in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 842–851, IEEE, 2013.
- [7] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, “How well do search engines support code retrieval on the web?,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 1, pp. 1–25, 2011.
- [8] M. J. Cafarella and O. Etzioni, “A search engine for natural language applications,” in *Proceedings of the 14th international conference on World Wide Web*, pp. 442–452, 2005.
- [9] J. Cambronero, H. Li, S. Kim, K. Sen, and S. 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*, pp. 964–974, 2019.
- [10] Z. Yao, J. R. Peddamail, and H. Sun, “Coacor: Code annotation for code retrieval with reinforcement learning,” in *The World Wide Web Conference*, pp. 2203–2214, 2019.
- [11] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, IEEE, 2018.
- [12] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” *arXiv preprint arXiv:2005.00653*, 2020.

- [13] P. Salza, C. Schwizer, J. Gu, and H. C. Gall, "On the effectiveness of transfer learning for code search," 2021.
- [14] A. Sordoni, M. Galley, M. Auli, C. Brockett, Y. Ji, M. Mitchell, J.-Y. Nie, J. Gao, and B. Dolan, "A neural network approach to context-sensitive generation of conversational responses," *arXiv preprint arXiv:1506.06714*, 2015.
- [15] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *Advances in neural information processing systems*, vol. 28, pp. 649–657, 2015.
- [16] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," *arXiv preprint arXiv:1606.01781*, 2016.
- [17] P. Pirapuraj and I. Perera, "Analyzing source code identifiers for code reuse using nlp techniques and wordnet," in *2017 Moratuwa Engineering Research Conference (MERCon)*, pp. 105–110, IEEE, 2017.
- [18] G. Press, "120 ai predictions for 2019," *Forbes*, 2019.
- [19] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, (New York, NY, USA), p. 31–41, Association for Computing Machinery, 2018.
- [20] C. Liu, X. Xia, D. Lo, Z. Liu, A. E. Hassan, and S. Li, "Simplifying deep-learning-based model for code search," *arXiv preprint arXiv:2005.14373*, 2020.
- [21] R. Dale, "The return of the chatbots," *Natural Language Engineering*, vol. 22, no. 5, pp. 811–817, 2016.
- [22] B. A. Shawar and E. Atwell, "Chatbots: are they really useful?," in *Ldv forum*, vol. 22, pp. 29–49, 2007.
- [23] P. B. Brandtzaeg and A. Følstad, "Why people use chatbots," in *International conference on internet science*, pp. 377–392, Springer, 2017.
- [24] M. Jain, P. Kumar, R. Kota, and S. N. Patel, "Evaluating and informing the design of chatbots," in *Proceedings of the 2018 Designing Interactive Systems Conference*, pp. 895–906, 2018.
- [25] E. Adamopoulou and L. Moussiades, "An overview of chatbot technology," in *Artificial Intelligence Applications and Innovations* (I. Maglogiannis, L. Iliadis, and E. Pimenidis, eds.), (Cham), pp. 373–383, Springer International Publishing, 2020.
- [26] E. Adamopoulou and L. Moussiades, "An overview of chatbot technology.," *Artificial Intelligence Applications and Innovations*, vol. 584, pp. 373–383, 2020.
- [27] S. A. Thorat and V. Jadhav, "A review on implementation issues of rule-based chatbot systems," in *Proceedings of the International Conference on Innovative Computing & Communications (ICICC)*, 2020.
- [28] I. V. Serban, C. Sankar, M. Germain, S. Zhang, Z. Lin, S. Subramanian, T. Kim, M. Pieper, S. Chandar, N. R. Ke, *et al.*, "A deep reinforcement learning chatbot," *arXiv preprint arXiv:1709.02349*, 2017.
- [29] M. Nuruzzaman and O. K. Hussain, "A survey on chatbot implementation in customer service industry through deep neural networks," in *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, pp. 54–61, IEEE, 2018.

- [30] R. Csaky, "Deep learning based chatbot models," *arXiv preprint arXiv:1908.08835*, 2019.
- [31] M. Dhyani and R. Kumar, "An intelligent chatbot using deep learning with bidirectional rnn and attention model," *Materials today: proceedings*, vol. 34, pp. 817–824, 2021.
- [32] L. Floridi and M. Chiriatti, "Gpt-3: Its nature, scope, limits, and consequences," *Minds and Machines*, vol. 30, no. 4, pp. 681–694, 2020.
- [33] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.
- [34] C. Gormley and Z. Tong, *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.
- [35] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, pp. 29–48, Citeseer, 2003.
- [36] L. Smith, "How scoring works in elasticsearch," 2016.
- [37] FastAPI, "Fastapi," 2021.
- [38] S. Janarthanam, *Hands-on chatbots and conversational UI development: build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills*. Packt Publishing Ltd, 2017.
- [39] G. D. Abowd, H.-M. Wang, and A. F. Monk, "A formal technique for automated dialogue development," in *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*, pp. 219–226, 1995.
- [40] M. F. McTear, "Modelling spoken dialogues with state transition diagrams: experiences with the cslu toolkit," *development*, vol. 5, no. 7, 1998.
- [41] D. Goddeau, H. Meng, J. Polifroni, S. Seneff, and S. Busayapongchai, "A form-based dialogue manager for spoken language applications," in *Proceeding of Fourth International Conference on Spoken Language Processing. ICSLP'96*, vol. 2, pp. 701–704, IEEE, 1996.
- [42] S. Seneff and J. Polifroni, "Dialogue management in the mercury flight reservation system," in *ANLP-NAACL 2000 Workshop: Conversational Systems*, 2000.
- [43] A. Raux and M. Eskenazi, "A finite-state turn-taking model for spoken dialog systems," in *Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics*, pp. 629–637, 2009.
- [44] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2020.
- [45] N. S. Patil, K. K. Shinde, A. R. Kulkarni, and A. M. Jagtap, "A study of revolutionizing real estate: Search engine using machine learning (ml), artificial intelligence (ai) and data analytics in elasticsearch algorithm," 2020.
- [46] B. A. Shawar, E. Atwell, and A. Roberts, "Faqchat as in information retrieval system," 2005.
- [47] S. Yi, "A chatbot by combining finite state machine , information retrieval , and bot-initiative strategy," 2017.

- [48] A. Stent, M. Marge, and M. Singhai, "Evaluating evaluation methods for generation in the presence of variation,"
- [49] C.-W. Liu, R. Lowe, I. V. Serban, M. Noseworthy, L. Charlin, and J. Pineau, "How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation," 2017.