Albert Cañadilla Domingo

DEVELOPMENT OF A LOCAL CHATBOT WITH A CUSTOMIZED KNOWLEDGE BASE

BACHELOR’S THESIS

Supervised by Dr. Pedro Antonio García López

Bachelor’s Degree in Computer Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona

2024

Abstract

A chatbot is a software application that relies on artificial intelligence to interact with a machine in a way similar to a human conversation. Its use has become popular in recent years with the arrival of commercial solutions, such as ChatGPT by OpenAI, Llama by Meta or Gemini by Google.

This work explores new techniques and tools to improve these systems and their implementation in local environments, as well as a more customizable service adapted to the needs of each individual as efficiently as possible.

It exploits not only portability but also adaptability. In a conventional system, it is necessary to train the model with a series of data according to the behavior you want to obtain. With the arrival of new data, it is necessary to retrain the model. Since model training is a temporary and economically expensive operation, it is not feasible to resort to it normally.

The alternative proposed in this work includes a different implementation in order to be able to provide external knowledge (for example from a set of documents) without having to go through this constant retraining of the model.

Resum

El xatbot és una aplicació de software que es basa en la intel·ligència artificial per interactuar amb una màquina de manera semblant a una conversa humana. El seu ús s'ha popularitzat en els anys recents amb l'arribada de solucions comercials, com ChatGPT de OpenAI, Llama de Meta o Gemini de Google.

Aquest treball explora de noves tècniques i eines per a la millora d'aquests sistemes i la seva implementació en entorns locals, així com en un servei més personalitzable i adaptat a les necessitats de cada individu el màxim eficient possible.

Explota no només la portabilitat sinó també l'adaptabilitat. En un sistema convencional, cal entrenar al model amb una sèrie de dades segons el comportament que es vulgui obtenir. Amb l'arribada de noves dades, cal reentrenar el model. Essent l'entrenament de models una operació temporal i econòmicament costosa, no és viable de recórre-hi amb normalitat.

L'alternativa proposada en aquest treball inclou una implementació per tal de poder aportar-li coneixement (per exemple, d'un conjunt de documents) extern sense haver de passar per aquest reentrenament constant del model.

Resumen

El chatbot es una aplicación de software que se basa en la inteligencia artificial para interactuar con una máquina de forma similar a una conversación humana. Su uso se ha popularizado en los años recientes con la llegada de soluciones comerciales, como ChatGPT de OpenAI, Llama de Meta o Gemini de Google.

Este trabajo explora nuevas técnicas y herramientas para la mejora de estos sistemas y su implementación en entornos locales, así como en un servicio más personalizable y adaptado a las necesidades de cada individuo lo máximo eficiente posible.

Explota no sólo la portabilidad sino también la adaptabilidad. En un sistema convencional, es necesario entrenar al modelo con una serie de datos según el comportamiento que se quiera obtener. Con la llegada de nuevos datos, es necesario reentrenar el modelo. Siendo el entrenamiento de modelos una operación temporal y económicamente costosa, no es viable recurrir a ella con normalidad.

La alternativa propuesta en este trabajo incluye una implementación a fin de poder aportarle conocimiento externo (por ejemplo, de un conjunto de documentos) sin tener que pasar por este reentrenamiento constante del modelo.

Dedication

This work is dedicated to my family, whose education, values and unconditional support have laid the foundation for my achievements and of what I have become. Thank you for always believing in me and for all your sacrifices.

Acknowledgements

I would like to thank my tutor Dr. Pedro Antonio García López for the opportunities he has given me during these last months, and for all his support and advice. This goes as well for the help and company received from all my colleagues in the Cloudlab research group.

I also want to extend my gratitude to my friends who have always been there, sharing memorable moments together, and also to the amazing people I have met during these years during my studies at college.

"I'd like to dedicate this to the losers. Because I tell you from my own experience: winning is one thing, but out of losing I always learned more for the future. So I got stronger from losing."

- Niki Lauda

Contents

List of Figures

List of Listings

List of Acronyms and abbreviations

AI	Artificial Intelligence
LLM	Large Language Model
NPL	Natural Processing Language
RAG	Retrieval Augmented Generation
ACID	Atomicity, Consistency, Isolation and Durability
OOP	Object Oriented Programming
GUI	Graphical User Interface
PoC	Proof of Concept
RACO	Reasoning About Colored Objects
STED	Salient Translation Error Detection
TSO	Tracking Shuffled Objects

1 Introduction

The advent of **artificial intelligence** (AI) has revolutionized numerous fields, including the development of chatbots. Chatbots are sophisticated software applications designed to simulate human conversation through text or voice interactions, becoming integral tools for businesses and individuals by enhancing customer service and efficient information retrieval. This thesis deepens into recent advancements in chatbot technology, focusing on improving functionality in local environments and providing a more personalized user experience through customized chatbots.

In recent years, the popularity of chatbots has surged with the introduction of high-profile solutions like **ChatGPT** by OpenAI, **Llama** by Meta, and **Gemini** by Google. These platforms have set new standards for conversational AI, demonstrating remarkable capabilities in understanding and generating human-like responses. Despite their success, there remains significant potential for further enhancement, particularly in customization and varied environment deployment.

Traditional chatbot systems typically require extensive model **training** with predefined data, a process that must be repeated with new data, making frequent retraining impractical due to its time and cost demands. This research work addresses the challenge of frequent retraining by proposing an alternative pipeline.

The proposed integration enables real-time updates and access to new information without retraining the model, resulting in more accurate and relevant responses that adapt to new information. For instance, one could **provide a set of documents** or files, enabling the chatbot to answer questions specifically about the information contained within those files. When a user poses a question, relevant data is retrieved and used as context of the question in order to generate accurate, context-specific responses. This capability allows the chatbot to handle queries about **proprietary** or **specialized content** that isn't covered by public solutions like the aforementioned ones, enhancing the ability to adapt to specific user needs and preferences.

In summary, this thesis aims to advance chatbot technology by exploring methods to enhance functionality, **adaptability**, and deployment. The proposed design and technologies to implement the solution represent address traditional model training limitations and enabling more efficient external knowledge integration and local implementation.

2 Objectives and motivations

To put us in situation, I was presented with a problem by a company. They wanted to develop an online chatbot to help their costumers navigation their website. Since they had a customized website, the bot had to be able to answer **specific questions** that public AI models could have not done. Additionally, the chatbot needed the capability to **reference** its information sources to redirect users for further details if they desired.

The first objective of this work is to **investigate** and learn about the listed technologies and concepts in the following sections. The aim is to understand how to create a portable model that can be adapted to different situations, while the study of those technologies will provide knowledge on the fields.

The main goal is to **integrate** all this knowledge to develop a solution that can be adjusted and customized according to the specific needs of each context or application.

The motivation behind this thesis is that it serves as a **learning experience**. By developing this tool, I am not only learning new technologies, but also pushing myself to tackle real-world problems. Furthermore, I see this project as an opportunity to prepare myself in a professional way to be more versatile and resourceful in my future career.

Ultimately, this tool is a testament to my commitment to continuous learning and my ambition to stay at the forefront of technological advancements.

For instance, front-end development and AI haven't been a great focus of my university studies. Along with Cloud computing, I think AI is a trending area that will grow exponentially in the future years.

To summarize, the main objectives of this thesis are:

- Investigate and learn about specific technologies and concepts relevant to developing an online chatbot tailored for customized websites.
- Understand how to create a portable model adaptable to various contexts by studying the identified technologies.
- Integrate acquired knowledge to develop a customizable solution meeting specific needs of diverse applications.
- Utilize the project as a learning experience to gain proficiency in new technologies and problem-solving skills.
- Prepare for future career endeavors by addressing real-world challenges and enhancing versatility and resourcefulness.
- Demonstrate commitment to continuous learning and staying updated with technological advancements, particularly in AI, cloud computing, and front-end development.

3 Planification

In this section I will clarify the **timeline** of the project and the initial planification of **tasks**. In the following fig. 1, there is an ordered list of tasks that I have followed in aim to develop my solution and accomplish my objectives:

Task Mode	Task Name	Duration	Start	Finish
	Determining the topic of the thesis	1 day	Wed 20/03/24	Wed 20/03/24
	Writing the thesis	28 days	Wed 01/05/24	Fri 07/06/24
	Learning and documentation about LocalAI and LLMs	2 days	Thu 21/03/24	Fri 22/03/24
	LocalAI implementation	4 days	Mon 25/03/24	Thu 28/03/24
	Learning and documentation about DAG and vector databases	2 days	Fri 29/03/24	Mon 01/04/24
	Selecting a vector database solution	1 day	Tue 02/04/24	Tue 02/04/24
	Learning and documentation about Langchain	2 days	Wed 03/04/24	Thu 04/04/24
	Learning and documentation about ChromaDB	2 days	Fri 05/04/24	Mon 08/04/24
	Implementation of Langchain and ChromaDB	4 days	Tue 09/04/24	Fri 12/04/24
	Integrating LocalAI with Langchain and ChromaDB	4 days	Mon 15/04/24	Thu 18/04/24
	Testing	1 day	Fri 19/04/24	Fri 19/04/24
	Fix possible errors	2 days	Mon 22/04/24	Tue 23/04/24
	Learning and documentation of Flask	1 day	Wed 24/04/24	Wed 24/04/24
	Flask implementation on Langchain service	2 days	Thu 25/04/24	Fri 26/04/24
	Testing	1 day	Mon 29/04/24	Mon 29/04/24
	Fix possible errors	2 days	Tue 30/04/24	Wed 01/05/24
	Learning and documentation about Flutter	2 days	Thu 02/05/24	Fri 03/05/24
	GUI development	10 days	Mon 06/05/24	Fri 17/05/24
	Backend and frontend integration	4 days	Mon 20/05/24	Thu 23/05/24
	Testing	1 day	Fri 24/05/24	Fri 24/05/24
	Fix possible errors	2 days	Mon 27/05/24	Tue 28/05/24
	Learning and documentation about no-sql databases	2 days	Wed 29/05/24	Thu 30/05/24
	Selecting a no-sql database solution	1 day	Mon 27/05/24	Mon 27/05/24
	Implementation of CouchDB	3 days	Fri 31/05/24	Tue 04/06/24
	Testing	1 day	Wed 05/06/24	Wed 05/06/24
	Fix possible errors	1 day	Thu 06/06/24	Thu 06/06/24

Figure 1: Planification

I also attach a timeline plot of the project, where the reader can see the tasks on fig. 2 in a more visually manner.

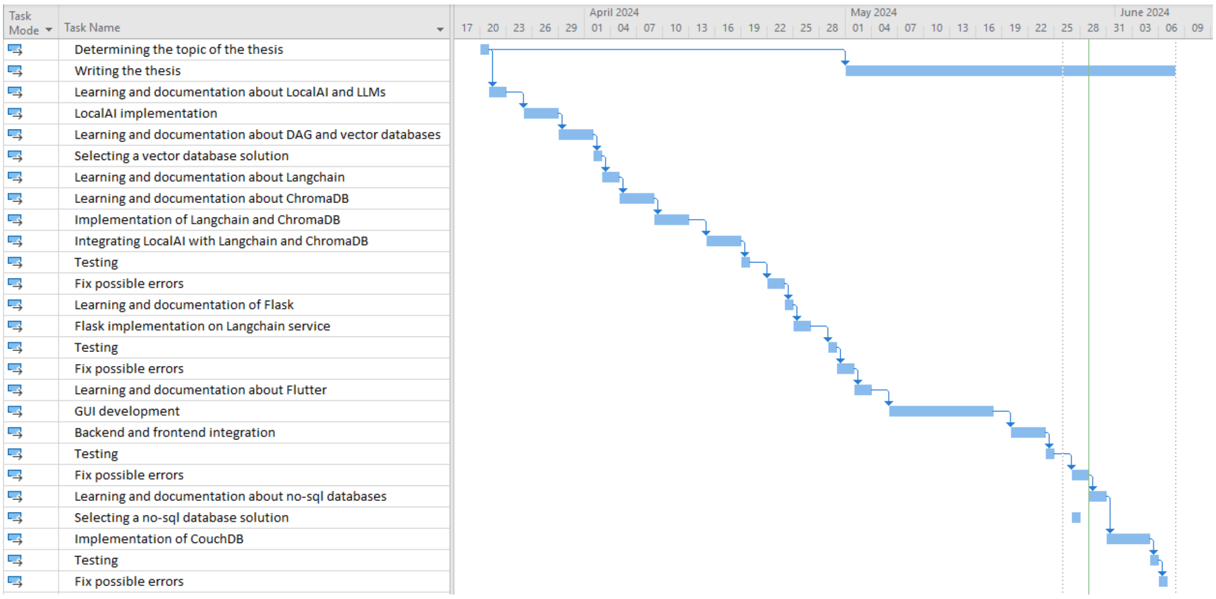

Figure 2: Planification timeline

4 Requirements

In this section, I will cover the requirements of the solution. I will consider both *functional* and *non-functional* requirements as integral components of the overall system design. Functional requirements will detail specific behaviours while non-functional requirements will address the quality attributes of the system.

4.1 Functional requirements

Here, I list all of the functional requirements of the pipeline.

1. Use many different compatible models.
2. Query the model about general data.
3. Upload files.
4. Extract and store data from websites.
5. Extract and store data from files.
6. Retrieve data from stored data.
7. Handle multiple conversations.
8. Query the model about specific data.

4.2 Non-Functional requirements

To provide a great user experience, the developed solution aims to accomplish these non-functional requirements. Not only that, but the goal for pursuing these requirements is to also provide a robust solution.

1. Accessible.
2. Deployable.
3. Failure transparent.
4. Usable.
5. Scalable.
6. Adaptable.
7. Responsive.
8. Interoperable

5 Requirements Analysis

In this section I will include a more detailed analysis of the requirements and the structure of the final solution in form of charts and diagrams.

Since some part of the back-end foundations come from pre-built solutions and the other part consisted of my own self-developed packages are not structured or object-oriented, I will contemplate the front-end analysis. The frontend structure accomplishes the object-oriented structure and also is entirely built by myself, instead of being pre-built.

5.1 Class Diagram

In the figure below (fig. 3) I describe the class structure of the built classes in my front-end code. In blue I represent stateful classes, and in orange I want to represent static, stateless classes that I specifically designed to supplement the other classes.

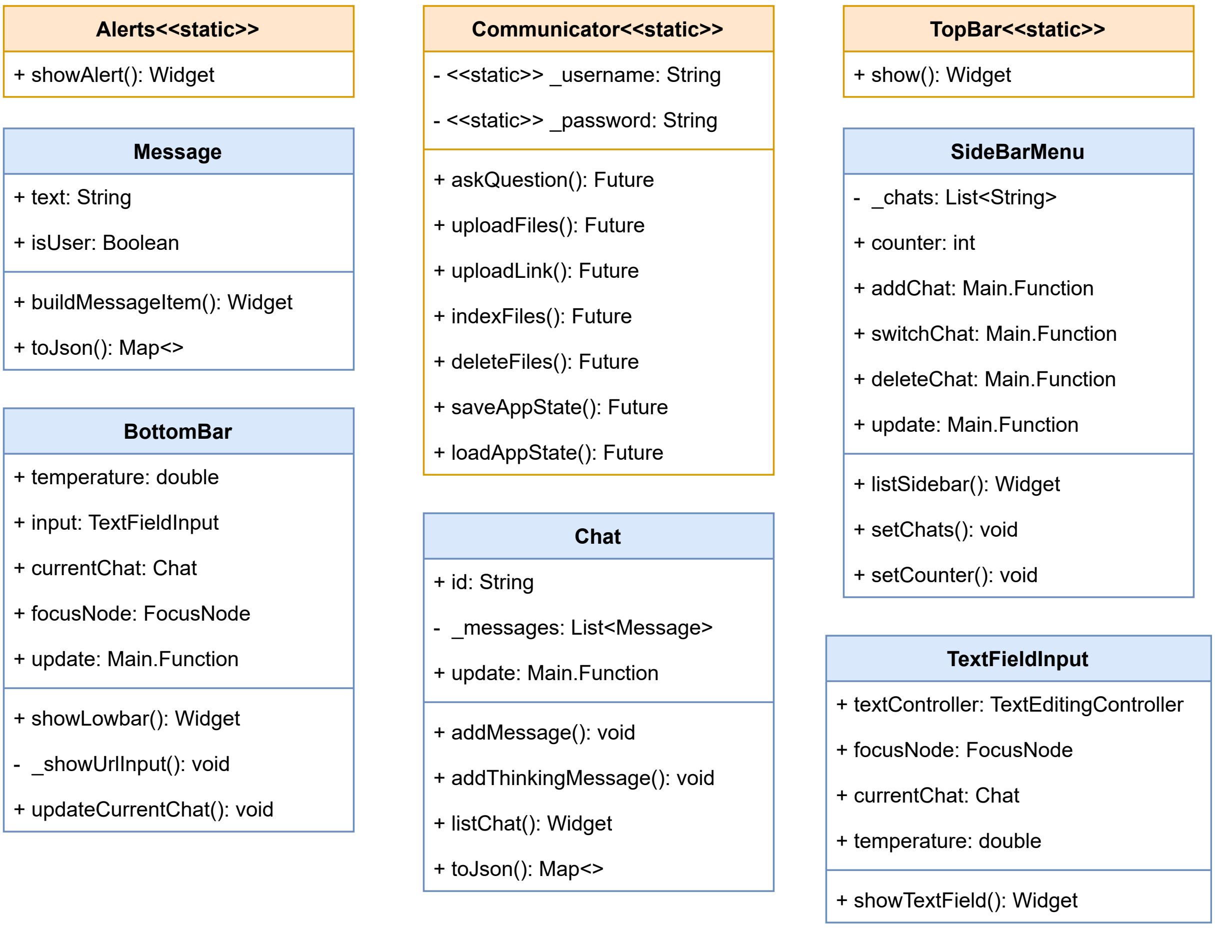

Figure 3: Global Schema

5.2 Use cases Diagram

In this section I describe the use cases, their dependences and the actors that interact with them. In the case of my proposed application, I can only contemplate a 'user' actor, as shown in section 5.2.

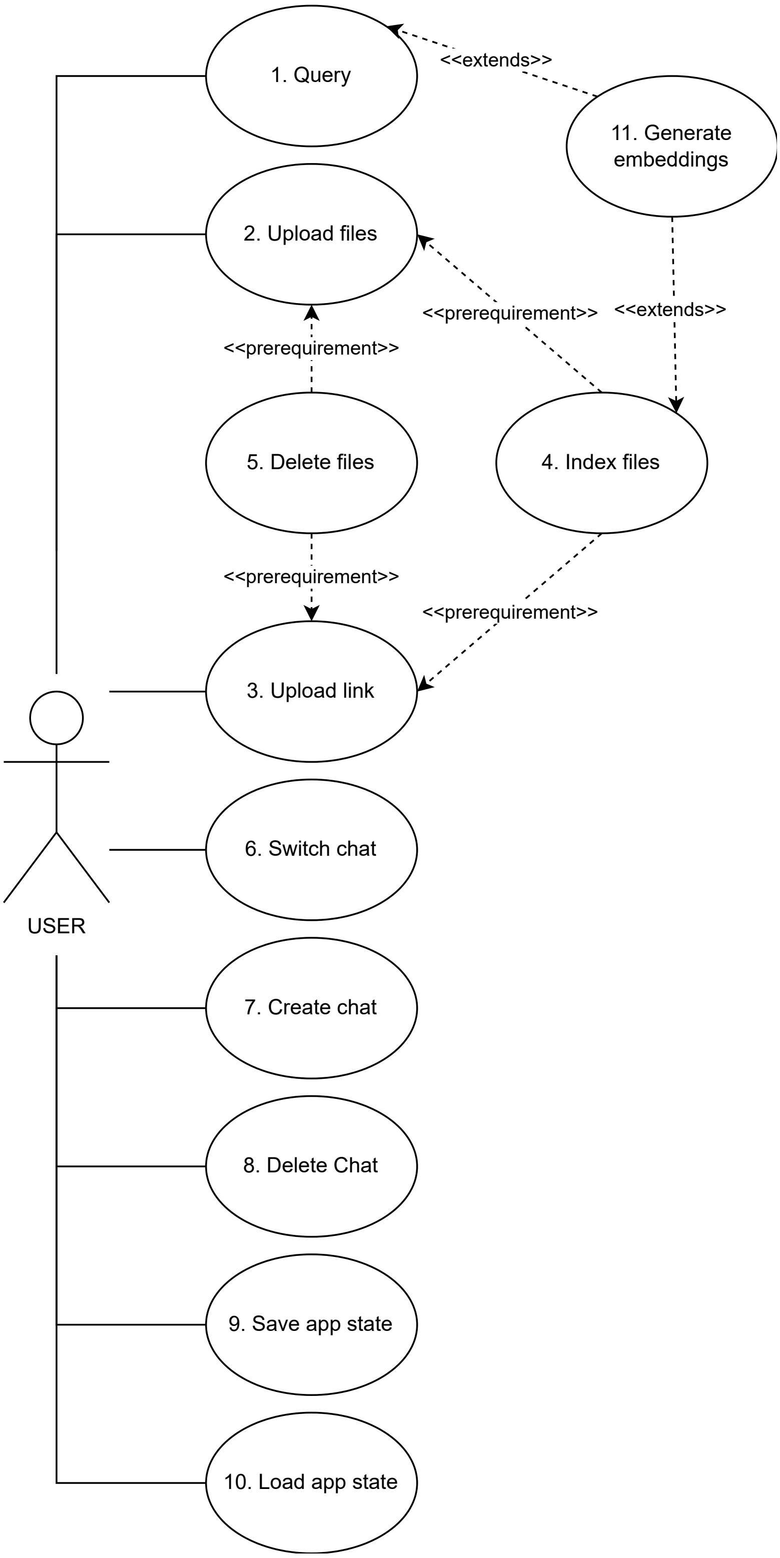

Figure 4: Use cases Diagram

5.2.1 Use case #1 Diagram

In this section I explain the use case number 1, via textual specification and with an additional diagram in fig. 5.

Functionality summary:	Allows the user to ask a question and receive a response from the chatbot.
Input parameters:	User query text
Output parameters:	Chatbot response text, Alerts for errors or status updates
Users:	User
Precondition:	User is interacting with the chatbot interface.
Postcondition:	The user receives an appropriate response to their query.
Main normal process:	1. User types a query in the text field and presses enter. 2. The chatbot (Communicator) processes the query. 3. The chatbot adds a thinking message indicating processing. 4. The chatbot returns a response to the user. 5. The chatbot adds the response to the chat.

Process alternatives and exceptions: **2a.** There is a client error (e.g., network issue).

2a1. An alert is shown to the user indicating the error.

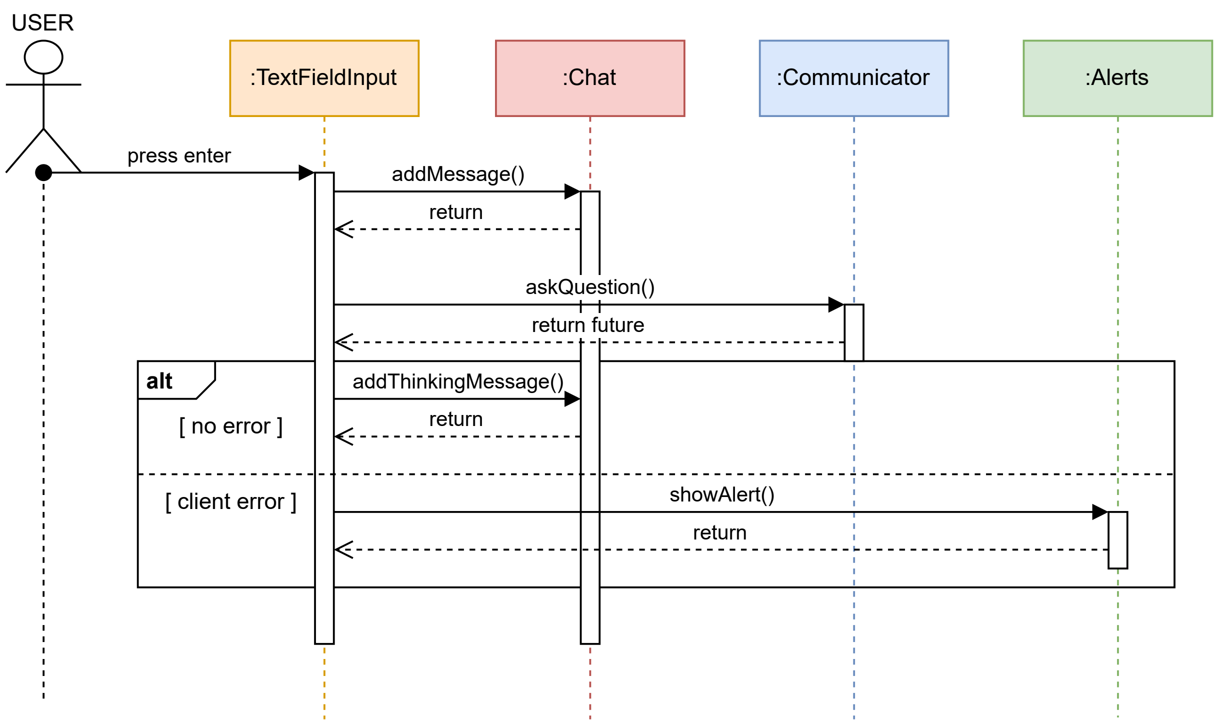

Figure 5: Query

5.2.2 Use case #2 Diagram

In this section I explain the use case number 2, via textual specification and with an additional diagram in fig. 6.

Functionality summary: Allows the user to upload files to the chatbot system.

Input parameters: Files to be uploaded

Output parameters: Upload status, Alerts for errors or status updates

Users: User

Precondition: User has files ready to upload.

Postcondition: Files are uploaded to the system, and the user is notified of the status.

Main normal process:

1. User presses the '+' button to initiate file upload.
2. The BottomBar component handles the uploadFiles function.
3. The Communicator processes the file upload.
4. The chatbot returns the status of the upload.
5. An alert is shown to the user indicating success or failure.

Process alternatives and exceptions: **3a.** There is a client error during file upload.

3a1. An alert is shown to the user indicating the error.

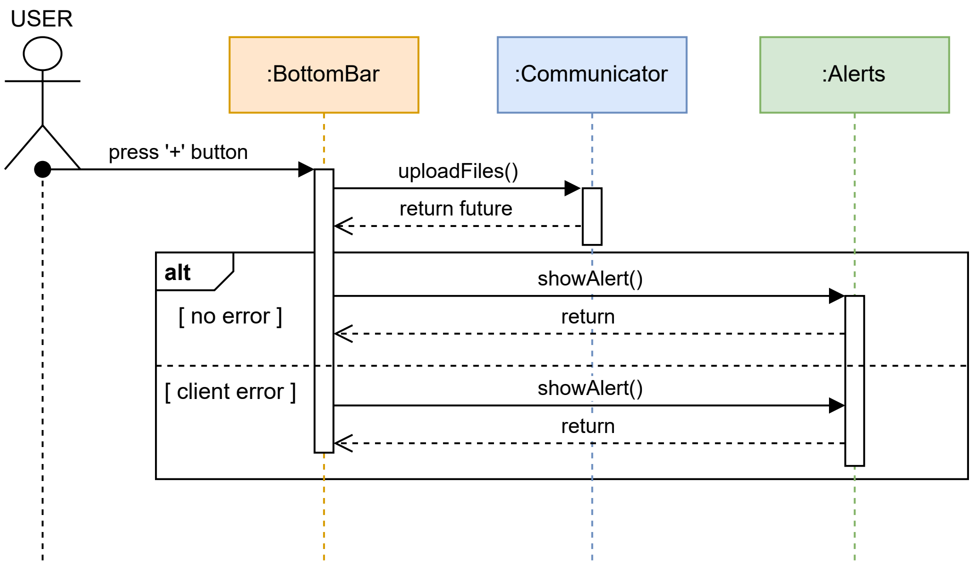

Figure 6: Upload files

5.2.3 Use case #3 Diagram

In this section I explain the use case number 3, via textual specification and with an additional diagram in fig. 7.

Functionality summary: Allows the user to upload content from a web link to the chatbot system.

Input parameters: URL to be uploaded

Output parameters: Upload status, Alerts for errors or status updates

Users: User

Precondition: User has a URL ready to upload.

Postcondition: Web content is uploaded to the system, and the user is notified of the status.

Main normal process:

1. User presses the link button to initiate URL upload.
2. The BottomBar component shows the URL input field.
3. The user enters the URL and submits it.
4. The Communicator processes the URL upload.
5. An alert is shown to the user indicating success or failure.

Process alternatives and exceptions: **4a.** There is a client error during URL upload.

4a1. An alert is shown to the user indicating the error.

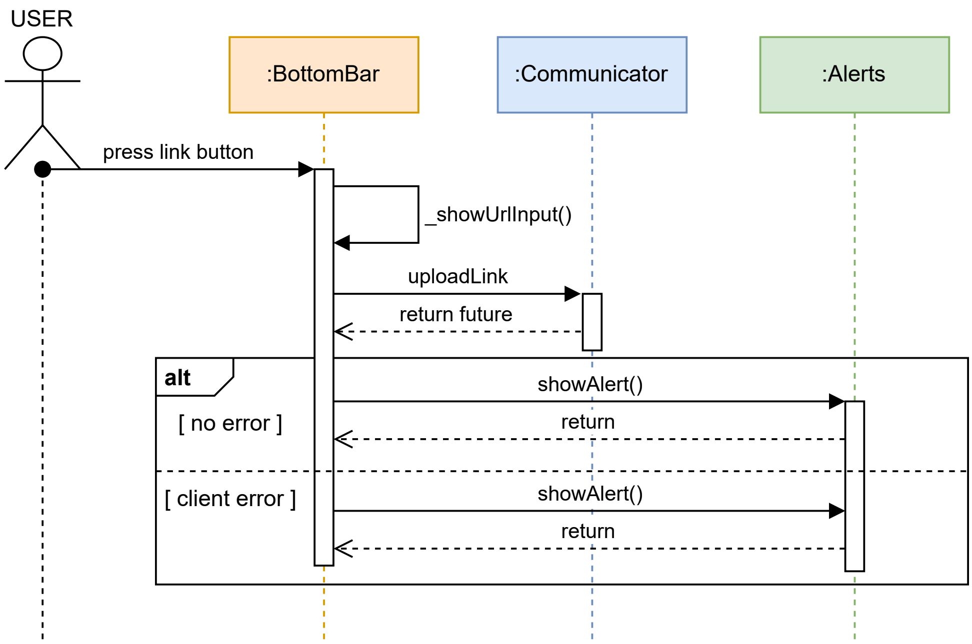

Figure 7: Upload link

5.2.4 Use case #4 Diagram

In this section I explain the use case number 4, via textual specification and with an additional diagram in fig. 8.

Functionality summary: Allows the user to index files within the chatbot system.

Input parameters: Files to be indexed

Output parameters: Indexing status, Alerts for errors or status updates

Users: User

Precondition: Files are uploaded to the system.

Postcondition: Files are indexed, and the user is notified of the status.

Main normal process:

1. User presses the index button to start the indexing process.
2. The BottomBar component handles the indexFiles function.
3. The Communicator processes the file indexing.
4. An alert is shown to the user indicating the status of indexing.

Process alternatives and exceptions: **3a.** There is a client error during file indexing.

3a1. An alert is shown to the user indicating the error.

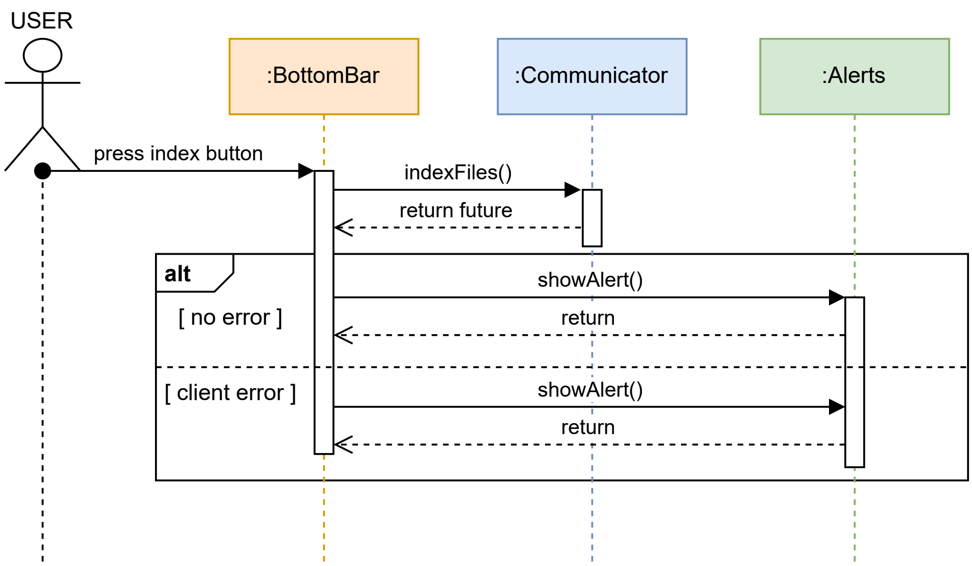

Figure 8: Index files

5.2.5 Use case #5 Diagram

In this section I explain the use case number 5, via textual specification and with an additional diagram in fig. 9.

Functionality summary: Allows the user to delete files from the system.

Input parameters: Files to be deleted

Output parameters: Deletion status, Alerts for errors or status updates

Users: User

Precondition: Files are present in the system.

Postcondition: Files are deleted, and the user is notified of the status.

Main normal process:

1. User presses the delete button to start the file deletion process.
2. The BottomBar component handles the deleteFiles function.
3. The Communicator processes the file deletion.
4. An alert is shown to the user indicating the status of deletion.

Process alternatives and exceptions: **3a.** There is a client error during file deletion.

3a1. An alert is shown to the user indicating the error.

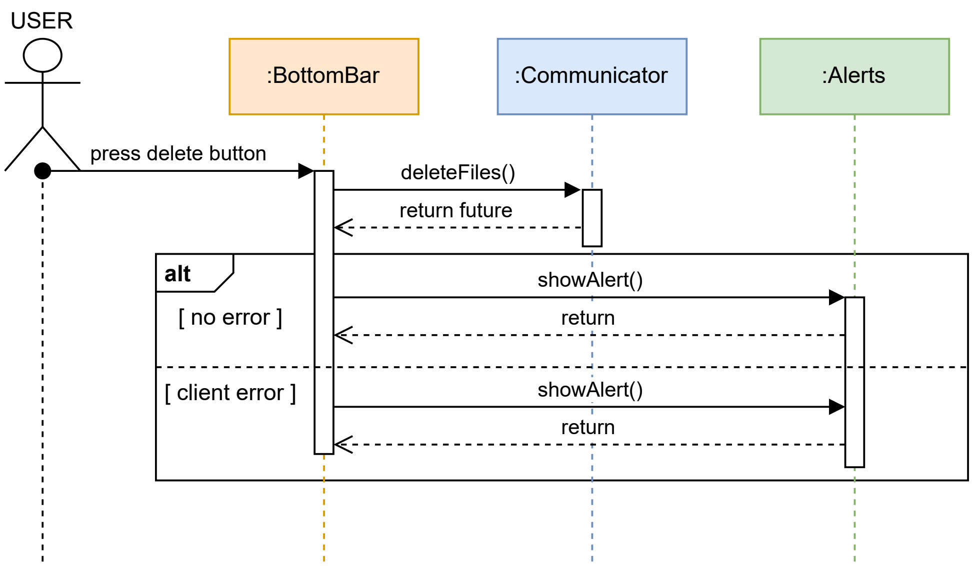

Figure 9: Delete files

5.2.6 Use case #6 Diagram

In this section I explain the use case number 6, via textual specification and with an additional diagram in fig. 10.

Functionality summary: Allows the user to switch between different chat sessions.

Input parameters: Chat session identifier

Output parameters: Updated chat interface

Users: User

Precondition: Multiple chat sessions are available.

Postcondition: The user is switched to the selected chat session.

Main normal process:

1. User presses the fold/unfold button in the TopBar to open the navigation rail.
2. User selects a different chat from the SideBar.
3. The SideBar handles the switchChat function.
4. The chat interface is updated to the selected chat session.

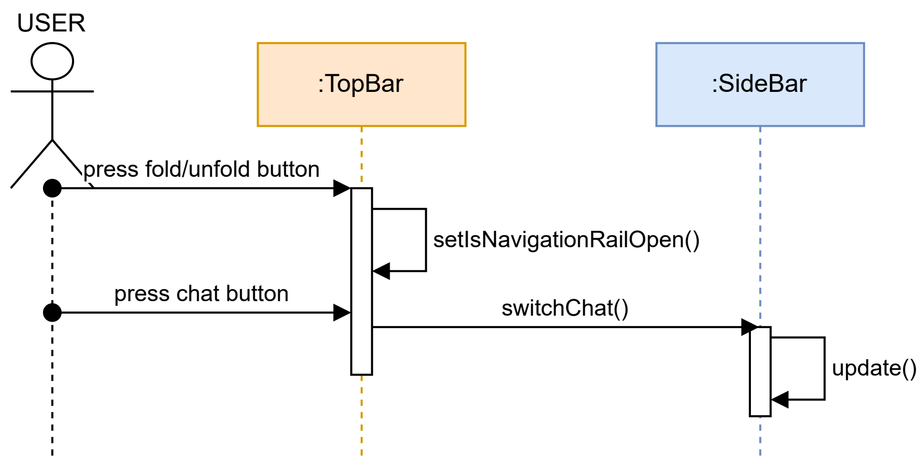

Figure 10: Switch chat

5.2.7 Use case #7 Diagram

In this section I explain the use case number 7, via textual specification and with an additional diagram in fig. 11.

Functionality summary:	Allows the user to create a new chat session.
Input parameters:	New chat name or identifier
Output parameters:	New chat session, Updated chat list
Users:	User
Precondition:	User is logged into the system.
Postcondition:	A new chat session is created and added to the list of available chats.
Main normal process:	1. User presses the fold/unfold button in the TopBar to open the navigation rail. 2. User presses the '+' button in the SideBar to create a new chat. 3. The SideBar handles the addChat function. 4. A new chat session is created and switched to. 5. The chat list is updated to include the new chat session.

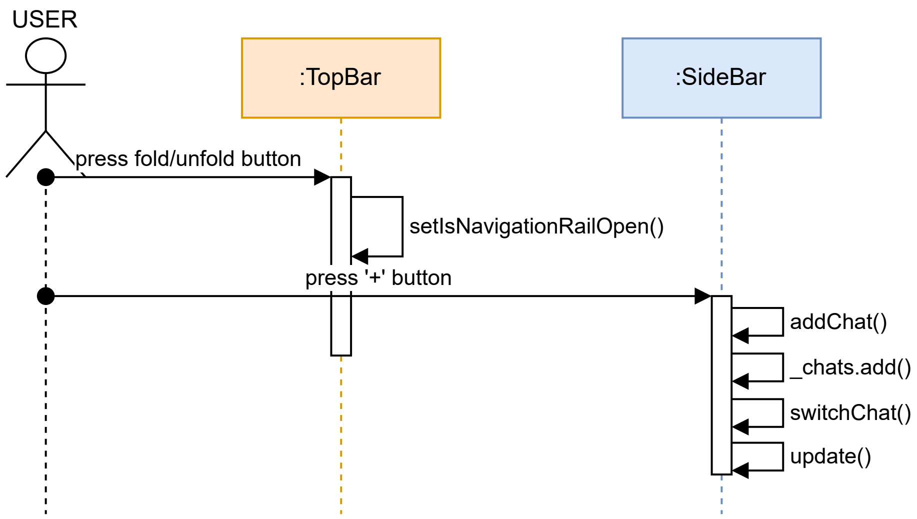

Figure 11: Create chat

5.2.8 Use case #8 Diagram

In this section I explain the use case number 8, via textual specification and with an additional diagram in fig. 12.

Functionality summary: Allows the user to delete an existing chat session.

Input parameters: Chat session identifier to be deleted

Output parameters: Updated chat list, Confirmation message

Users: User

Precondition: User has one or more chat sessions.

Postcondition: The selected chat session is deleted and removed from the chat list.

Main normal process:

1. User presses the fold/unfold button in the TopBar to open the navigation rail.
2. User presses the delete button in the SideBar for the chat session to be deleted.
3. The SideBar handles the deleteChat function.
4. The selected chat session is deleted and removed from the chat list.
5. The chat list is updated to reflect the deletion.

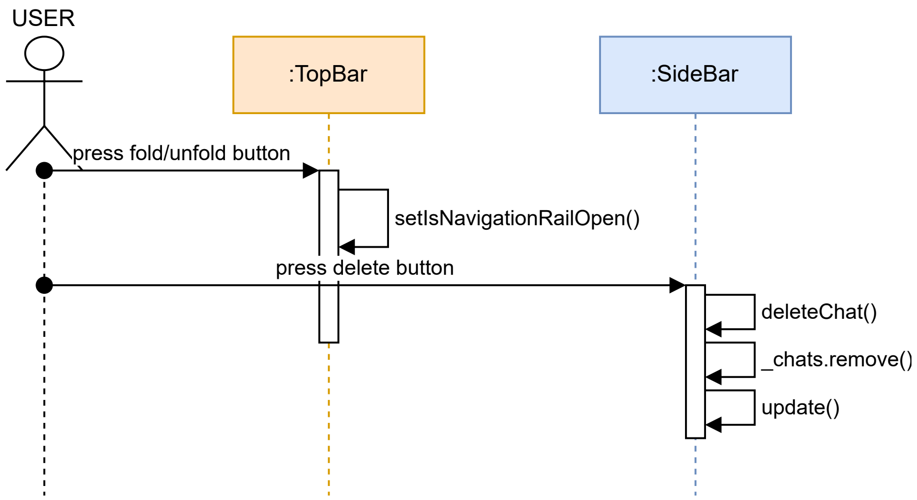

Figure 12: Delete chat

5.2.9 Use case #9 Diagram

In this section I explain the use case number 9, via textual specification and with an additional diagram in fig. 13.

Functionality summary: Allows the user to save the current state of the application.

Input parameters: Current app state data

Output parameters: Save status, Alerts for errors or status updates

Users: User

Precondition: User has made changes to the app state.

Postcondition: The current app state is saved in the database, and the user is notified of the status.

Main normal process:

1. User presses the save button in the TopBar to save the app state.
2. The TopBar handles the saveAppState function.
3. The Communicator processes the save request.
4. An alert is shown to the user indicating the save status.

Process alternatives and exceptions: **3a.** There is a client error during the save process.

3a1. An alert is shown to the user indicating the error.

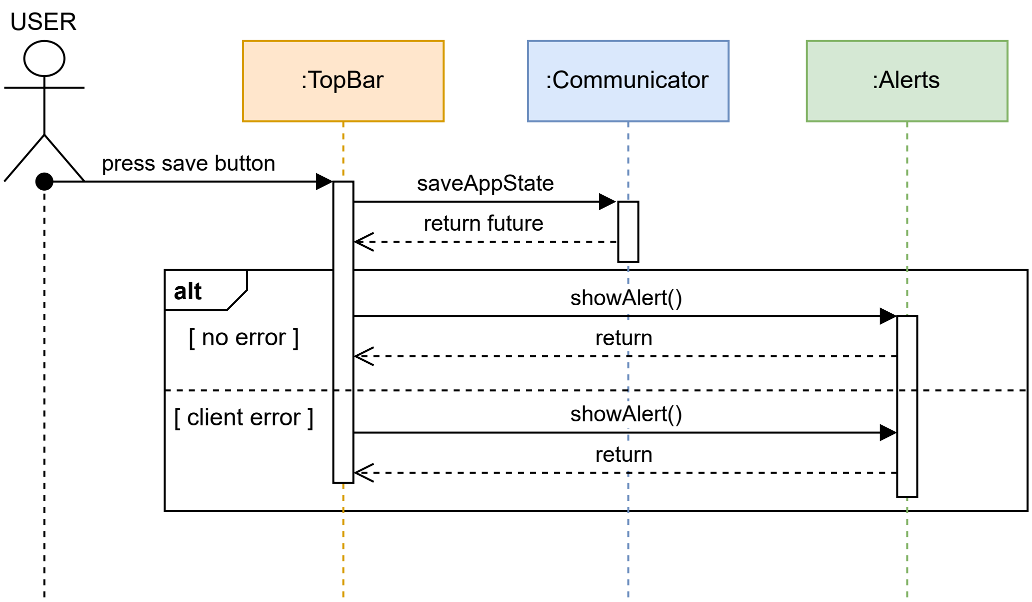

Figure 13: Save app state Schema

5.2.10 Use case #10 Diagram

In this section I explain the use case number 10, via textual specification and with an additional diagram in fig. 14.

Functionality summary: Allows the user to load the previously saved state of the application.

Input parameters: None (app state is loaded automatically)

Output parameters: Loaded app state, Alerts for errors or status updates

Users: User

Precondition: There is a saved app state in the database.

Postcondition: The app state is loaded, and the user is notified of the status.

Main normal process:

1. User enters the app, triggering the loadAppState function.
2. The TopBar handles the loadAppState function.
3. The Communicator processes the load request.
4. The app state is loaded and applied.
5. An alert is shown to the user indicating the load status.

Process alternatives and exceptions: **3a.** There is a client error during the load process.

3a1. An alert is shown to the user indicating the error.

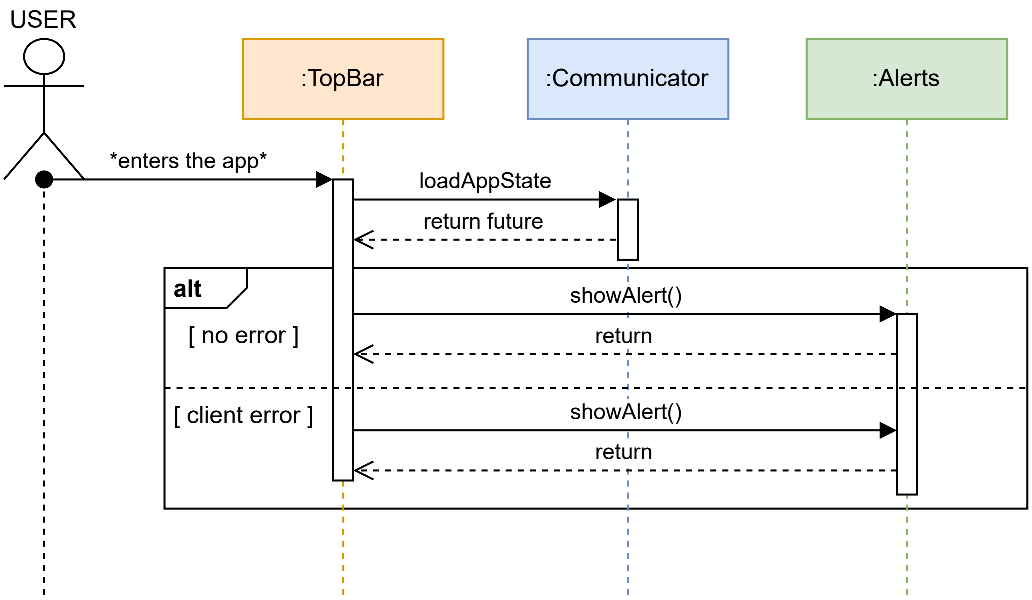

Figure 14: Load app state

6 Theoretical background

In this section I will cover all of the technical and theoretical aspects of the thesis, in order to provide the reader with the necessary knowledge to comprehend the proposed solution and all the reasoning behind it.

6.1 Large Language Models

A large language model (LLM)[21] is a sophisticated language processing system built upon a neural network architecture with a vast number of parameters. These models, they are usually trained over a large amount of data following unsupervised learning methods. The primary goal of an LLM is to understand and generate human-like text by learning the statistics and patters from the training data. Unlike traditional models, LLMs leverage the abundance of textual data available on the internet to autonomously learn linguistic structures and semantics. This capability is what allows LLMs to perform natural language processing (NPL) tasks, such as text generation, summarization and translation, or even answering specific questions about some data with a high degree of accuracy and fluency.

Thousands of public LLMs are currently available in various external repositories, facilitating widespread access and adoption in the research and development community. One prominent example is the HuggingFace[10] platform, which serves as a comprehensive repository and hub for pre-trained language models. HuggingFace provides an extensive collection of models that can be easily downloaded and integrated into various applications. These models range from general-purpose language models to specialized models fine-tuned for specific tasks or domains.

It also supports a wide array of NLP tasks and offers tools for seamless integration, such as the transformers library, which provides a user-friendly interface for accessing and utilizing these models despite not being used on this thesis. Additionally, HuggingFace encourages community contributions, allowing users to share their fine-tuned models and datasets, making the site a more free and open-sourced facility for developers and researchers.

Another key dynamic feature about LLMs is the temperature parameter, which basically controls the randomness of the generated text. It is a crucial parameter that influences how conservative or creative the model's responses are.

If we take a look at a more concise definition, we can see that it is stated that:

> The temperature[1] parameter is a scalar value that adjusts the probability distribution over the model's vocabulary during text generation. In terms of range, the temperature is typically a positive real number, ranging from close to 0 up to 2, depending on the model.

During text generation, the model computes output probabilities prior to temperature adjustment, employing a softmax function applied to the logits, representing raw output scores for each possible next token. Temperature then modulates these logits before softmax application.

Low Temperature (<1): When temperature is less than 1, the model's output distribution becomes sharper. This makes high-probability tokens even more likely while reducing the chances of lower-probability tokens. The result is more deterministic and focused text generation.

Temperature = 1: When temperature is exactly 1, the model generates text based on the raw probabilities given by the logits maintaining its default behavior without any additional scaling. This equilibrium state preserves the inherent characteristics of the model's responses.

High Temperature (>1): When temperature is greater than 1, the output distribution becomes flatter. This broader spectrum enhances the chances of selecting less probable tokens, encouraging a richer and more imaginative array of text outputs. However, this may come at the expense of coherence, potentially leading to less structured or coherent responses.

6.2 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG)[22] is an advanced method in natural language processing that combines the strengths of retrieval-based and generative models. The primary objective of RAG is to enhance the quality and accuracy of responses in various applications, such as chatbots, question-answering systems, and content creation tools. Unlike traditional generative models, which rely solely on pre-trained knowledge, RAG incorporates an additional retrieval step. This means that the model can access a vast database of documents or knowledge sources to find relevant information before generating a response. By doing so, it ensures that the generated content is not only contextually relevant but also factually accurate and up-to-date.

The RAG framework operates in two main stages: retrieval and generation. During the retrieval stage, the model searches through a pre-indexed collection of documents to find the most pertinent pieces of information related to the input query. This step is typically powered by vector databases that efficiently handle large-scale data searches using embeddings.

Embeddings are numerical representations that capture the semantic meaning of the data. It usually consist of a vector, where each position has a different value, which is typically a numerical value, and corresponds to an attribute of the represented object.

In the generation stage, the model takes the retrieved documents and uses them to inform and guide the generation of its response. This hybrid approach allows RAG models to produce more informative and context-aware answers, bridging the gap between static knowledge bases and the dynamic needs of users. By leveraging retrieval-augmented generation, developers can create more robust and adaptable AI systems that excel in a wide range of practical applications.

In this proposed solution, RAG will be addressed by using vector databases and generated embeddings.

6.3 Vector Database

One of the main keys to the operation of the proposed solution is the Vector Database [9]. In a vector database, information is stored in a way that enables efficient similarity searches and clustering. By representing data points as vectors in a high-dimensional space, we can leverage mathematical techniques to find and group similar items.

Conceptually speaking, we can associate this as a clustering model, where data points (or chunks of information) that are close to each other in the N-dimensional space are considered similar. The proximity of these points is determined by the values of their respective attributes, allowing the database to quickly identify and retrieve related data based on their spatial relationships.

To further illustrate, imagine each data point in the database as a unique position within a vast multi-dimensional landscape. The coordinates of each point are defined by its vector, and the distance between any two points indicates their similarity. For instance, in a database storing images, each vector might represent various features of an image such as color, texture, and shape. Images with similar features would be located near each other in this space, enabling efficient retrieval of similar images through simple spatial queries.

Hereby, in fig. 15 is an illustrated and simple example of the concept explained, of what would be a query of a `kitten` in a clustering environment, and how the query would be placed in the animals area, nearby the `cat` class:

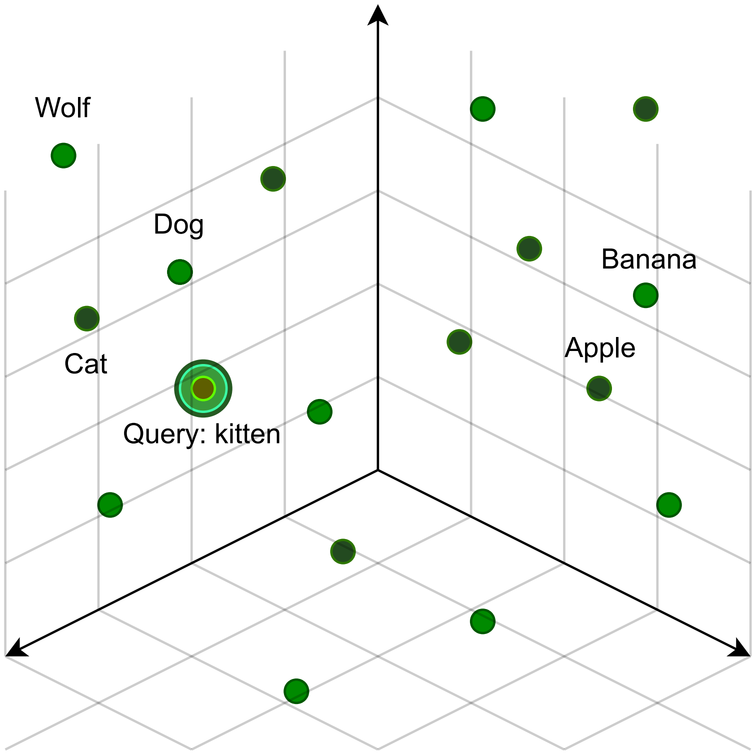

Figure 15: Vector database space representation.

The data indexing process begins by creating embeddings for the files or data to be stored. The process starts with partitioning the data into chunks of a certain size. Each chunk is then sent to a LLM that has been specially trained to generate these embeddings. The LLM processes each chunk separately, producing a vector of N attributes, where each attribute has a distinct value representing a specific feature of the data. Once the vector is generated, it is paired with the corresponding data chunk and stored in the vector database.

When querying the model using the indexed data, the process is repeated with the query text. The query is partitioned into chunks if necessary, and each chunk is sent to the same specific LLM to generate the corresponding vectors. These query vectors are then compared to the indexed vectors in the vector database to identify the most similar ones. The similarity comparison leverages the spatial properties of the vectors in the N-dimensional space, allowing for efficient retrieval of the relevant data chunks.

Upon identifying the most similar vectors, the associated data chunks are returned as the query results. This approach ensures that the most contextually relevant information is retrieved, and then given to the querying LLM model as context for the user's question, enhancing the accuracy and relevance of the responses. Below in fig. 16 is a schematic of the data processing and indexing pipeline, representing the flow from data embedding to query response.

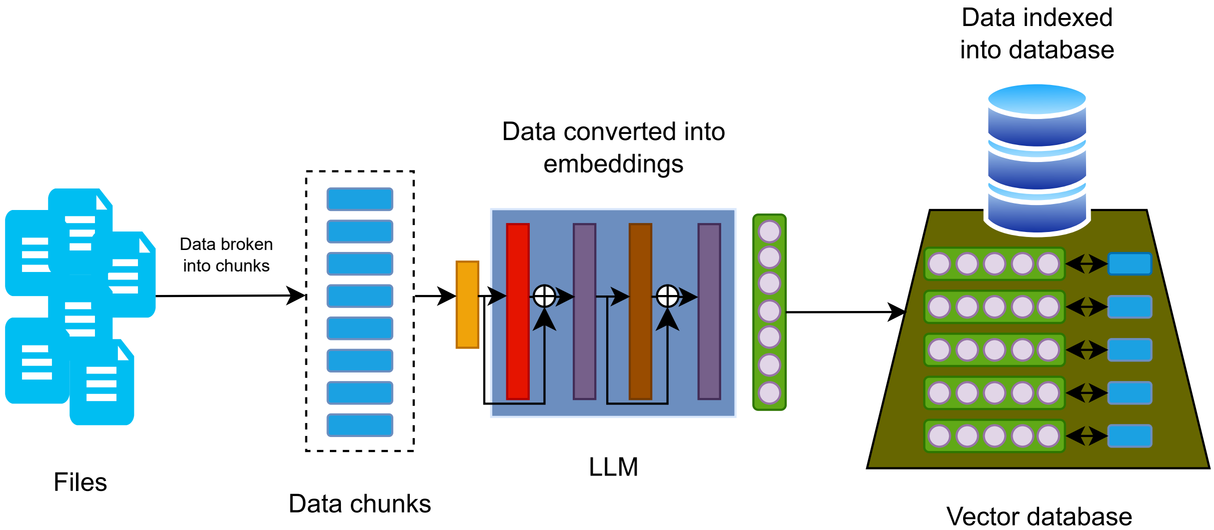

Figure 16: Embedding generation pipeline.

One of the key strengths of this pipeline is the ability to provide meaningful data as context to the model and therefore answer the specific questions the users ask, even if the topic of discussion is not publicly available or subsequent to the model training date.

Another key strength is its feature to provide validation and reasoning to the given answer. The explanatory model enhances transparency and justification by citing original sources for the retrieved results. This is made possible by the metadata associated with the data chunks in the vector database, which includes the source of the information. When the data is retrieved, this metadata is also returned, allowing the system to list these sources in its response to the user.

7 Used technologies

In the subsections below I will cover which backend implementations I used to cover the parts aformentioned in section 6.

7.1 Large Language Model

For implementing the LLM model in my project, I chose **LocalAI** solution. LocalAI[17] is an open-source, easy to use framework designed to interact with almost any LLM model locally by using OpenAI's API. One of its key advantages is its support for multiple *backends*, making it versatile and adaptable for various use cases. Some of these backends for running LLMs are:

- llama.cpp
- bert
- whisper
- stablediffusion
- langchain-huggingface
- piper
- bark
- exllama
- dolly.

In fig. 17 you can observe how LocalAI works in a more detailed manner. The reader can observe it deploys an OpenAI-like API to interact with the LocalAI service. This will manage the available LLMs using different backends, depending on the needs of each LLM model.

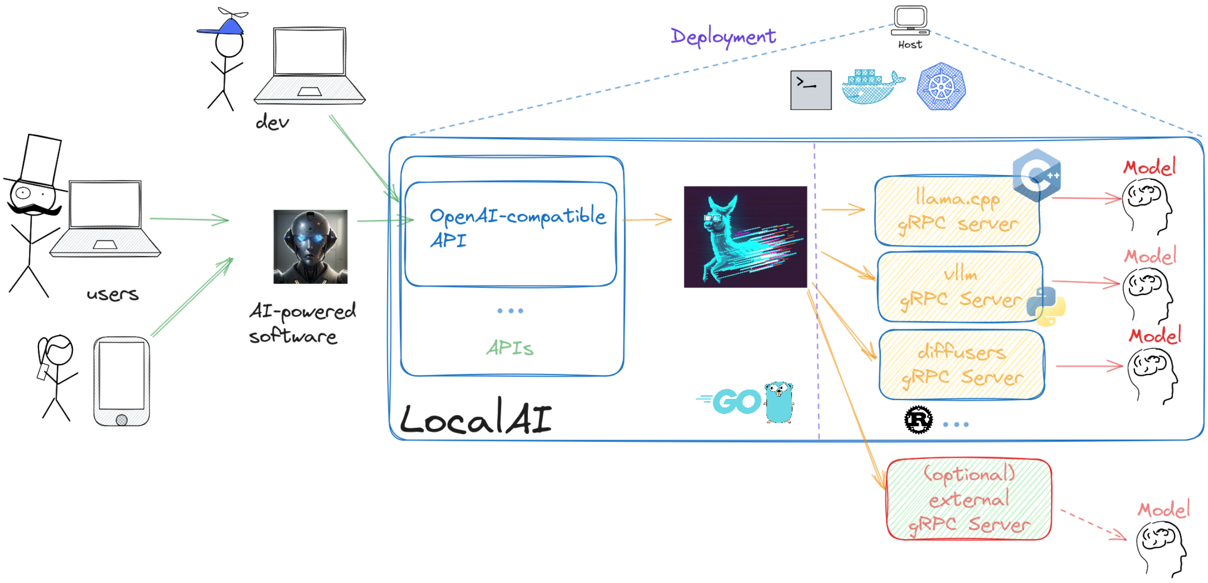

Figure 17: LocalAI architecture representation. Extracted from [14]

Given the constraints of running the model on a CPU-only setup, I opted for the **llama.cpp** backend. This backend is open-source, fast and privacy-focused, aligning with the project's requirements. Moreover, it is integrated with the **LangChain framework**, which I will discuss in section 9.2. The **llama.cpp** backend offers several model options available for download, all of which would be valid and suit my necessities. For the demonstration purposes of this work, I selected the **nous-hermes-llama2-13b.Q8_0.gguf** model.

It is important to note that running a model solely on a CPU results in **lower performance** compared to GPU-accelerated models. GPUs are specifically optimized for the **parallel processing** tasks required in neural network computations, leading to significantly faster performance. However, exploring GPU optimization falls outside the scope of this thesis. Also, the provided developing infrastructure did not dispose of GPU-oriented nodes, but only large CPU processing nodes with also loads of memory (which is one of the big requirements to run those kind of models on hardware).

For generating the aformentioned embeddings, I will also deploy a **bert** backend based model: **mudler/all-MiniLM-L6-v2** [16]. This model specializes in the generation of the vectors that will be stored in the vector database.

7.2 Vector Database

To get the *RAG* behaviour on the backend *pipeline* I chose to go with **vector databases**, since they offer great performance for the required stages of RAG technique.

There are numerous open-source, high-performance vector databases available in the market today, each offering unique features and capabilities. Among the most reputable options are **Pinecone**, **Qdrant**, and **ChromaDB**. After some consideration, I have opted to implement ChromaDB as the vector database solution for this project. The decision to choose ChromaDB[6] was influenced by several factors, including its robust features, scalability, and compatibility with the project's requirements.

Langchain[18] is a *framework* designed to streamline the development of applications utilizing LLMs, simplifying complex tasks such as integration and data pre-processing. By providing a suite of tools for pre-processing input data and post-processing model outputs, Langchain facilitates seamless interaction with LLMs. For instance, it offers functions for data cleaning, text tokenization, and formatting model outputs into user-friendly formats.

Moreover, Langchain offers **compatibility** with various vector databases (in which we can find the already mentioned ChromaDB). This provides developers like me the flexibility to choose the most suitable option for their specific requirements, allowing for efficient data management and retrieval processes.

As an open-source project, Langchain benefits from comprehensive documentation and an active community of developers contributing to it. Additionally, Langchain's compatibility with popular models such as OpenAI's ChatGPT and its API further enhances its utility and appeal for this work, since the compatibility with LocalAI's API.

With LocalAI facilitating model communication, ChromaDB managing vector data storage, and Langchain streamlining development tasks, the combined solution offers a versatile and efficient platform for building intelligent applications. All in all, it becomes also free and open-source, as well as providing a more than desirable perfect integration between them out of the box.

7.3 Application Database

Given the JSON-like nature of the information handled by the web application, implementing a NoSQL database would provide the necessary flexibility and scalability, appart from the fact that such a simple demo application does not require the design of tables and relations.

NoSQL databases are designed to handle unstructured or semi-structured data, making them ideal for applications dealing with JSON-like data formats. The key advantages of NoSQL databases include:

1. **Flexibility**: Schema-less data models allow for easy adjustments as application requirements evolve.
2. **Scalability**: Horizontal scaling capabilities to handle large volumes of data and high traffic.
3. **Performance**: Optimized for read and write operations, often resulting in faster access times.

The main NoSQL database that I have seen during my degree studies has been Redis, which is an *in-memory* database with the basic PUT GET DELETE queries. One big disadvantage of Redis is that is single-threaded, which means that it only uses one CPU of your hardware. That might be a bottleneck in some applications, for which exists Dragonfly. Dragonfly is an open-source, multi-threading alternative to Redis that performs way better.

I took both options into consideration, but since the amount of queries will be minimum, Redis will not bottleneck the performance and therefore the high-performant Dragonfly will not be needed.

Another database I found interesting has been CouchDB[2]. This alternative runs as a service and can be interacted with via an API of its own. It also supports disk-persistence, despite that the performance is therefore reduced in comparison with like the other two.

Each database mentioned before has its unique strengths and potential drawbacks, but since Redis and Dragonfly are *memory-oriented* and I aim to have persistence even between boots, I expect persistence to disk. Therefore, **CouchDB** is the answer.

I want to note that, despite being a NoSQL database, which usually have BASE properties, CouchDB offers **ACID** (Atomicity, Consistency, Isolation, Durability) properties.

On-disk, CouchDB never overwrites committed data or associated structures, ensuring the database file is always in a consistent state. In these cases, the data must be first deleted given their unique ID, to then upload the newer data afterwards.

Document updates (ADD and DELETE) are serialized, except for binary blobs which are written concurrently. Database readers are never locked out and never have to wait on writers or other readers.

CouchDB also uses *B-trees* to *index* documents by their ID and Sequence ID, aiding in efficient change tracking. The append-only update approach ensures that even during document updates, the database remains consistent.

Unlike traditional SQL databases, CouchDB stores data in semi-structured documents, which avoids the complexity of maintaining multiple tables and rows found in traditional databases. For querying and organizing this semi-structured data, CouchDB employs a dynamic view model. Views allow for the aggregation, joining, and reporting of data on-demand without altering the underlying documents. These views are defined in special design documents, which can also replicate across databases, ensuring that both data and application logic are consistently replicated.

7.4 Other technologies

In all these sections I talked about technologies I just deployed to use at some point of the application. Here, I will briefly talk about other technologies I used to develop myself the rest of the structure.

7.4.1 Front-end Implementation

For more versatility, I decided to create a **web application**, so that it can run on any operating system. I have used the Flutter[13] framework to develop my design, inspired by the current ChatGPT web solution. **Flutter** is a front-end framework that uses a programming language called **Dart**. Since I intend to have stateful element on the interface, I find mandatory to develop it using the OOP rather than a functional paradigm language. It is also really easy to use, and it lets you export to HTML, CSS and Javascript.

7.4.2 Communication and relation

To enable the communication between the client and the vector database, I used the Flask package to create a custom API that encapsulates the defined functions in the vector database's Python script.

Flask[12] is a micro web *framework* for Python that simplifies the creation of web applications. It listens for incoming HTTP requests and maps these requests to the appropriate functions based on the URL route specified. When a request is received, Flask executes the corresponding function and returns the response data in JSON format to the client. This process allows seamless integration and communication between the client and the server, making it an ideal choice.

7.4.3 Dockerization

To ensure flexibility and ease of **deployment** across diverse environments, I have planned to containerize each component individually using **Docker**. This enables users to easily set up and manage each element according to their specific requirements.

8 Design

Here I will display some global and detailed graphs about the designed structure for the proposed solution. The main goal is to provide graphical, easy to read and process information to the reader so that it can comprehend all of the structure.

8.1 Global Schema

The design of the environment is very simple and straightforward. The primary mode of interaction is through an API, which can be accessed in two ways: through a GUI or by manually sending POST and GET requests. The GUI provides a user-friendly interface for those who prefer a visual approach, and better suits my testing and demonstration purposes. The following fig. 18 illustrates the overall architecture and interaction flow between the user interface and the back-end API:

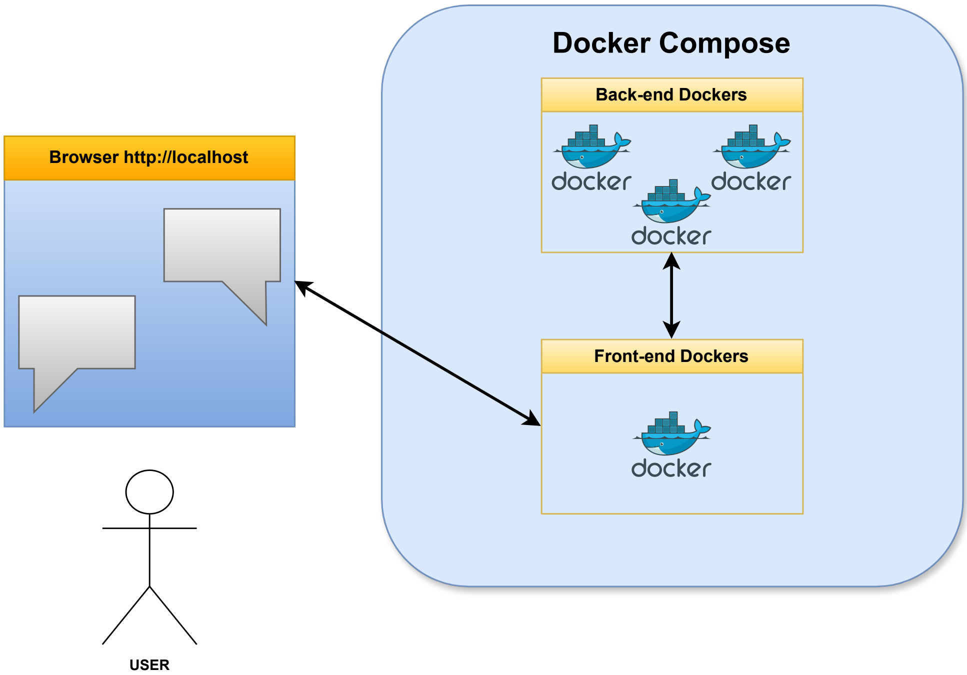

Figure 18: Global Schema

8.2 Backend Schema

The GUI API primarily interfaces with the vector database service. This service is crucial as it interacts with the large language models (LLMs) to generate vector embeddings from textual data, which are then stored in the vector database. Additionally, the vector database service handles the task of passing user queries to the LLMs along with relevant context data, ensuring accurate and contextually aware responses.

In addition to this primary interaction, the GUI also connects directly to the CouchDB database using CouchDB's own API. This connection is essential for persisting the state of the application, which includes storing all user chats and their respective messages. This dual interaction ensures that the application operates smoothly and retains critical user interaction data. The following fig. 19 illustrates this architecture:

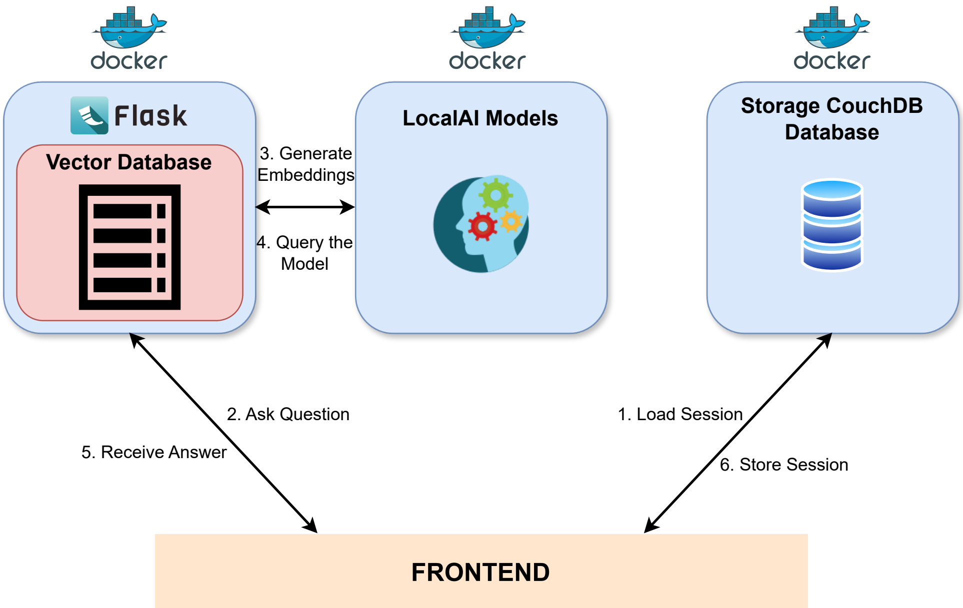

Figure 19: Back-end design

8.3 Frontend Schema

The Graphical User Interface (GUI) design for this application will resemble the ChatGPT website layout. A side panel, positioned on the left-hand side, serves as a navigation hub. From this panel, users can select existing conversations or initiate new ones, providing a way to manage multiple conversations.

Occupying the central portion of the interface is the chat window, where messages from the currently active conversation are displayed in a clear and organized manner. This area will be designed to ensure readability and ease of interaction, supporting the dynamic flow of conversation between the user and the application.

At the bottom of the interface, the input field is placed at the center, allowing users to type their queries. Alongside this input field are several buttons that facilitate interaction with the back-end functionalities. These buttons will enable users to perform tasks such as uploading context files to a vector database, deleting or indexing these files, and adjusting model parameters like temperature.

This arrangement of elements ensures that users have all necessary tools at reach, while copying the current ChatGPT layout facilitates adaptation for its users. The fig. 20 below provides a visual representation of the described interface, showing the layout of each individual component in greater detail.

Figure 20: Front-end design

9 Implementation

In this section of the thesis, I will detail the implementation process of my solution, including the technologies I decided to use and the reason of my decisions.

Before delving deeper into the technical and detailed aspects of this section, it is important to note that all necessary files required to run the pipeline, along with the source code for each component, will be made available in my public repository on GitHub. For those interested in accessing these resources, please refer to [5] for further details. This repository will include comprehensive documentation, ensuring that all steps and processes are transparent and reproducible, aiming to facilitate easier replication and validation of the findings presented in this thesis.

9.1 Back-end Implementation

9.1.1 LLM

To implement LocalAI, I pulled the base public image from DockerHub localai/localai:v2.16.0-ffmpeg-core [19]. Once downloaded, I had to start a container with this given command:

```
docker run --net=host --detach --name cllic localai/localai --threads 40
```

If the reader is willing to replicate the application, he must know that he can use any other model by running a bash in the running container and pulling a new model with the following command:

```
wget https://gpt4all.io/models/ggml-gpt4all-j.bin -O ggml-gpt4all-j
```

Models can also be downloaded from the LocalAI gallery, which also runs on the pulled Docker. In addition, there are many options available on the HuggingFace portal.

Complementary to this, users have the flexibility to customize each model by writing `yaml` configuration files, which enable detailed adjustments, model fine-tuning or even merging multiple models at once.

It is mandatory to restart the container in order for the newer models to work. In any case, the documentation for LocalAI is added in the references section of this thesis.

In my case, as you may have noticed, I indicated the localAI the number of vCPUs I wanted to assign to the model, and in order to increase performance I selected the majority of the availables in the computing node I was running the container in.

9.2 Vector Database

To implement the Langchain vector database, I used ChromaDB library, as long as sub-dependencies of the Langchain python library, such as OpenAIEmbeddings. Here is where the compatibility of LocalAI with OpenAI's API plays a huge role in the development. In the listings below listing 1 and listing 2, the reader can see the two main actions done by the python script covering the vector database:

```
texts = CharacterTextSplitter(chunk_size="size",
↪ chunk_overlap="overlap").split_documents(data)
embedding = OpenAIEmbeddings(model="embegging-model-name",
↪ openai_api_base=base_path)
vectordb = Chroma.from_documents(documents=texts, embedding=embedding,
↪ persist_directory=vectordb_persist_dir)
vectordb.persist()
```

Listing 1: Langchain indexing process

```
embedding = OpenAIEmbeddings(model="embegging-model-name",
↪ openai_api_base=base_path)
llm = ChatOpenAI(temperature="temperature", model_name="answering-model-name",
↪ openai_api_base=base_path)
vectordb = Chroma(persist_directory=vectordb_persist_dir,
↪ embedding_function=embedding)
qa = RetrievalQA.from_llm(llm=llm,
↪ retriever=VectorStoreRetriever(vectorstore=vectordb))
answer = qa.invoke("your question here")
```

Listing 2: Langchain querying process

In section 9.5.1 you can find more information about this custom API. You can also check the source files in the GitHub repository [5].

9.3 Application Database

In order to use CouchDB for the application I also had to pull the public base image from Dockerhub couchdb:latest [3]. It needed some manual preparation in order to properly work for the application. For that case, I created a custom Dockerfile.

It is still something that I wonder, to make the user manually create the base tables for the database to work instead or providing them out of the box. Anyway, just by building that new Dockerfile image, I managed to create a custom image that works out of the box with my application implementation as shown in listing 3:

```
FROM couchdb:latest

COPY data /opt/couchdb/data
```

Listing 3: CouchDB Dockerfile

For this, I had to create a container from couchdb vanilla, create the necessary tables, to therefore copy the created files to my local storage. Then, I could copy those files into the new image.

9.4 Front-end Implementation

The Flutter application is basically composed by these 4 main components:

On one hand, the main page serves as the central hub, showcasing the ongoing conversation with an organized display of all messages exchanged between users.

On the other hand, a side menu provides navigation over the available chats, and also the ability to delete created chat rooms. This feature empowers users to easily switch between different conversation threads, exponentially improving the potential of the pipeline.

The top bar also provides with the application title, a button to fold and unfold the side menu and another button to save the current application state.

Lastly, a convenient bottom bar facilitates user interaction with the vector database while also serving as a command center for inputting questions into the model.

All in all, it basically follows the Design described in section section 8.3 . To be more concisely, hereby I describe all the undetailed buttons that haven't been explained yet:

- A button to delete the selected chat room.
- A slidebar to control the temperature parameter of the model.
- A button to send the question to the pipeline.
- A button to upload files to the vector database.
- A button to attach URLs, so their contents also get uploaded to the vector database.
- A button to index the vector database, generating embeddings for the stored data. This action deletes all the previously stored vectors, resetting the ChromaDB itself.

- A button to delete the uploaded files and contents of the submitted URLs.
- A button to save the actual app state.

Flutter is a front-end framework that uses an Object-oriented programming (OOP) language called Dart. Since I intend to have stateful element on the interface, I find mandatory to develop it using the OOP rather than a functional paradigm language. It is also really easy to use, and it lets you export to html, css and javascript.

The solution establishes that each mainly part of the interface is corresponded a class. Each class will have functions to build the necessary widgets and variables to store the state in-memory. So basically, we have:

- BottomBar class.
- Chat class.
- Message class.
- SidebarMenu class.
- TextField class.
- TopBar class.

I also decided to implement two more classes, which for their inteded purpose, will be static. Their aim is to provide a clean and organized communicnation relation with the back-ends explained in the earlier point. These classes (listed below) will use the APIs I talk about in section 9.5 :

- Alerts class.
- Communication class.

To facilitate the deployment of the frontend, I exported the Flutter project into JSON, HTML, and CSS scripts by using the Flutter tool `flutter build web`. Following the export, I proceeded to create a Dockerfile to containerize the application, selecting a Python 3 image as a baseline.

In the Dockerfile, I incorporated instructions to copy the directory containing the exported files into the container's file system. This ensures that all necessary frontend resources are available within the container environment. To enable the frontend to be served upon the container's startup, I also configured the Dockerfile to execute the command `python3 -m http.server 8989`. This command initiates a simple HTTP server on port 8989, effectively serving the exported files and making the frontend accessible via a web browser.

By using Python's built-in HTTP server, the solution remains lightweight, easy and straightforward, minimizing overhead and maintaining the focus on the main objectives of this thesis.

9.5 Communication and relation

Here I discuss the different APIs designed and implemented by myself in order to communicate extra parts, such as the explained here developed by myself that still don't have a functional API.

9.5.1 VectorDB Flask server

To enable the communication between the client and the vector database, I used the Flask package to create a custom API that encapsulates the defined functions in the vector database's Python script.

Flask is a micro web framework for Python that simplifies the creation of web applications. It listens for incoming HTTP requests and maps these requests to the appropriate functions based on the URL route specified. When a request is received, Flask executes the corresponding function and returns the response data in JSON format to the client. This process allows seamless integration and communication between the client and the server, making it an ideal choice.

By leveraging Flask, I have been able to efficiently build a functional API for interacting with the vector database, ensuring smooth and effective data exchange between the client and server.

For each functionality, I implemented a corresponding function and annotated it with the @app.route() decorator, which defines the URL endpoint for that function.

For example, consider the example shown in listing 4 coming from the deployed script:

```
@app.route('/files/clean', methods=['POST'])
def clean_files():
    os.system('rm storage/*')
    return "Files cleared"

@app.route('/current', methods=['GET'])
def get_current_model():
    return current_model

@app.route('/models', methods=['GET'])
def get_list_models():
    response = 'curl http://localhost:8080/v1/models'
    for model in response['data']:
        available_models.append(model['id'])
    return available_models
```

Listing 4: VectorDB API code snippet

This code snippet sets up various endpoints, each specifying the allowed HTTP methods. When an endpoint is accessed, the associated function is executed, and the server returns an HTTP response with the function's return value as a JSON value.

For example, when a POST request is sent to `http://hostIP:5000/files/clean`, the `clean_files()` function is invoked, which removes all files in the storage directory and returns a confirmation message "Files cleared".

Similarly, sending a GET request to `http://hostIP:5000/current` will trigger the `get_current_model()` function, returning the current model in use.

Finally, a GET request to `http://hostIP:5000/models` calls the `get_list_models()` function. This function queries the local LLM API at `http://localhost:8080/v1/models` to retrieve a list of available models. The response data is then parsed and stored in a list, which is subsequently returned to the user, providing the frontend with the necessary model information.

To start the Flask service, I used the command `flask -app backend run`, which initializes the Flask application and makes it ready to handle incoming HTTP requests. This will run automatically when starting the container, as I will explain on section 9.6 .

In listing 5 there is a simplified full script, with all of the functions and routes listed:

```
app = Flask(__name__)
CORS(app)

@app.route('/files/upload', methods=['POST'])
def upload_files():

@app.route('/from-link', methods=['POST'])
def fromlink():

@app.route('/files/index', methods=['POST'])
def index_files():

@app.route('/files/clean', methods=['POST'])
def clean_files():

@app.route('/current')
def get_current_model():

@app.route('/models')
def get_list_models():

@app.route('/query', methods=['POST'])
def ask():
```

Listing 5: VectorDB API operations list

9.5.2 Flutter HTTP server

To implement the front-end, I exported the Flutter project and migrated it to a separate Docker container with Python installed. Within this container, I configured the default working directory to point to the location of the exported files. Afterwards, whenever the container is started, it runs a simple HTTP server using the command `python3 -m http.server` (similar to using Apache2, as both serve the purpose of running HTTP servers to host web applications), which serves the front-end files and makes the application accessible via a web browser. This method simplifies the deployment over static files, providing a lightweight and efficient solution for the front-end delivery.

9.6 Dockerization

To ensure flexibility and ease of deployment across diverse environments, I have planned to containerize each component individually using Docker, as I already explained before. This enables users to easily set up and manage each element according to their specific requirements. While the design mandates that all components reside on the same machine, it's adaptable for distributed setups via SSH tunneling. Crucially, each container is configured to share the network of the host to avoid IP addresses and ports misconfiguration.

By encapsulating each part within its own container, users have the liberty to distribute the workload across multiple nodes if needed, as already mentioned. This modular setup accommodates instances where certain components of the pipeline demand higher computational resources. Users can dynamically allocate resources and balance the load across various nodes, optimizing performance and resource utilization.

Moreover, for users with a single node capable of handling the entire system's workload, or for ones that want an even easier way to deploy the pipeline, I have crafted a specific Docker Compose YAML file. This file streamlines the deployment process with a single command in a matter of minutes, simplifying the setup even more. This straightforward approach is intended to approach the AI and LLMs to more people, even if they have close to no knowledge on the IT area.

10 Proof of concept

This section also works as the validation of the project. To validate the proposed techniques and demonstrate their effectiveness in enhancing chatbots.

I will talk about the design, implementation, and evaluation of the Proof of Concept (PoC), which involved several steps:

- **Data Preparation**: A dataset composed by various unrelated contexts was prepared. This dataset is used to populate the vector database with relevant and different knowledge.
- **Vector Database Population**: Knowledge vectors is created from the training data and additional external sources. These vectors are then stored in the vector database, indexed for efficient retrieval.
- **Dynamic Knowledge Integration**: Mechanisms have been implemented to allow the chatbot to query the vector database in real-time, retrieving relevant knowledge vectors based on user inputs. This integration enables the chatbot to provide updated and contextually appropriate responses without retraining.
- **Local Deployment**: The entire system has been configured to run on a local machine if wanted by the user. This setup involved ensuring that all components, including the vector database and chatbot core engine, operated efficiently in a local environment.

The PoC system was evaluated based on several criteria:

- **Adaptability**: The system's ability to incorporate new knowledge dynamically.
- **Response Accuracy**: The accuracy of the chatbot's responses was measured by comparing its outputs to the original source data.
- **Customization**: The effectiveness of the customization was evaluated by allowing users to specify their temperature and observing the chatbot's ability to adapt its responses accordingly.

The PoC system demonstrated the feasibility and benefits of integrating vector databases into chatbot systems. Key findings include:

- **Enhanced Adaptability**: The chatbot could easily integrate new information from the vector database, improving its relevance and accuracy without undergoing retraining.
- **Response Accuracy**: The results showed that the pipeline improved the chatbot's accuracy by providing more relevant and contextually appropriate answers.
- **Local Deployment**: The system operated effectively in a local environment, ensuring data privacy and security while providing a custom context.
- **Improved Customization**: User could customize the chatbot's behavior to meet their specific needs, resulting in more personalized and engaging interactions.

These results confirm the **potential** of the proposed techniques to address the limitations of traditional chatbot systems, offering a more flexible approach to conversational AI. That is in spite of the loss of performance giving the testing environment in terms of hardware capabilities.

10.1 Results

In this section, I have included a series of images from a selection of the tests conducted. The following images below (fig. 21, fig. 22, fig. 23, fig. 24, fig. 25, fig. 26 and fig. 27) provide a clear and comprehensive of the results obtained. By examining these pictures, one can better understand the possible outcome behaviour that can be obtained out of this thesis.

To start with, some documents and URLs were uploaded and indexed in the system, so to give the proper and intended context to the model. These are Lithops documentation websties such as [26], [24] and [25], LocalAI main website [15], URV website of my bachelor [27] and a FIA official PDF document of a racing incident during a Formula1 Grand Prix [11]. The average answering time of all these questions was 67s.

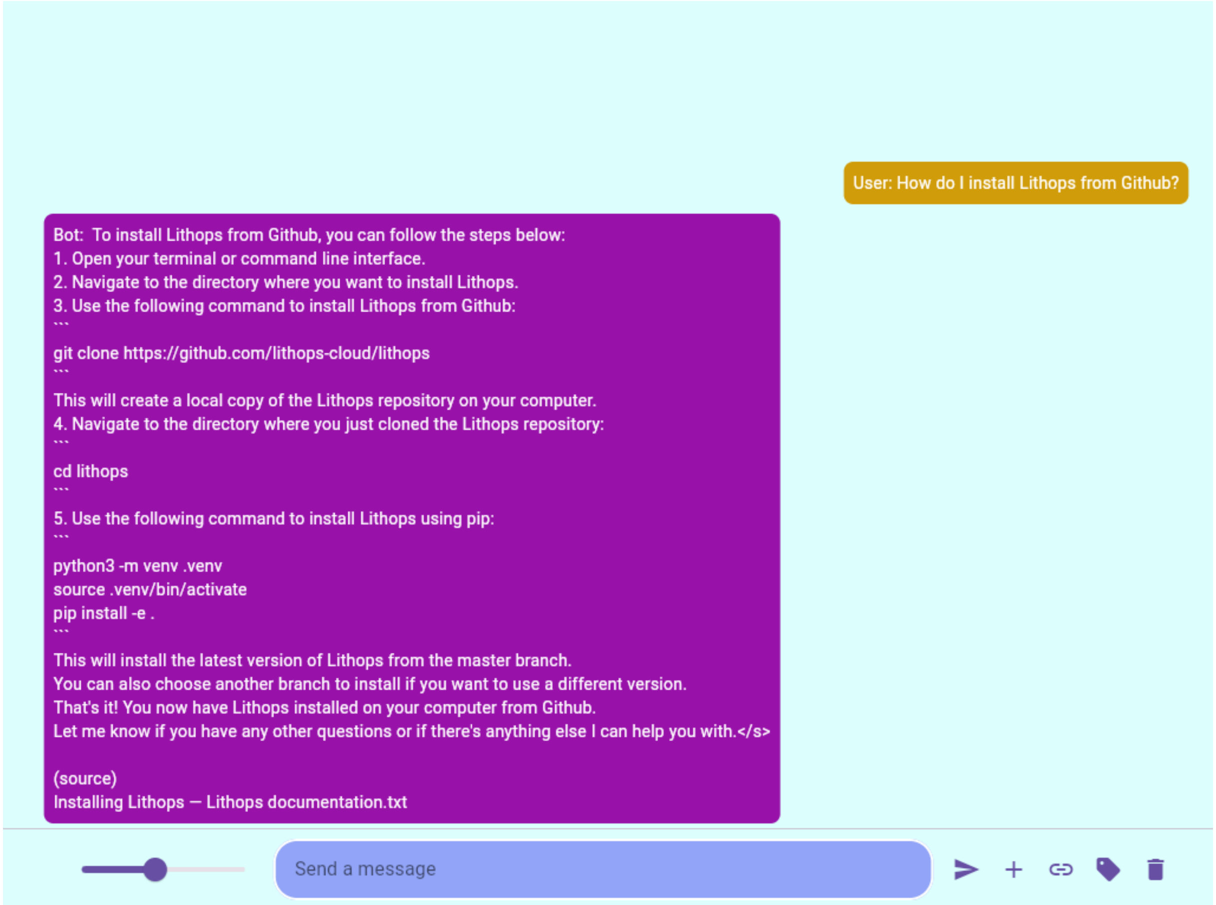

Figure 21: Asking how to install Lithops from Github. From website [26]

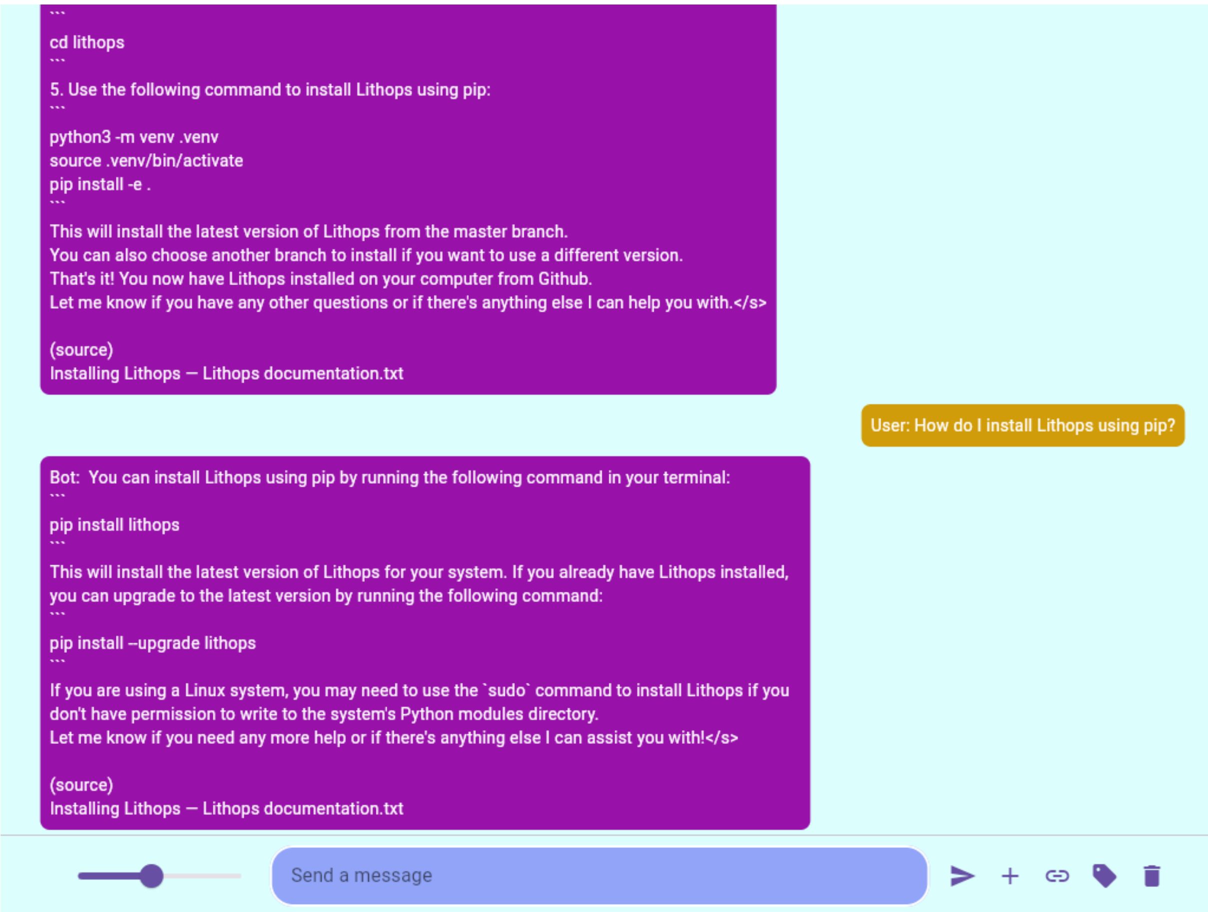

Figure 22: Asking how to install Lithops from pip. From website [26]

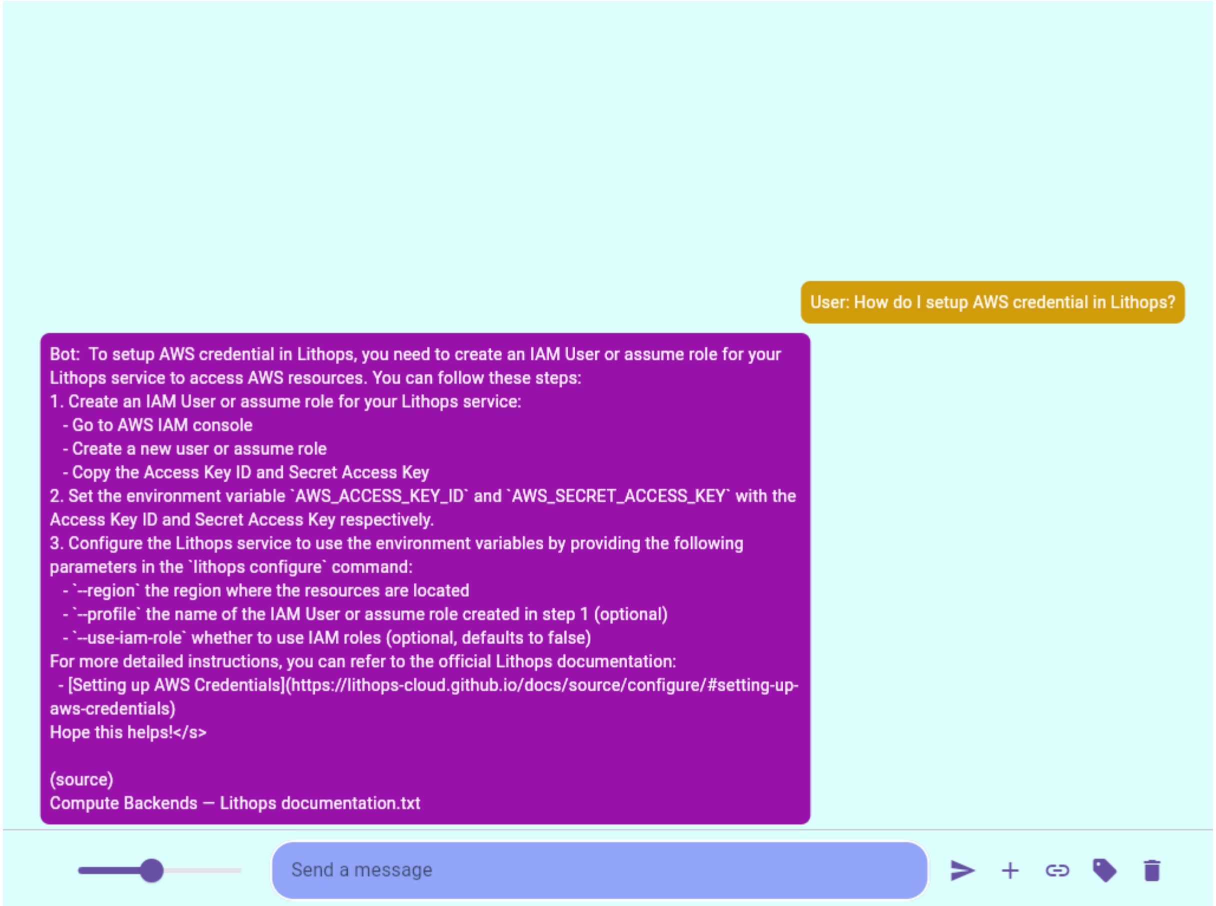

Figure 23: Asking how to setup AWS credentials for Lithops. From website [24]

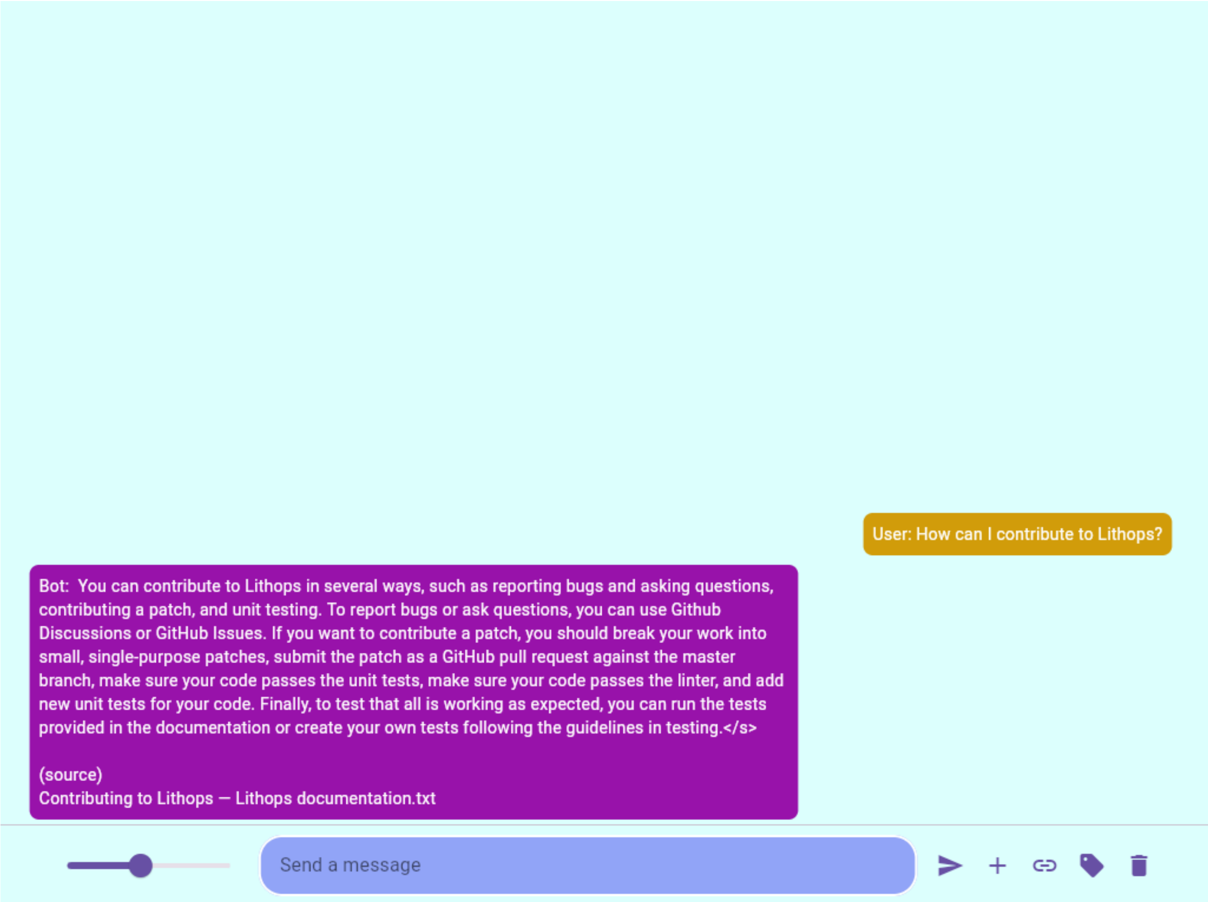

Figure 24: Asking how to contribute to Lithops. From website [25]

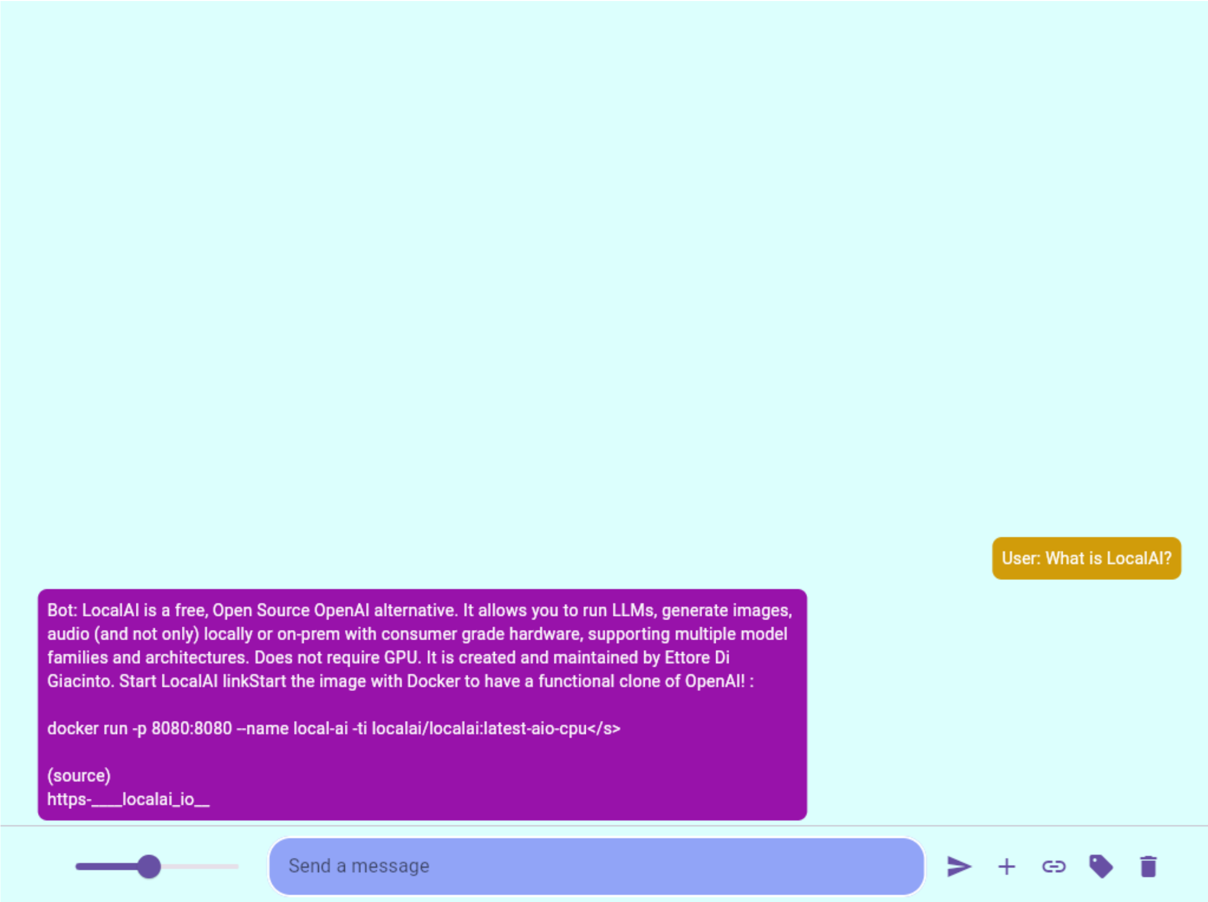

Figure 25: Asking what LocalAI is. From website [15]

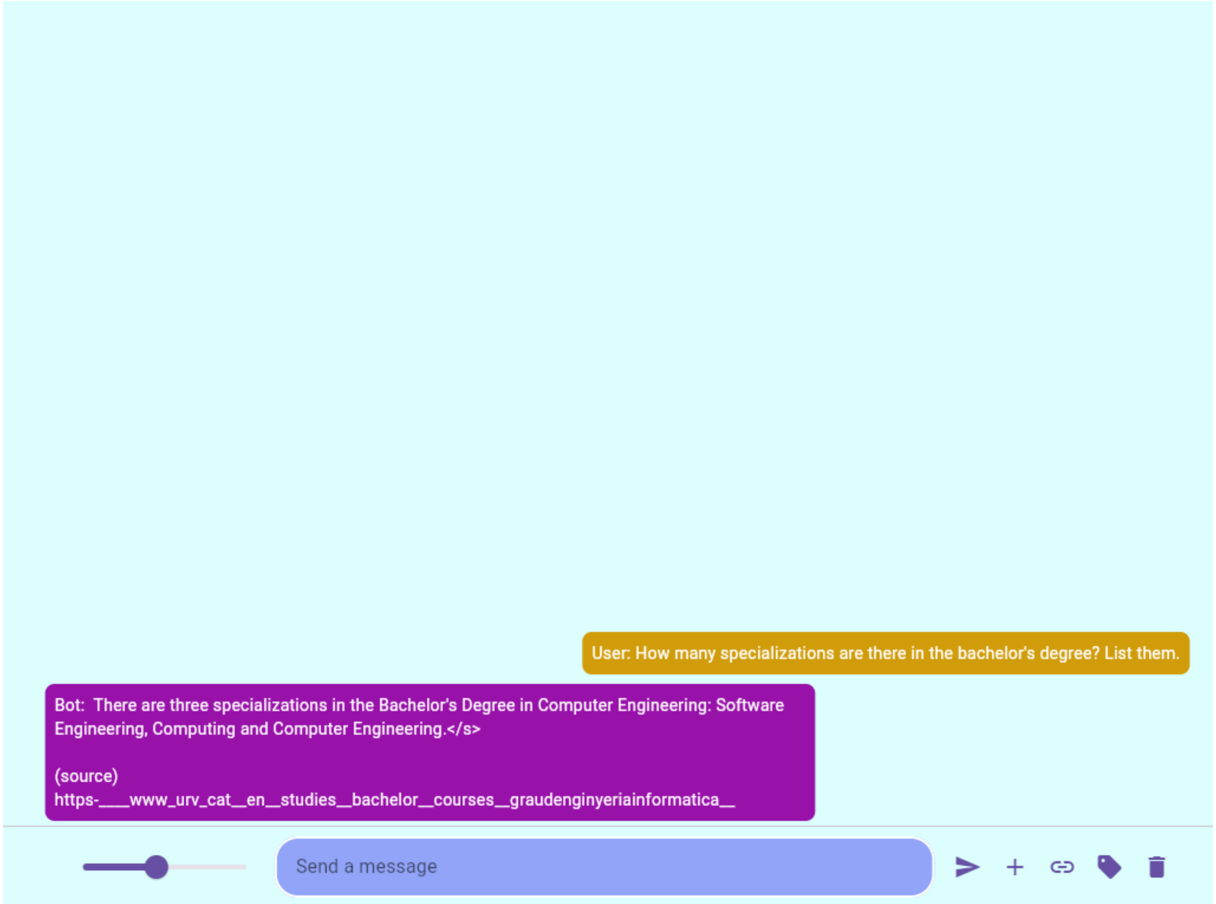

Figure 26: Asking about Bachelor's Degree in Computer Engineering in Universitat Rovira i Virgili. From website [27]

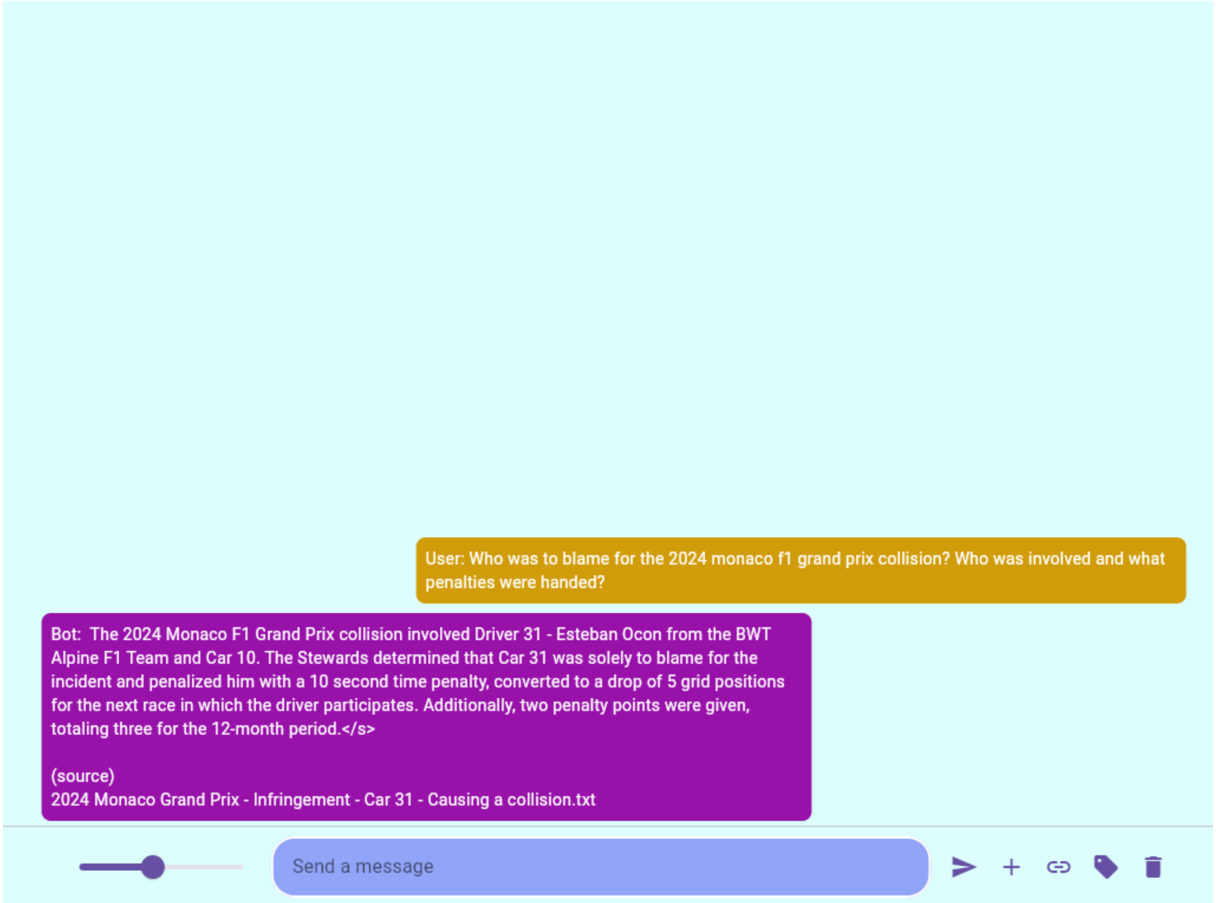

Figure 27: Asking about an FIA statement of a Formula1 race. From document [11]

11 Testing and validation

In this section I will describe all the testing procedures I have followed in order to check and guarantee a perfect functionality of the project. There will be unit tests and integration tests as well.

11.1 Unit tests

In this subsection I describe all the individual unit tests I have applied to the project app to check its functionality.

Test Case 1: Query Verify that the chatbot responds appropriately to a user query.

- **Test Steps:**
  1. Enter a query in the chatbot interface.
  2. Press enter or send message button.
- **Expected Result:** The chatbot should process the query and provide a response within a reasonable time frame.
- **Actual Result:** The message is shown, and eventually the chatbot gives an answer.
- **Pass/Fail:** Pass

Test Case 2: Upload Files Verify that files can be successfully uploaded to the chatbot system.

- **Test Steps:**
  1. Click the '+' button to initiate file upload.
  2. Select files to upload.
- **Expected Result:** Files should be uploaded to the system, and a success message should be displayed.
- **Actual Result:** The files are correctly uploaded to the container filesystem and formatted if necessary.
- **Pass/Fail:** Pass

Test Case 3: Upload Link Verify that a URL can be successfully uploaded to the chatbot system.

- **Test Steps:**
  1. Click the link button to initiate URL upload.
  2. Enter a valid URL.
  3. Submit the URL.
- **Expected Result:** The URL content should be uploaded to the system, and a success message should be displayed.
- **Actual Result:** The URL contents will be obtained, unless Javascript is required to get that. In that case, it fails to gather the contents.
- **Pass/Fail:** Partial Pass

Test Case 4: Index Files Verify that files can be successfully indexed within the chatbot system.

- **Test Steps:**
  1. Click the index button to start the indexing process.
- **Expected Result:** Files should be indexed, and a success message should be displayed.
- **Actual Result:** Files are indexed correctly, and a message is shown.
- **Pass/Fail:** Pass

Test Case 5: Delete files Verify that files can be successfully deleted from the chatbot system.

- **Test Steps:**
  1. Click the delete button to start the file deletion process.
- **Expected Result:** Files should be deleted, and a success message should be displayed.
- **Actual Result:** Files stored in the container's filesystem are removes successfully.
- **Pass/Fail:** Pass

Test Case 6: Switch Chat Verify that the user can switch between different chat sessions.

- **Test Steps:**
  1. Open the navigation rail.
  2. Select a different chat session from the sidebar.
- **Expected Result:** The chat interface should update to display the selected chat session.
- **Actual Result:** The user can switch to another created chat, even if the previous one is waiting for a response.
- **Pass/Fail:** Pass

Test Case 7: Create Chat Verify that the user can create a new chat session.

- **Test Steps:**
  1. Open the navigation rail.
  2. Click the '+' button to create a new chat session.
- **Expected Result:** A new chat session should be created, and the chat list should be updated to include it.
- **Actual Result:** A new chat session is created and added to the side menu, swapping the user's current session to the new one.
- **Pass/Fail:** Pass

Test Case 8: Delete Chat Verify that the user can delete an existing chat session.

a. **Test Steps:**
   (a) Open the navigation rail.
   (b) Select the chat session to be deleted from the sidebar.
   (c) Click the delete button.

b. **Expected Result:** The selected chat session should be deleted, and the chat list should be updated to remove it.

c. **Actual Result:** The user can successfully remove any existing chat room, removing it from the side menu. If it results to be the current session the user is in, it will change to the first session available. If the removed chat was waiting for a response, once it arrives it will generate an error, bot will not stop the app from functioning correctly.

d. **Pass/Fail:** Partially Pass.

Test Case 9: Save App State Verify that the user can save the current state of the application.

- **Test Steps:**
  1. Click the save button in the TopBar.
- **Expected Result:** The current state of the application should be saved, and a success message should be displayed.
- **Actual Result:** The state of the app is saved in the Couchdb database correctly. If the users expects a response at the moment of saving that still hasn't arrived, it will not be saved.
- **Pass/Fail:** Pass

Test Case 10: Load App State Verify that the user can load the previously saved state of the application.

- **Test Steps:**
  1. Open the application.
- **Expected Result:** The previously saved state of the application should be loaded automatically, and a success message should be displayed.
- **Actual Result:** Once we restart the website application, the last saved session is restored successfully.
- **Pass/Fail:** Pass

11.2 Failing tests

In this subsection I run failing tests to check the failure handling by the user interface, including the error communication to the user.

Failing Test 1: Flask unavailable Verify that app shows an error if Flask service is not available and fails to provide an answer.

- **Test Steps:**
  1. Steps from the Unit Test cases number 1, 2, 3, 4 and 5.
- **Expected Result:** The app state does not change and an error message is shown to the user, describing that what is failing is the Flask service.
- **Actual Result:** The error message specifying the error is shown.
- **Pass/Fail:** Pass

Failing Test 2: Couchdb unavailable Verify that app shows an error if CouchDB container is not available and fails to provide an answer.

- **Test Steps:**
  1. Steps from the Unit Test cases number 9 and 10.
- **Expected Result:** The app state does not change, it cannot neither load nor save the app state and an error message is shown to the user, describing that what is failing is the CouchDB service.
- **Actual Result:** The error message specifying the error is shown and the app state is not loaded nor saved.
- **Pass/Fail:** Pass

Failing Test 3: LocalAI unavailable Verify that app shows an error if LocalAI container is not available and fails to provide an answer.

- **Test Steps:**
  1. Steps from the Unit Test cases number 1 and 4.
- **Expected Result:** The app state does not change, the files are not indexed or the answer is not provided, showing a "-" as a response from the chatbot. An error message is shown to the user, describing that what is failing is the LocalAI service.
- **Actual Result:** The error message specifying the error is shown.
- **Pass/Fail:** Pass

In conclusion, 11 out of the 13 tests fully pass, and the other 2 remaining tests pass partially. This last expression means that while the error is not handled and the exception not contemplated, it does not compromise the working operation of the app in any way, as well as the user experience.

12 Performance

While hosting a Large Language Model (LLM) and its associated system in a local environment offers numerous advantages, it also presents certain trade-offs, with **performance** being the primary concern, depending on the hardware and the chosen model. Local deployment allows for enhanced data privacy and security, as sensitive information remains within the confines of the local network, mitigating the risk of data breaches that can occur with cloud-based solutions. Additionally, it provides greater control over the environment, enabling customization and optimization tailored to specific needs and workflows.

These benefits are often overshadowed by significant performance challenges. Unlike cloud-based or other distributed solutions solutions that leverage high-performance computing resources, local environments typically lack the same level of hardware capabilities. This is especially evident when running models on **CPUs** instead of GPUs. CPUs, while versatile and adequate for many computational tasks, are not optimized for the parallel processing demands of LLMs, leading to a notable worse performance. The latency introduced by CPU-bound processing becomes a huge **bottleneck**.

In my specific case, the performance degradation when running models on a CPU is pronounced. The inference times are significantly longer, which not only slows down the development and testing cycles but also impacts the user experience if the model is intended for real-time interaction. For instance, tasks that would take seconds on a GPU can extend to **minutes** on a CPU, making the system impractical for any application requiring swift turnaround.

Moreover, the lack of specialized hardware accelerators in a local setup means that scaling the system to handle larger models or more concurrent users becomes a challenge. This can hinder the ability to experiment with more complex models or to iterate rapidly, which is crucial in a research and development setting.

All this goes for the performance in terms of responsiveness and speed. There are many other factors when evaluating a LLM performance. For instance, you can benchmark its perplexity, human evaluation, used grammar or readability. These other factors can also qualify LLMs capacity to perform on a diversity of tasks.

12.1 Global Standard Benchmarking

In this subsection I detail the exact Standard Benchmarks I used to compare the performance of the used model in contrast to commercial solutions You find the comparison of the following benchmarks on fig. 28.

- **ARC Challenge[8]:** Checks for advanced reasoning skills.
- **ARC Easy[8]:** Checks for basic reasoning skills.
- **BoolQ[7]:** Checks for natural language understanding through boolean questions.
- **HellaSwah[30]:** Challenges models with nuanced reasoning as well as sentence completion.
- **OpenBookQA[29]:** Tests comprenhension of science-based questions.
- **WinoGrande[20]:** Assesses common-sense reasoning thorugh winograd schema styled questions.
- **PIQA[28]:** Evaluates understanding of physical interactions and commonsense.

I have also studied the performance difference on different tasks from Google's *Beyond the Imitation Game benchmark* (**BIG-bench**)[23]. It is composed by a diverse collection of 204 tasks contributed by various recognized authors. These tasks cover a wide range of topics, including linguistics, mathematics, common-sense reasoning, social bias, software development, biology or physics knowledge. In this thesis I compare some of those tasks in fig. 29 and in fig. 30.

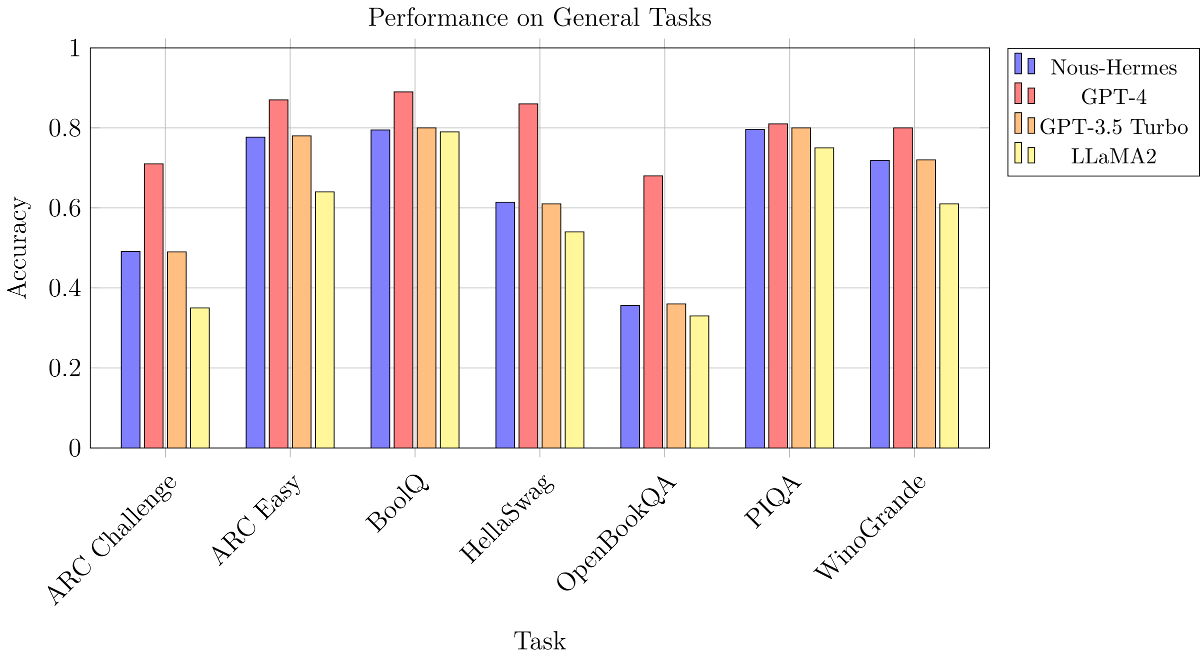

Figure 28: Performance on General Tasks

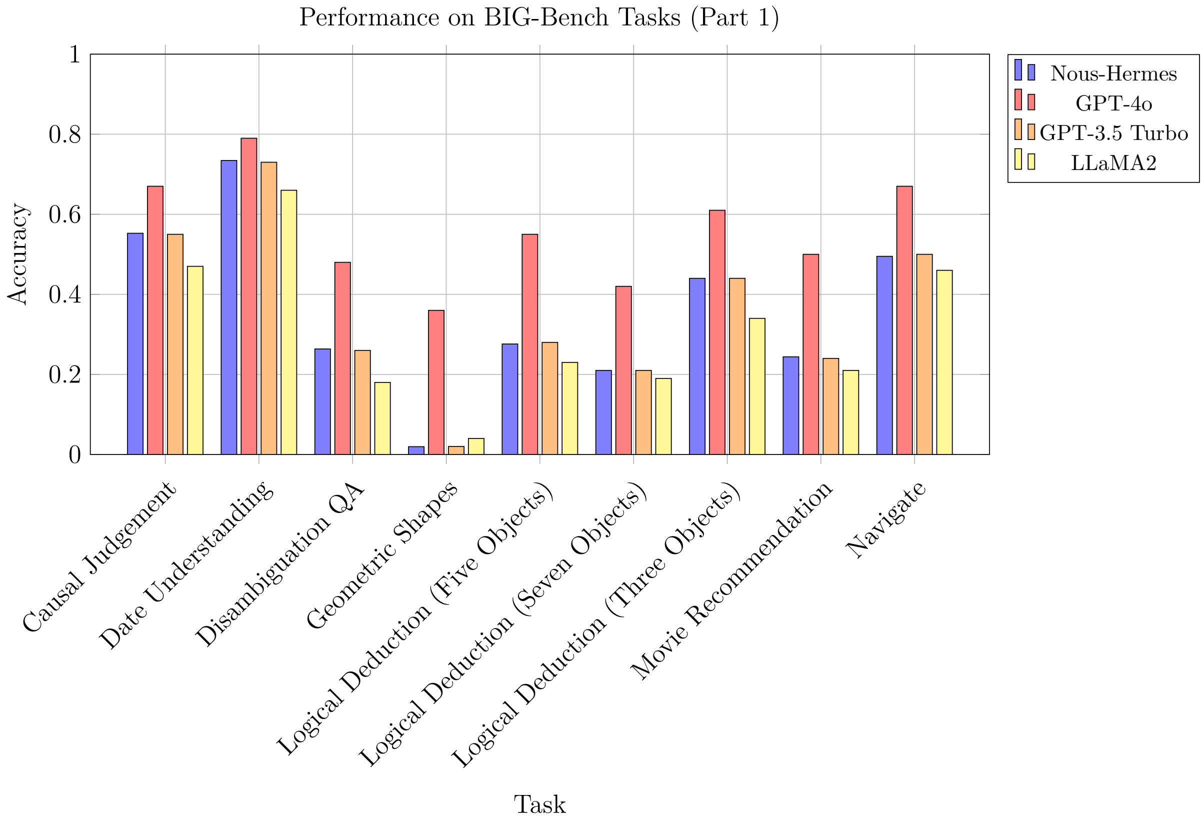

Figure 29: Performance on BIG-Bench Tasks (Part 1)

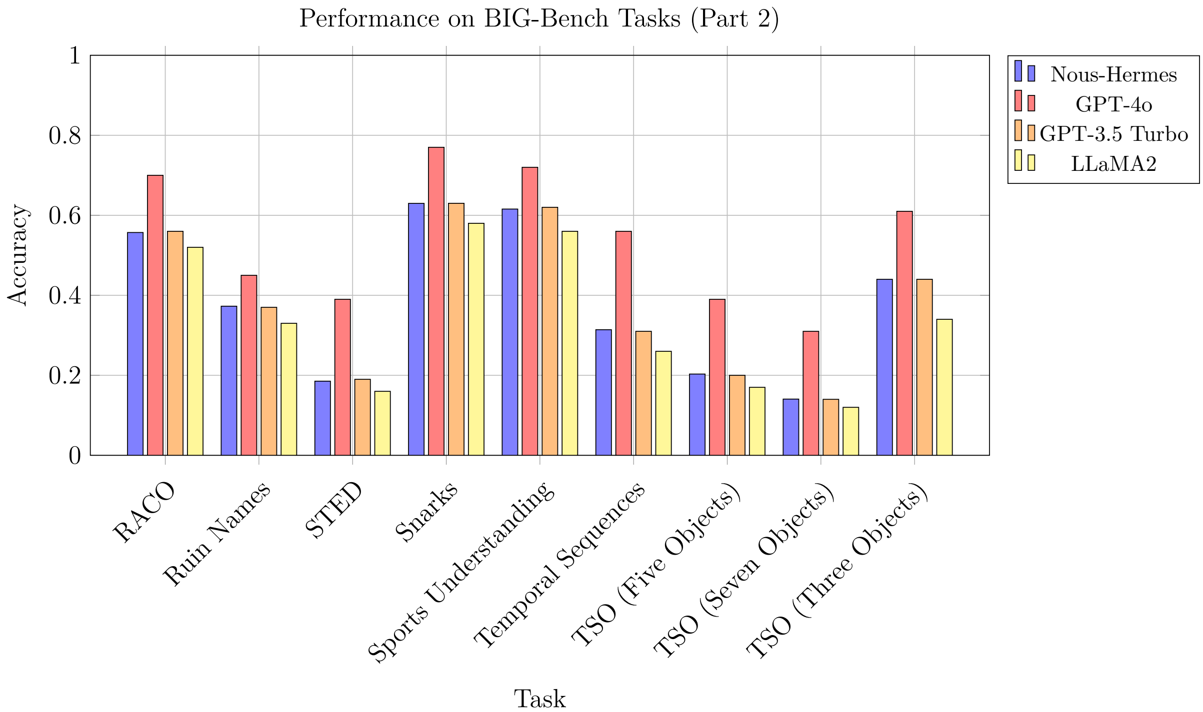

Figure 30: Performance on BIG-Bench Tasks (Part 2)

12.2 Deployment performance test

To provide a more understandable comparison of the time performance between the deployment of my solution and other commercial solutions, I have made a simple test on my chatbot and ChatGPT-4o. I have uploaded a PDF document [11] to both of them and asked the same questions. Here are the answers obtained and the elapsed time by each model:

Model: ChatGPT
Question: What are the competitors reminded of?
Elapsed time: 1.5s
Answer: [4] `Competitors are reminded that they have the right to appeal certain decisions of the Stewards in accordance with Article 15 of the FIA International Sporting Code and Chapter 4 of the FIA Judicial and Disciplinary Rules, within the applicable time limits`

Model: My solution (nous-hermes-llama2-13b.Q8_0.gguf)
Question: What are the competitors reminded of?
Elapsed time: 88s
Answer: `They are reminded that they have the right to appeal certain decisions of the Stewards, in accordance with Article 15 of the FIA International Sporting Code and Chapter 4 of the FIA Judicial and Disciplinary Rules, within the applicable time limits.`

In both scenarios, the chatbots referenced the specific file from which they successfully extracted the information to answer the given questions. While both answers are of similar quality, a severe difference in terms on computational time can be noted.

As already explained before, there is a huge performance difference from running a model using CPU only an on an ordinary hardware platform. This tiny test underscores the responsive gap between its deployment and a globally-scale distributed deployed model.

When tasked with providing general information, the models usually have to deal with less data. Therefore, the response time is also reduced. the elapsing time also relates to the response and the complexity of the user's demands. On average, when asked about concrete general information, my solution achieves a response time of over **40 seconds**. This improvement demonstrates that when not querying the model with context or demanding big responses such as generating big pieces of code, there is a substantial reduction in the time required to process and deliver answers.

13 Conclusions

The realization of this thesis has shown how chatbots have undergone a significant evolution from simple rule-based programs to the sophisticated architecture they currently have. This continuous evolution has enabled chatbots not only to answer questions, but also to understand complex contexts, adapt their responses and continuously learn from interactions with users. Their application in more and more sectors demonstrates their enormous potential to improve efficiency and service quality.

It has also demonstrated the possibility of deploying a local, safe and private alternative of the current commercial chatbots, and also its ability to adapt to any context with an almost free cost. The integration of LLM models and vector databases not only allows to offer more contextual answers and continuous improvement with a minimum cost, but also offers powerful tools for the analysis of data and large volumes of information.

What I take away from this is the idea that it has also allowed me to complete my professional training and knowledge as a computer scientist working with new fields, concepts and tools.

13.1 Future work

This project represents an initial exploration into these technologies and their integration at a low, foundational level. Building upon this groundwork opens exciting possibilities, such as the exploration of running distributed LLMs models within a Kubernetes cluster.

This mentioned approach harnesses the scalability and orchestration capabilities of Kubernetes to efficiently manage and scale LLMs across multiple nodes, unlocking new possibilities for enhancing the performance and versatility of their language processing applications. This experimental thesis serves as a springboard for further research and development in the field of distributed artificial intelligence.

References

[1] Amansinghalml. *Temperature — LLMs*. URL: https://medium.com/@amansinghalml_33304/temperature-llms-b41d75870510.

[2] Apache. *API Reference*. URL: https://docs.couchdb.org/en/stable/api/index.html.

[3] Apache. *couchdb*. URL: https://hub.docker.com/_/couchdb.

[4] Albert Cañadilla. *ChatGPT chatroom performance test*. URL: https://chatgpt.com/share/82b38794-d87a-44a5-a918-beefb3ef9892.

[5] Albert Cañadilla. *Thesis GitHub repo*. URL: https://github.com/acanadil/TFG.

[6] ChromaDB. *Chroma, an AI-native open-source vector database*. URL: https://python.langchain.com/v0.2/docs/integrations/vectorstores/chroma/.

[7] Christopher Clark et al. *BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions*. 2019. arXiv: 1905.10044 [cs.CL].

[8] Peter Clark et al. *Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge*. 2018. arXiv: 1803.05457 [cs.AI].

[9] Cloudflare. *What is a vector database?* URL: https://www.cloudflare.com/learning/ai/what-is-vector-database/.

[10] Hugging Face. *HuggingFace main page*. URL: https://huggingface.co/.

[11] FIA. *2024 Monaco Grand Prix - Infringement - Car 31 - Causing a collision*. URL: https://www.fia.com/sites/default/files/decision-document/2024%20Monaco%20Grand%20Prix%20-%20Infringement%20-%20Car%2031%20-%20Causing%20a%20collision.pdf.

[12] Flask. *User's guide*. URL: https://flask.palletsprojects.com/en/3.0.x/.

[13] Flutter. *Flutter documentation*. URL: https://docs.flutter.dev/.

[14] Ettore Di Giacinto. *LocalAI - Architecture*. URL: https://localai.io/docs/reference/architecture/.

[15] Ettore Di Giacinto. *LocalAI - Overview*. URL: https://localai.io/.

[16] Ettore Di Giacinto. *mudler/all-MiniLM-L6-v2*. URL: https://huggingface.co/mudler/all-MiniLM-L6-v2/tree/main.

[17] Ettore Di Giacinto. *Text generation (GPT)*. URL: https://localai.io/features/text-generation/.

[18] Langchain. *Docs Introduction*. URL: https://python.langchain.com/v0.2/docs/introduction/.

[19] LocalAI. *localai/localai*. URL: https://hub.docker.com/r/localai/localai.

[20] Keisuke Sakaguchi et al. *WinoGrande: An Adversarial Winograd Schema Challenge at Scale*. 2019. arXiv: 1907.10641 [cs.CL].

[21] Amazon Web Services. *What are Large Language Models (LLM)?* URL: https://aws.amazon.com/en/what-is/large-language-model/.

[22] Amazon Web Services. *What Is RAG?* URL: https://aws.amazon.com/what-is/retrieval-augmented-generation/.

[23] Aarohi Srivastava et al. *Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models*. 2023. arXiv: 2206.04615 [cs.CL].

[24] The Lithops Team. *AWS Lambda*. URL: https://lithops-cloud.github.io/docs/source/compute_config/aws_lambda.html.

[25] The Lithops Team. *Contributing to Lithops*. URL: https://lithops-cloud.github.io/docs/source/contributing.html.

[26] The Lithops Team. *Installing Lithops*. URL: https://lithops-cloud.github.io/docs/source/install_lithops.html.

[27] Universitat Rovira i Virgili. *Bachelor's Degree in Computer Engineering*. URL: https://www.urv.cat/en/studies/bachelor/courses/graudenginyeriainformatica/.

[28] Weiguo Xia et al. *Analysis of a Nonlinear Opinion Dynamics Model with Biased Assimilation*. 2020. arXiv: 1912.01778 [math.OC].

[29] Chen Yuan et al. *Power Flow Analysis Using Graph based Combination of Iterative Methods and Vertex Contraction Approach*. 2018. arXiv: 1809.01398 [cs.DC].

[30] Rowan Zellers et al. *HellaSwag: Can a Machine Really Finish Your Sentence?* 2019. arXiv: 1905.07830 [cs.CL].