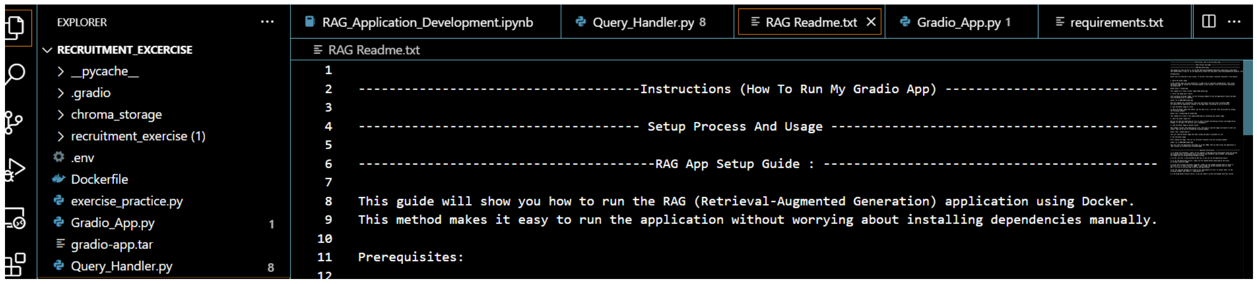
Project: Multilingual RAG Model for Finnish and English Language Support

Objective:

This project aims to build a multilingual Retrieval-Augmented Generation (RAG) model to handle text processing and generation tasks in both Finnish and English. The model will use the LangChain framework, supporting various NLP operations, including language identification, translation, and information retrieval. This setup is crucial for applications that need to process and generate content in multiple languages, specifically Finnish and English, offering seamless language support for end-users.

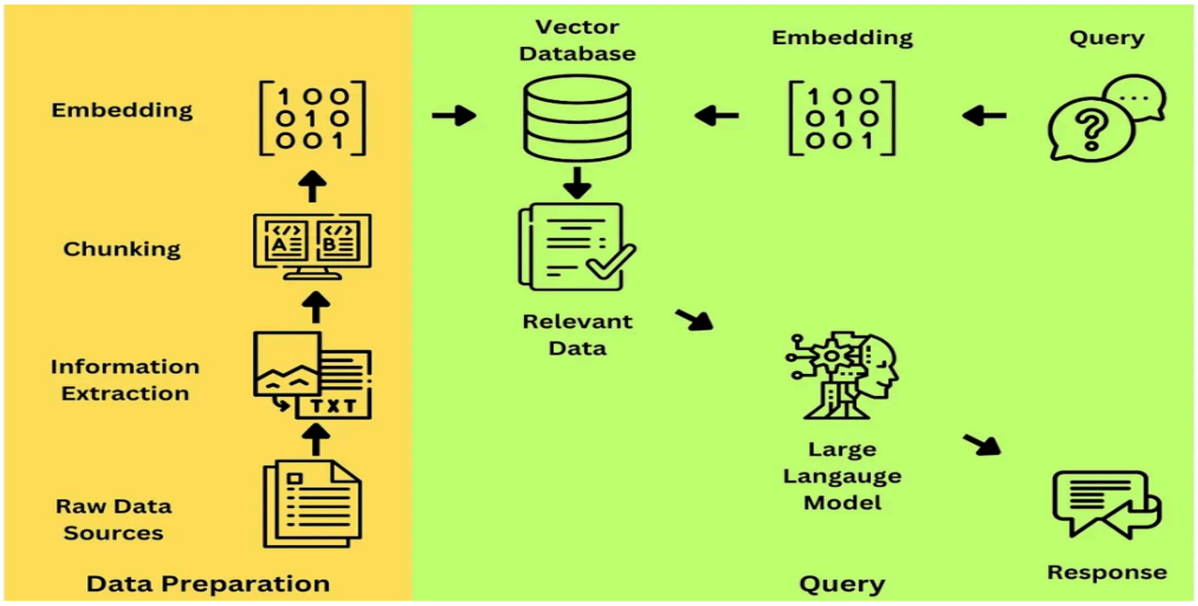
Instructions for setting up and running the chatbot in a Docker container :

All instructions can be found in the RAGReadme.txt file in the project directory.



```

-----------------------------------Instructions (How To Run My Gradio App) ---------------------------

---------------------------------- Setup Process And Usage -----------------------------------------

-------------------------------------RAG App Setup Guide : ------------------------------------------

This guide will show you how to run the RAG (Retrieval-Augmented Generation) application using Docker.
This method makes it easy to run the application without worrying about installing dependencies manually.

Prerequisites:
```

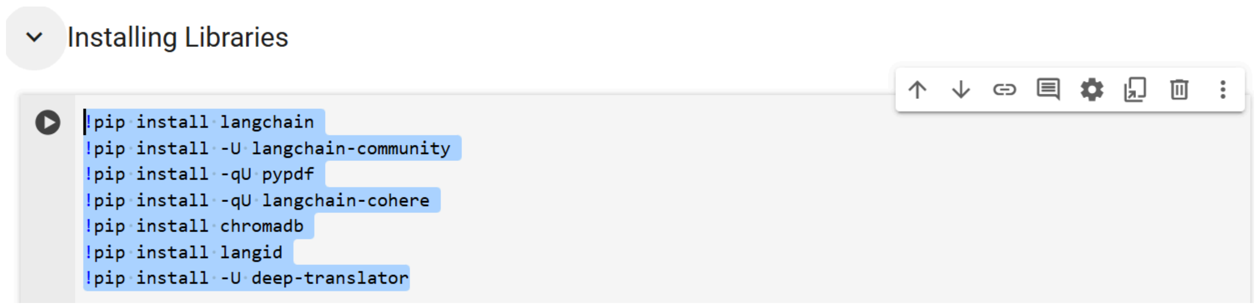
Technique/Architecture I Used :



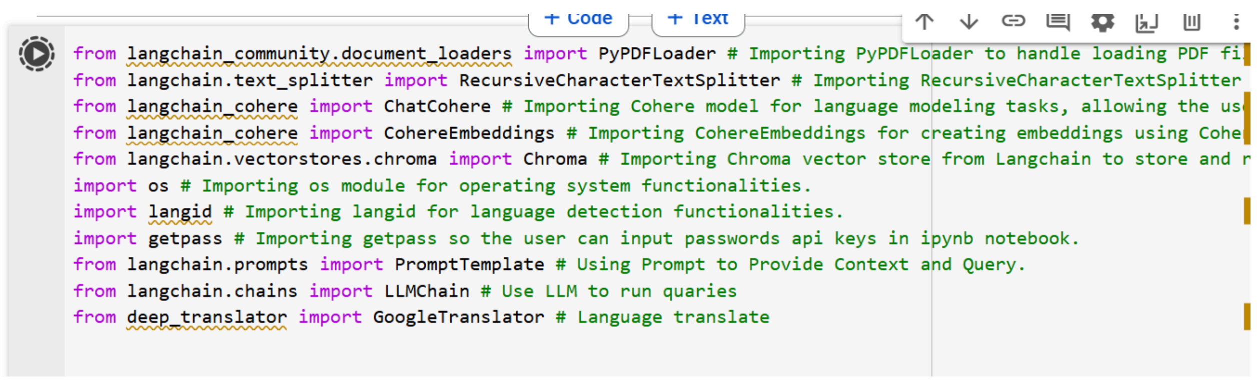
RAG Architecture Explanation :

Libraries I Used :

The script installs several key libraries required for the project, including LangChain, the core framework for building applications with large language models. It updates to the latest version of LangChain Community to access additional features and integrations. For handling PDF files, it installs PyPDF, enabling the extraction of text and data from PDFs. LangChain with Cohere is included to integrate Cohere's language models, enhancing the NLP capabilities of the project. ChromaDB is installed to manage and query vector embeddings, essential for information retrieval in a RAG model. Additionally, LangID is included for language identification, which helps in determining the language of text inputs. Finally, Deep Translator is installed for automatic translation between multiple languages, allowing the model to support both Finnish and English.


Installing Libraries

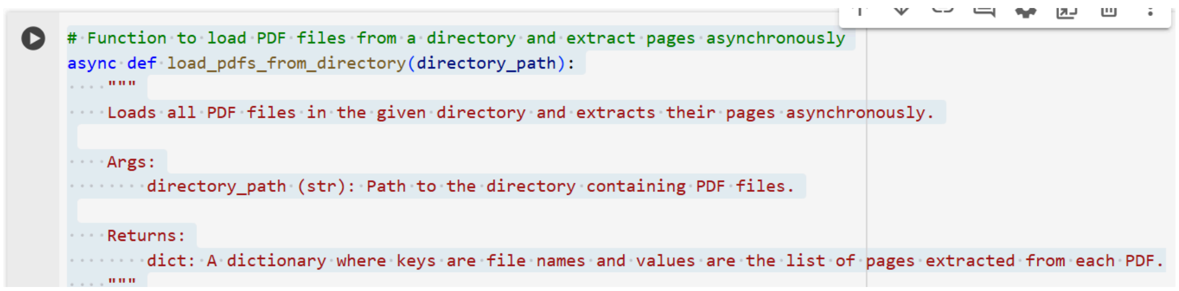
```
!pip install langchain
!pip install -U langchain-community
!pip install -qU pypdf
!pip install -qU langchain-cohere
!pip install chromadb
!pip install langid
!pip install -U deep-translator
```



```
from langchain_community.document_loaders import PyPDFLoader # Importing PyPDFLoader to handle loading PDF fi
from langchain.text_splitter import RecursiveCharacterTextSplitter # Importing RecursiveCharacterTextSplitter
from langchain_cohere import ChatCohere # Importing Cohere model for language modeling tasks, allowing the us
from langchain_cohere import CohereEmbeddings # Importing CohereEmbeddings for creating embeddings using Cohe
from langchain.vectorstores.chroma import Chroma # Importing Chroma vector store from Langchain to store and r
import os # Importing os module for operating system functionalities.
import langid # Importing langid for language detection functionalities.
import getpass # Importing getpass so the user can input passwords api keys in ipynb notebook.
from langchain.prompts import PromptTemplate # Using Prompt to Provide Context and Query.
from langchain.chains import LLMChain # Use LLM to run quaries
from deep_translator import GoogleTranslator # Language translate
```

Loading The Data :

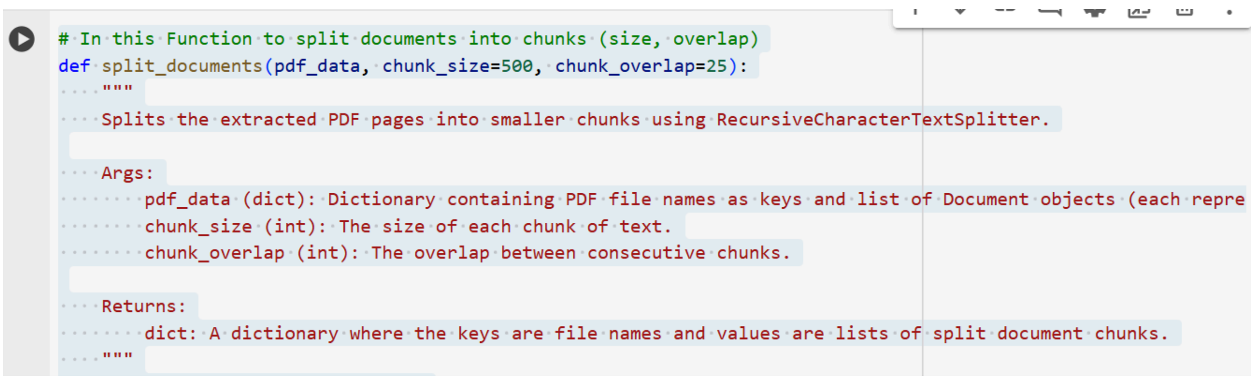
This function `load_pdfs_from_directory`, loads all PDF files from a specified directory and extracts their pages asynchronously. It takes the directory path as input and iterates through the files in the directory, checking for PDF files. For each PDF, it uses the `PyPDFLoader` to extract the pages asynchronously and stores them in a dictionary, where the keys are the file names and the values are the lists of pages extracted from each PDF. After processing, the total number of PDFs loaded is displayed along with a confirmation message.



```python
# Function to load PDF files from a directory and extract pages asynchronously
async def load_pdfs_from_directory(directory_path):
    """
    Loads all PDF files in the given directory and extracts their pages asynchronously.

    Args:
        directory_path (str): Path to the directory containing PDF files.

    Returns:
        dict: A dictionary where keys are file names and values are the list of pages extracted from each PDF.
    """
```

Chunking Data :

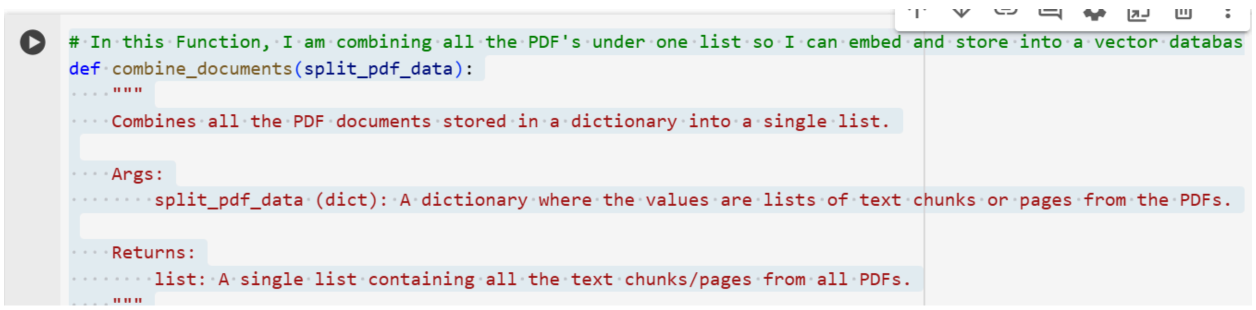
This function, split_documents, splits the extracted PDF pages into smaller chunks using the RecursiveCharacterTextSplitter. It takes in a dictionary (pdf_data) where the keys are PDF file names and the values are lists of documents (each representing a page). The function allows customization of chunk size (chunk_size) and overlap between consecutive chunks (chunk_overlap). It uses the RecursiveCharacterTextSplitter to break down the documents into smaller, manageable parts. The resulting chunks are stored in a dictionary (split_docs), which is returned after processing all the documents. Finally, the script prints the total number of split chunks across all documents.



```python
# In this Function to split documents into chunks (size, overlap)
def split_documents(pdf_data, chunk_size=500, chunk_overlap=25):
    """
    Splits the extracted PDF pages into smaller chunks using RecursiveCharacterTextSplitter.

    Args:
        pdf_data (dict): Dictionary containing PDF file names as keys and list of Document objects (each repre
        chunk_size (int): The size of each chunk of text.
        chunk_overlap (int): The overlap between consecutive chunks.

    Returns:
        dict: A dictionary where the keys are file names and values are lists of split document chunks.
    """
```

Combining Data :

The combine_documents function merges all the text chunks or pages from the individual PDF documents into a single list. It takes as input a dictionary (split_pdf_data) where the values are lists containing text chunks or pages from the PDFs. The function uses Python's sum() method to concatenate all the lists into one. This single combined list can then be used for further processing, such as embedding the content and storing it in a vector database. The result is stored in the variable combine_all_documents, which now contains all the combined text from the PDFs.

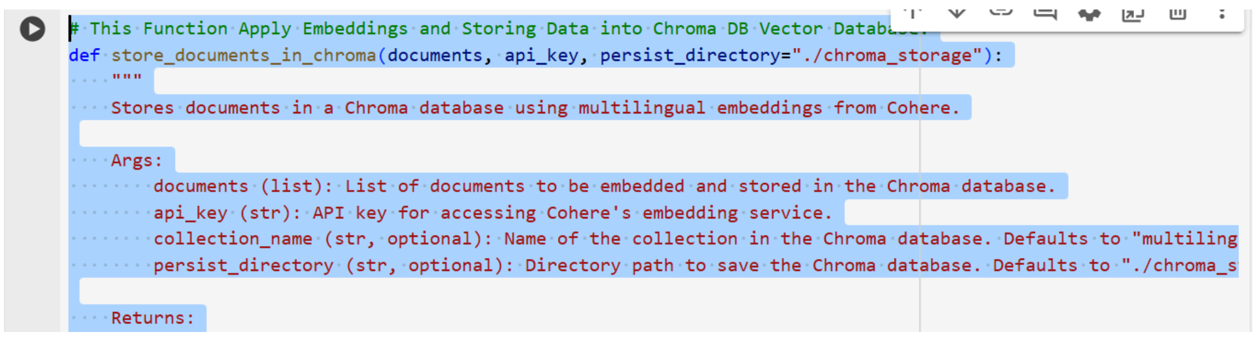


```python
# In this Function, I am combining all the PDF's under one list so I can embed and store into a vector databas
def combine_documents(split_pdf_data):
    """
    Combines all the PDF documents stored in a dictionary into a single list.

    Args:
        split_pdf_data (dict): A dictionary where the values are lists of text chunks or pages from the PDFs.

    Returns:
        list: A single list containing all the text chunks/pages from all PDFs.
    """
```

Embeddings And Data Storage :

The store_documents_in_chroma function is responsible for embedding documents and storing them in a Chroma vector database using Cohere's multilingual embeddings. The function takes a list of documents (documents), a valid API key (api_key), and an optional directory path (persist_directory) to store the Chroma database. It sets the API key for accessing Cohere's embedding service, initializes the embedding model with multilingual capabilities, and uses Chroma's from_documents method to store the embedded documents in the specified collection. The data is then persisted in the specified directory. A confirmation message is printed once the documents are successfully stored.

To use the function, the user needs to provide a valid Cohere API key, which is securely obtained using getpass.getpass. After calling this function, the documents will be embedded and saved to the Chroma vector database for later retrieval and querying.

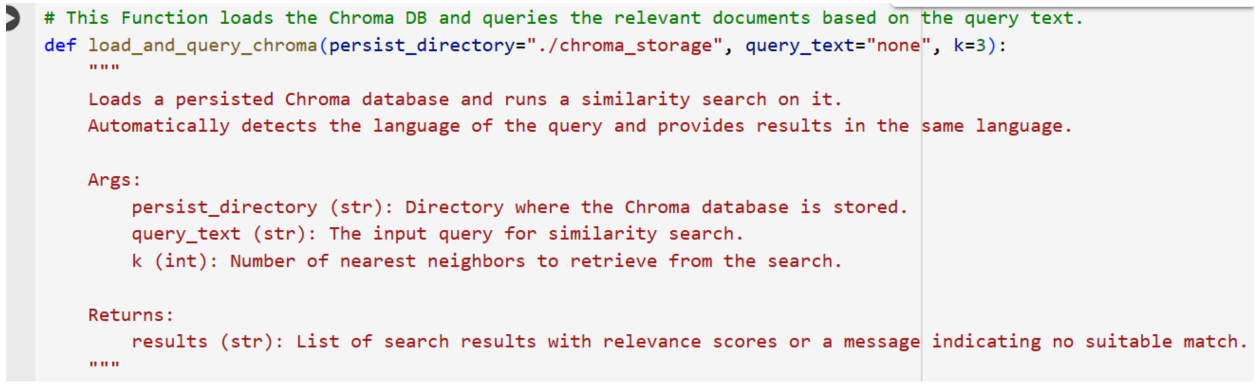


```python
# This Function Apply Embeddings and Storing Data into Chroma DB Vector Datab
def store_documents_in_chroma(documents, api_key, persist_directory="./chroma_storage"):
    """
    Stores documents in a Chroma database using multilingual embeddings from Cohere.

    Args:
        documents (list): List of documents to be embedded and stored in the Chroma database.
        api_key (str): API key for accessing Cohere's embedding service.
        collection_name (str, optional): Name of the collection in the Chroma database. Defaults to "multiling
        persist_directory (str, optional): Directory path to save the Chroma database. Defaults to "./chroma_s

    Returns:
```

- `embed-multilingual-v2.0` or `embed-multilingual-v3.0`:
  - **Best for**: High-accuracy, cross-language semantic tasks, especially for retrieval-based systems, clustering, and any application that demands robust multilingual embeddings.
  - **Trade-off**: Higher latency and computational costs.
  - **Version choice**: `v3.0` typically provides better embeddings than `v2.0`, so go with `embed-multilingual-v3.0` for more accuracy.

Retrieval Using Vector Similarity Search:

The `load_and_query_chroma` function loads a persisted Chroma database and queries it based on a provided input (`query_text`). It uses a multilingual embedding model from Cohere to perform similarity searches in the Chroma vector database, returning relevant results based on the query. The function automatically detects the language of the query using the `langid` library and ensures that the system supports queries in either English or Finnish. If the query language is detected as unsupported, it will return an appropriate message. The function performs a similarity search and checks the relevance of the results based on a threshold score of 0.3. If suitable results are found, they are extracted, formatted, and returned. If the query language differs from the language of the search results, the results are translated to match the query language using the Google Translator API. If the results are already in the same language, they are returned as is.

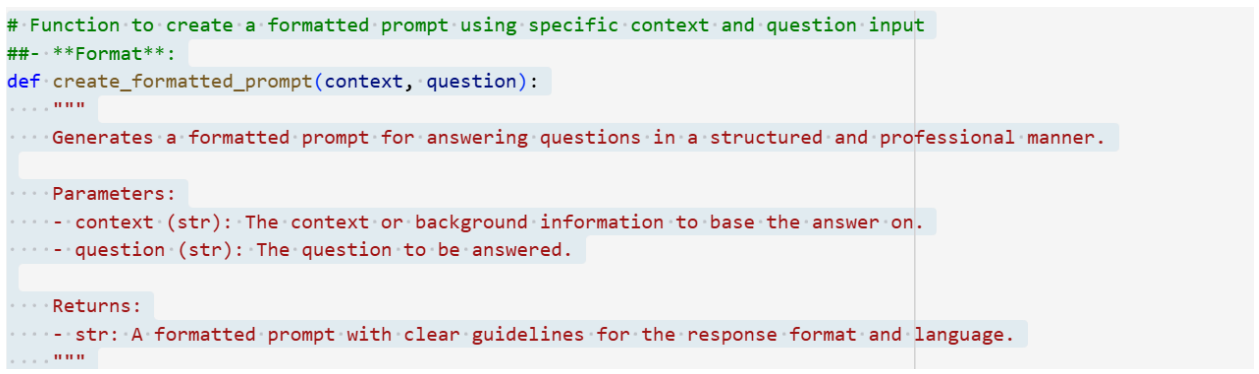


```python
# This Function loads the Chroma DB and queries the relevant documents based on the query text.
def load_and_query_chroma(persist_directory="./chroma_storage", query_text="none", k=3):
    """
    Loads a persisted Chroma database and runs a similarity search on it.
    Automatically detects the language of the query and provides results in the same language.

    Args:
        persist_directory (str): Directory where the Chroma database is stored.
        query_text (str): The input query for similarity search.
        k (int): Number of nearest neighbors to retrieve from the search.

    Returns:
        results (str): List of search results with relevance scores or a message indicating no suitable match.
    """
```

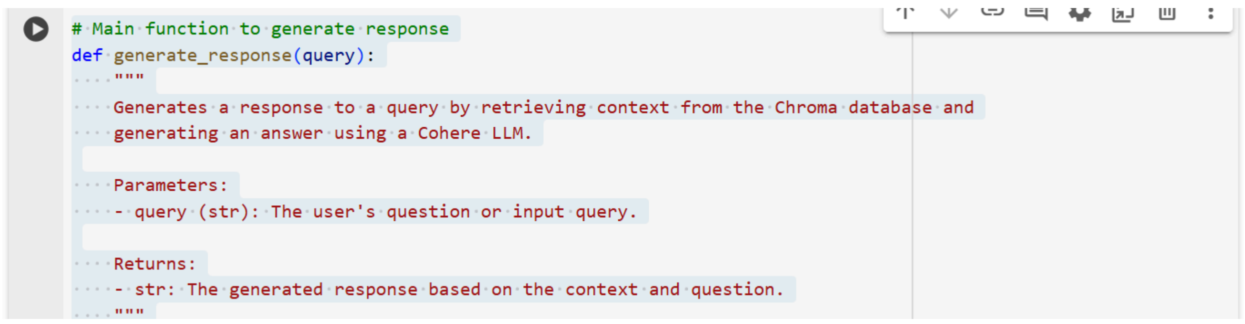
Combining Context and Query :

The create_formatted_prompt function generates a well-structured, professional prompt for answering questions. It takes two inputs: context (the background information to base the answer on) and question (the query to be answered). The function returns a formatted prompt that emphasizes accuracy, clarity, and relevance while ensuring that the response follows a structured and professional style. The prompt is designed to guide the assistant in providing concise, fact-based answers or descriptive paragraphs depending on the nature of the response.

The generate_response function uses this formatted prompt to generate a response to a user's query. It first retrieves the context related to the query from the Chroma database using the load_and_query_chroma function. Then, it passes the context and the user's query to the create_formatted_prompt function to create the appropriate prompt. The function then uses Cohere's LLM (Language Learning Model) to generate an answer based on the context and query, using the LLMChain to run the model and return the generated response.



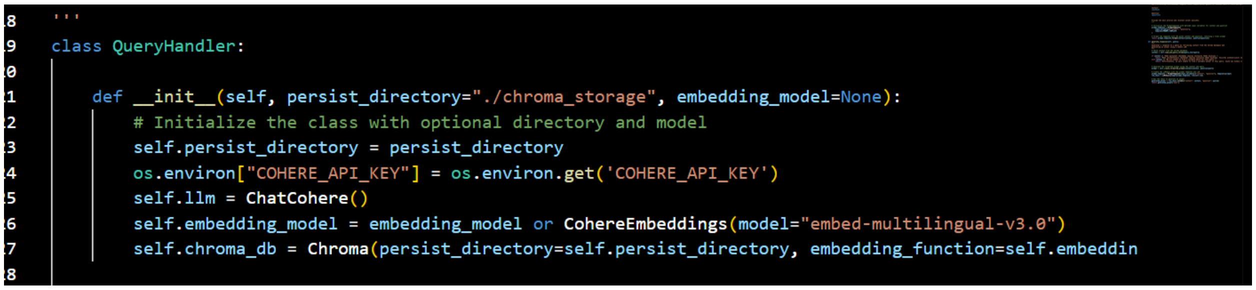
```
# Function to create a formatted prompt using specific context and question input
##- **Format**:
def create_formatted_prompt(context, question):
    """
    Generates a formatted prompt for answering questions in a structured and professional manner.

    Parameters:
    - context (str): The context or background information to base the answer on.
    - question (str): The question to be answered.

    Returns:
    - str: A formatted prompt with clear guidelines for the response format and language.
    """
```



```
# Main function to generate response
def generate_response(query):
    """
    Generates a response to a query by retrieving context from the Chroma database and
    generating an answer using a Cohere LLM.

    Parameters:
    - query (str): The user's question or input query.

    Returns:
    - str: The generated response based on the context and question.
    """
```

Query Handler Class :

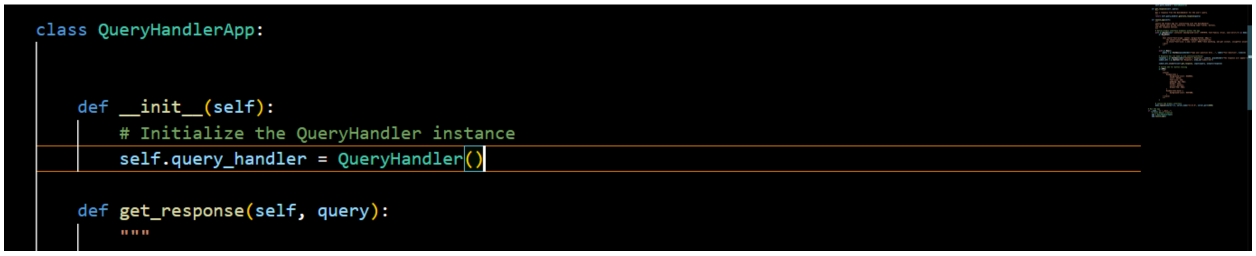
The QueryHandler class is responsible for processing user queries, retrieving relevant context from a Chroma database, and generating accurate, context-aware responses. It supports multilingual queries, automatically detecting the language of the query and translating results when necessary. The class utilizes Cohere's embeddings and language models to enhance the similarity search and response generation. It formats prompts for structured, professional answers and ensures that responses are relevant to the user's question, even supporting both English and Finnish languages.



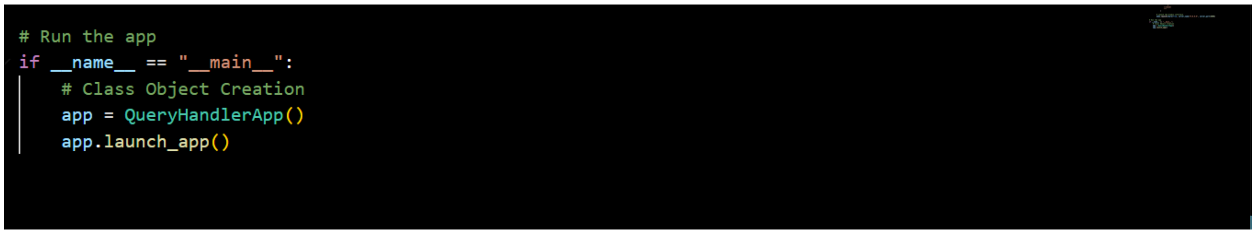
```
'''
class QueryHandler:

    def __init__(self, persist_directory="./chroma_storage", embedding_model=None):
        # Initialize the class with optional directory and model
        self.persist_directory = persist_directory
        os.environ["COHERE_API_KEY"] = os.environ.get('COHERE_API_KEY')
        self.llm = ChatCohere()
        self.embedding_model = embedding_model or CohereEmbeddings(model="embed-multilingual-v3.0")
        self.chroma_db = Chroma(persist_directory=self.persist_directory, embedding_function=self.embeddin

```

Query Handler App Class :

The QueryHandlerApp class provides a **Gradio-based** interface for interacting with a chatbot assistant. The app initializes with a QueryHandler instance to manage query responses and includes a launch_app method for setting up and running the Gradio UI. This interface offers a clean layout with a header, input textbox for user questions, and a response display area. Styled buttons trigger responses, enhancing user experience. When executed, the app runs on a local server, allowing users to type questions and receive instant, relevant answers from the chatbot in a user-friendly format.



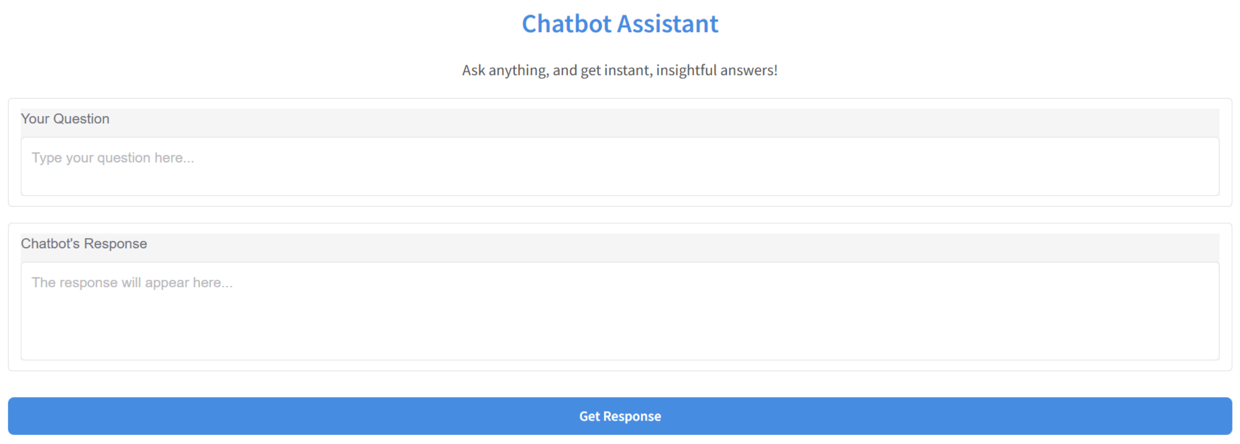
```
class QueryHandlerApp:


    def __init__(self):
        # Initialize the QueryHandler instance
        self.query_handler = QueryHandler()

    def get_response(self, query):
        """
```

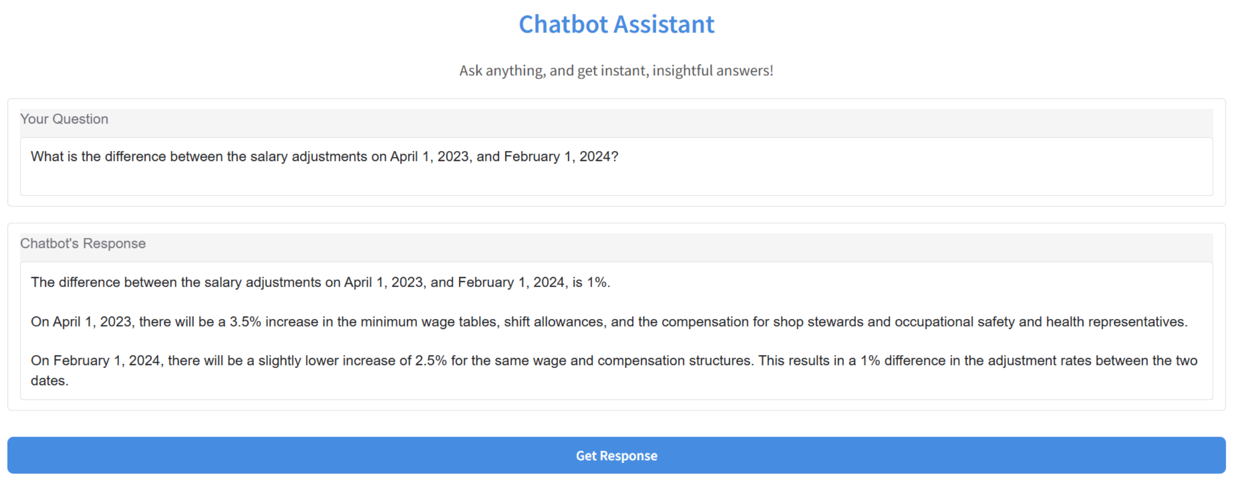


```
# Run the app
if __name__ == "__main__":
    # Class Object Creation
    app = QueryHandlerApp()
    app.launch_app()
```

Front End RAG Question Answer Chatbot Using Gradio :

I use the Gradio app in Python to connect the front-end interface with the query-handling app class for generating responses from the RAG (Retrieval-Augmented Generation) backend. The backend utilizes Chroma DB, where the documents are already stored, enabling the system to retrieve relevant information and generate accurate answers to user queries. This simple front-end chatbot allows users to interact with the RAG model seamlessly, enhancing the overall question-answering experience.


Chatbot Assistant
Ask anything, and get instant, insightful answers!
Your Question
Type your question here...
Chatbot's Response
The response will appear here...
Get Response

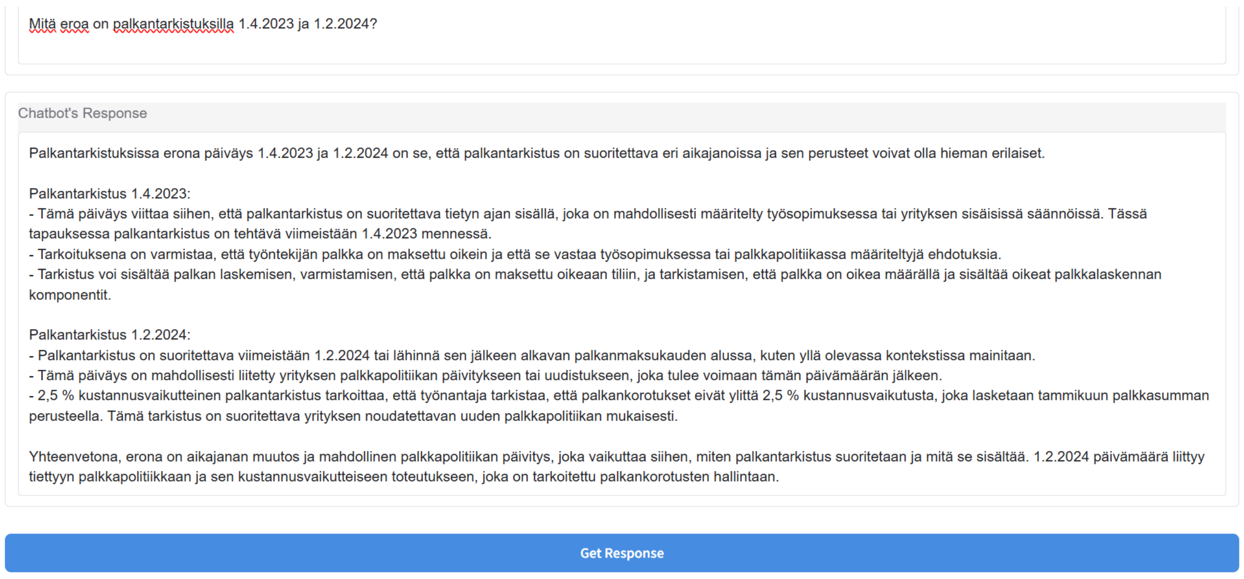


What is the difference between the salary adjustments on April 1, 2023, and February 1, 2024?

The difference between the salary adjustments on April 1, 2023, and February 1, 2024, is 1%.

On April 1, 2023, there will be a 3.5% increase in the minimum wage tables, shift allowances, and the compensation for shop stewards and occupational safety and health representatives.

On February 1, 2024, there will be a slightly lower increase of 2.5% for the same wage and compensation structures. This results in a 1% difference in the adjustment rates between the two dates.


Mitä eroa on palkantarkistuksilla 1.4.2023 ja 1.2.2024?

Palkantarkistuksissa erona päiväys 1.4.2023 ja 1.2.2024 on se, että palkantarkistus on suoritettava eri aikajanoissa ja sen perusteet voivat olla hieman erilaiset.

Palkantarkistus 1.4.2023:
- Tämä päiväys viittaa siihen, että palkantarkistus on suoritettava tietyn ajan sisällä, joka on mahdollisesti määritelty työsopimuksessa tai yrityksen sisäisissä säännöissä. Tässä tapauksessa palkantarkistus on tehtävä viimeistään 1.4.2023 mennessä.
- Tarkoituksena on varmistaa, että työntekijän palkka on maksettu oikein ja että se vastaa työsopimuksessa tai palkkapolitiikassa määriteltyjä ehdotuksia.
- Tarkistus voi sisältää palkan laskemisen, varmistamisen, että palkka on maksettu oikeaan tiliin, ja tarkistamisen, että palkka on oikea määrällä ja sisältää oikeat palkkalaskennan komponentit.

Palkantarkistus 1.2.2024:
- Palkantarkistus on suoritettava viimeistään 1.2.2024 tai lähinnä sen jälkeen alkavan palkanmaksukauden alussa, kuten yllä olevassa kontekstissa mainitaan.
- Tämä päiväys on mahdollisesti liitetty yrityksen palkkapolitiikan päivitykseen tai uudistukseen, joka tulee voimaan tämän päivämäärän jälkeen.
- 2,5 % kustannusvaikutteinen palkantarkistus tarkoittaa, että työnantaja tarkistaa, että palkankorotukset eivät ylitä 2,5 % kustannusvaikutusta, joka lasketaan tammikuun palkkasumman perusteella. Tämä tarkistus on suoritettava yrityksen noudatettavan uuden palkkapolitiikan mukaisesti.

Yhteenvetona, erona on aikajanan muutos ja mahdollinen palkkapolitiikan päivitys, joka vaikuttaa siihen, miten palkantarkistus suoritetaan ja mitä se sisältää. 1.2.2024 päivämäärä liittyy tiettyyn palkkapolitiikkaan ja sen kustannusvaikutteiseen toteutukseen, joka on tarkoitettu palkankorotusten hallintaan.

Short Setup Process and Usage Guide :

Setup Process :

Install dependencies: Use the command pip install -r requirements.txt to install necessary packages.

API Key Configuration:

Add your API key to the .env file in the project directory.

Run the App Locally:

Execute python Gradio_App.py and access the chatbot interface at http://127.0.0.1:8000.

Docker Setup:

Build the Docker container: docker build -t gradio-app .

Run the Docker container:

```
docker run -p 8000:8000 gradio-app
```

Access the chatbot at http://127.0.0.1:8000 after running the container.

Usage :

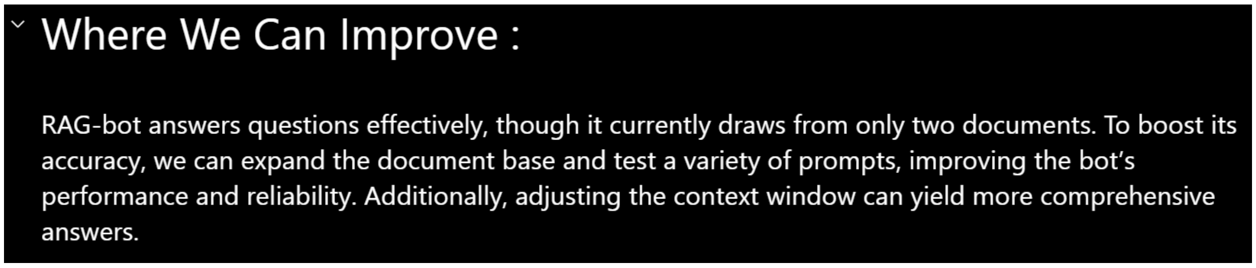
Upload More Documents:

To upload additional documents, open the RAG_Application_Development.ipynb file and specify the directory path. All documents should be placed in this directory to ensure they load into the Chroma storage database.

Interact with the Chatbot:

In the Gradio interface, type a question or query, and the chatbot will retrieve relevant information using the stored documents and provide context-aware responses.

Improvement Area For Our Chatbot :

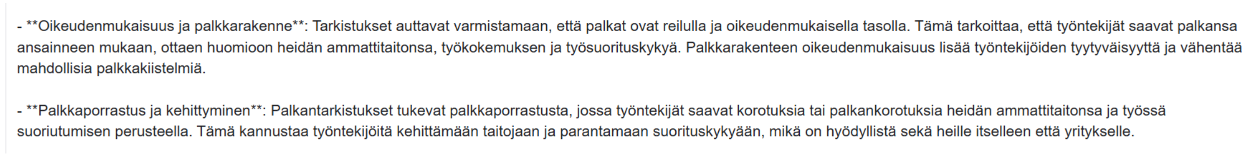


Where We Can Improve :

RAG-bot answers questions effectively, though it currently draws from only two documents. To boost its accuracy, we can expand the document base and test a variety of prompts, improving the bot's performance and reliability. Additionally, adjusting the context window can yield more comprehensive answers.

I Tried But I wasn't able to make the text bold as I am also not an expert In Front End :

Yes, We See Estrick In the Text These Should Be Bold :



```
- **Oikeudenmukaisuus ja palkkarakenne**: Tarkistukset auttavat varmistamaan, että palkat ovat reilulla ja oikeudenmukaisella tasolla. Tämä tarkoittaa, että työntekijät saavat palkansa ansainneen mukaan, ottaen huomioon heidän ammattitaitonsa, työkokemuksen ja työsuorituskykyä. Palkkarakenteen oikeudenmukaisuus lisää työntekijöiden tyytyväisyyttä ja vähentää mahdollisia palkkakiistelmiä.

- **Palkkaporrastus ja kehittyminen**: Palkantarkistukset tukevat palkkaporrastusta, jossa työntekijät saavat korotuksia tai palkankorotuksia heidän ammattitaitonsa ja työssä suoriutumisen perusteella. Tämä kannustaa työntekijöitä kehittämään taitojaan ja parantamaan suorituskykyään, mikä on hyödyllistä sekä heille itselleen että yritykselle.
```