Cresen PharmaGPT: Architecture

Document Search

1. **Introduction**

**A black text on a white background

Description automatically generated**

In the vast ecosystem of artificial intelligence and conversational models, while platforms like OpenAI's GPT offer an expansive range of generic knowledge, they can occasionally fall short when navigating highly specialized domains. Recognizing this gap, Cresen PharmaGPT emerges as a groundbreaking solution, specifically engineered to cater to niche and domain-specific inquiries.

While most standard GPT interfaces, including renowned platforms like Bing, rely heavily on publicly available data, their breadth can sometimes compromise depth in specific areas. When posed with intricate, domain-specific questions, these models might provide generalized or even misaligned answers due to the absence of such specialized content in their foundational datasets.

Cresen PharmaGPT provides a solution in this space. Powered by the robustness of GPT, it doesn't just rely on pre-trained knowledge. Instead, it seamlessly integrates Azure OpenAI's LLM models with advanced retrieval systems. This unique fusion enables the bot to ingest, comprehend, and index vast arrays of PDF documents. These documents serve as a supplementary knowledge reservoir, ensuring that every user query is met with the most accurate and contextually relevant response.

The ability to source and derive insights from a myriad of PDF documents sets Cresen PharmaGPT apart. By doing so, it can bridge the knowledge gap often seen in generic chatbots, ensuring precision and expertise, especially in specialized sectors.

All the resources used to develop this chatbot are from Azure.

**2. Overview of the Workflow**

* **Document Storage**:
  + Store documents in a local directory or Azure Blob Storage.
* **Document Processing**:
  + Sequentially process each PDF using Python code.
  + Extract text content from the PDFs with Python PDF reading modules.
  + Fragment the extracted text into smaller chunks to fit the GPT model's token limit.
* **Model Initialization**:
  + Initialize the GPT model 3.5 Turbo 16k from Azure OpenAI service.
  + Use Azure OpenAI embedding "text-embedding-ada-002" to vectorize the fragmented text chunks.
* **Vector store Configuration and Creation**:
  + Set up and configure Azure Cognitive Search Vector store.
  + Input the fragmented text chunks and their embeddings into the vector store to create an Azure Cognitive Search index.
* **Integration with Langchain Conversation Agent**:
  + Integrate the Azure Cognitive Search index with the Langchain conversation agent's retriever for relevant context retrieval.
* **Migration to Flask Application**:
  + Embed the conversation functionality within a Python Flask application.
  + Configure data endpoints to interface user queries with chatbot responses.
* **Integration into MonitorMate Product**:
  + The Flask application within the MonitorMate product as a new tab for easy user accessibility.

**3. Detailed Workflow**

**3.1 Document Storage**

* **Local Directory/Azure Blob Storage**:

A close up of a screen

Description automatically generated**Local Directory**

1. **Base Classes and Setup**:
   * **BasePDFLoader** is a foundational class to manage PDF files.
     + Identifies if a given file path is local or a URL.
     + Downloads the file temporarily if it's a URL.
     + Verifies if the provided file path is valid.
     + Cleans up temporary files upon deletion.
   * The PyPDFLoader class extends the **BasePDFLoader**.
     + It initializes with the file path and employs the PyPDFParser to extract content.
     + Has functions **load()** and **lazy\_load()** to fetch documents.
2. **Directory PDF Loading**:
   * The **PyPDFDirectoryLoader** class is designed to handle a directory of PDFs.
     + Configurable options like:
       - **glob**: Pattern to find files.
       - **silent\_errors**: Whether to silently handle errors.
       - **load\_hidden**: Option to load hidden files.
       - **recursive**: Load files recursively.
       - **n\_threads**: Number of threads for concurrent processing.
     + Filters items based on their visibility and specified patterns.
     + Progress bar (**tqdm**) to visualize the loading process.
     + Multithreaded approach for parallel processing of files using **ThreadPool**.
     + Keeps track of unique files processed.
     + Errors during the loading process can be logged or raised.
3. **Main Execution**:
   * Initializes the **PyPDFDirectoryLoader** with a directory named "pdfs".
   * Accumulates documents.
   * Computes statistics on the loaded documents, such as:
     + Total number of pages.
     + Total number of characters.
     + Total number of tokens (approximated as characters divided by 4).
     + Estimated price for embedding based on token count.
   * Displays the number of unique PDFs processed.



****

This code serves as a utility to efficiently load, process, and derive insights from a directory filled with PDF documents, taking advantage of multithreading to boost performance.

**Azure Blob Storage**

****

**Functionality**:

1. **Initialization (\_\_init\_\_)**:
   * Sets up a connection to a specific blob in Azure Blob Storage using a connection string, container name, and blob name.
   * Downloads the blob into a temporary file on the local system.
   * Initializes a PDF parser (**PyPDFParser**) to process the content of the PDF.
2. **\_download\_blob\_to\_temp\_file**:
   * Creates a temporary file on the local system.
   * Downloads the blob's content into this file using a stream.
   * Returns the path of the temporary file.

**AzureBlobPDFDirectoryLoader:**

This class is a bulk loader that processes multiple PDFs from an Azure Blob Storage container concurrently.

**Functionality**:

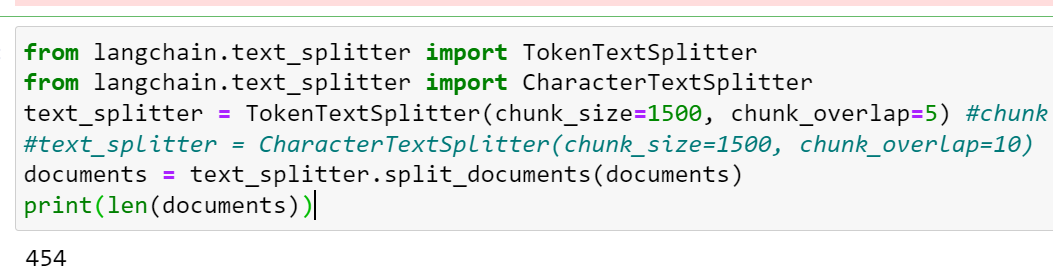
1. **Initialization (\_\_init\_\_)**:
   * Sets basic attributes like connection string, container name, number of threads for parallel processing, and whether to silently handle errors.
2. **load**:
   * Connects to the Azure Blob Storage container using the provided connection string.
   * Lists all blobs (files) inside the specified container.
   * For each blob:
     + Downloads and processes its content in parallel using multiple threads.
     + Attaches the blob's name as metadata for each processed document.
     + Handles exceptions, either logging them or raising them based on the **silent\_errors** flag.

**Main Execution:**

1. An instance of **AzureBlobPDFDirectoryLoader** is created targeting the "drug-pdfs" container in Azure Blob Storage.
2. All documents (PDFs) in the specified Azure Blob Storage container are downloaded, processed, and added to the **documents** list.
3. The code calculates and prints:
   * Total number of documents processed.
   * Total number of characters across all documents.
   * A derived value for the total number of tokens.
   * An estimated price for embedding based on the total number of tokens.

The code offers an integration to download, process, and extract data from PDFs stored in Azure Blob Storage. It provides both individual blob processing (**AzureBlobPDFLoader**) and bulk processing for all PDFs in a container (**AzureBlobPDFDirectoryLoader**). After retrieving all documents from the Azure container "drug-pdfs", the code prints a summary of the data size and an estimated price for embedding.

**3.2 Document Processing and Text Extraction**

****

**Text Extraction**:

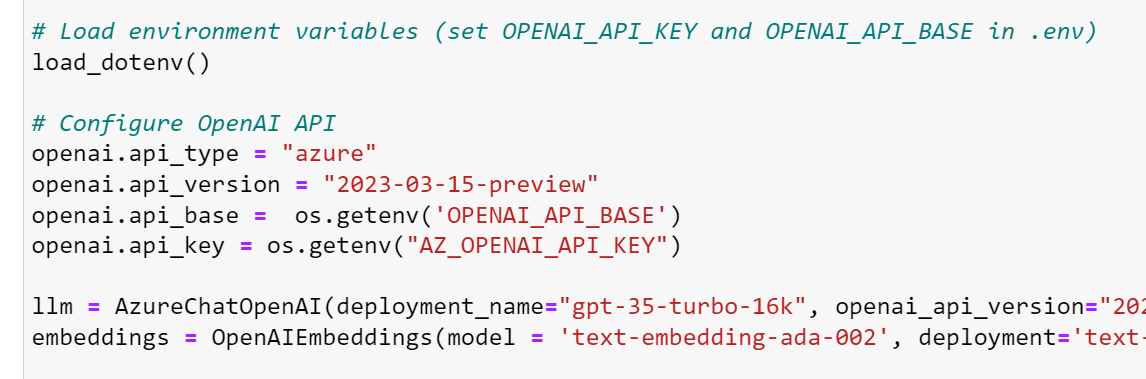
Text extraction from PDFs is a critical task in this pipeline. The code utilizes the **PyPDFParser** class, which uses library **PyPDF2**. This class is responsible for parsing the PDF content and converting it into a format suitable for further processing.

**Text Fragmentation**:

Post extraction, the text data from PDFs might be vast, especially for lengthy documents. Processing such vast data in one go can be computationally challenging and may not align with the input constraints of some models, like the GPT model. The GPT model has a maximum token limit, and processing beyond this limit will result in errors.

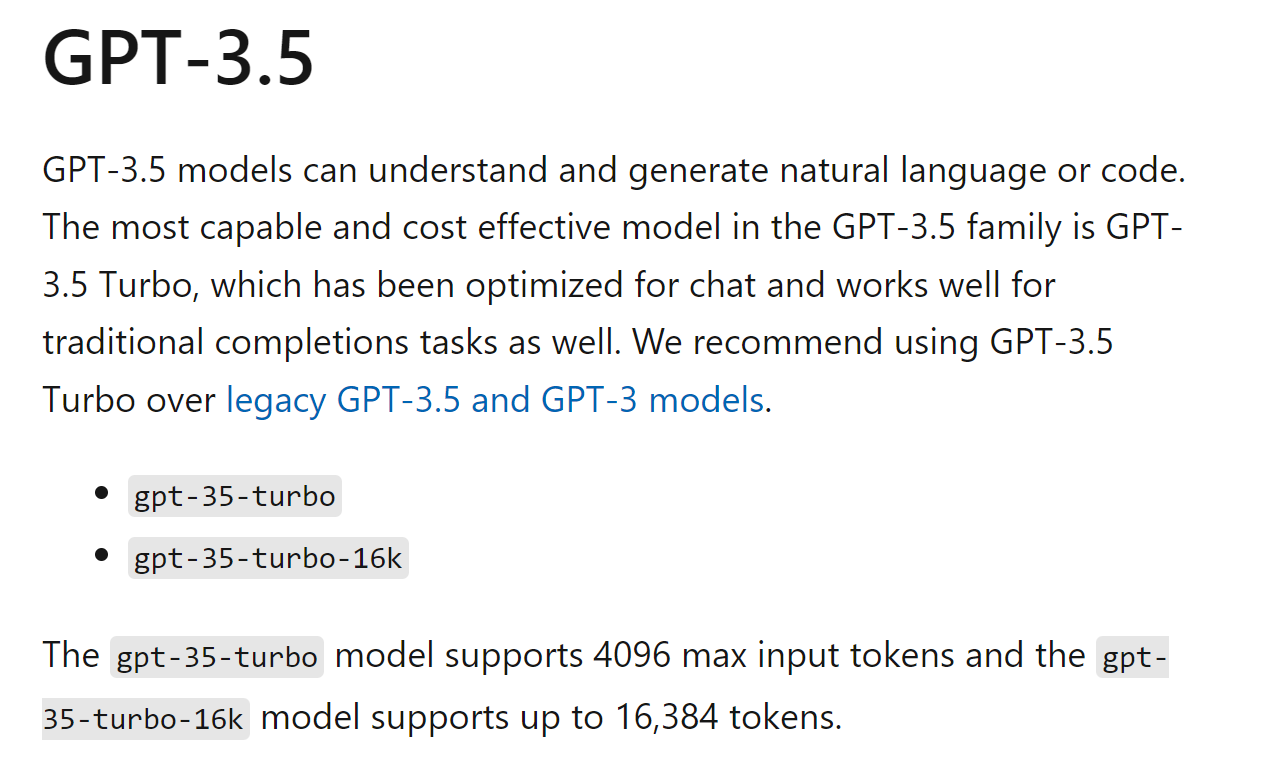
To manage this, the code employs a text-splitting strategy. The **TokenTextSplitter** and **CharacterTextSplitter** classes from the **langchain.text\_splitter** module are designed to break down the text into smaller, manageable chunks, ensuring each chunk stays within a model's input constraints. The **TokenTextSplitter**, as the name suggests, fragments text based on tokens, ensuring chunks of text are of a specified size. The inclusion of a 'chunk overlap' ensures that no information is lost between the splits, providing a small overlap of tokens or characters between consecutive chunks.

In the above code snippet, the **TokenTextSplitter** is initialized with a chunk size of 1500 and an overlap of 5 tokens. This means that each chunk will contain approximately 1500 tokens, with the last 5 tokens of one chunk being the first 5 of the next, ensuring continuity and context preservation. The **CharacterTextSplitter** works similarly but operates on characters instead of tokens.

**3.3 Model Initialization**

The keys are stored in .env file in the same directory.

**GPT Model 3.5 Turbo 16k**:

<https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models#gpt-35>

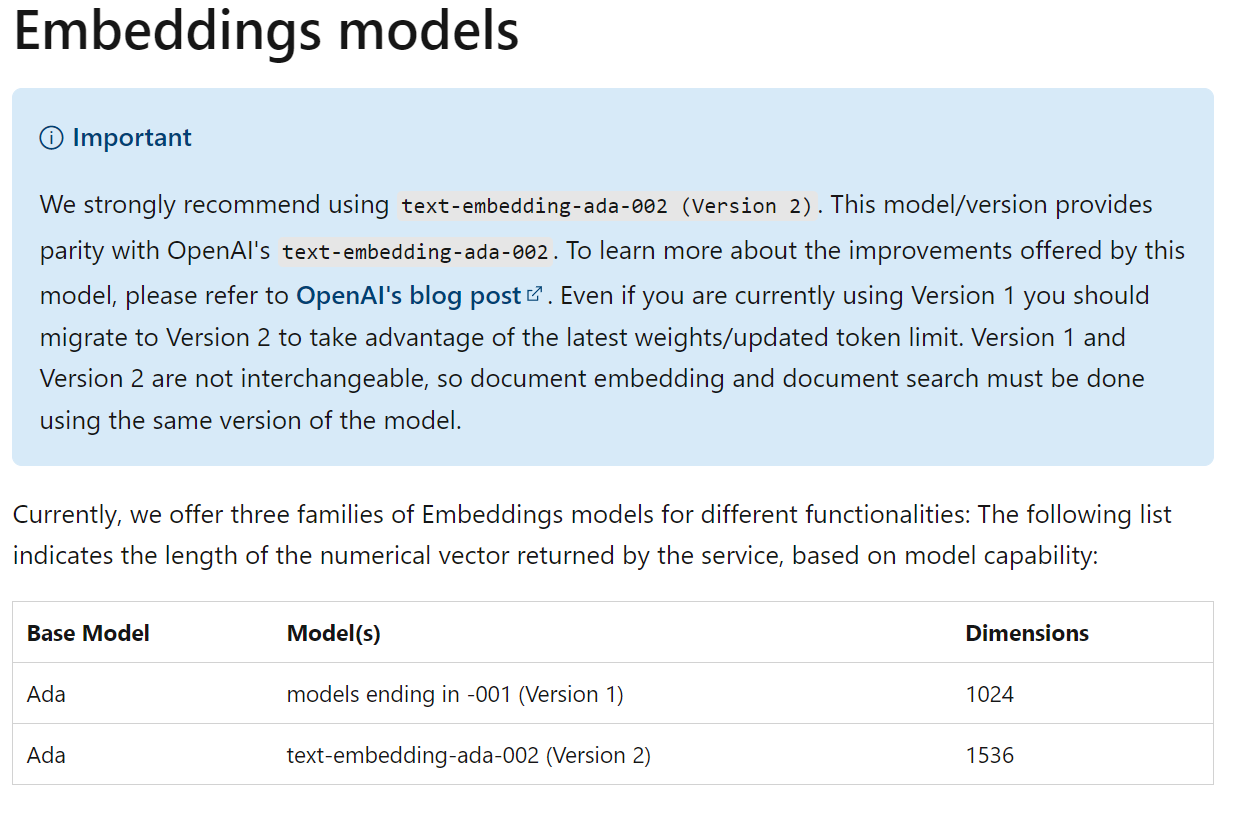
**Description**: The GPT (Generative Pre-trained Transformer) Model 3.5 Turbo 16k is one of the latest iterations of OpenAI's transformer-based language models. The "16k" indicates the model's ability to handle up to 16,000 tokens in a single pass. Such capacity allows it to process longer conversations, documents, or sequences of text, making it highly efficient for diverse natural language processing tasks.

**Functionalities**:

* **Natural Language Understanding**: GPT 3.5 Turbo excels in understanding context, sentiment, and nuances in text. This ability allows it to interpret user input effectively and provide meaningful responses.
* **Context Preservation**: Its large token capacity ensures that longer conversations maintain context, making it adept at multi-turn conversations where understanding previous interactions is crucial.
* **Versatility**: Beyond chat services, GPT 3.5 Turbo can be used for summarization, translation, question-answering, and even code generation.

**Service to Chat**: In chat applications, the GPT 3.5 Turbo 16k provides real-time, coherent, and contextually relevant responses. Its design ensures that the chatbot retains the context of the conversation, allowing for seamless interactions. Moreover, with its ability to understand and generate human-like text, the user experience is highly interactive, akin to chatting with a human.

**Text-Embedding-Ada-002**:



**Description**: Text-Embedding-Ada-002 is a dedicated embedding model whose primary role is to transform textual data into numerical vectors. These vectors, often termed embeddings, represent the semantics of the text in multi-dimensional space. Embeddings are crucial for machine learning tasks as they provide a way to quantify and compare textual information.

**Role in Fragmented Text Chunks**:

* **Vectorization**: This model takes the fragmented text chunks and converts each one into a dense vector. Each dimension in this vector captures some aspect of the text's semantics.
* **Semantic Encoding**: The embeddings retain the semantic essence of the text. Texts with similar meanings will have embeddings close to each other in the vector space, enabling similarity checks and categorization tasks.
* **Data Reduction**: By representing long text fragments as fixed-size vectors, the model effectively reduces the dimensionality of the data while preserving its informational content.
* **Model Interoperability**: Once the fragmented texts are vectorized, they can be fed into a variety of machine learning models for tasks such as clustering, classification, or anomaly detection. It bridges the gap between raw text data and models that require numerical input.
* **Query Search & Comparison**: Using the embeddings, one can perform operations like semantic search. For instance, given a query, one can find the most semantically relevant text chunk by comparing the query's embedding with the embeddings of the text chunks.

In essence, Text-Embedding-Ada-002 plays a foundational role in transforming the fragmented text chunks into a format suitable for context based chatbots.

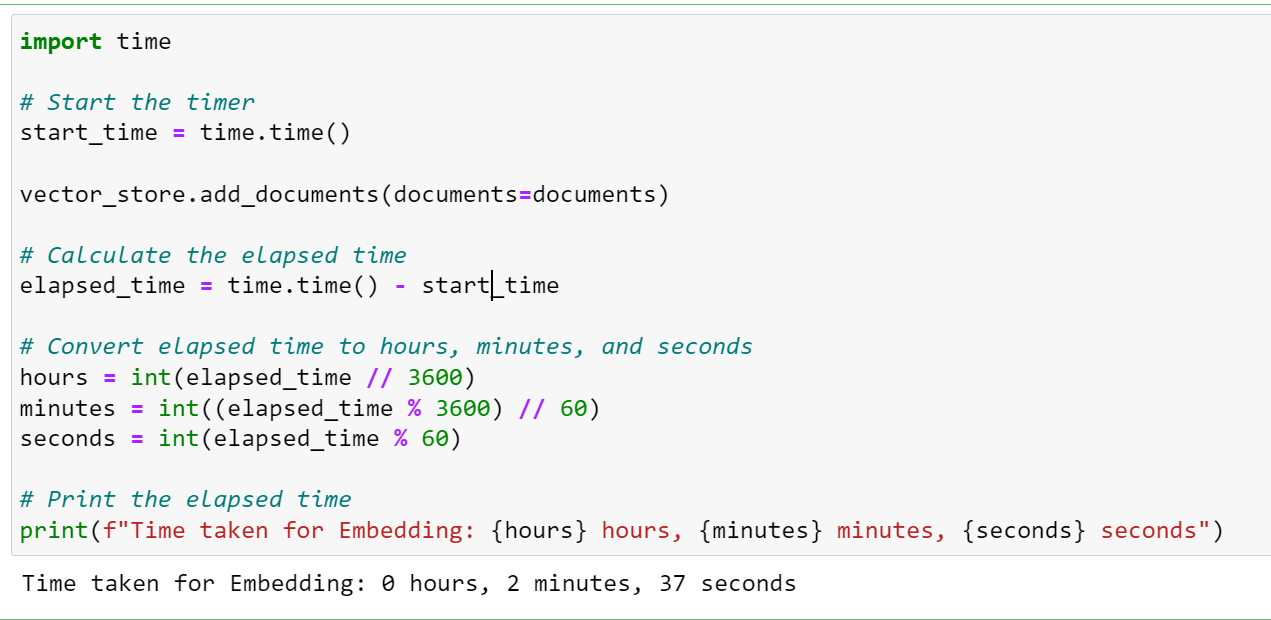
**3.4 Vector store Configuration**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer code

Description automatically generated**

****

**Azure Cognitive Search Vector Store**:

**Initialization & Configuration**: Azure Cognitive Search Vector Store facilitates the storage of numerical vectors generated from textual data. To initiate, one would typically define the address and password of the Azure Search Service, as depicted in the code screenshot where environment variables store these configurations (**AZURE\_SEARCH\_SERVICE\_ENDPOINT** and **AZURE\_SEARCH\_ADMIN\_KEY**).

**Purpose**: The core objective of the Azure Cognitive Search Vector Store is to house embeddings (vectors) derived from raw textual data. These embeddings capture the semantic essence of the text and are instrumental in various machine learning and data retrieval tasks, such as semantic search and document clustering.

**Advantages**:

1. **Scalability**: Azure platform's robustness ensures the storage can handle vast amounts of data seamlessly.
2. **Security**: Given the sensitivity of some data, Azure provides top-notch security configurations.
3. **Integration**: Azure Cognitive Search seamlessly integrates with other Azure services, enhancing workflow efficiency.
4. **Semantic Search**: Storing both text and its corresponding vector enables powerful semantic search capabilities.

**Azure Cognitive Search Index**:

**Process**: Once the fragmented text chunks are transformed into embeddings via a model like Text-Embedding-Ada-002, these vectors, along with their corresponding raw data, are ready to be ingested into the Azure Cognitive Search Index.

1. **Embeddings Initialization**: The **OpenAIEmbeddings** class is initialized, pointing to a specific deployment and model. It's configured to chunk the data and connect to an Azure-hosted OpenAI endpoint.
2. **Vector Store Configuration**: The **AzureSearch** class is initialized with the endpoint details, the index name, and the embedding function. It's further configured semantically using the **SemanticSettings** class, which prioritizes certain fields in the data like content and metadata.
3. **Data Ingestion**: The **add\_documents** method of the **AzureSearch** instance ingests the list of documents (both raw data and embeddings) into the Azure Cognitive Search Index.

**A screenshot of a computer

Description automatically generatedOutcome**: The result is a rich index on Azure that holds both the raw textual data and its numerical representation. This dual storage allows for conventional keyword-based searches and advanced semantic searches based on the vector space.

**3.5 Integration into Langchain Conversation Agent's Retriever**



**Accessibility of the Azure Cognitive Search Index**:

The Langchain conversation agent leverages Azure Cognitive Search Index to provide intelligent conversational experiences. Once the raw text and its corresponding embeddings are stored in the index, the conversation agent can access this indexed data through specific APIs and libraries, which are abstracted by Langchain’s inbuilt functions.

**Retriever Mechanism**:

The Retriever plays a crucial role in fetching the most relevant context based on user queries. Here's a breakdown:

1. **Prompt Templates**: The initial step is to define the nature of the interaction using prompt templates. The provided code features:
   * **System Message Prompt**: This sets the context for the conversation agent, which in this case is "CresenPharmaGPT". It gives explicit guidelines on how the agent should behave and respond, making it act like a healthcare compliance expert.
   * **User Message Prompt**: A straightforward format to capture the user's question.
   * **Chat Prompt**: This merges the system and user prompts, structuring the chat.
2. **Retriever Initialization**:
   * The **vector\_store.as\_retriever** method initializes the retriever. It's set to use a similarity search type, aiming to fetch the top 8 (**k:8**) most relevant documents based on the embedded vector's similarity.
   * The **ConversationalRetrievalChain** class connects the Logical Language Model (**llm**) and the retriever. It ensures that during the conversation, the retriever is invoked to fetch relevant documents based on the provided user query.
3. **Querying Mechanism**:
   * When a user poses a query, it's transformed into a prompt using the earlier defined templates.
   * The retriever, now connected to the Azure Cognitive Search Index, searches for the most semantically similar documents.
   * Relevant documents are fetched based on the similarity of their embedded vectors to the vector representation of the user's query.

In essence, the Langchain conversation agent seamlessly taps into the Azure Cognitive Search Index to extract the most pertinent information based on the context set by the user's query. This integration ensures that the conversation is both contextual and informative, delivering a richer user experience.

**3.6 Migration to Flask Application**

****

****

**Role of Python Flask Framework**: Flask is a lightweight web framework in Python that's used for building web applications. In the context of managing the conversation functionality, Flask facilitates the creation and management of web services, enabling the chatbot to interface with users via web browsers. This helps in hosting and exposing the conversational capabilities of the Langchain system to users, allowing for real-time interaction.

**Endpoints Configuration**:

1. **Root (/)**:
   * **Purpose**: This endpoint serves the main web page of the application, often used as the landing page.
   * **Function**: Returns the **index.html** template which may serve as the user interface for the chat.
2. **Data (/data)**:
   * **Purpose**: The main interaction point for the user to communicate with the chatbot.
   * **Function**: Accepts a POST request containing the user’s question. Processes the question, retrieves the relevant answer from the conversation agent, and appends it to the chat history. It then returns the answer in JSON format.
3. **Clear (/clear)**:
   * **Purpose**: Provides a mechanism to reset or clear the chat history for the current session.
   * **Function**: Accepts a POST request to reset the chat history and returns a JSON response confirming the action.

**Code Overview**:

1. **Environment Configuration**:
   * The environment variables are loaded using **load\_dotenv()**. This helps in securely accessing secrets and configurations like API keys and endpoints.
2. **OpenAI Configuration**:
   * The OpenAI module is initialized with Azure as the API type, with details like API keys and versions fetched from environment variables.
3. **Initialization**:
   * Two primary models are initialized: **AzureChatOpenAI** (for chat) and **OpenAIEmbeddings** (for embeddings). The former facilitates communication using the GPT-3.5 Turbo model, and the latter aids in generating embeddings for given text using the ADA model.
4. **Vector Store Configuration**:
   * An Azure Search vector store instance is created. This acts as the bridge between the embedded textual data and the Azure Cognitive Search services.
5. **Retrieval Chain Setup**:
   * Conversational prompts are set up to guide the behaviour of the chat model.
   * The retrieval chain integrates the logical language model with the vector store retriever to provide an interactive and informative conversational experience.
6. **Flask Application Setup**:
   * A new Flask application instance is created, and CORS (Cross-Origin Resource Sharing) is enabled to allow requests from different origins.
   * Routes (or endpoints) are defined for the application to manage user interactions and deliver appropriate responses.
7. **Session Management**:
   * The Flask app uses sessions to manage and store chat history for individual users. This ensures continuity in conversation across multiple interactions.

The application effectively migrates the conversational capabilities of the Langchain system to a web environment, ensuring real-time, interactive, and user-friendly access to the chatbot's functionalities.

**3.7 Integration into MonitorMate Product**

This flask application and its endpoints are displayed as a new tab in our MonitorMate product.

**A screenshot of a computer

Description automatically generated**

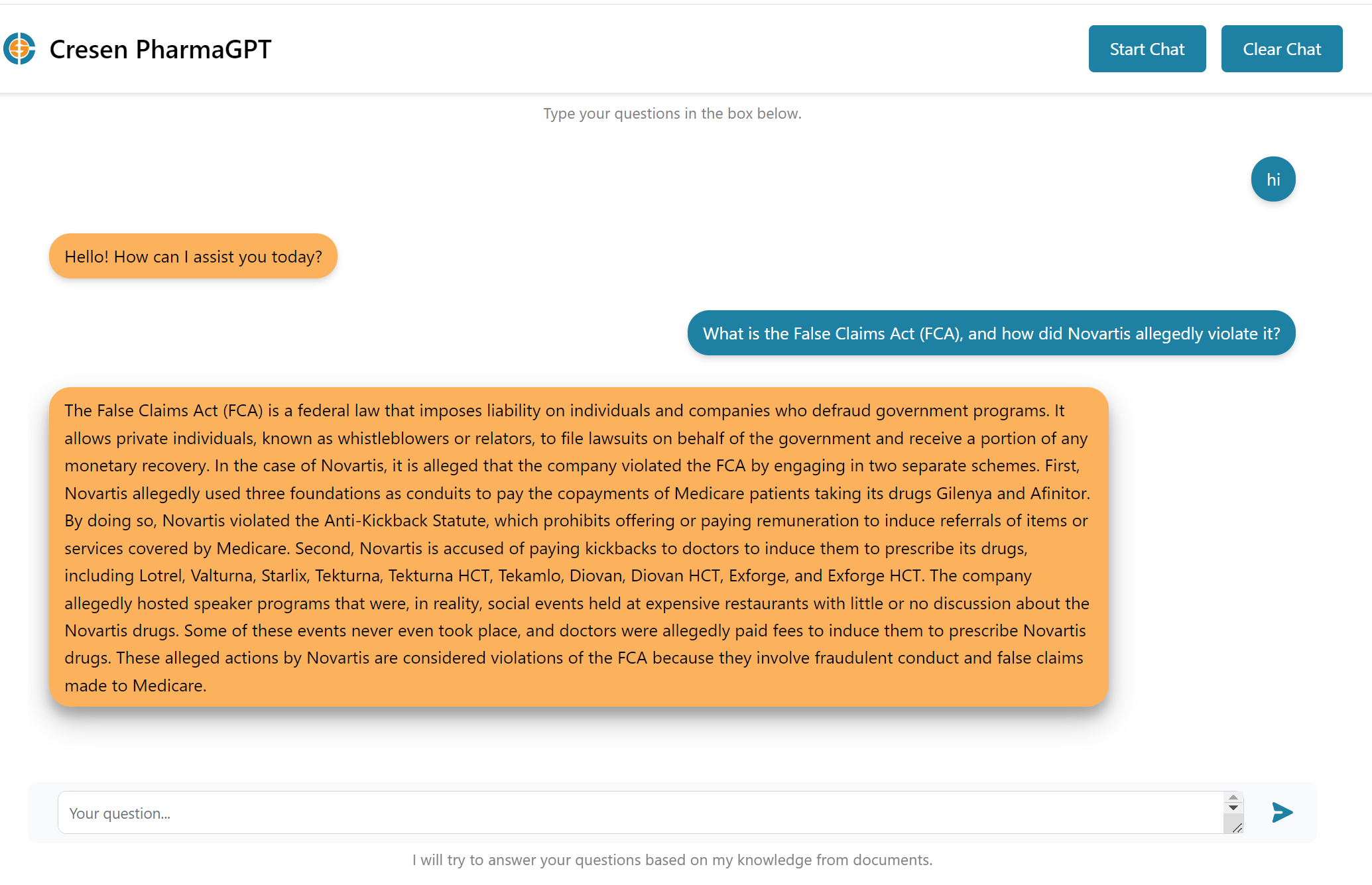
**1. Frontend Chatbot Integration:**

* **Chatbot Icon Creation:** On the homepage, a Chatbot Icon will be added. When clicked, it will redirect users to the Chatbot component, accessible via the link cresendemomm-monitoring/chatbot/'.
* **Chatbot Component Design:** A dedicated Chatbot component will be designed.
* **Dynamic Div Bundling:** Dynamic div elements will be utilized to bundle both user requests and chatbot responses.
* **Microservices Integration:** Requests sent by the user will trigger calls to relevant microservices.

**2. Intermediate Microservices:**

* **Feign Calls to REST APIs:** The microservices will make Feign calls to REST API endpoints. Two methods will be implemented, each returning ResponseType, ResponseData, and ResponseStatus.
* **Configuration via Application.yml:** Hitpoints (e.g., 'https://cresenpharmagpt.azurewebsites.net/') will be saved in the application.yml file for easy management of API calls.
* **Logging in Database:** Upon receiving responses from the APIs, the microservices will save both the request and response data using objects in the database.
* **Returning Responses to Frontend:** The microservices will then return the processed responses to the Angular frontend application.

**4. Cresen PharmaGPT Interface**

****

**5. Security and Compliance**

All the resources and endpoints used in this product are within Azure.

**Data, privacy, and security for Azure OpenAI Service :**

<https://learn.microsoft.com/en-us/legal/cognitive-services/openai/data-privacy?context=%2Fazure%2Fcognitive-services%2Fopenai%2Fcontext%2Fcontext>

**Microsoft Products and Services Data Protection Addendum (DPA) :**

<https://www.microsoft.com/licensing/docs/view/Microsoft-Products-and-Services-Data-Protection-Addendum-DPA>

**6. References**

Azure Cognitive Search :

<https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search>

Langchain Conversation Retrieval Agent :

<https://python.langchain.com/docs/use_cases/question_answering/how_to/conversational_retrieval_agents?ref=blog.langchain.dev>

Python Flask :

<https://pythonbasics.org/what-is-flask-python/>

Azure OpenAI :

<https://azure.microsoft.com/en-us/products/ai-services/openai-service>

Vector search within Azure Cognitive Search :

<https://learn.microsoft.com/en-us/azure/search/vector-search-overview>

Retrieval Augmented Generation (RAG) in Azure Cognitive Search :

<https://learn.microsoft.com/en-gb/azure/search/retrieval-augmented-generation-overview>