**Bearchat**

Bearchat is a course assistant RAG chatbot developed as a proof of concept for the University of Cincinnati's Lindner College of Business. The RAG (Retrieval-Augmented Generation) chatbot is a sophisticated conversational AI system designed to provide accurate and contextually relevant responses by leveraging advanced search and language models. This tool is designed to provide students with tailored support for their coursework queries. By leveraging UC’s proprietary documents and instructors' course manuals, Bearchat generates precise, context-specific responses, enhancing the academic experience and streamlining access to essential information. Its integration with the university's academic resources ensures that students receive relevant and timely assistance, ultimately fostering a more efficient and informed learning environment.

It integrates Azure AI Search and OpenAI's GPT-4-o to handle queries based on both a knowledge base and historical interaction memory.

**Frontend:**

For POC, Q&A interface was developed using Streamlit

**Streamlit app (used in document\_management.py and main app script)**

**•Interface:** Built with Streamlit, providing a user-friendly interface for interacting with the chatbot.

**•Document Upload:** Users can upload documents via the Streamlit interface. The file is sent to an Azure Function for processing.

**•Question Handling:** Users can ask questions, which are sent to an Azure Function for generating answers.

**•** The app provides a simple interface for users to ask questions. User questions are sent to an Azure Function which returns answers. Questions and answers are stored and displayed in the app interface. The app ensures that users can see a history of their interactions.

**Backend:**

**Azure Functions**

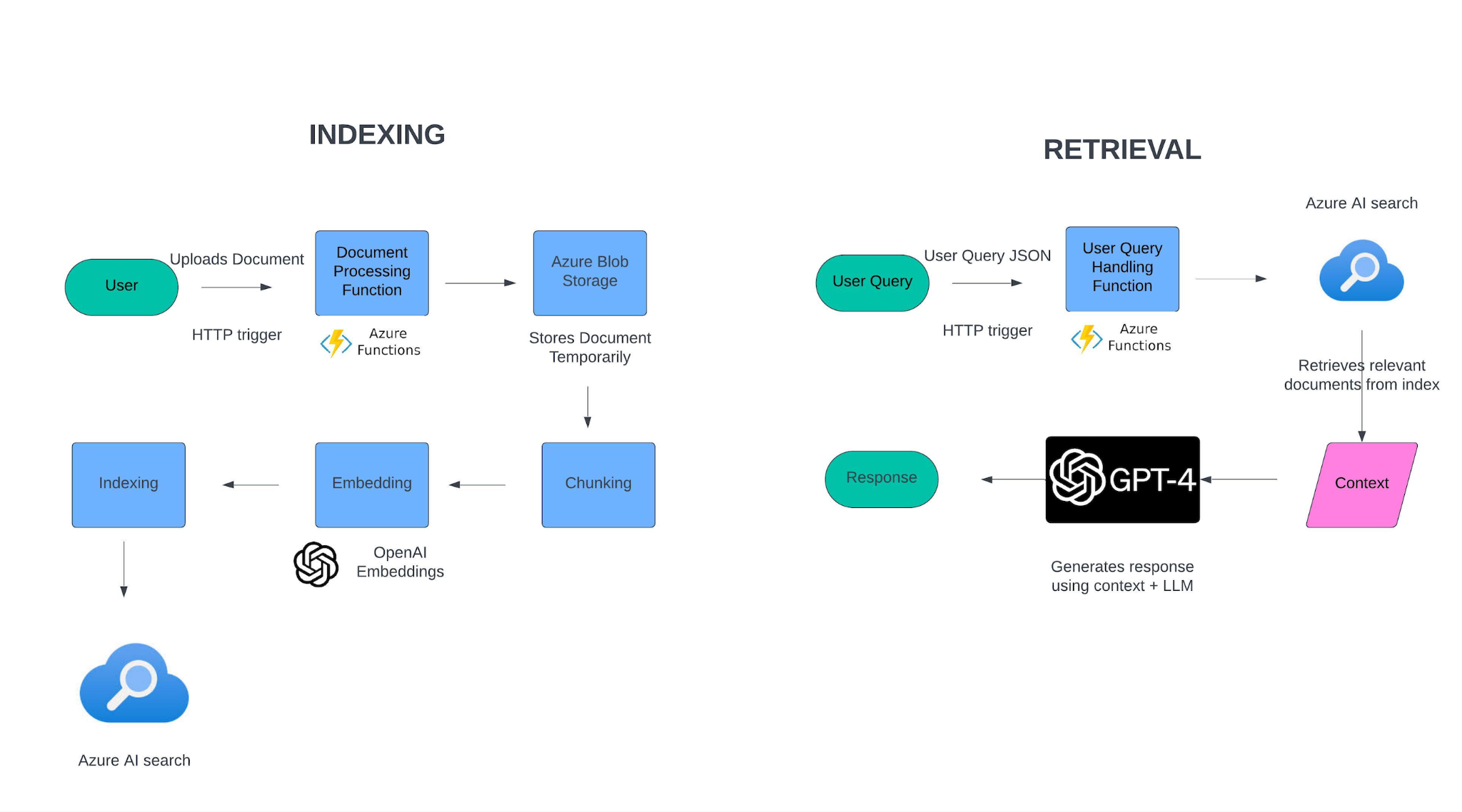
**Function App:** Registers the DataProcessing and userQuery blueprints, this is the main entry point for your Azure Functions application

**1. Document Processing:** Processes uploaded documents (e.g., PDFs), extracts text, chunks it into smaller pieces, embeds the text using OpenAI's embedding models, and indexes it in Azure Search.

**2. User Query Handling:** Handles user queries, retrieves relevant documents from the index, generates a response using OpenAI's GPT model, and provides an answer.

**Architecture:**

The chatbot system is composed of several key components. To delve deep into these components we divide them into 2 categories . INDEXING and RETRIEVAL

****

## Indexing

Indexing involves preparing and storing data so that it can be efficiently retrieved later. This process includes:

### 1. Document Processing

* **Component**: DataProcessing\_blueprint.cpython-311.pyc
* **Function**: Processes uploaded PDF documents, extracts text

**2. Chunking Process**

* **Purpose:** Break large documents into manageable pieces.
* **Tools Used:** RecursiveCharacterTextSplitter from langchain\_text\_splitters.
* **Process:**

1. The documents are read, and text content is extracted using PyPDF2.
2. The text is chunked into manageable sizes for embedding.
3. Define separators (sentence endings and word breaks)
4. Concatenate document text.
5. Split text into chunks of ~300 characters with 20 character overlap.

**3. Embedding Process**

**•Purpose:** Convert text chunks into vector representations.

**•Tool Used:** OpenAI Embedding Model (text-embedding-3-small).

**•Process:**

1. Clean text to remove unwanted characters.
2. Generate embeddings using OpenAI's API.

**4. Indexing Process**

**•Purpose: Store embedded text chunks in a searchable index.**

**•Tool Used: Azure AI Search.**

**•Process:**

1. Create unique IDs for each chunk.
2. Store chunk content, embedding, and page number.

### Other core components

**OpenAI API**: Used for generating responses and embeddings.

* **Embedding Service**: Generates vector embeddings for the documents and queries.
* **Chat Completion Service**: Generates natural language responses based on the provided context and user queries.

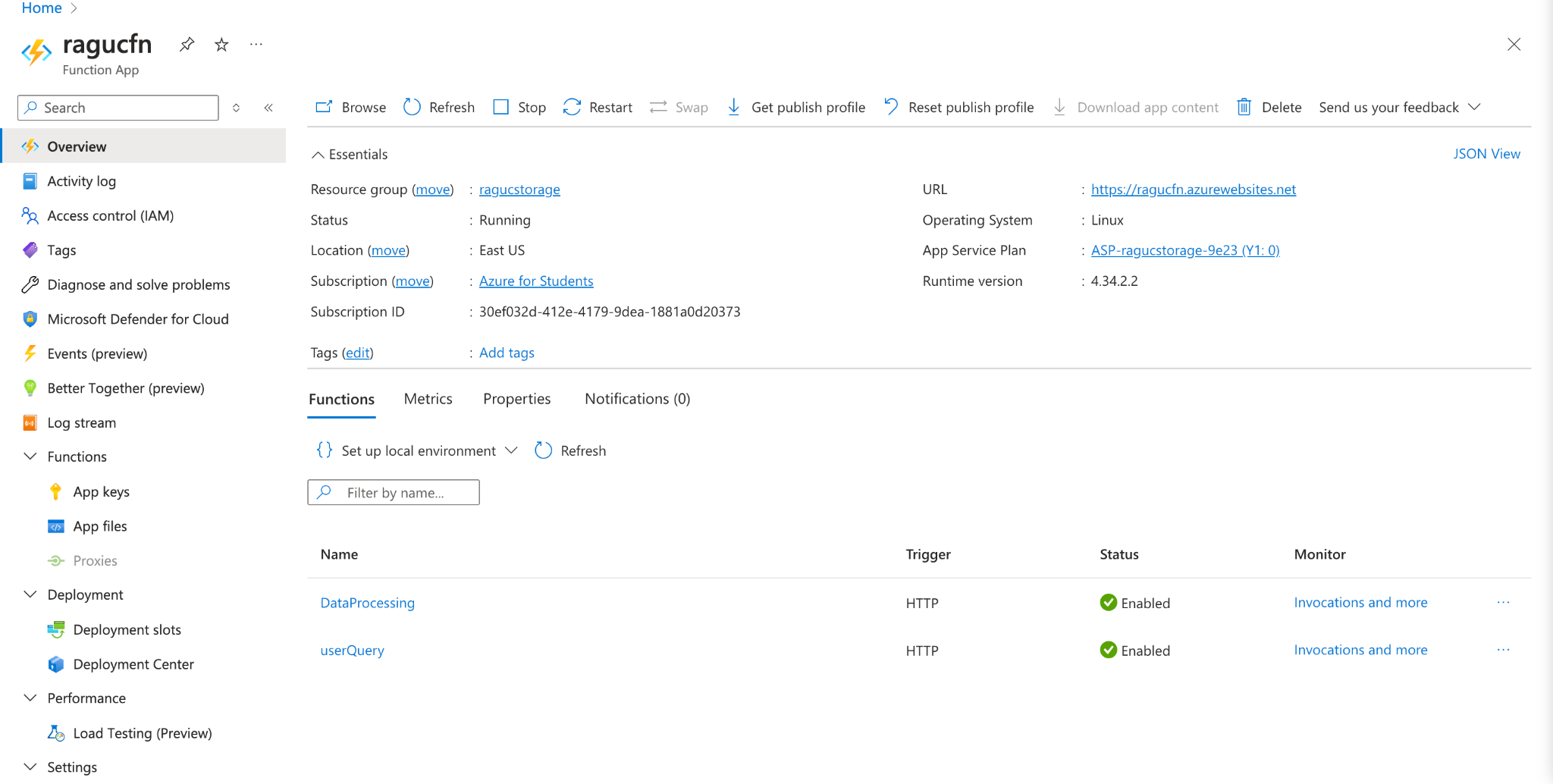
**Azure Blob Storage**: Stores the uploaded documents temporarily for processing.

**Azure configurations:**

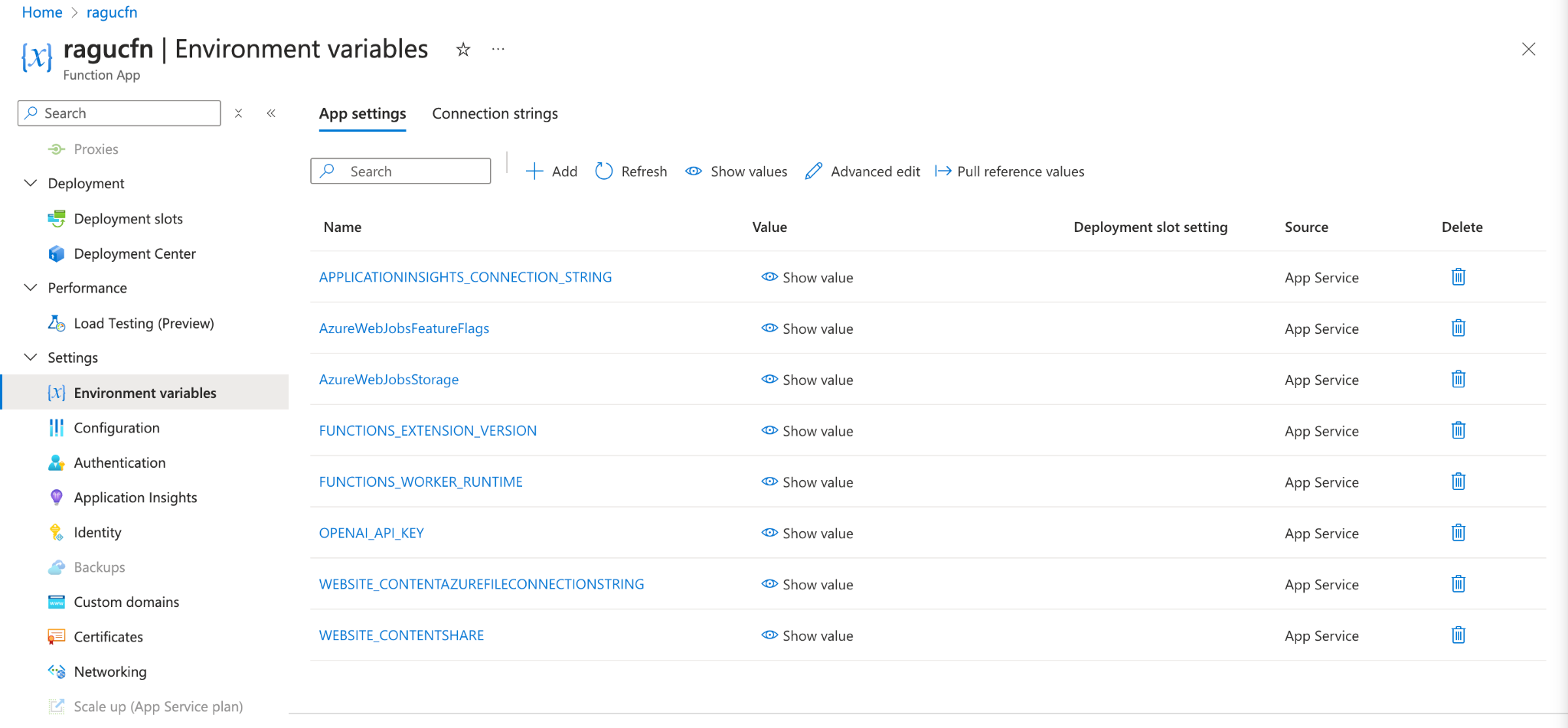
1. **Storage** : a storage container to store function and ai search

2. **Azure Functions App:** The core of the RAG chatbot.

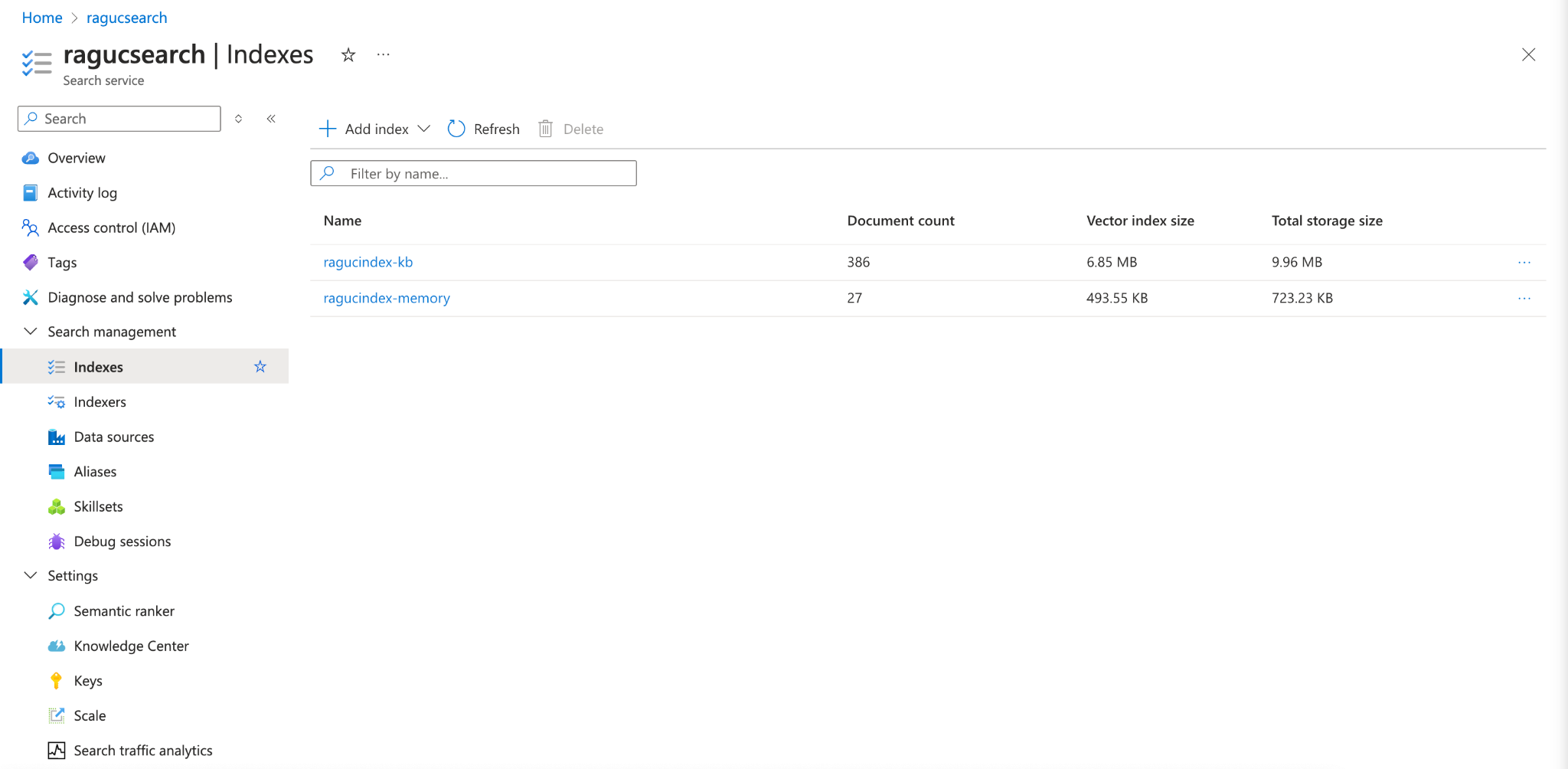
* HTTP Trigger Functions:
  + **DataProcessing Function**: Handles file uploads, processes documents, and indexes them in Azure AI Search.
  + **UserQuery Function**: Handles user queries, retrieves relevant documents, and generates responses using Gpt 4-o.



* Adding OPEN AI key to the environment variable in function should be done manually. Unlike the Azure app functions , OPEN AI key doesnt register itself when code in pushed.

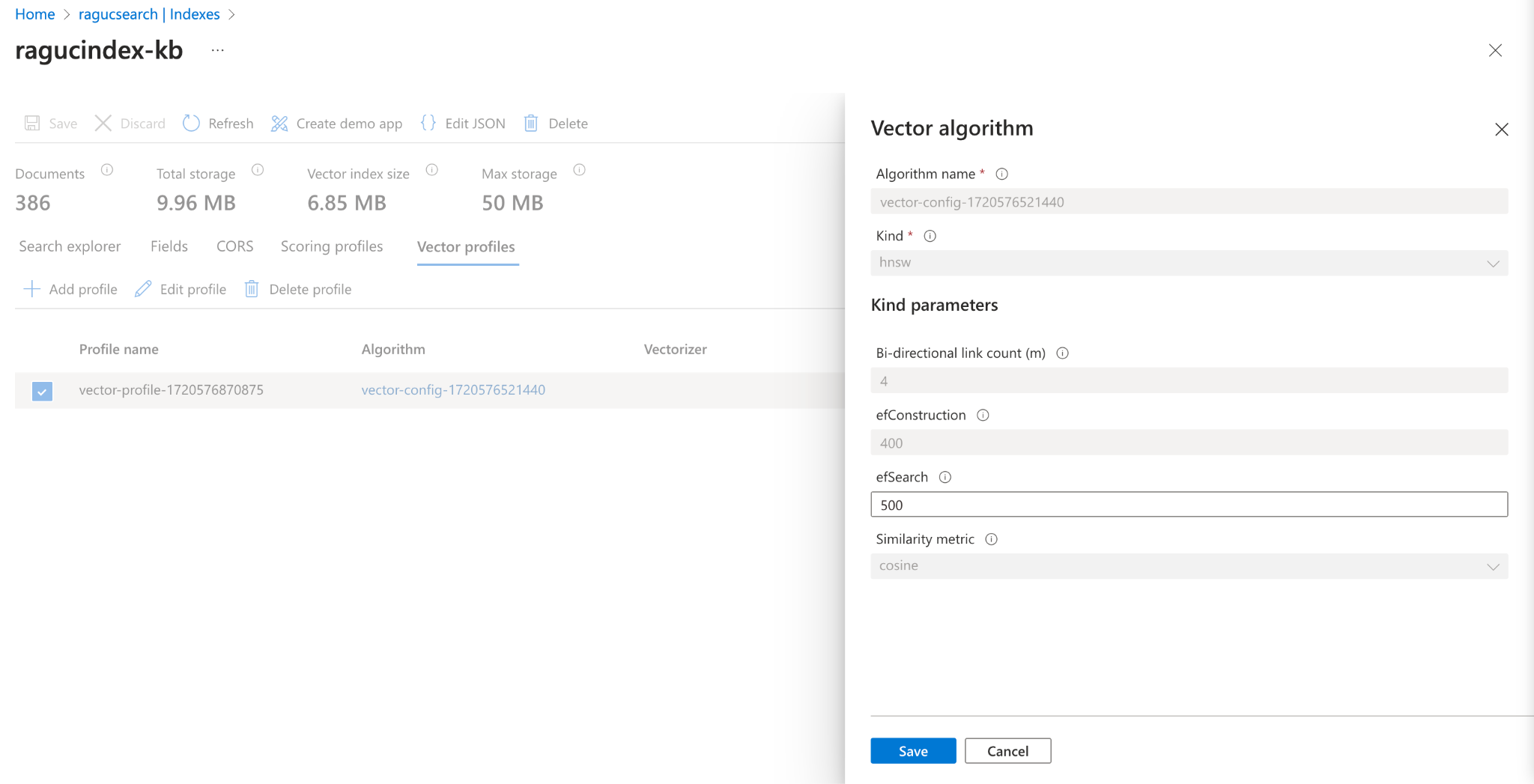


1. **Azure AI Search :** includes built-in vector search capabilities, which allow for similarity searches using vector representations of data.
2. **Knowledge Base Index:** Stores and retrieves knowledge documents that provide answers to user queries.
3. **Memory Index:** Stores and retrieves past interactions (queries and responses) to provide contextual answers.



In both indexes, vector profile configurations are :

* Vector profile kind: hnsw ((Hierarchical Navigable Small World graph)
* Bi-directional link count (m): 4
* efConstruction: 400 (controls the accuracy and construction time of the index)
* efSearch: 500 (controls the accuracy and speed of the search query)
* Similarity metric: cosine (used to measure the similarity between vectors)

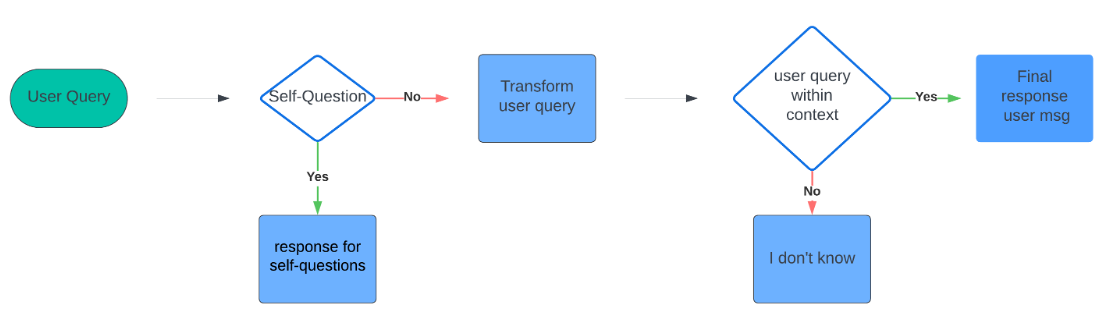
****

**RETRIEVAL**

Retrieval involves querying indexed data to find relevant information and generating responses based on that information.

* **Query Handling:** User queries are vectorized using the same embedding model.
* Relevant documents are retrieved using Azure Search.
* Prompt Engineering to extract the required response based on context, conversation history and language model.
* OpenAI GPT-4 model generates responses based on retrieved documents.

**Prompt Engineering:**Structure our prompts for GPT-4o in the following way :

****

**classify\_self\_questions\_system\_prompt:** Instructs the model to classify user inputs as either self-questions or not.

#selfquestion : a greeting , introductory remark

**response\_for\_self\_questions\_system\_prompt:** Provides instructions to LLM for responding to self-questions with appropriate greetings.

**transform\_query\_system\_message:** Guides the model to rephrase follow-up queries as standalone questions using previous conversation in the session

**final\_response\_system\_prompt:** Instructs the model to answer only using provided context or indicate if the information is not available

**final\_response\_user\_message:** Template for formatting the user's message with context. Uses source knowledge , previous history and LLM language capability to structure answer

* Classifies the query to check if it's a general or specific question using classify\_self\_questions\_llm.
* Retrieves relevant documents from the index using get\_relevant\_documents.
* Generates a response based on the retrieved documents and the query using final\_response\_from\_llm.
* If the query is classified as a general question, it responds with a predefined answer using self\_response\_from\_llm.

## Code Files

### get\_knowledge\_from\_index.py

* **Function**: Retrieves relevant documents from the knowledge base index based on a query.

### put\_memory\_into\_index.py

* **Function**: Uploads interaction history to the memory index for contextual retrieval.

### get\_relevant\_memory.py

* **Function**: Retrieves relevant past interactions from the memory index.

### response\_for\_self\_questions.py

* **Function**: Handles responses for self-related questions using OpenAI.

### transform\_query\_llm\_tool.py

* **Function**: Reformulates follow-up queries based on previous conversation history.

### chunk.py

* **Function**: Chunks text into manageable sizes for embedding.

### index\_documents.py

* **Function**: Indexes documents into Azure Cognitive Search with vector embeddings.

### text\_embed.py

* **Function**: Generates text embeddings for semantic search.

### config.json

* **Purpose**: Contains configuration settings for API keys, endpoints, and index names.

### DataProcessing\_blueprint.cpython-311.pyc

* **Function**: Azure Function blueprint for document processing.

### filesUpload.py

* **Function**: Script for uploading files to the Azure Function.

### function\_app.py

* **Function**: Registers Azure Functions for document processing and user queries.

### document\_management.py

* **Function**: Streamlit application for uploading documents and interacting with the chatbot.

**Conclusion**

The RAG chatbot represents a cutting-edge integration of AI and search technologies, offering highly relevant and contextually rich responses. By combining Azure Cognitive Search with OpenAI's language models and embedding techniques, the chatbot delivers high-quality answers based on both static knowledge and dynamic interaction history.

With upcoming enhancements, including custom AI avatars and lip-synced, voice-enabled interactions, the chatbot will provide an even more personalized and immersive user experience, bridging the gap between static information retrieval and dynamic, human-like communication .