SIG788

Engineering AI solutions

High Distinction Task 8

Arunkumar Balaraman
S223919051

Contents

..... 1
Target = High Distinction 3
Shortage Product AI Bot 3
Introduction 3
Objective: 3
Technological Implementation: 3
Data Utilization: 3
Approach: 4
OpenAI Api: 4
Azure Blob Storage & its containers: 6
Azure AI Search: 7
Azure Speech Services: 7
Azure Computer Vision: 8
Create Azure Speech to Text: 8
Visual Studio Code: 9
Pre-requisites: 9
Configure .env file: 11
Python SDK: 12
Dataset selection for Bot: 12
Data Load: 12
Azure Search Vector 15
Creating the bot 18
Conclusion: 30
Cleaning Up Activity: 30
References: 31

Target = High Distinction

Shortage Product AI Bot

Introduction

This project aims to develop an intelligent bot, named **Shortage Product AI Bot**, to automate and **personalize product recommendations** for customers using **advanced AI technologies** hosted using Azure & Flask. This bot is designed to address challenges faced by pharmacies due to the frequent occurrence of drug shortages and price concessions as described by the Pharmaceutical Services Negotiating Committee (PSNC).

Objective:

The objective of this project is to create a **multi-modal**, intelligent recommendation system that can interact with customers using **text, voice & image**. This system will utilize Azure's cognitive services and a custom AI developed with OpenAI technology to deliver personalized drug recommendations based on current stock levels, price concessions, and customer preferences.

Technological Implementation:

- **OpenAI API Access**: Utilizes OpenAI's models for natural language processing to interpret and respond to customer queries along with embeddings.
- **Azure Resource Group**: Manages all project resources in a consolidated manner to ensure efficient access and cost management.
- **Azure Blob Storage**: To tore project data files in a scalable, secure environment.
- **Azure Container in storage**: To load the project file.
- **Azure AI Search**: The embedded data using OpenAI embedding systems has been stored in Azure AI Search which provides advanced search capabilities across the stored data to quickly retrieve relevant product information based on customer queries.
- **Azure Speech Services**: to convert speech to text, allowing seamless voice interactions with the bot.
- **Azure Computer Vision**: Analyzes images to enhance the bot's understanding using OCT and response to visual input.
- **Flask**: Serves as application's backend, handling requests and serving the user interface.
- **Visual Studio Code**: Used for writing and testing the bot's code.

Data Utilization:

Data from the PSNC website, specifically the generic shortages list from January, February, and March 2024, were used to train the bot. This data helps the bot understand current market dynamics and product availability to make accurate recommendations.

Link: https://cpe.org.uk/funding-and-reimbursement/reimbursement/price-concessions/

Approach:

The development of the **Shortage Product AI Bot** is personalized to address the specific challenges posed by drug shortages and price concessions. The process is structured to well gather data, develop a responsive bot, and integrate it with essential services for seamless operation.

Data Collection

Source: Data is sourced from the PSNC (CPE) for monthly price concession files and drug shortage lists from **January to March 2024**. This dataset provides comprehensive insights into the availability and pricing of drugs, essential for the bot's functionality.

Purpose: The collected data enables the bot to understand current market trends, drug availability, and the impact of price concessions on pharmacy operations. This understanding allows the bot to make informed recommendations to pharmacies.

Bot Development

Shortage Product AI Bot uses the OpenAI API for natural language processing to understand and respond to customer queries. All resources are managed through an Azure Resource Group, and data files are stored in Azure Blob Storage. The project file is loaded using an Azure Container in storage. The data, processed using OpenAI embedding systems is stored in Azure AI Search for quick retrieval of product information.

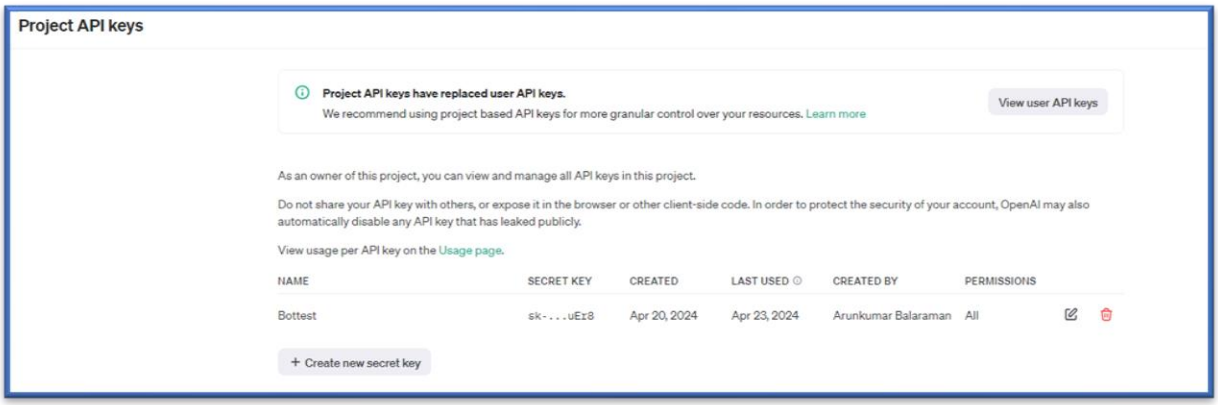
Azure Speech Services convert speech to text for seamless voice interactions and Azure Computer Vision analyses images for enhanced understanding. Flask handles requests and serves the user interface and all code is written and tested in Visual Studio Code.

Now let’s create each service in Azure & Open AI Api.

OpenAI Api:

The OpenAI API is a bridge to OpenAI's models. It is used to integrate advanced AI capabilities in this project without needing to understand the complexities of the model architecture. This API is used for natural language processing to interpret and respond to various queries along with the embeddings.

Created OpenAI API key



Embeddings

OpenAI's text embeddings measure the relatedness of text strings. Embeddings are commonly used for:

- Search (where results are ranked by relevance to a query string)
- Clustering (where text strings are grouped by similarity)
- Recommendations (where items with related text strings are recommended)
- Anomaly detection (where outliers with little relatedness are identified)
- Diversity measurement (where similarity distributions are analyzed)
- Classification (where text strings are classified by their most similar label)

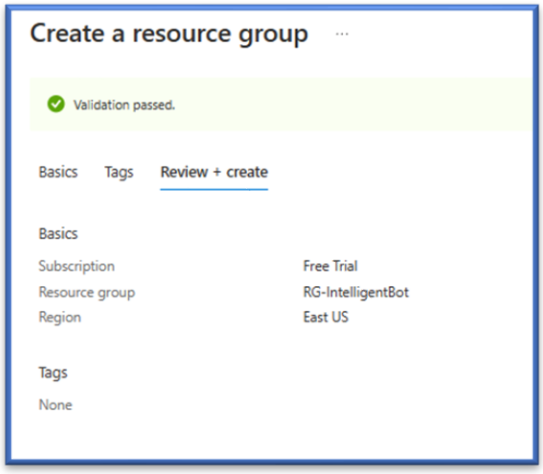
Used the model **text-embedding-3-small**

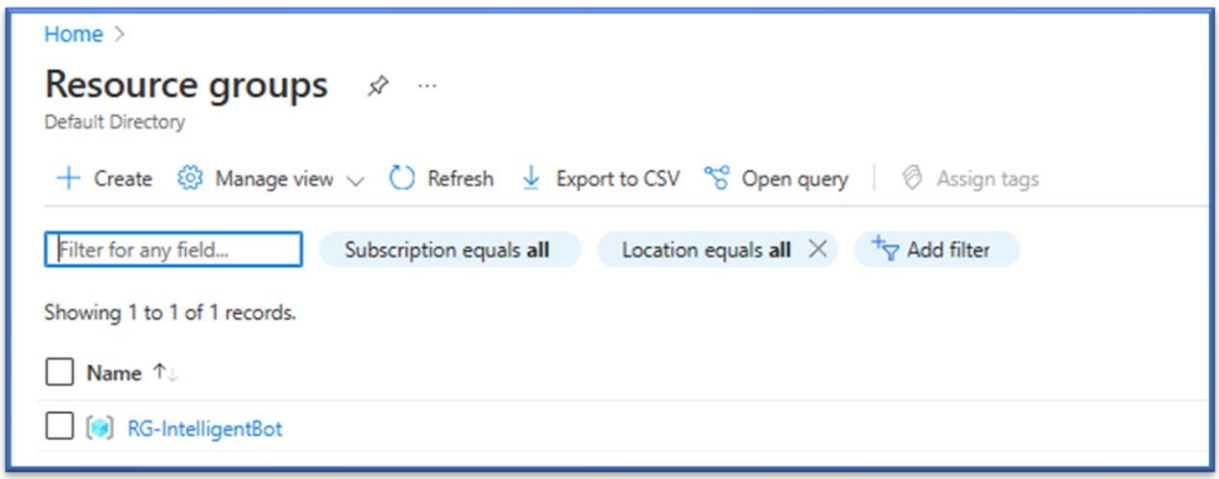
Azure Resource Group:

Manages all project resources in a consolidated manner to ensure efficient access and cost management.

Signed in to Azure Portal: Azure Portal and sign in with Microsoft account credentials & created Azure Resource group.

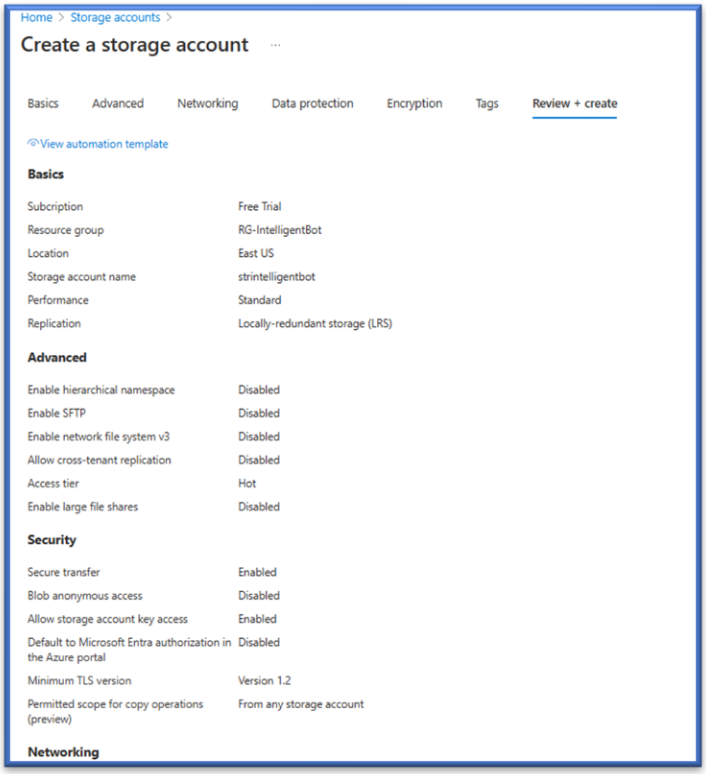
Link: https://portal.azure.com/

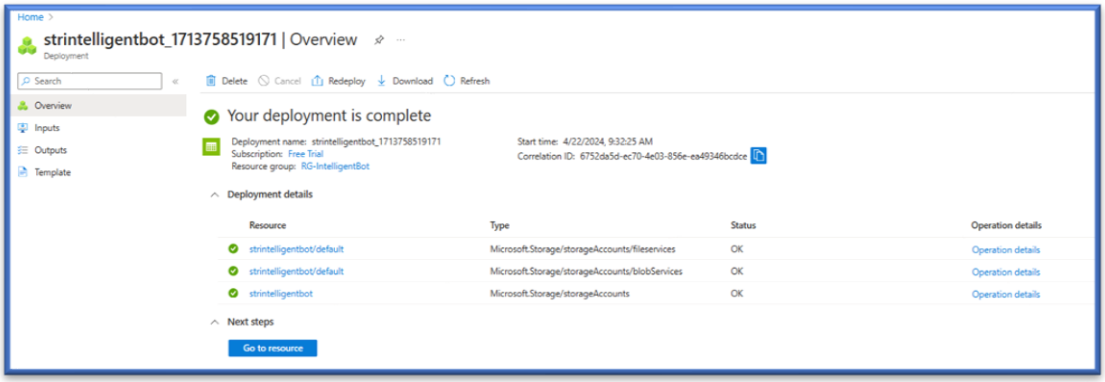


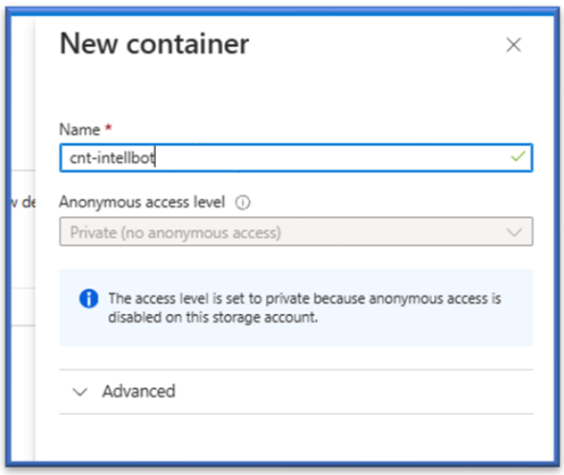


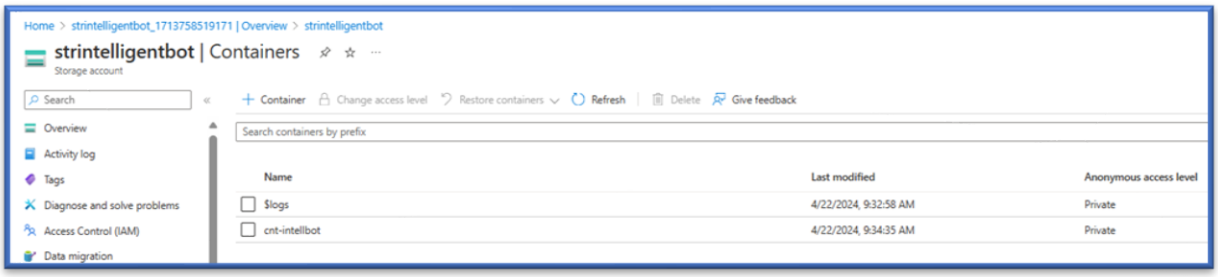
Azure Blob Storage & its containers:

To store project data files in a scalable, secure environment within its container



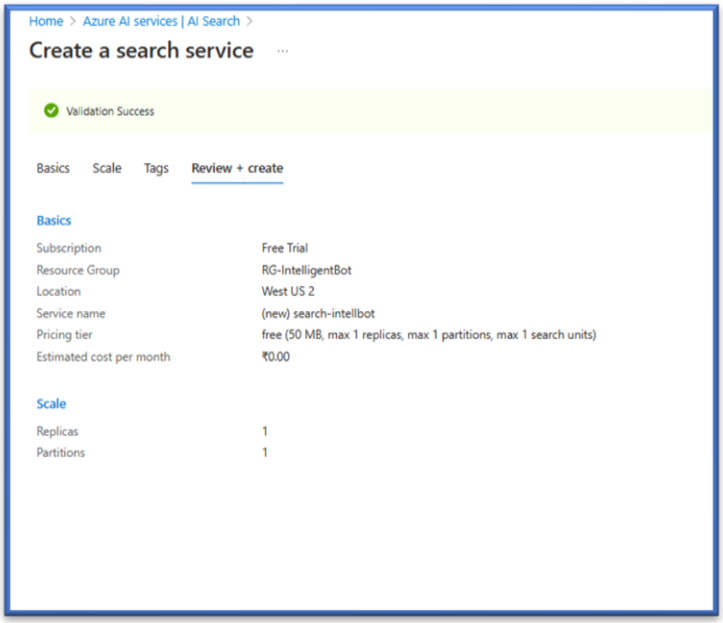


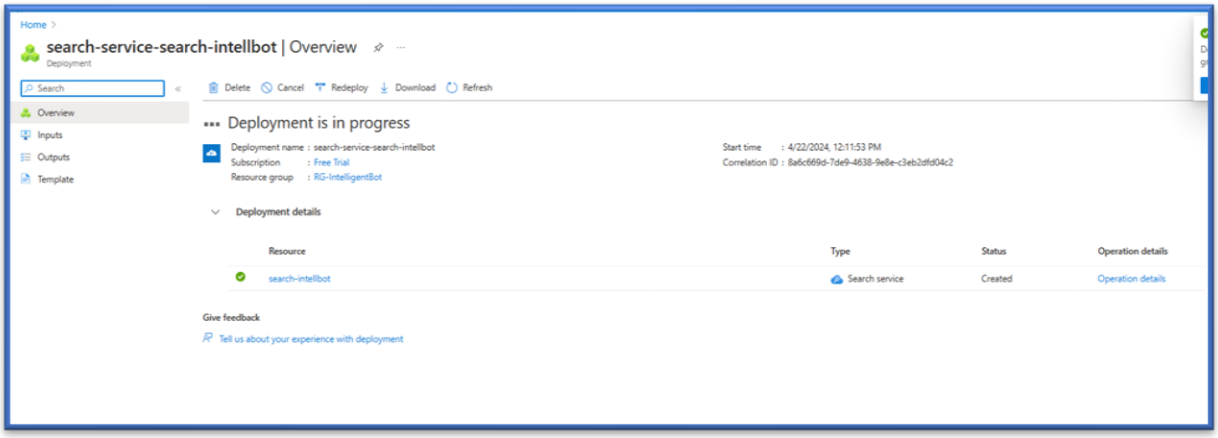




Azure AI Search:

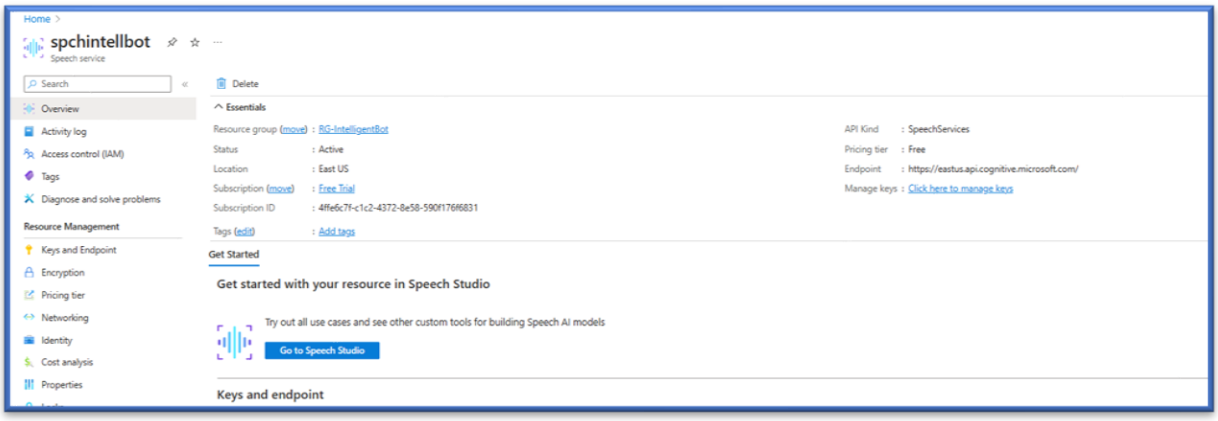
The **embedded data** using OpenAI embedding systems has been stored in Azure AI Search which provides advanced search capabilities across the stored data to quickly retrieve relevant product information based on customer queries.





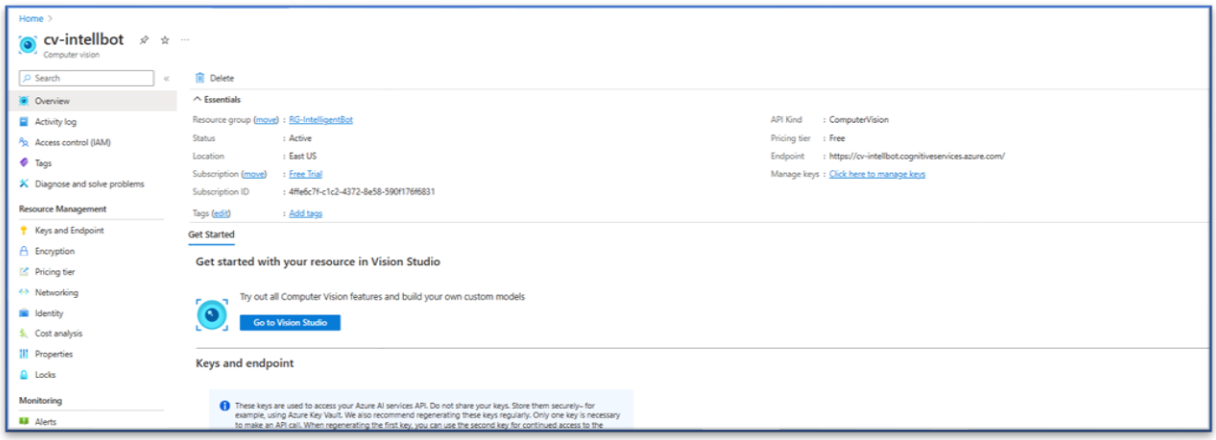
Azure Speech Services:

To convert speech to text, allowing seamless voice interactions with the bot.

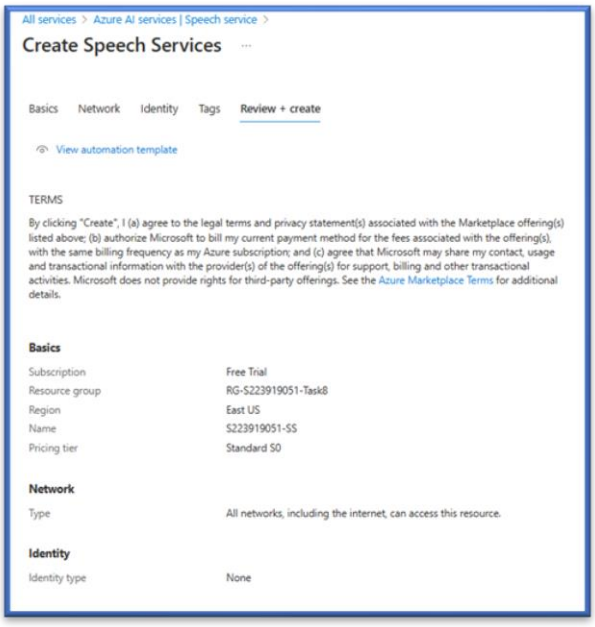


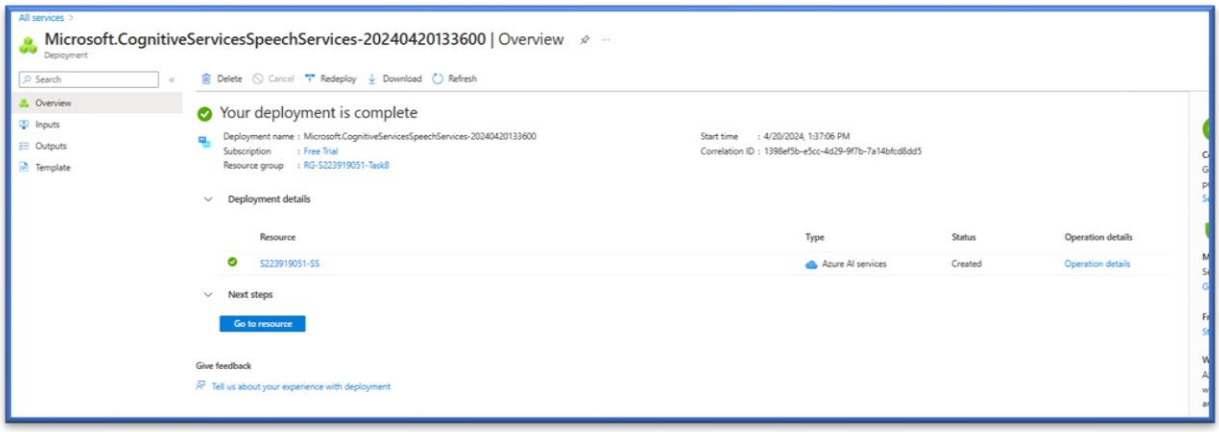
Azure Computer Vision:

Analyses images to enhance the bot's understanding using OCR (Optical character recognition) and response to visual input.



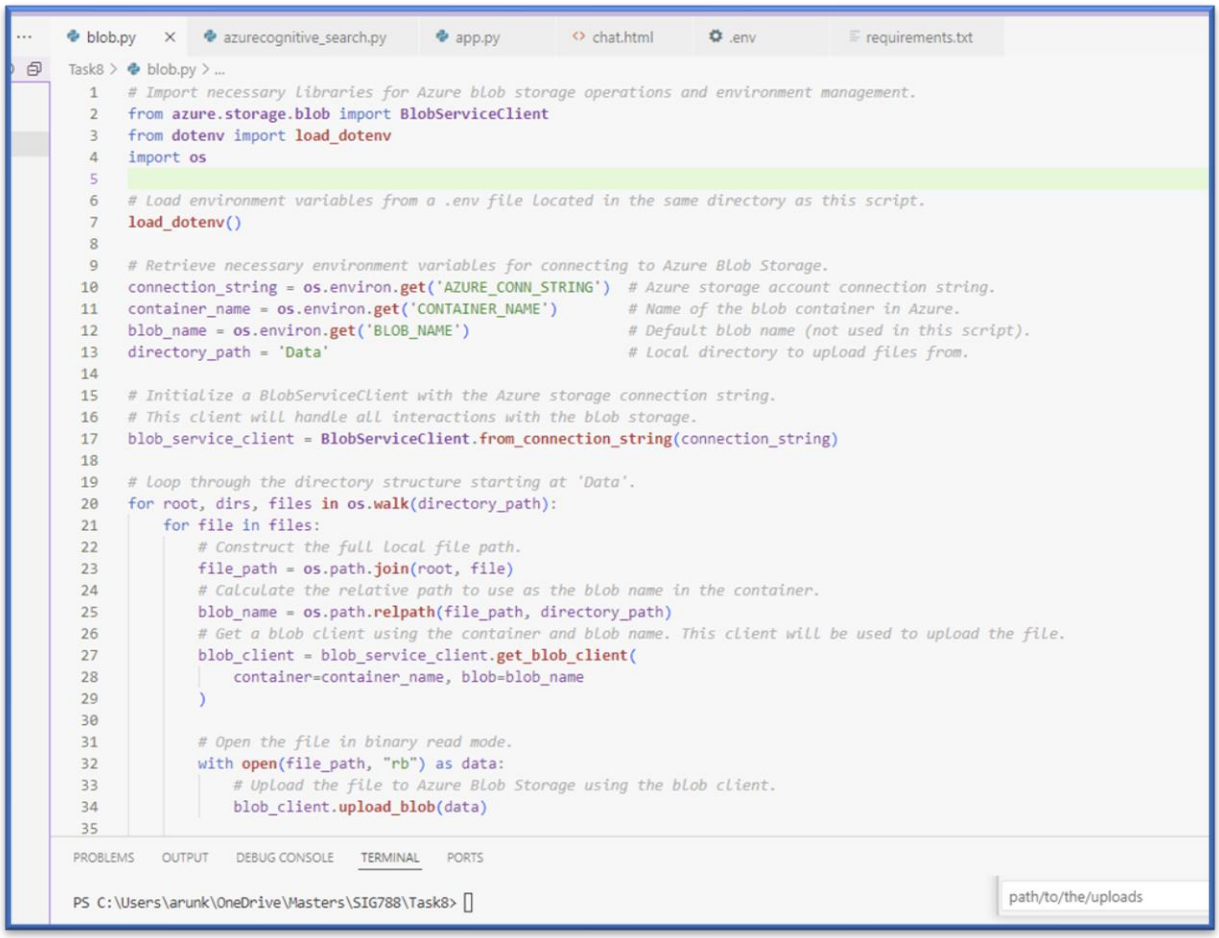
Create Azure Speech to Text:





Visual Studio Code:

Used for writing and testing the bot's code.



Pre-requisites:

Install pre-requisites for **Shortage Product AI Bot** in Visual Studio Code terminal.

- pip install azure-search-documents
- pip install azure-storage-blob
- pip install langchain
- pip install python-dotenv
- pip install openai

- pip install tiktoken
- pip install unstructured
- pip install -U langchain-openai
- pip install -U langchain-community
- pip install "unstructured[csv]"
- pip install azure-cognitiveservices-speech
- pip install scipy
- pip install flask-cors
- pip install azure-cognitiveservices-vision-computervision

pip install azure-search-documents

This package installs the Azure SDK for Python that allows interaction with Azure AI Search, a cloud search service with built-in AI capabilities. It provides tools for importing, indexing, and querying data to quickly find relevant results based on queries.

pip install azure-storage-blob

This command installs the Azure Blob Storage client library for Python. It provides methods for managing blob storage on Azure, including uploading, downloading, and listing blob items, which are essential for handling large amounts of unstructured data.

pip install langchain

Langchain is a Python library designed for building language applications using chain-of-thought prompting strategies. This package facilitates the integration of different language models and tools, simplifying the creation of sophisticated language processing pipelines.

pip install python-dotenv

This package is used to read key-value pairs from a .env file and set them as environment variables.

pip install openai

This command installs the official OpenAI Python client library, which allowed to access and utilize the API provided by OpenAI, including capabilities like GPT-3, embedding, and other AI models for natural language processing and understanding.

Pip install tiktoken

TikToken is a Python library used for efficiently parsing and handling tokens, particularly useful in NLP applications were managing a large number of text tokens.

pip install unstructured

This package provides utilities to handle unstructured data in Python. It simplifies operations such as data extraction, transformation, and storage, which are critical in projects dealing with non-standard data formats.

pip install -U langchain-openai

This installs specific components of the Langchain library tailored for integrating OpenAI's models. It enhances Langchain's capabilities to interface directly with OpenAI's services, making it easier to implement advanced NLP features.

pip install -U langchain-community

This installs the Langchain Community package, which includes additional tools and functionalities developed by the Langchain open-source community. These tools often extend Langchain's core capabilities with new features and improvements.

pip install "unstructured[csv]"

This installs the Unstructured library with additional support for handling CSV files, providing tools for managing and transforming unstructured data contained in CSV formats.

pip install azure-cognitiveservices-speech

This package installs the Azure Speech SDK for Python, enabling developers to integrate speech processing capabilities such as speech-to-text.

pip install flask-cors

This package is a Flask extension for handling Cross-Origin Resource Sharing (CORS), making it possible to configure how Flask app handles cross-domain requests, essential for web applications exposed to the web.

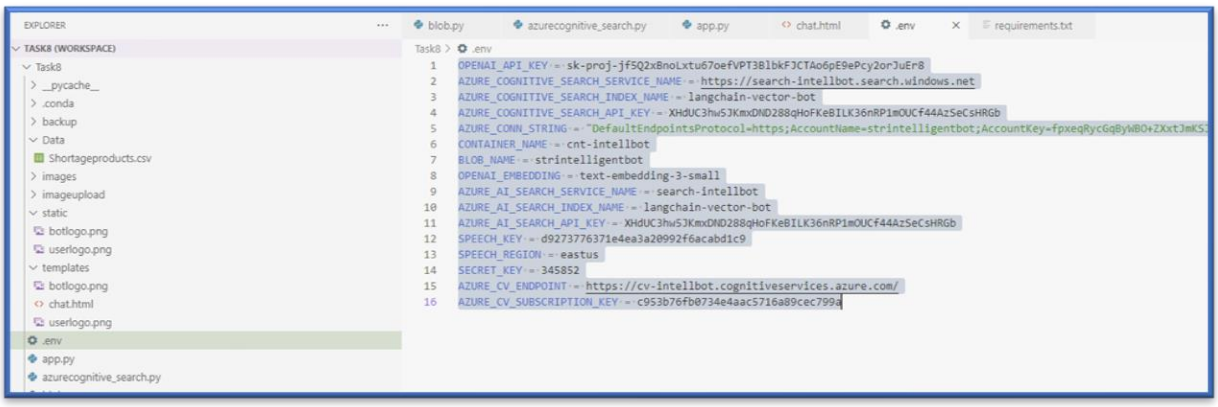
pip install azure-cognitiveservices-vision-computervision

This installs the Azure Computer Vision client library for Python. It provides tools for processing and analysing visual data in this case we used OCR.

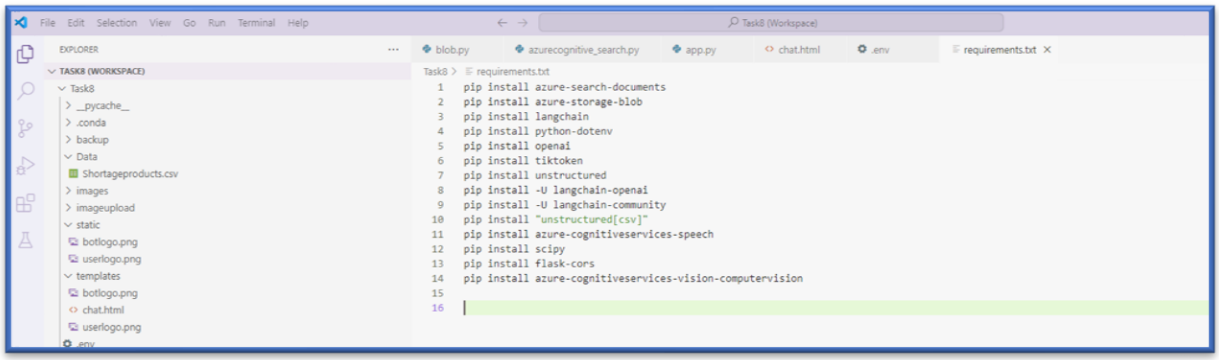
Configure .env file:

For **Shortage Product AI Bot** we used .env in visual studio to secure the credentials, endpoints and keys.

Note: I have deleted all Keys after the project is complete & nothing would work



Also included requirements.txt to reference and replicate the project.

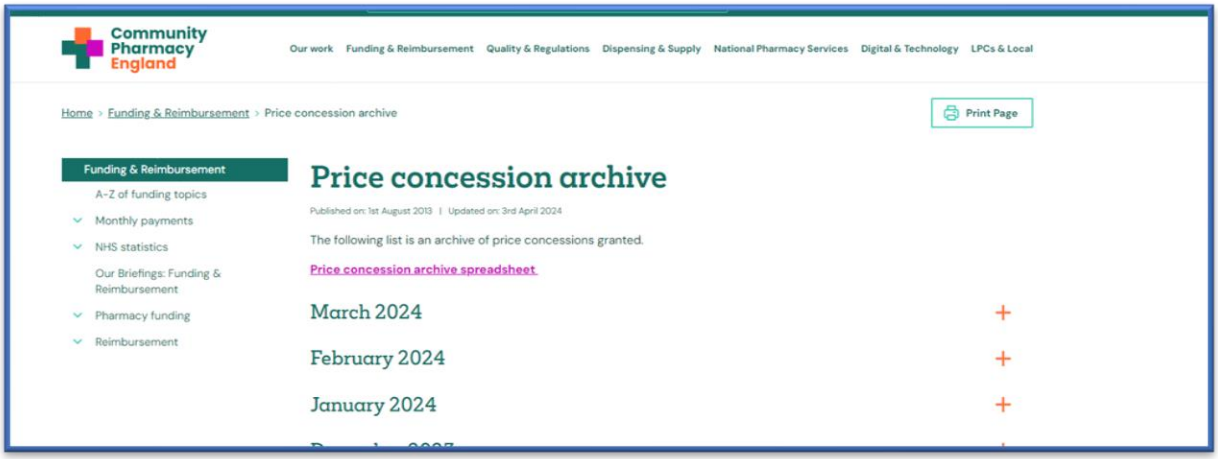


Python SDK:

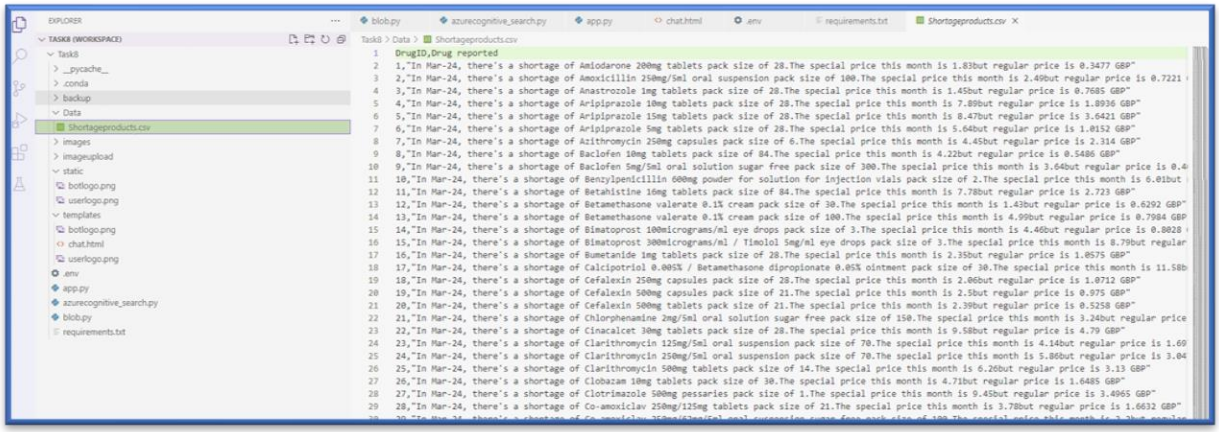
Dataset selection for Bot:

I've curated the **Shortage Products** dataset from the **PSNC website** and used it for our project to efficiently help the pharmacies for the **Shortage Product AI Bot**. This bot will provide timely and accurate information about drug shortage for UK healthcare products, which is vital for efficiently to handle pharmacy operations.

Dataset:



The downloaded file is placed in the folder **Data**



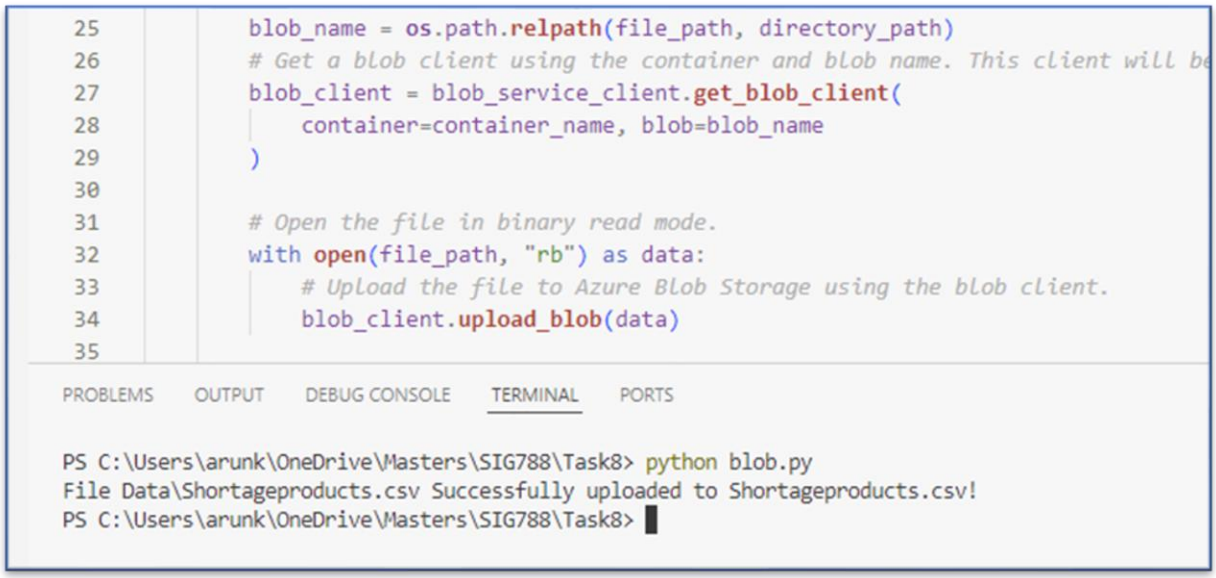
Data Load:

For the creation of the **Shortage Product AI Bot** services using **above mentioned services**, I've performed the following steps:

- Loaded the **Shortage Product dataset** directly from the PSNC website & using Python. The same dataset has been loaded into the Azure BLOB storage using python SDK. Here's the Python code used:

```
# Import necessary libraries for Azure blob storage operations and environment management.
from azure.storage.blob import BlobServiceClient
from dotenv import load_dotenv
import os

# Load environment variables from a .env file located in the same directory as this script.
load_dotenv()

# Retrieve necessary environment variables for connecting to Azure Blob Storage.
connection_string = os.environ.get('AZURE_CONN_STRING')  # Azure storage account connection string.
container_name = os.environ.get('CONTAINER_NAME')       # Name of the blob container in Azure.
blob_name = os.environ.get('BLOB_NAME')              # Default blob name (not used in this script).
directory_path = 'Data'                          # Local directory to upload files from.

# Initialize a BlobServiceClient with the Azure storage connection string.
# This client will handle all interactions with the blob storage.
blob_service_client = BlobServiceClient.from_connection_string(connection_string)

# loop through the directory structure starting at 'Data'.
for root, dirs, files in os.walk(directory_path):
    for file in files:
        # Construct the full local file path.
        file_path = os.path.join(root, file)
        # Calculate the relative path to use as the blob name in the container.
        blob_name = os.path.relpath(file_path, directory_path)
        # Get a blob client using the container and blob name. This client will be used to upload the file.
        blob_client = blob_service_client.get_blob_client(
            container=container_name, blob=blob_name
        )

        # Open the file in binary read mode.
        with open(file_path, "rb") as data:
            # Upload the file to Azure Blob Storage using the blob client.
            blob_client.upload_blob(data)

        # Print a success message indicating the file and its blob path.
        print(f"File {file_path} Successfully uploaded to {blob_name}!")
```

Output:



Process:

The above code is used to upload all the files from a local '**Data'** directory to an **Azure Blob Storage container**. The data file is also placed in the Azure Blob container storage. The blob name in the container corresponds to the relative path of the file in the '**Data'** directory. The same has been organized the files in the Azure Blob Storage.

Steps:

Import Libraries: Omporting necessary libraries. BlobServiceClient is used for Azure Blob Storage operations, os is used for interacting with the operating system, and dotenv is used to manage environment variables.

Load Environment Variables: The load_dotenv() function loads environment variables from a file named .env in the working directory.

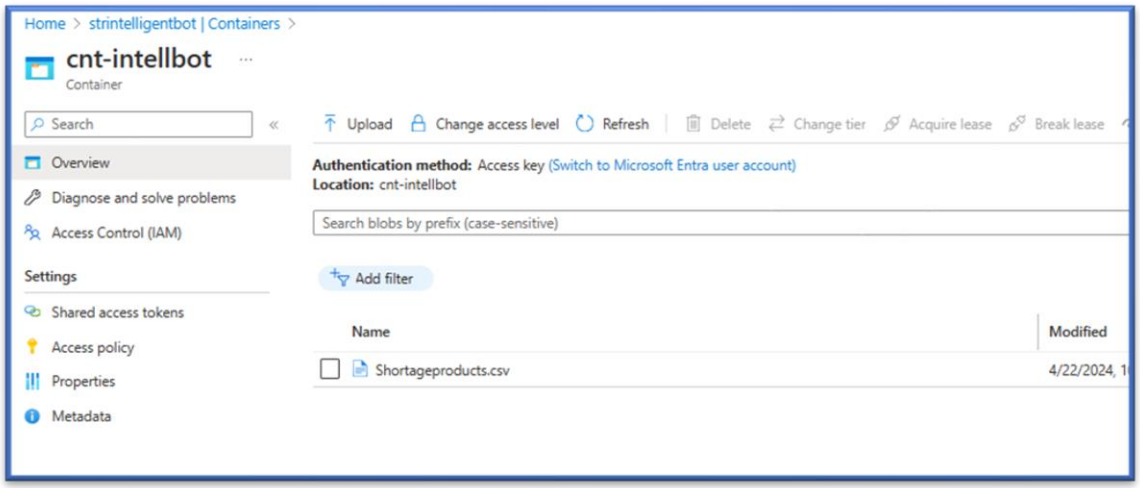
Retrieve Connection Details: Code retrieves the **Azure storage account connection string**, the name of the **blob container** in Azure, and the default blob name from the environment variables.

Initialize BlobServiceClient: The **BlobServiceClient** is initialized with the Azure storage connection string. This client will handle all interactions with the blob storage.

Upload Files: Code then loops through all the files in the **'Data'** directory. For each file, constructing the full local file path and creates relative path to use as the blob name in the container. It gets a blob client using the container and blob name. This client is used to upload the file to Azure Blob Storage.

Success Message: After file is uploaded, a success message is printed indicating the file and its blob path.

I have provided the screenshot on the file has been loaded into the Azure Blob.



As now the data has been loaded into Azure Blob, now let's proceed to create Azure Search Vector storage using Open AI API embeddings.

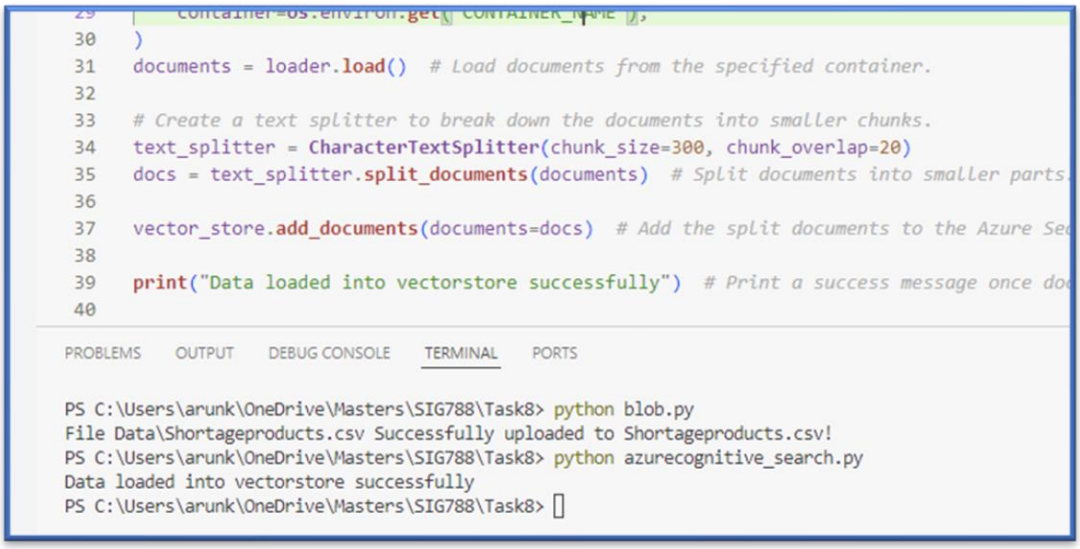
Azure Search Vector

Code:

```
import os  # Import the OS library for interacting with the operating system.
from langchain_openai import OpenAIEmbeddings  # Import OpenAI Embeddings from the langchain library.
from langchain.vectorstores.azuresearch import AzureSearch  # Import Azure Search from the langchain library.
from langchain_community.document_loaders import AzureBlobStorageContainerLoader  # Import document loader for Azure Blob Storage.
from langchain.text_splitter import CharacterTextSplitter  # Import text splitter for breaking text into smaller pieces.
from dotenv import load_dotenv  # Import dotenv to load environment variables from a .env file.

load_dotenv()  # Load environment variables from the .env file.

# Retrieve the model name for OpenAI embeddings and Azure search service details from environment variables.
model: str = os.environ.get('OPENAI_EMBEDDING')
vector_store_address: str = os.environ.get('AZURE_COGNITIVE_SEARCH_SERVICE_NAME')

# Create an instance of OpenAI Embeddings with specified model and chunk size.
embeddings: OpenAIEmbeddings = OpenAIEmbeddings(deployment=model, chunk_size=1)
index_name: str = os.environ.get('AZURE_COGNITIVE_SEARCH_INDEX_NAME')

# Setup Azure Search with the endpoint, API key, and the index name.
vector_store: AzureSearch = AzureSearch(
    azure_search_endpoint=vector_store_address,
```

```
    azure_search_key=os.environ.get("AZURE_COGNITIVE_SEARCH_API_KEY"),
    index_name=index_name,
    embedding_function=embeddings.embed_query,
)

# Setup a loader to load documents from an Azure Blob Storage container.
loader = AzureBlobStorageContainerLoader(
    conn_str=os.environ.get("AZURE_CONN_STRING"),
    container=os.environ.get("CONTAINER_NAME"),
)
documents = loader.load()  # Load documents from the specified container.

# Create a text splitter to break down the documents into smaller chunks.
text_splitter = CharacterTextSplitter(chunk_size=300, chunk_overlap=20)
docs = text_splitter.split_documents(documents)  # Split documents into smaller parts.

vector_store.add_documents(documents=docs)  # Add the split documents to the Azure Search vector store.

print("Data loaded into vectorstore successfully")  # Print a success message once documents are loaded.
```

Output



The above code is used to load and split documents from an Azure Blob Storage container, and then add these split documents to an Azure Search vector store for further processing or querying. The **OpenAI embeddings** are used to embed the queries for the **Azure Search**.

Steps:

import Libraries: The necessary libraries are imported including libraries for interacting with the **operating system**, loading **environment variables**, handling **OpenAI embeddings**, **Azure Search**, document loading from Azure Blob Storage, and text splitting.

Load Environment Variables: The load_dotenv() function is used to load environment variables from a .env file.

Retrieve Connection Details: The code retrieves the model name for **OpenAI embeddings** and **Azure search service details** from the environment variables.

Initialize OpenAI Embeddings: An instance of **OpenAI Embeddings** is created with the specified model and chunk size.

Setup Azure Search: Azure Search is set up with the **endpoint**, **API key**, and the **index name**. The **embedding function** from the **OpenAI embeddings** instance is also passed to it.

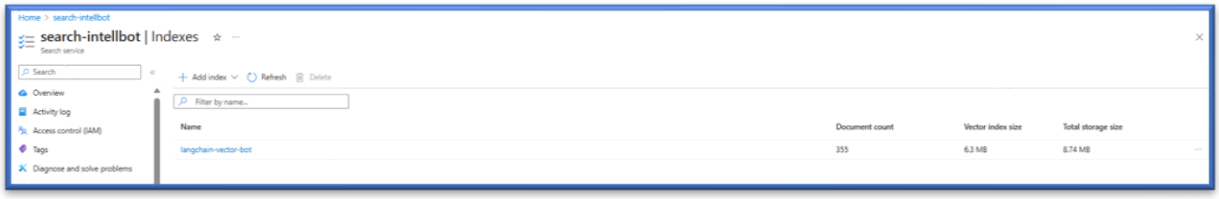
Load Documents from Azure Blob Storage: Loader has been set up to load documents from an Azure Blob Storage container using the connection string and container name retrieved from the environment variables. The documents are then loaded from the specified container.

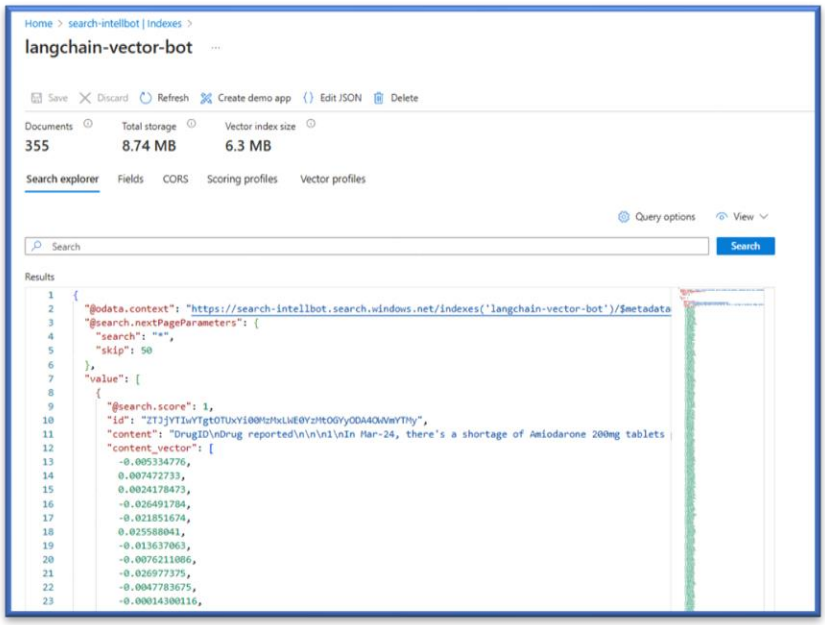
Split Documents: A text splitter is created to break down the documents into smaller chunks. The documents are then split into smaller parts.

Add Documents to Azure Search Vector Store: The split documents are added to the **Azure Search vector store**.

Success Message: A success message is printed after the document are successfully loaded into the vector store.

Azure Search Vector Storage:





I have succesSfully integrated **OpenAI's Embeddings model**, specifically the **text-embedding-3-small** variant, into our **Azure AI Search service** to enhance **semantic search capabilities**. search index, **'langchain-vector-bot'**, which consists of 355 documents that represent the searchable corpus.

By leveraging the **OpenAI model**, each document has been transformed into a **high-dimensional vector**, effectively capturing the **semantic meaning of the text**. This conversion enables the **search functionality** to go beyond keyword matching, allowing for a deeper understanding of the context and content within each document. The vectorized representations has 6.3 MB of the index's total 8.74 MB storage on Azure's servers.

Creating the bot

Code:

```
import os
import azure.cognitiveservices.speech as speechsdk
from flask import Flask, request, jsonify, render_template, session, redirect, url_for
from werkzeug.utils import secure_filename
from dotenv import load_dotenv
from langchain.retrievers import AzureCognitiveSearchRetriever
from langchain.chains import ConversationalRetrievalChain
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.prompts import PromptTemplate
from flask_cors import CORS
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from msrest.authentication import CognitiveServicesCredentials
from azure.cognitiveservices.vision.computervision.models import OperationStatusCodes
import time

# Load environment variables from .env file
load_dotenv()

# Initialize the Flask application
app = Flask(__name__)
app.secret_key = os.getenv('SECRET_KEY', '123')  # Use a secret key from environment
variables or default to '123'

# Set up Azure Computer Vision client using credentials from environment variables
endpoint = os.getenv('AZURE_CV_ENDPOINT')
subscription_key = os.getenv('AZURE_CV_SUBSCRIPTION_KEY')
computervision_client = ComputerVisionClient(endpoint,
CognitiveServicesCredentials(subscription_key))

# Allow cross-origin requests on all routes
CORS(app)

def recognize_from_microphone():
```

```
    # Set up the speech recognition with Azure Cognitive Services
    speech_config = speechsdk.SpeechConfig(subscription=os.getenv('SPEECH_KEY'),
region=os.getenv('SPEECH_REGION'))
    speech_config.speech_recognition_language = "en-US"
    audio_config = speechsdk.audio.AudioConfig(use_default_microphone=True)
    speech_recognizer = speechsdk.SpeechRecognizer(speech_config=speech_config,
audio_config=audio_config)

    # Perform speech recognition
    result = speech_recognizer.recognize_once_async().get()

    # Handle the results of speech recognition
    if result.reason == speechsdk.ResultReason.RecognizedSpeech:
        return result.text
    elif result.reason == speechsdk.ResultReason.NoMatch:
        return "No speech could be recognized."
    elif result.reason == speechsdk.ResultReason.Canceled:
        return f"Speech recognition canceled: {result.cancellation_details.reason}"

@app.route('/recognize_speech', methods=['POST'])
def recognize_speech():
    # Endpoint to recognize speech from a microphone input
    text = recognize_from_microphone()
    return jsonify({'text': text})

# Set up memory for conversation history
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True, output_key="answer")

def load_chain():
    # Configure the conversational chain with Langchain and Azure Cognitive Search
    prompt_template = """You are a helpful assistant for questions about UK Shortage
Products.
    {context}
    Question: {question}
    Answer here:"""
    PROMPT = PromptTemplate(template=prompt_template, input_variables=["context",
"question"])
    retriever = AzureCognitiveSearchRetriever(content_key="content", top_k=10)

    return ConversationalRetrievalChain.from_llm(
        llm=ChatOpenAI(),
        memory=memory,
        retriever=retriever,
        combine_docs_chain_kwargs={"prompt": PROMPT},
    )

chain = load_chain()
```

```python
@app.route('/', methods=['GET', 'POST'])
def home():
    # Home page that handles both displaying and posting chat messages
    if 'history' not in session:
        session['history'] = []

    if request.method == 'POST':
        user_input = request.form.get('user_input', '').strip()
        if user_input:
            response = chain.run(question=user_input)
            session['history'].extend([
                {'text': user_input, 'user': True},
                {'text': response, 'user': False}
            ])
            session.modified = True
        else:
            print("No input to process.")

    return render_template('chat.html', history=session.get('history', []))

# Set allowed extensions for image uploads
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}
UPLOAD_FOLDER = 'imageupload'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

def allowed_file(filename):
    # Check if the uploaded file is an allowed type
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/upload_image', methods=['POST'])
def upload_image():
    # Handle image uploads and process them for OCR
    if 'image' not in request.files:
        return redirect(request.url)
    file = request.files['image']
    if file.filename == '':
        return redirect(request.url)
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(filepath)
        question = get_text_from_image(filepath)

        if question:
            response = chain.run(question=question)
            session['history'].extend([
```

```
            {'text': question, 'user': True},
            {'text': response, 'user': False}
        ])
        session.modified = True
        return redirect(url_for('home'))
    else:
        return "No text recognized or text was empty."
  return 'File type not supported'

def get_text_from_image(image_path):
  # Use Azure Computer Vision to extract text from an uploaded image
  with open(image_path, "rb") as image_stream:
    read_response = computervision_client.read_in_stream(image_stream, raw=True)

  read_operation_location = read_response.headers["Operation-Location"]
  operation_id = read_operation_location.split("/")[-1]

  while True:
    read_result = computervision_client.get_read_result(operation_id)
    if read_result.status not in ['notStarted', 'running']:
      break
    time.sleep(1)

  if read_result.status == OperationStatusCodes.succeeded:
    text = []
    for text_result in read_result.analyze_result.read_results:
      for line in text_result.lines:
        text.append(line.text)
    return " ".join(text)
  return "No text recognized"

if __name__ == '__main__':
  app.run(debug=True)
```

Chat.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Shortage Product AI Chat</title>
<style>
  body {
    font-family: 'Arial', sans-serif;
    background-color: #f7f7f7;
    display: flex;
    justify-content: center;
    align-items: center;
```

```
    height: 100vh;
    margin: 0;
}
.chat-container {
    width: 400px;
    height: 600px;
    border-radius: 8px;
    background-color: #fff;
    box-shadow: 0 4px 8px rgba(0,0,0,0.2);
    display: flex;
    flex-direction: column;
    overflow: hidden;
}
.chat-header {
    background-color: #4CAF50;
    color: #fff;
    padding: 16px;
    font-size: 24px;
    text-align: center;
}
.chat-body {
    padding: 10px;
    overflow-y: auto;
    flex-grow: 1;
    background: #e5ddd5;
    display: flex;
    flex-direction: column;
}
.message {
    display: flex;
    align-items: center;
    padding: 10px;
    border-radius: 18px;
    color: white;
    max-width: 75%;
    word-wrap: break-word;
}
.bot-message {
    background-color: #007bff;
    align-self: flex-start;
    text-align: left;
}
.user-message {
    background-color: #34b7f1;
    align-self: flex-end;
    text-align: right;
}
.logo {
```

```
        display: flex;
        justify-content: center;
        align-items: center;
        width: 50px;
        height: 50px;
        background-color: #ccc;
        border-radius: 50%;
        color: black;
        font-weight: bold;
        font-size: 14px;
        text-transform: uppercase;
    }
    .chat-footer {
        padding: 10px;
        background-color: #f0f0f0;
        display: flex;
    }
    .chat-footer input[type="text"] {
        flex-grow: 1;
        padding: 10px;
        margin-right: 10px;
        border: 1px solid #ccc;
        border-radius: 18px;
        outline: none;
        height: 40px;
    }
    .chat-footer input[type="submit"] {
        padding: 10px 20px;
        background-color: #4CAF50;
        border: none;
        border-radius: 18px;
        color: #fff;
        cursor: pointer;
        outline: none;
        height: 40px;
    }

</style>
</head>
<body>
    <div class="chat-container">
        <div class="chat-header">
            Shortage Product AI
        </div>
        <div class="chat-body" id="chatBody">
            {% for entry in history %}
                <div class="message {{ 'user-message' if entry.user else 'bot-message' }}">
                    <div class="message-text">{{ entry.text }}</div>
```

```
            </div>
        {% endfor %}
      </div>
      <div class="chat-footer">
        <div class="chat-footer">
          <form action="/" method="post">
            <button type="button" id="microphone-btn" onmousedown="startRecording()"
onmouseup="stopRecording()" ontouchstart="startRecording()" ontouchend="stopRecording()">🎤
Hold to Speak</button>
            <input type="text" name="user_input" id="user_input" placeholder="Ask a question..."
autocomplete="off">
            <input type="submit" value="Send">
          </form>
          <form action="/upload_image" method="post" enctype="multipart/form-data">
            <input type="file" name="image" accept="image/*">
            <input type="submit" value="Upload Image">
          </form>
        </div>

      </div>

    </div>
    <script>
      var recognition;
      function startRecording() {
        var SpeechRecognition = SpeechRecognition || webkitSpeechRecognition;
        recognition = new SpeechRecognition();
        recognition.continuous = false; // Set to false as we want the recognition to stop after a single
result
        recognition.interimResults = false; // We only want final results
        recognition.lang = 'en-US'; // Set the language of the recognition
        recognition.start(); // Start recognition

        recognition.onresult = function(event) {
          var transcript = event.results[0][0].transcript;
          var userInputField = document.getElementById('user_input');
          userInputField.value = transcript; // Set the recognized text to the input field
          console.log('Transcript:', transcript);
          recognition.stop(); // Stop recognition after receiving the first result
        };

        recognition.onend = function() {
          if (userInputField.value) { // Check if there is text to submit
            document.querySelector('form').submit(); // Submit the form
          }
        };
      }
```

```
    function stopRecording() {
      if (recognition) {
        recognition.stop();  // This will also trigger the onend event
      }
    }
  </script>


</body>
</html>
```

App.py

Environment Variables: loading environment variables which store confidential information like the secret key and Azure service credentials.

Flask App Initialization: A new Flask application instance is created, along with configuration for CORS (Cross-Origin Resource Sharing) to allow the app to handle requests from different origins.

Azure Services Setup: Setting up clients for Azure AI Services, **Speech** and **Computer Vision**, using the credentials from the environment variables. These services enable the bot to understand **speech input** and **analyse images**.

Speech Recognition: Function **recognize_from_microphone()** is defined to handle speech recognition using **Azure's Speech SDK**. Function listens for speech input from the default microphone, processes it, and returns the recognized text.

Speech Recognition Endpoint: Route **/recognize_speech** is set up to handle **POST** requests that trigger the speech recognition function and return the recognized text as JSON.

Conversation Memory: Application with **ConversationBufferMemory** object to keep track of the conversation history. This memory is used to maintain context during a conversation with the bot.

Conversational Retrieval Chain: The **load_chain()** function configures a **ConversationalRetrievalChain** from the **Langchain** library, which uses Azure Cognitive Search for document retrieval, and **OpenAI's Chat model** for generating responses.

Home Page Endpoint: Route / serves the home page and handles both displaying and posting chat messages. It uses session storage to retain chat history.

Image Upload and Processing: It defines an allowed file extension list and sets up an upload folder. Route /upload_image is created to handle image uploads, save them securely, and process them using Azure's OCR (Optical Character Recognition) to extract text.

OCR Function: The **get_text_from_image()** function sends images to the **Azure Computer Vision service** to extract text, which can then be used as input to the conversational chain.

Main Block: Block checks if the script is the main program and, if so, runs the **Flask app**.

Chat.html

HTML Structure: The document uses HTML5 and is set to English. Meta tags are used for proper rendering and touch zooming.

Styling: Inline CSS styles are used to create a modern chat interface. The chat window is centered on the page.

Chat Header: The header contains the title “Shortage Product AI”.

Chat Body: This section displays the chat history. Messages are styled differently based on whether they’re from the user or the bot.

Chat Footer: This section contains the input form for sending messages. It includes a microphone button for **speech-to-text** functionality, a text input for typing messages, and a send button. There’s also a form for **image uploads**.

Speech-to-Text Functionality: JavaScript is used for speech recognition. The recognized speech is populated into the text input field.

Speech Recognition Trigger: The microphone button is wired to start and stop the speech recognition.

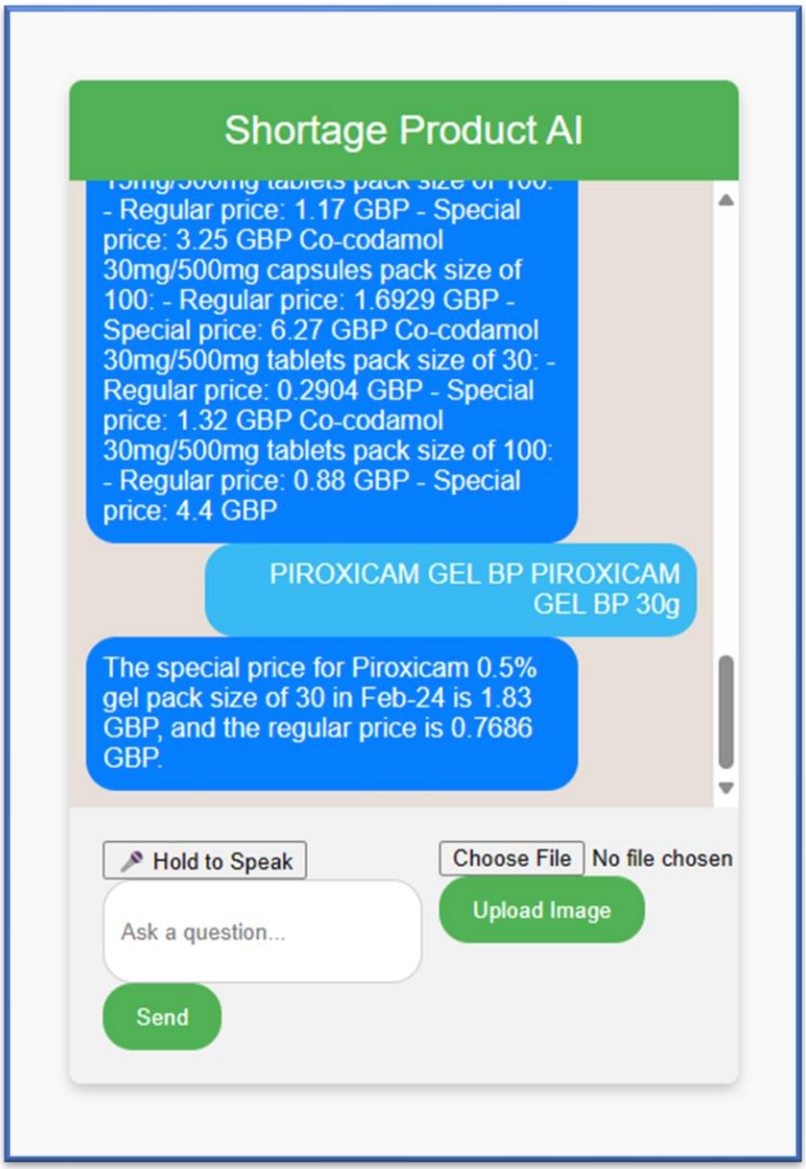
Automatic Form Submission: The form is automatically submitted when speech recognition ends and there’s text in the input field.

Image Upload: Users can upload images, which are sent to a server-side endpoint for processing.

Responsive Design: The chat interface adjusts to the screen size, making it accessible on both desktop and mobile devices.

Integration with Flask: The HTML integrates with a Flask application, allowing to interact with the bot in multiple ways. The Flask application uses these inputs to converse with users, leveraging Azure services to enhance the bot’s capabilities.

Output



I am very pleased to be successful in completion of AI-driven chatbot project, designed to address the critical issue of pharmaceutical product shortages. This chatbot, showcased in the attached screenshot, exemplifies the integration of multiple advanced technologies to deliver real-time, user-friendly interactions and information retrieval.

Key Capabilities and Features:

Semantic Search Integration: Utilizing OpenAI's text-embedding-3-small embeddings model, we have empowered Azure AI Search to understand and retrieve information on product shortages semantically. This allows the bot to provide contextually relevant information, such as specific product prices and stock levels, as evidenced by the detailed responses regarding "Piroxicam Gel" pricing in the user interface.

Speech-to-Text Interaction: The chatbot features a 'Hold to Speak' function that enables voice-based queries. This function capitalizes on **Azure's Speech SDK**, allowing users to communicate with the bot in a natural and accessible manner.

Image Processing and Text Extraction: An innovative image upload capability allows users to input data via pictures, such as photographs of product lists or prescriptions. Leveraging **Azure's Computer Vision OCR**, the system extracts text from images, making the data available for processing and response by the chatbot.

Responsive Web Design: The user interface, as part of the Flask application, is designed to be clean and intuitive, ensuring accessibility across devices. The conversation history is presented in an easily navigable format, with distinct visual cues for user and bot messages.

Conversational Memory: Through ConversationBufferMemory, the chatbot maintains the context of interactions, providing continuity across a session. This aspect is particularly important when dealing with complex inquiries about product availability and shortages.

The project objectives has been met to have reliable tool for healthcare professionals and supply chain managers to quickly obtain information on medication availability and pricing changes. The final product not only meets these objectives but also enhances user experience through multi-modal interaction capabilities—**text, voice, and image inputs**.

Also by changing the secret key of flask application, the Bot would be initiated for the new chat interface for the new users.

```
SECRET_KEY = 345852
```

Link: http://127.0.0.1:5000

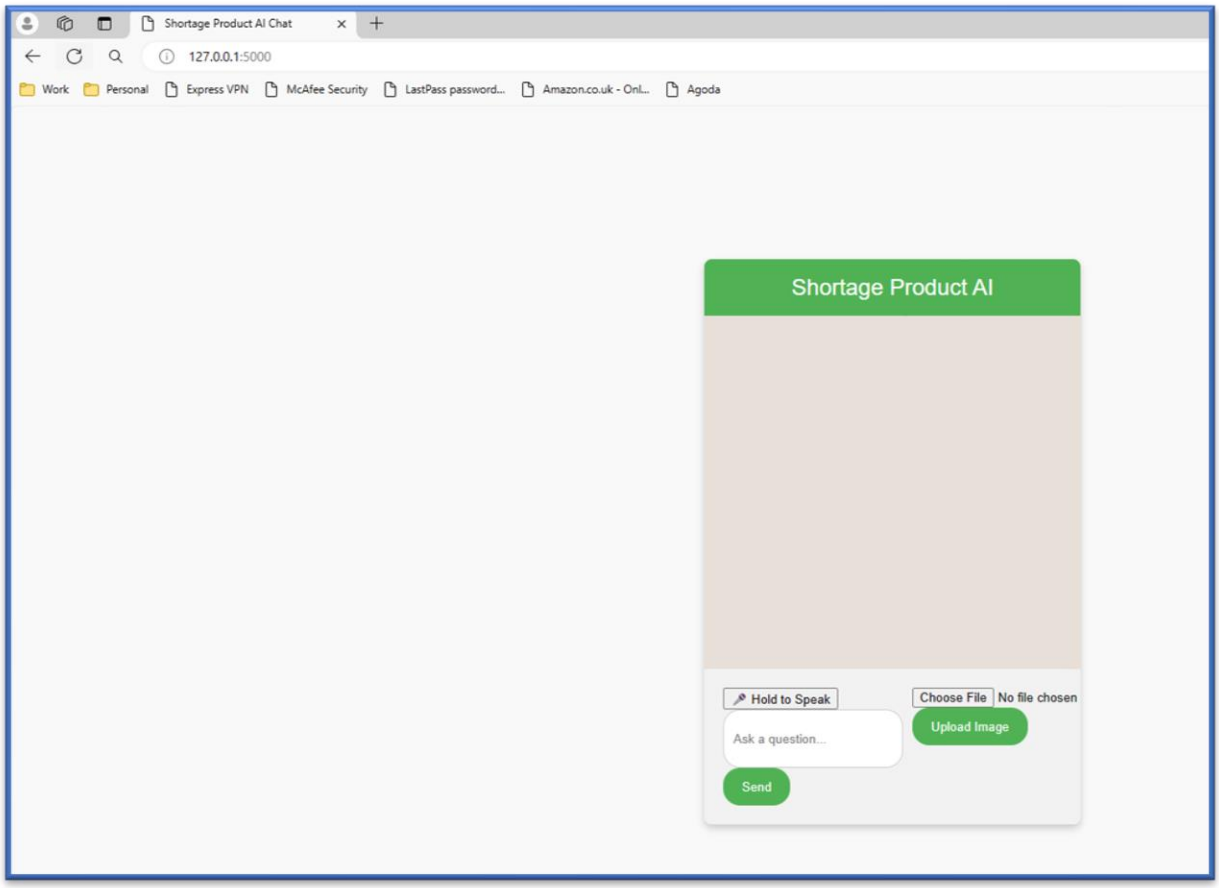
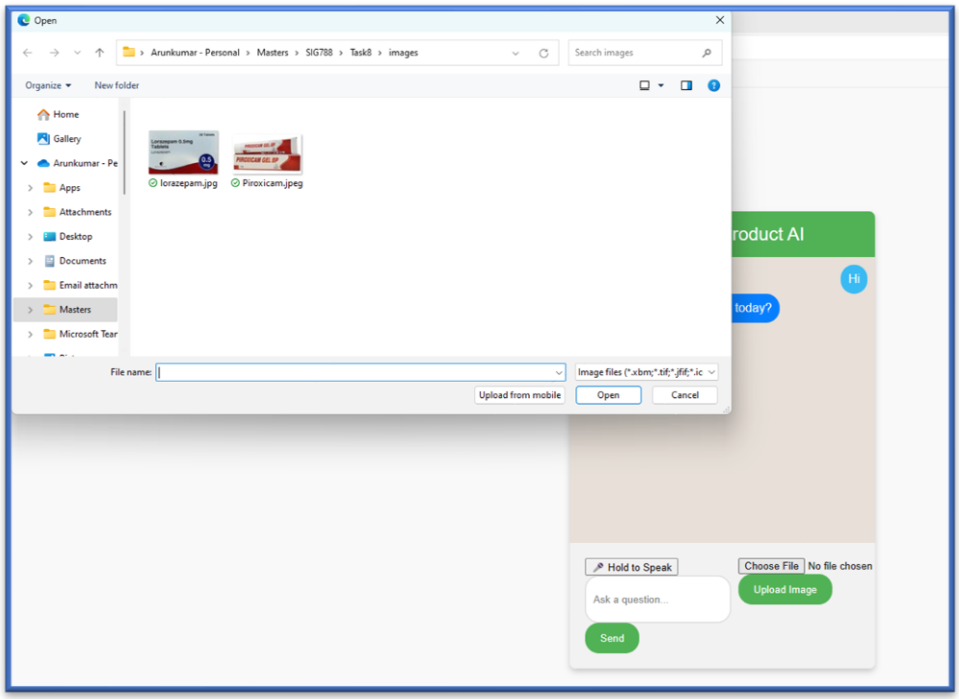
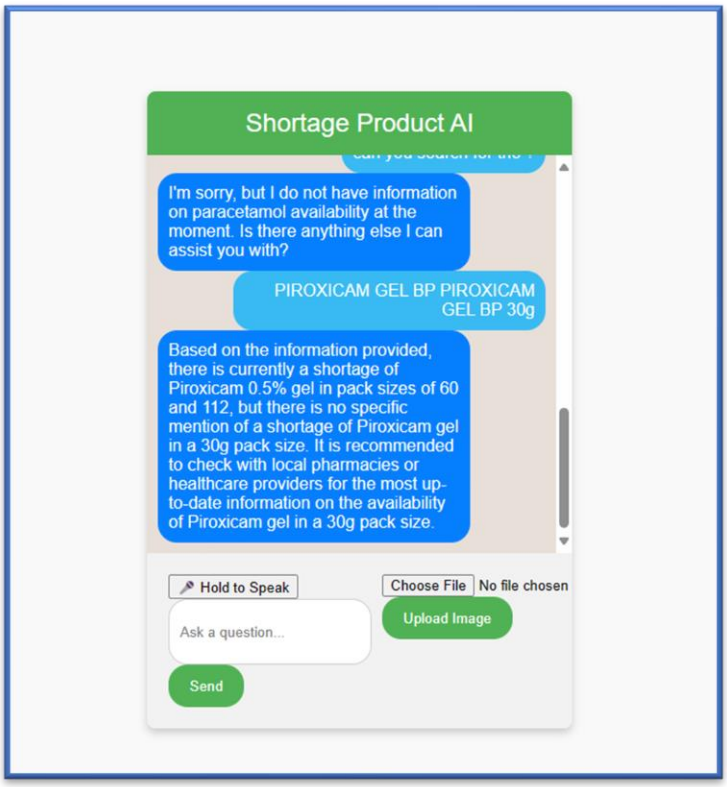


Image selection:

I have provided the image input and retrived the results.



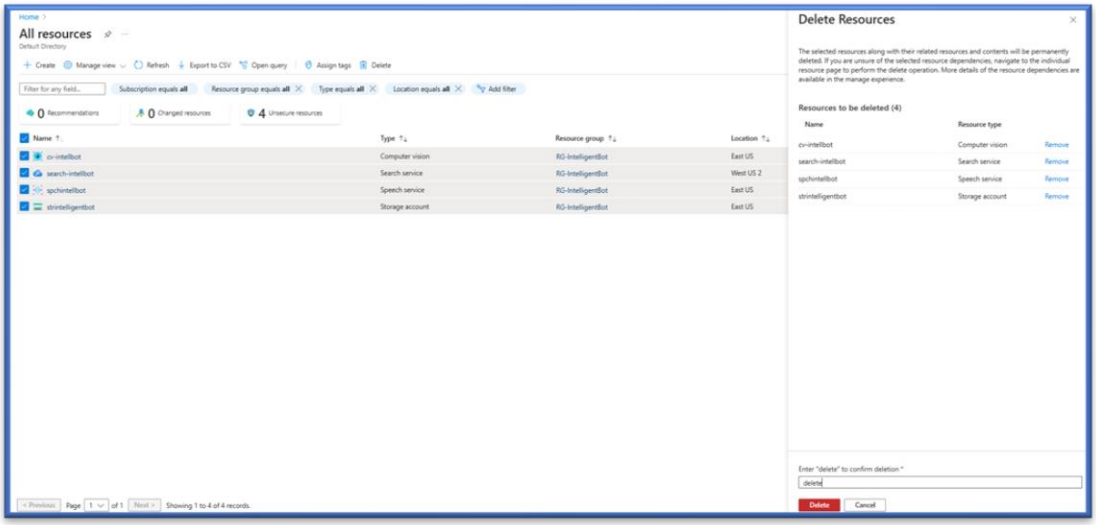


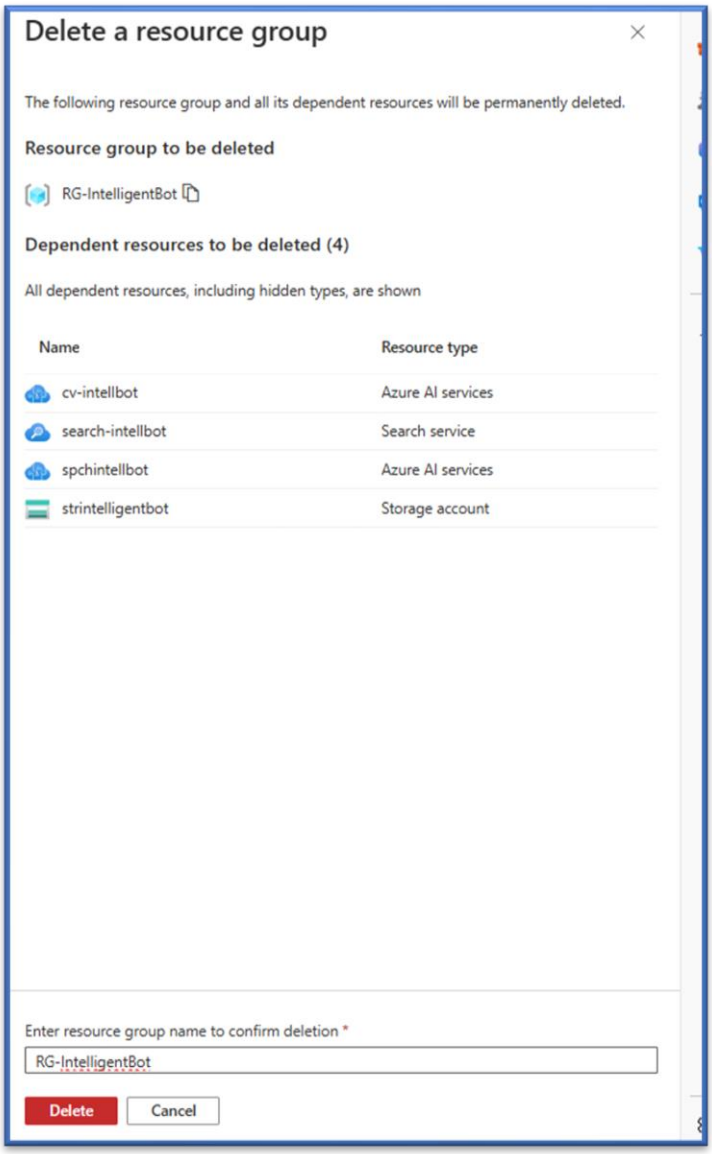
Conclusion:

Project stands as a testament to the potential of AI to transform industry-specific challenges into opportunities for innovation and improved service delivery. The bot is now well-equipped to assist in managing the complexities of **pharmaceutical supply chains**, thereby contributing to better healthcare outcomes.

Cleaning Up Activity:

As completed the project, cleaning up the resources and resource group to reduce cost on the Azure subscription.





References:

- Microsoft Learn (no.date.) "**Azure Cognitive Language Service Question Answering client library for Python - version 1.1.0**" Available at: https://learn.microsoft.com/en-us/python/api/overview/azure/ai-language-questionanswering-readme?view=azure-python
- Great Learning (2024) " **Week 5 - Language Understanding & Azure AI Fundamentals**" Mentor session Available at: https://olympus.mygreatlearning.com/mentorship_recordings/2317305
- Community pharmacy England (no.date.) "**Price Concession**" available at: https://cpe.org.uk/funding-and-reimbursement/reimbursement/price-concessions/
- Great Learning (2024) "**Case study on Computer Vision & Custom Vision using Python SDK**" Available at: https://olympus.mygreatlearning.com/courses/109553?module_id=747540
- Microsoft Learn (no.date.) "**Quickstart: Image Analysis**" Available at: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/quickstarts-

sdk/image-analysis-client-library?tabs=windows%2Cvisual-studio&pivots=programming-language-python

- Microsoft Learn (no.date.) "**Call the Image Analysis 3.2 API**" Available at: https://learn.microsoft.com/en-us/azure/ai-services/computer-vision/how-to/call-analyze-image?tabs=python

- Great Learning (2024) "**Case study on Computer Vision & Custom Vision using Python SDK**" Available at: https://olympus.mygreatlearning.com/courses/109553?module_id=747540

- Alam Smith (Sep 23, 2021) "**Azure Computer Vision using Python**" Available at: https://www.youtube.com/watch?v=3Zfcn1tsSwE&t=142s

- Coding Crashcourses (18th Aug 2023) "**LangChain on Microsoft Azure - ChatBot with Azure Web Service & Azure Cognitive Search**" Available at: https://www.youtube.com/watch?v=WAedZvSDZAI&t=188s