Contents

[OPEN AI 2](#_Toc217671645)

[GPT3 VERSUS GPT4 2](#_Toc217671646)

[USING OPENAI MODELS 2](#_Toc217671647)

[SIMPLE API CALL TO OPEN AI 2](#_Toc217671648)

[CREATING EMBEDDINGS 3](#_Toc217671649)

[IMAGE GENERATION USING DALL-E 4](#_Toc217671650)

[SPEECH TO TEXT CONVERSION 5](#_Toc217671651)

[AZURE OPEN AI & AZURE FOUNDRY 5](#_Toc217671652)

[AZURE OPEN AI 5](#_Toc217671653)

[PROVISIONING OPENAI SERVICE 7](#_Toc217671654)

[CREATING DEPLOYMENT 8](#_Toc217671655)

[CODE 9](#_Toc217671656)

[AZURE AI SEARCH 10](#_Toc217671657)

[HOW AI SEARCH WORKS 11](#_Toc217671658)

[HOW VECTOR WORKS IN AI SEARCH 12](#_Toc217671659)

[IMPLEMENTATION 13](#_Toc217671660)

[RESOURCES 13](#_Toc217671661)

[BUILDING RAG WITH AZURE AI SEARCH, AZURE OPEN AI AND LANGCHAIN 20](#_Toc217671662)

[SETTING UP 21](#_Toc217671663)

[STEP 1: CREATE AZURE TABLE BASED ON CSV FILE 22](#_Toc217671664)

[STEP 2: CREATE AZURE AI SEARCH SERVICE AND INDEXER 23](#_Toc217671665)

[STEP 3: CREATE AI SEARCH INDEXER 23](#_Toc217671666)

[INITIALIZE THE RETRIEVER, PROMPT, AND LLM 27](#_Toc217671667)

[PROCESSING CHAIN AND USER INPUT LOOP 28](#_Toc217671668)

# OPEN AI

## GPT3 VERSUS GPT4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Model Name | Type | Capabilities | Usecase |
| GPT-4o | Multimodal LLM | Text, image, audio input/output; real-time reasoning | Chatbots, assistants, multimodal search |
| o3-series | Text-only LLM | Advanced reasoning, long-context understanding | Research, technical writing, complex workflows |
| o1-series | Lightweight LLM | Fast, efficient responses with decent reasoning | Mobile apps, real-time systems |
| Whisper | Audio model | Speech multilingual transcription -to-text, | Voice assistants, accessibility tools |
| DALL·E | Image model | Text-to-image generation and editing | Design, prototyping, creative content |
| TTS (Text-to-Speech) | Audio model | Converts text to natural-sounding speech | Voice synthesis, accessibility |
| Embedding Models | Vector model | Converts text into high-dimensional vectors for semantic similarity | Search, clustering, recommendation systems |

# USING OPENAI MODELS

## SIMPLE API CALL TO OPEN AI

|  |  |
| --- | --- |
| INSTALL OPEN AI MODULE | pip install **openai**  **or**  uv add **openai** |
| from openai import OpenAI  from dotenv import load\_dotenv  import os  load\_dotenv('openai.env')  # Access the environment variables from the .env file  api\_key = os.environ.get('OPENAI\_API\_KEY')  from openai import OpenAI  client = OpenAI()  response = client.chat.completions.create(    model="gpt-4",    messages=[      {"role": "user", "content": "Who is Prime Minister of India and give some bullet points of his achievements"},   ]  )  print(response) | Interaction with OpenAI's GPT-4 model using the official OpenAI Python library.   * [**load\_dotenv('openai.env')**](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) **:** Loads environment variables from a file named [openai.env](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html), which contains the API keys. * [**os.environ.get('OPENAI\_API\_KEY')**](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html): Retrieves the OpenAI API key from the environment variable * The [OpenAI](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) client is then instantiated, which will be used to send requests to the API. * A chat completion request is made using the **[client.chat.completions.create()](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)** method. The request specifies the model (gpt-4) and provides a list of messages, simulating a conversation * The model processes this input and generate a response. |

## CREATING EMBEDDINGS

|  |  |
| --- | --- |
| from openai import OpenAI  from dotenv import load\_dotenv  import os  load\_dotenv('openai.env')  # Access the environment variables from the .env file  api\_key = os.environ.get('OPENAI\_API\_KEY')  from openai import OpenAI  client = OpenAI()  response = client.embeddings.create(  input="cat",  model="text-embedding-3-small"  )  print(response.data[0].embedding) | * Embeddings are a way to represent data—especially text, images, or other complex inputs—as numerical vectors in a high-dimensional space. These vectors capture the **semantic meaning** or **contextual relationships** of the data, making it easier for machines to process and understand. * In Simple terms - Imagine we want to teach a computer what words mean. Instead of giving it dictionary definitions, we give each word a unique set of numbers (a vector) that reflects how it's used in language. * Words with similar meanings or contexts will have similar vectors |

|  |  |
| --- | --- |
| What do we mean by vectors in a high-dimensional space,  It means it’s a representation of data (like words, images, or sentences) as points in a space with many dimensions—often **hundreds or even thousands**.  What is a Vector?  A **vector** is just a list of numbers. For example:   * A 2D vector: [3.5, -1.2] → can be plotted on a flat plane. * A 3D vector: [2.1, 0.5, -3.3] → can be visualized in 3D space.   But embeddings often use **300, 768, or even 4096 dimensions**—far beyond what we can visualize.  Why High-Dimensional?  Because more dimensions allow us to **capture more nuanced relationships**. For example:   * In a 300-dimensional space, each dimension might encode something like:   + Gender association   + Verb tense   + Sentiment   + Topic relevance   + Contextual usage   So the word **“apple”** might be close to:   * “fruit” in one context * “iPhone” in another   Depending on how the embedding is trained. | |
| Imagine each word is a dot in a huge invisible cloud. Words that **mean similar things** or are **used in similar contexts** are **closer together** in this cloud. For example - Imagine a 2D scatter plot where each **dot** represents a word. Words with similar meanings or contexts would appear **close together**.  For example:   * **Cluster 1 (Royalty)**: king, queen, prince, princess * **Cluster 2 (Fruits)**: apple, banana, orange, grape * **Cluster 3 (Vehicles)**: car, bus, train, truck * **Cluster 4 (Animals)**: cat, dog, lion, tiger   Each cluster forms because the embedding vectors for those words are similar in high-dimensional space, and dimensionality reduction (like t-SNE or PCA) helps us visualize that in 2D. |  |

## IMAGE GENERATION USING DALL-E

|  |
| --- |
| from openai import OpenAI  from dotenv import load\_dotenv  import os  load\_dotenv('openai.env')  # Access the environment variables from the .env file  api\_key = os.environ.get('OPENAI\_API\_KEY')  from openai import OpenAI  client = OpenAI()  response = client.images.generate(  model="dall-e-3",  prompt="a Black furry Dog with black eyes with a dog collar and a white cat with blue eyes with a necklace",  size="1024x1024",  quality="standard",  n=1,  )  image\_url = response.data[0].url  print(image\_url) |

## SPEECH TO TEXT CONVERSION

|  |
| --- |
| from openai import OpenAI  from dotenv import load\_dotenv  import os  load\_dotenv('openai.env')  # Access the environment variables from the .env file  api\_key = os.environ.get('OPENAI\_API\_KEY')  client = OpenAI()  audio\_file= open("/Users/kshitijjoy\_1/Downloads/deep\_fake\_video.mp4", "rb")  transcription = client.audio.transcriptions.create(  model="whisper-1",  file=audio\_file  )  print(transcription.text) |

# AZURE OPEN AI & AZURE FOUNDRY

* Azure OpenAI Service is part of the Azure AI Foundry ecosystem, but Foundry offers more tools and flexibility for building full-fledged AI applications.

|  |  |
| --- | --- |
| Platform | Scope |
| Azure AI Foundry | * **Azure AI Foundry** is a **comprehensive platform** designed to help you build **generative AI applications**—like chatbots, copilots, and intelligent agents. * It’s like a **workshop** where we get all the tools, models, and infrastructure needed to create smart, interactive AI systems. |
| Azure OpenAI Service | * **Azure OpenAI Service** is a **specialized service** within Azure that gives you **direct access to OpenAI’s models** (*like GPT-4, GPT-4o, DALL·E, Whisper*). * We use it when we want to **generate text, images, transcribe audio**, or perform semantic search using these models. |

HOW THE ARE RELATED

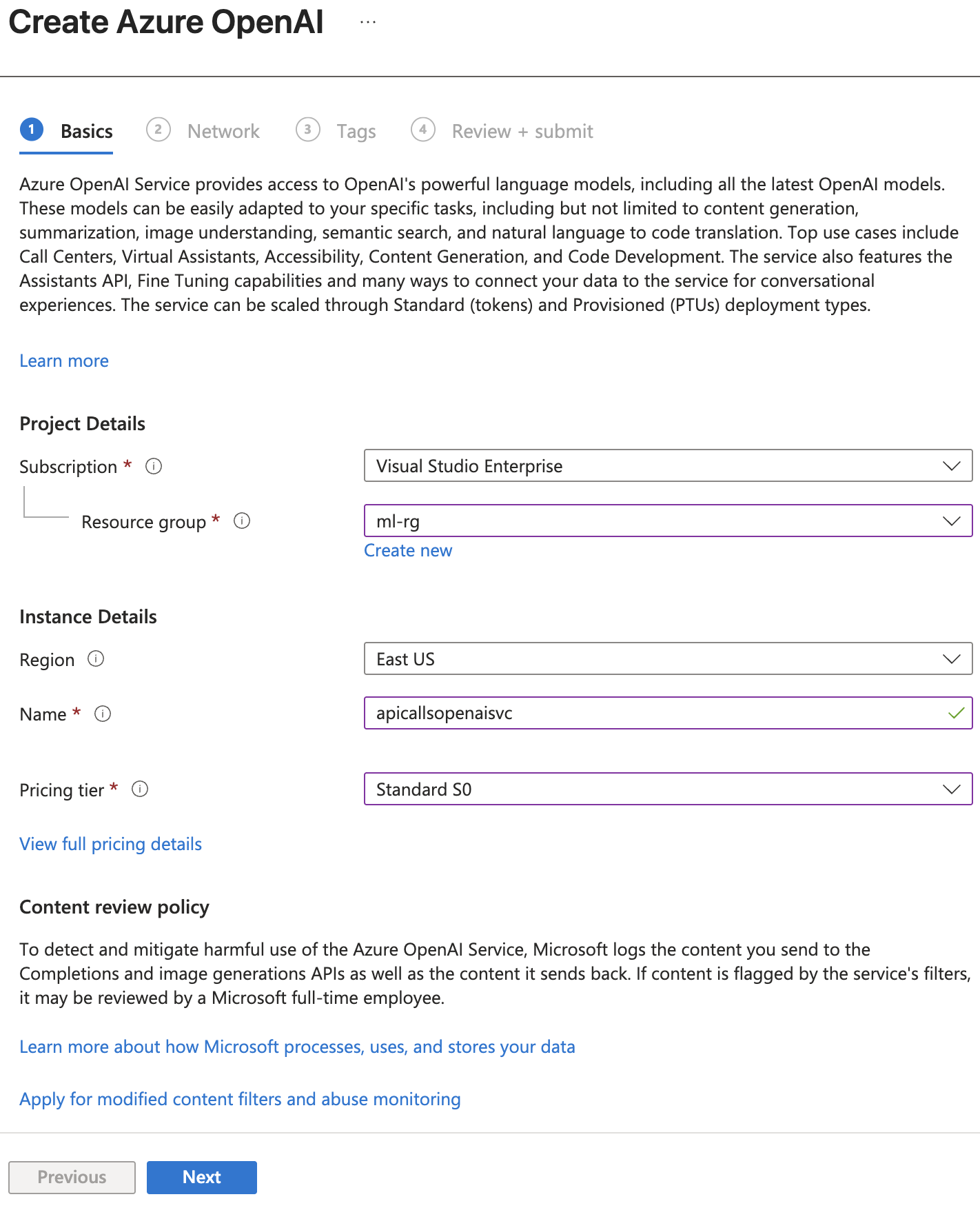
* **Azure OpenAI Service** is a **toolbox** with powerful tools (models).
* Azure AI Foundry is the entire workshop that includes:
  + That toolbox (OpenAI models)
  + Other toolboxes (Meta, Hugging Face models)
  + Workbenches (SDKs, orchestration tools)
  + Safety gear (governance, monitoring)
  + Collaboration zones (project workspaces)
* Azure OpenAI Service is a subset of Azure AI Foundry—we can use it inside Foundry, but Foundry gives us much more.

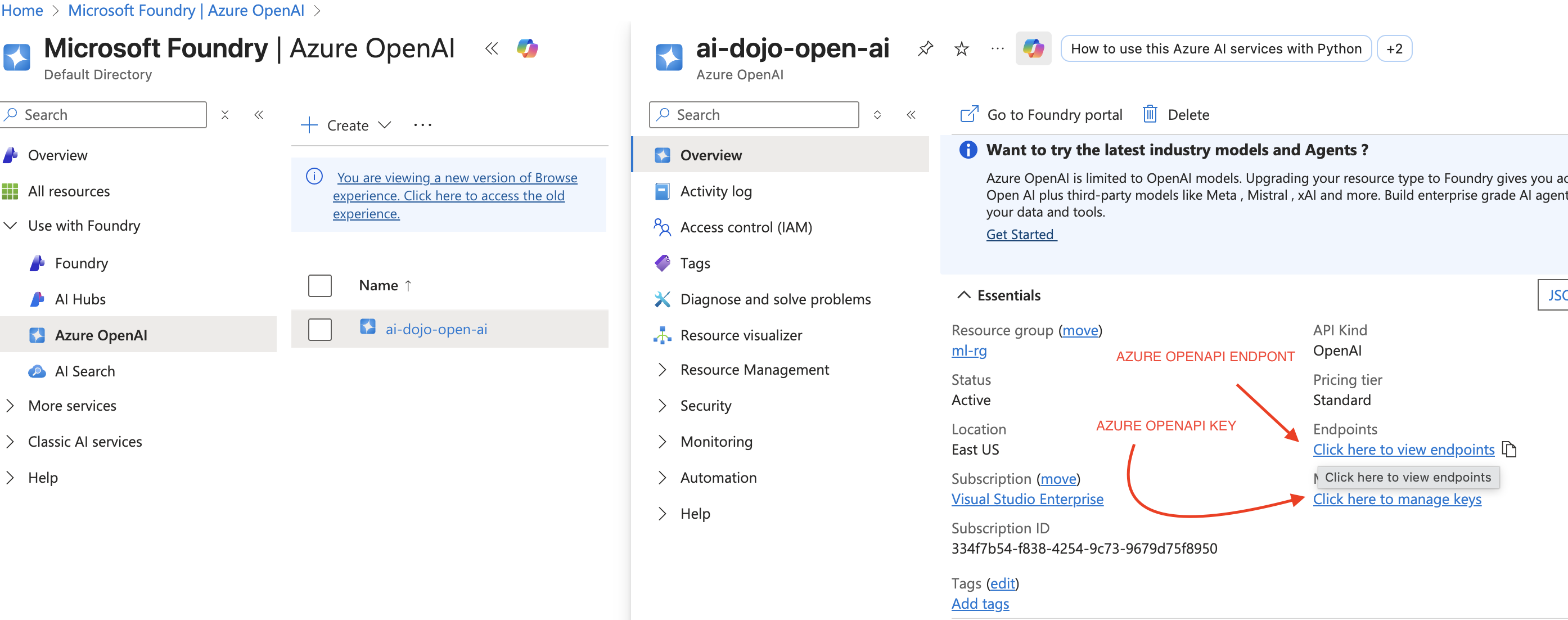
# AZURE OPEN AI

OPEN AI VERSUS AZURE OPEN AI

|  |  |
| --- | --- |
| OPENAI - INSTANTIATION | AZURE OPEN AI - INSTANTIATION |
| import os  from openai import OpenAI  client = OpenAI(  api\_key=os.getenv("AZURE\_OPENAI\_API\_KEY")  ) | import os  from openai import AzureOpenAI  client = AzureOpenAI(  azure\_endpoint = os.getenv("AZURE\_OPENAI\_ENDPOINT"),  api\_key=os.getenv("AZURE\_OPENAI\_API\_KEY"),  api\_version="2024-05-01-preview"  ) |
| * **Hosted by OpenAI** on their own servers. * Call API directly using an **API key**. * **Authentication**: Only needs **api\_key**. * **Use Case**: Great for general apps, prototypes, and quick integrations. | * **Hosted by Microsoft Azure**. * Same OpenAI models, but with **enterprise-grade security, compliance, and integration**. * **Authentication**: Needs:   + **azure\_endpoint** (Azure resource URL)   + **api\_key**   + **api\_version**   **EXTRA BENEFITS**:   * Private networking (VNET, Private Endpoint). * RBAC (Role-Based Access Control). * Compliance (HIPAA, GDPR). * Integration with **Azure AI Search** for RAG. |
| completions =client.completions.create(  model=os.getenv("**MODEL\_NAME**"),  prompt="What is life?",  )   * **Here it takes model name** | completions =client.completions.create(  model=os.getenv("AZURE\_OPENAI\_DEPLOYMENT"),  prompt="What is life?",  )   * This must match the custom deployment name we choose from model name. |
| completions =client.chat.completions.create(  model=’gpt-4’,  messages=**<messages>**,  ) | completions =client.chat.completions.create(  model=<DEPLOYMENT\_NAME>,  messages=**<messages>**,  ) |
| completions =client.embeddings.create(  model=’text-embeddings-ada-002’,  input=<input>,  ) | completions =client.completions.create(  model=<DEPLOYMENT\_NAME>,  input=<input>,  ) |

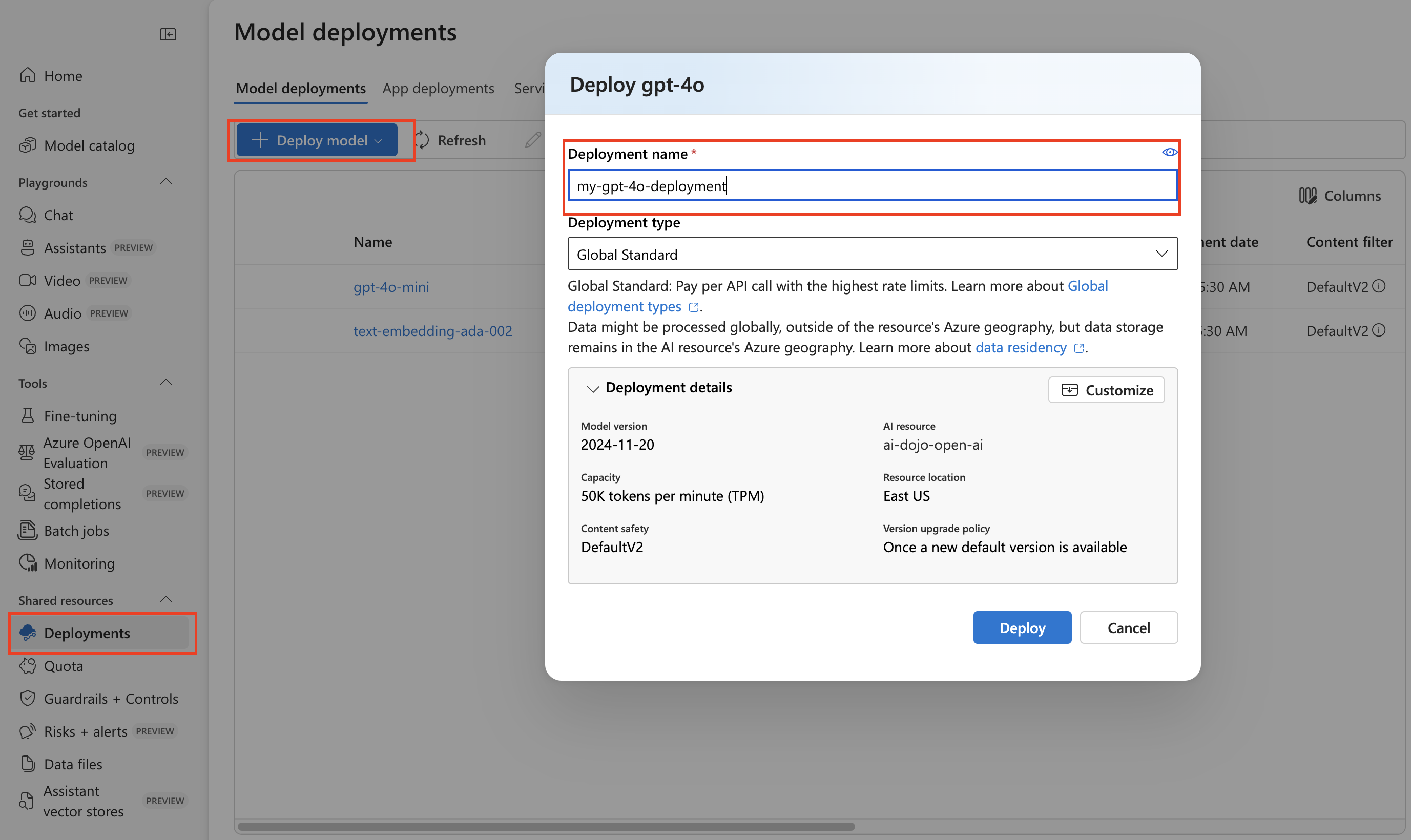
## PROVISIONING OPENAI SERVICE

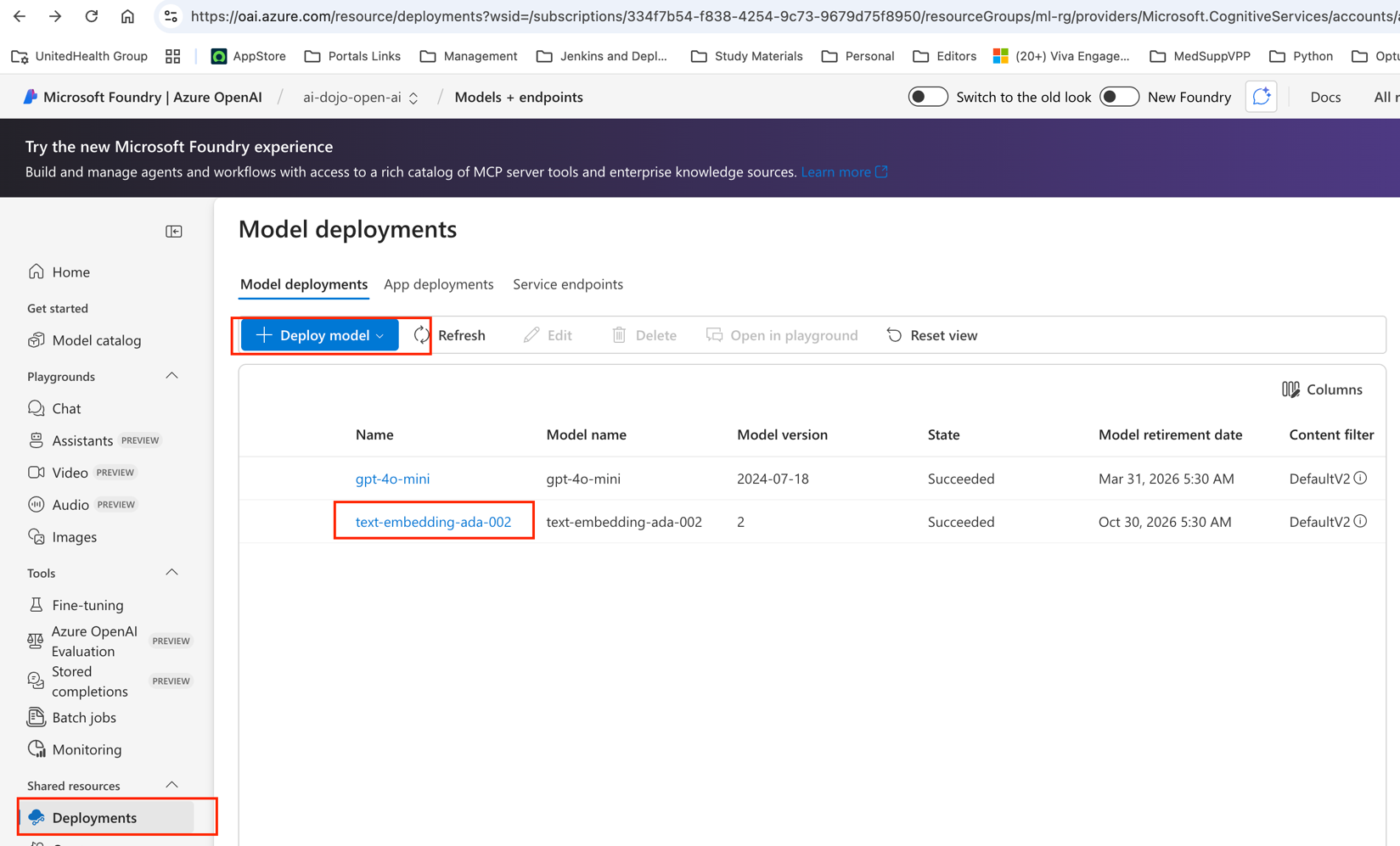




### CREATING DEPLOYMENT

* In **Azure OpenAI**, we **cannot call the base model’s name directly** (e.g., **gpt-3.5-turbo** or **gpt-4**). Instead, we must **create a deployment** in Azure OpenAI Studio and use the **deployment name** in API calls.
* The deployment is linked to a **base model** (e.g., GPT-3.5 Turbo, GPT-4).
* The deployment can be done from chat’s playground where can configure:
  + **Deployment name** (custom name we will use in our code).
  + **Model version** (e.g., 0301 or auto-update).
  + **Token limits** and **content filters**.





* Once created, the deployment appears in the **Chat Playground** and can be used for testing and API calls.

|  |  |
| --- | --- |
| Hence in the code, we need to refer the **deployment name**, not the model name. | response = client.chat.completions.create(      model="your-deployment-name",      messages=[{"role": "user", "content": "Hello"}]  ) |

### CODE

|  |
| --- |
| from openai import AzureOpenAI  from dotenv import load\_dotenv  import os  load\_dotenv()  client = AzureOpenAI(  azure\_endpoint = os.getenv("AZURE\_OPENAI\_ENDPOINT"),  api\_key=os.getenv("AZURE\_OPENAI\_API\_KEY"),  api\_version=os.getenv("AZURE\_OPENAI\_API\_VERSION")  )  response = client.chat.completions.create(  model=os.getenv("AZURE\_OPENAI\_DEPLOYMENT"),  messages=[  {"role": "user", "content": "Give me the names of US Presidents till now"},  ]  )  print(response) |

A screenshot of a computer

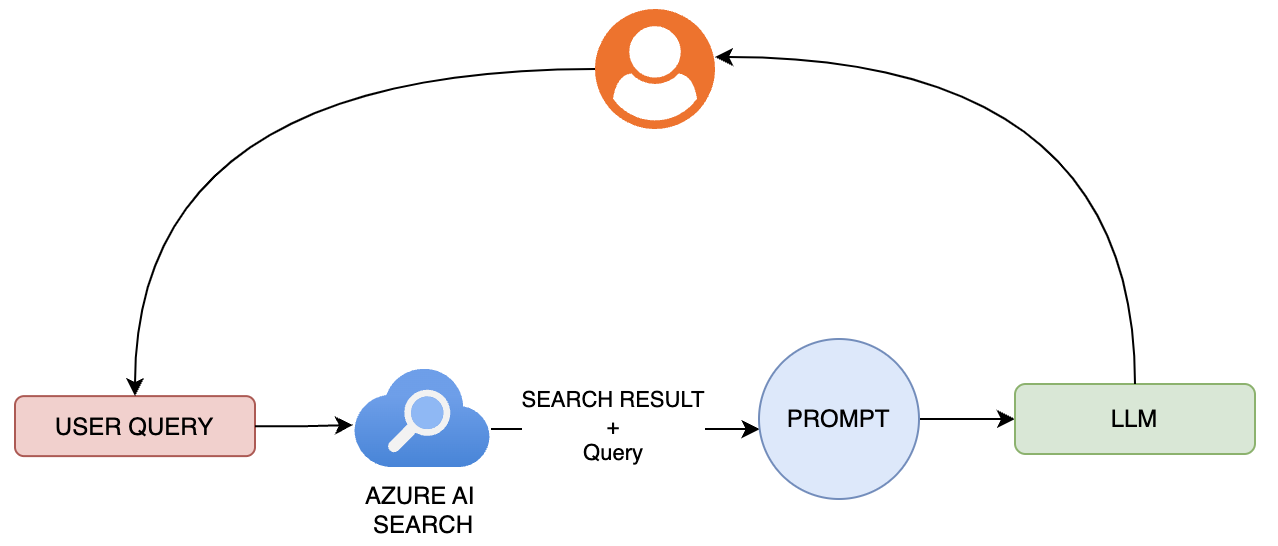
AI-generated content may be incorrect.

# AZURE AI SEARCH

* Azure AI Search (previously Azure Cognitive Search) is a managed search service that helps you retrieve information securely and at scale.
* Enables search on your private data, not on public LLM training data. This is essential for RAG (Retrieval-Augmented Generation) because it uses your own documents to answer queries.

|  |
| --- |
| Think of **Azure AI Search** as a **smart librarian** for your company’s documents.   * A librarian doesn’t write books; they **help you find the right book quickly**. * Similarly, Azure AI Search doesn’t generate answers by itself—it **searches your private data** (files, PDFs, databases) and gives the most relevant pieces. |

How Does It Fit into RAG?

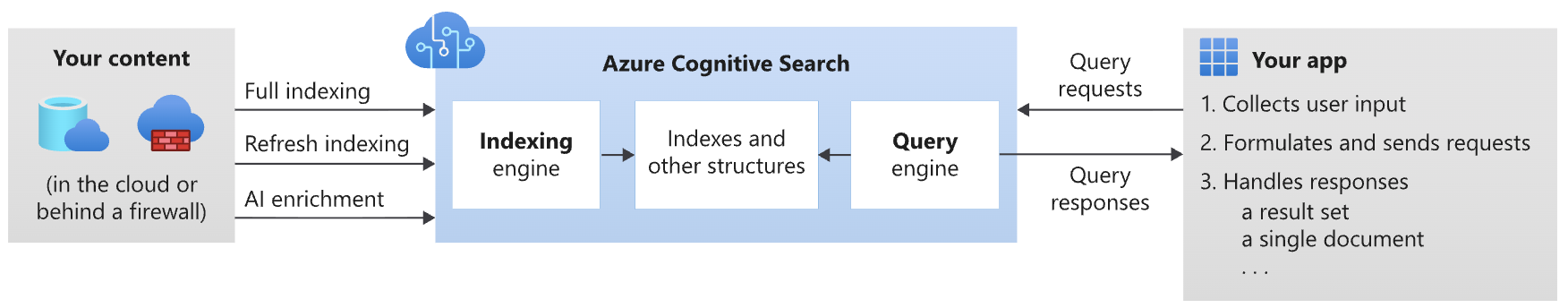


RAG FLOW:

1. User Query → Sent to Azure AI Search.
2. Search Results → Combined with the query to form a prompt.
3. Prompt → Sent to the LLM for generating a final answer.

|  |
| --- |
| EXAMPLE:   * Imagine you ask: “What is our company’s parental leave policy?” * LLM doesn’t know your company’s private rules. * So, we use Azure AI Search to **fetch the correct policy from company’s HR documents**, then give that to the AI model. * The AI combines your question + the retrieved info and gives a **precise, trustworthy answer**. |

### HOW AI SEARCH WORKS



1. Your Content
   1. This is your private data: PDFs, Word files, databases (like Cosmos DB etc.), blob storage.
   2. Example: HR policy PDFs in Azure Blob Storage.

AZURE AI SEARCH (or AZURE COGNITIVE SEARCH)

1. INDEXING ENGINE
   1. Azure AI Search creates an **index** (like a catalog in a library).
   2. It breaks big documents into **chunks** (paragraphs or sections) so searching is fast.
   3. Example: A 100-page HR handbook is split into smaller pieces.
   4. Adds **AI enrichment** (OCR, language detection, key phrase extraction).
   5. **Example**:
      1. If your HR policy is a scanned image, OCR converts it into searchable text.
      2. It also extracts key phrases like “parental leave” or “paid time off.”
2. **INDEXES AND STRUCTURES**
   1. These are the **data structures** that store your searchable content.
   2. Think of them as the “organized shelves” in the library.

|  |
| --- |
| * Inside **Azure AI Search**, when your content (documents, PDFs, databases) is processed by the **Indexing Engine**, it creates an **inverted index**. * Think of it as a **reverse dictionary**:   + A normal dictionary maps **word → meaning**.   + An inverted index maps **word → list of documents where the word appears**. * This inverted index is stored in the **Indexes and other structures** box in the diagram. * Later, the **Query Engine** uses this inverted index to quickly find documents that match your search terms.   WHY IS IT IMPORTANT?   * It makes **keyword-based search fast**. * Instead of scanning every document, Azure AI Search looks up the inverted index to find which documents contain the words you searched for.   EXAMPLE  Imagine you have HR documents stored in Azure Blob:   * **Doc1**: “Parental leave provides 12 weeks paid leave.” * **Doc2**: “Paid time off policy for employees.”   Azure AI Search builds an inverted index that links each unique word to the documents where it appears and tracks how often the word shows up, which helps with ranking results.   * **Word → Documents** * **Parental → [Doc1]** * **Leave → [Doc1]** * **Paid → [Doc1, Doc2]** * **Policy → [Doc2]**   Now, when your app sends a query **“parental leave policy”**, the **Query Engine** uses this inverted index to instantly find **Doc1 and Doc2** without scanning all files.  COMBINED WITH OTHER FEATURES   * Inverted index is used for **keyword search**. * Azure AI Search also adds **semantic search** and **vector search** for meaning-based queries.   + Keyword search finds exact matches.   + Vector search finds similar meaning (e.g., “paid time off” ≈ “leave benefits”). |

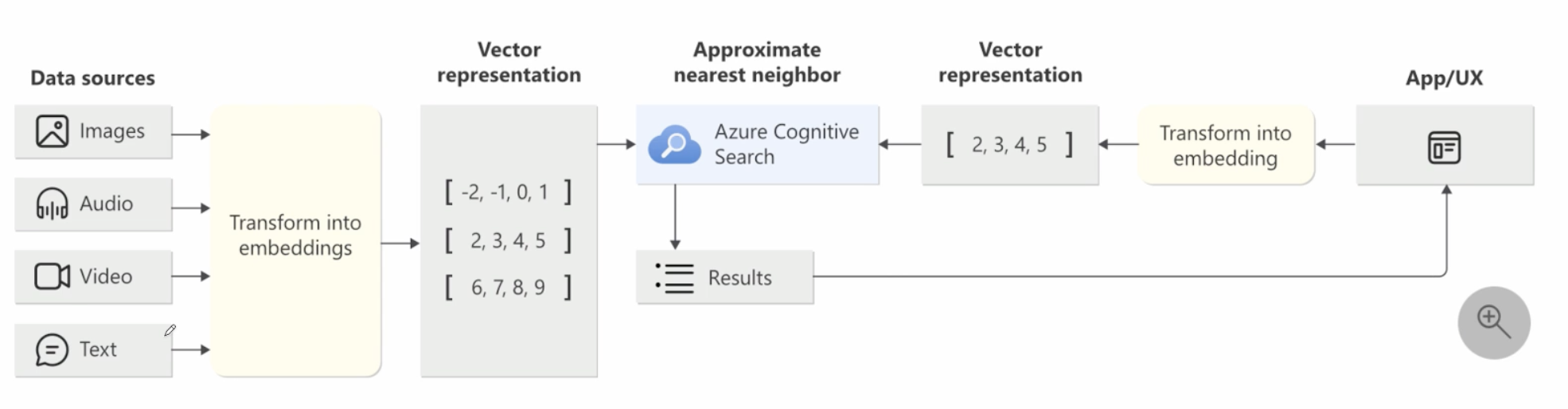
1. QUERY ENGINE
   1. When you ask a question, it finds the most relevant chunks based in ranking (called relevance ranking ).
   2. Example: You search “parental leave policy” → It finds the section in the HR handbook that talks about parental leave.
2. Ranking
   1. It ranks results by relevance (most useful first).
   2. Example: If “parental leave” appears 10 times in one document and only once in another, the first one ranks higher.

### HOW VECTOR WORKS IN AI SEARCH

What is Vector Search?

* Vector search helps find **similar meaning**, not just exact words.
* Instead of matching keywords, it uses **embeddings**—numerical representations of text that capture context and meaning.

How It Works in Azure AI Search?



**DATA STORE**

* **Data Sources -** Documents such as PDFs, text files, images, and audio are securely stored in repositories like Azure Blob or Cosmos DB. For instance, HR policy documents may reside within Azure Blob Storage.
* **Transform data into vector embeddings :** 
  + Each document or its segments is converted into numerical vectors using an embedding model, such as OpenAI’s Ada.
  + These vectors encapsulate the semantic meaning of the information.
  + For example, phrases like “Bank of the river” and “Money in the bank” will generate distinct embeddings due to their differing contextual meanings.

**Persist embeddings within the index**

* Azure AI Search maintains these vector representations alongside the standard inverted index, enabling enhanced search and retrieval functionality.

**DATA** **RETRIEVAL**

* **User query also converted into embeddings. i.e** when you ask: “What is parental leave policy?”, your query is turned into a vector too.
* Azure AI Search compares your query vector with document vectors and finds the closest matches in meaning (not just exact words).
* Example:
  + Query: “Paid time off policy”
  + Result: Document about “Leave benefits” because they are semantically similar.
* **Ranking and response**
  + - Results are ranked by relevance and sent back to your app.
    - Your app shows the most relevant chunks.

## IMPLEMENTATION

Prerequisites for RAG with Azure AI Search

**CREATE A STORAGE ACCOUNT**

* We need storage for documents like PDFs and Word files.
* Usually we use **Azure Blob Storage** to upload files

**CREATE AN EMBEDDING DEPLOYMENT**

* Embedding models will transform the content into embeddings (vectors) to enable semantic and vector searches.
* Example: Every paragraph within your HR policy is converted into a numeric vector that represents its meaning.

**CREATE A CHAT MODEL DEPLOYMENT**

* Purpose: To generate responses based on the content retrieved.
* Method: Implement a GPT model (such as GPT-3.5 or GPT-4) within Azure OpenAI.
* For example, when a user inquires about the parental leave policy, the chat model utilizes relevant retrieved information to provide a comprehensive answer.

**CREATE AZURE AI SEARCH SERVICE**

This is the main search engine that organizes your documents and handles keyword and vector searches. Setup requires:

* **Storage account** for document storage.
* **Embedding deployment** to create vectors for indexing.

**BRING YOUR OWN DATA**

* RAG uses your private data in addition to the model’s training data.
* Upload documents to Blob Storage; Azure AI Search ingests and indexes them.
* Examples include HR handbooks, Medicare plan PDFs, and internal FAQs.

**DATA INGESTION & INDEXING**

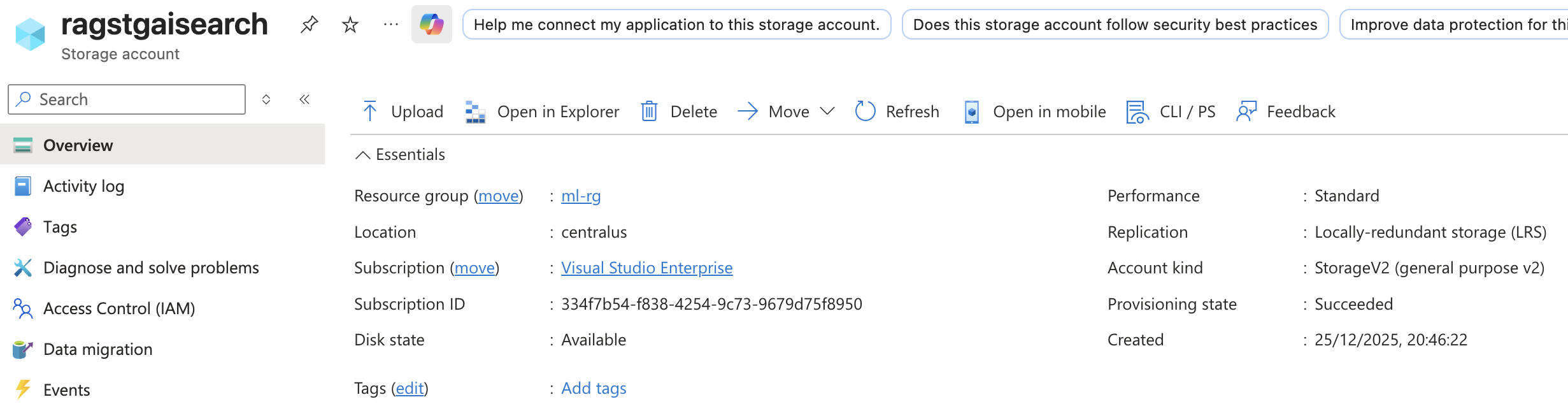
* Purpose: To enable efficient data searchability.
* Procedure:
  + Divide documents into well-defined segments, such as paragraphs or sections.
  + Implement AI-based enrichment processes, including Optical Character Recognition (OCR), language identification, and key phrase extraction.
  + Generate vector embeddings for each segment.
  + Store all embeddings in the index to facilitate rapid retrieval.

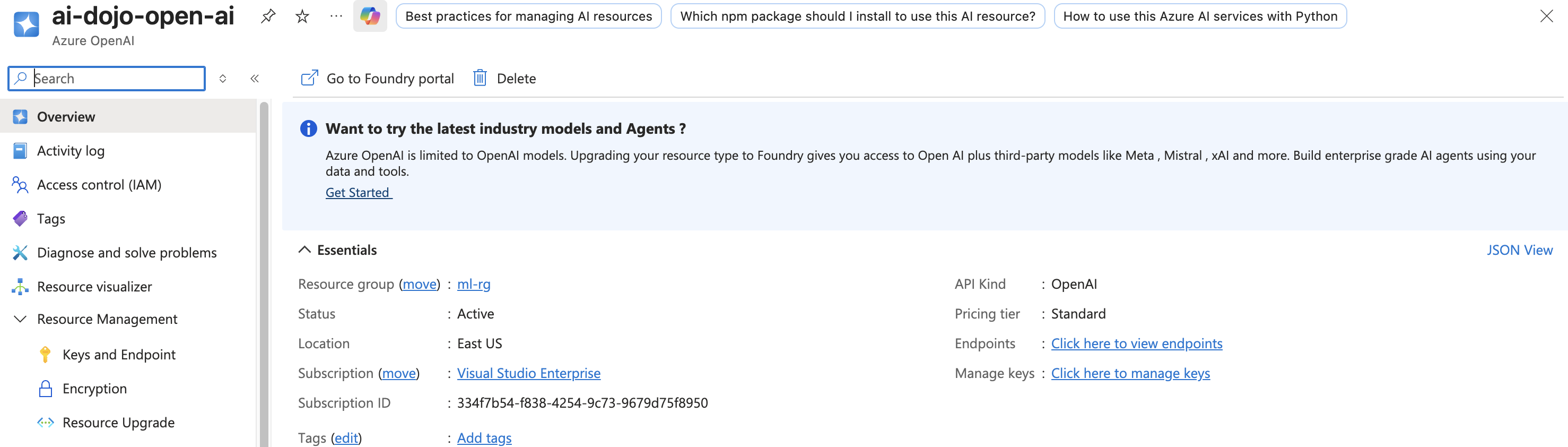
### RESOURCES

|  |
| --- |
| In the below use case   1. From the chat model we will as a question, if the model is trained on OLD data set it will reply based on that training data   Once we add a recent knowledge to the model(by uploading the relevant document) it can able give response based on recent data set. |

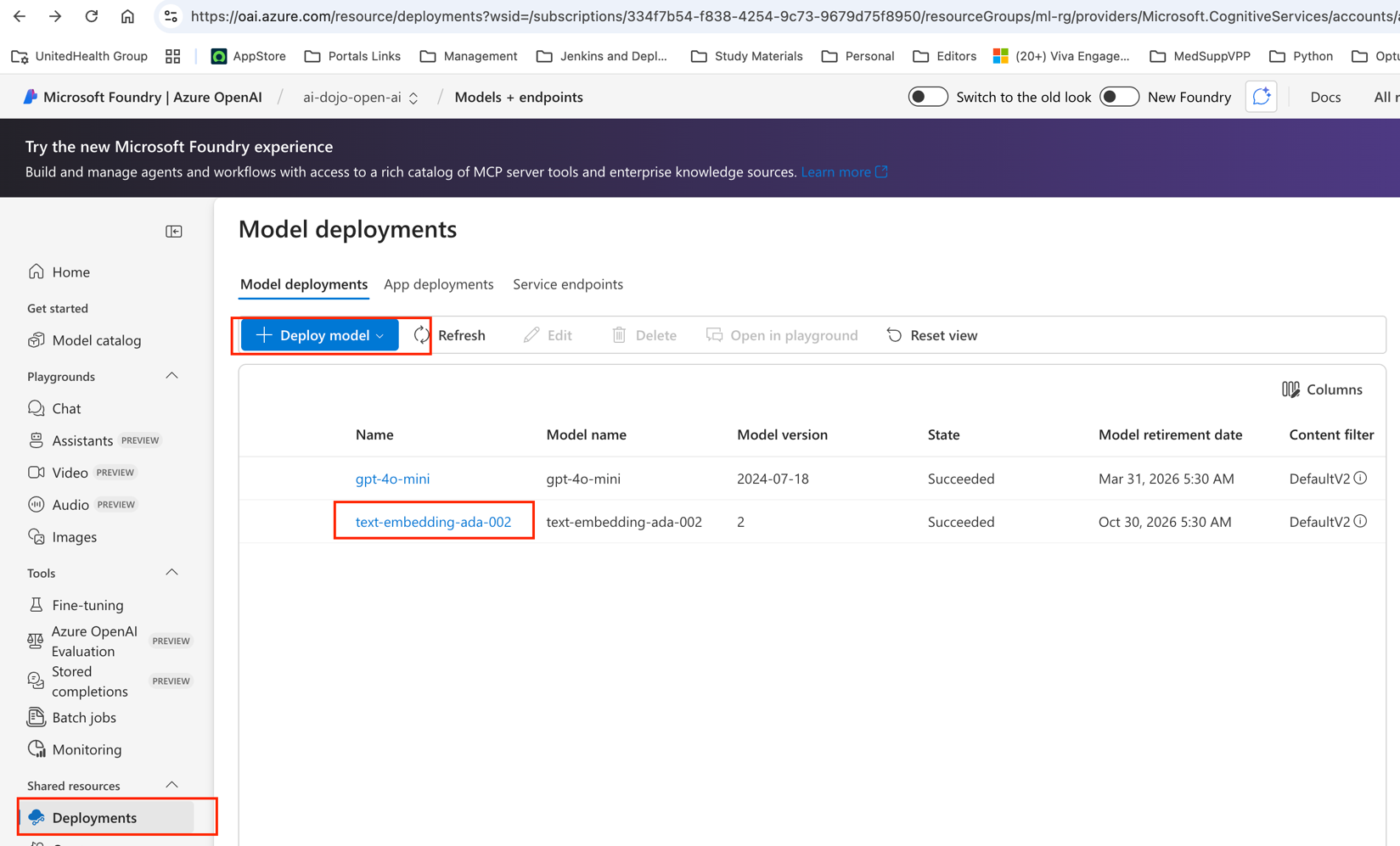
|  |  |
| --- | --- |
| RESOURCE TYPE | RESOURCE NAME |
| CREATE A STORAGE ACCOUNT | * Name - ragstgaisearch * Region – Central US |
| DEPLOY AZURE OPEN AI RESOURCE –  Create a model deployment model of below types | * Name- [ai-dojo-open-ai](https://portal.azure.com/#resource/subscriptions/334f7b54-f838-4254-9c73-9679d75f8950/resourceGroups/ml-rg/providers/Microsoft.CognitiveServices/accounts/ai-dojo-open-ai) * Region – East US |
| EMBEDDING MODEL | text-embedding-ada-002 |
| CHAT DEPLOYMENT | gpt-4o-mini |
| AI SEARCH | Name- ragwithazureaisearch  Region- Central US |

CREATING STORAGE ACCOUNT

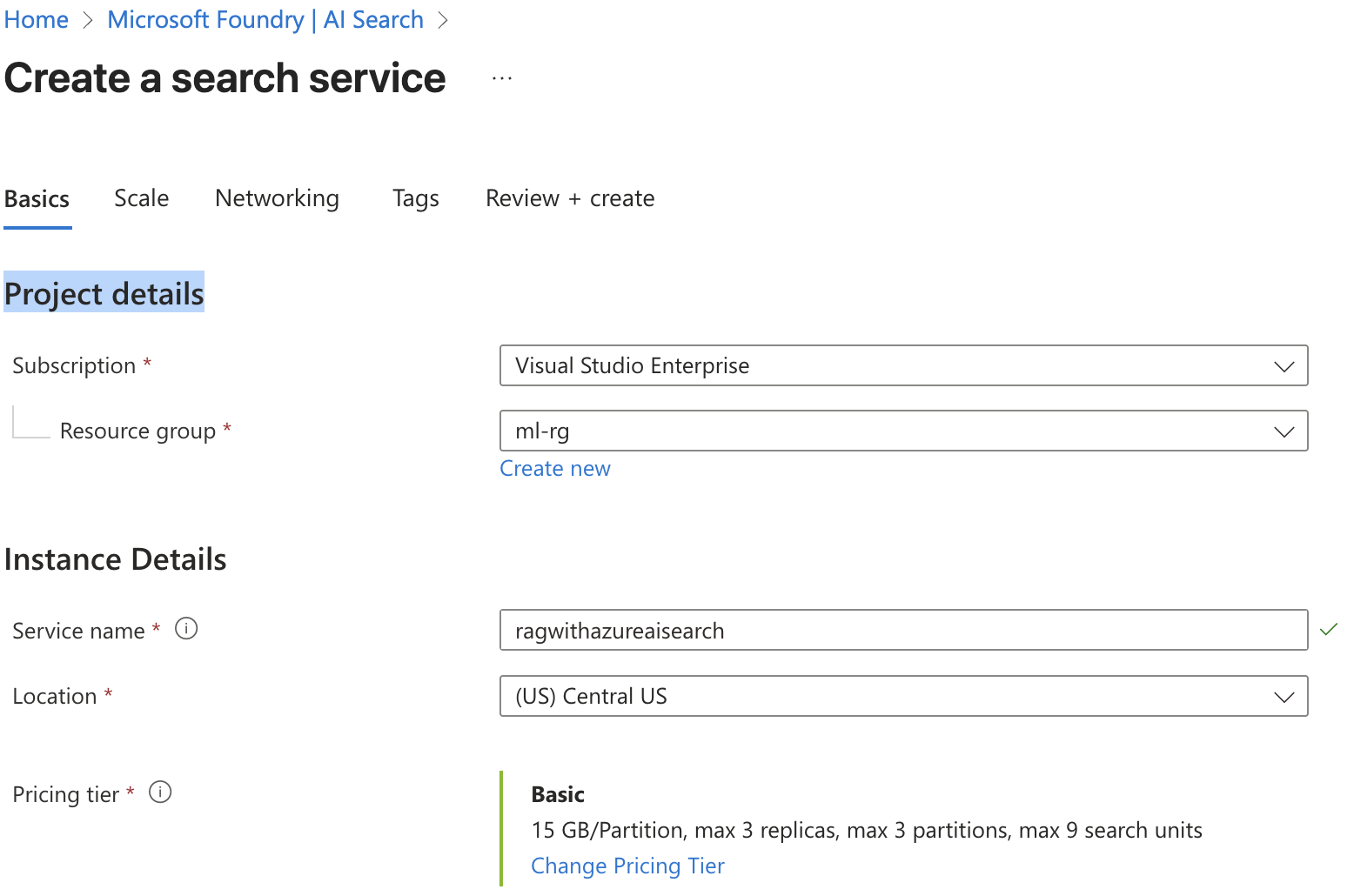
 CREATING AZURE OPEN AI RESOURCE



#### CREATE DEPLOYMENTS

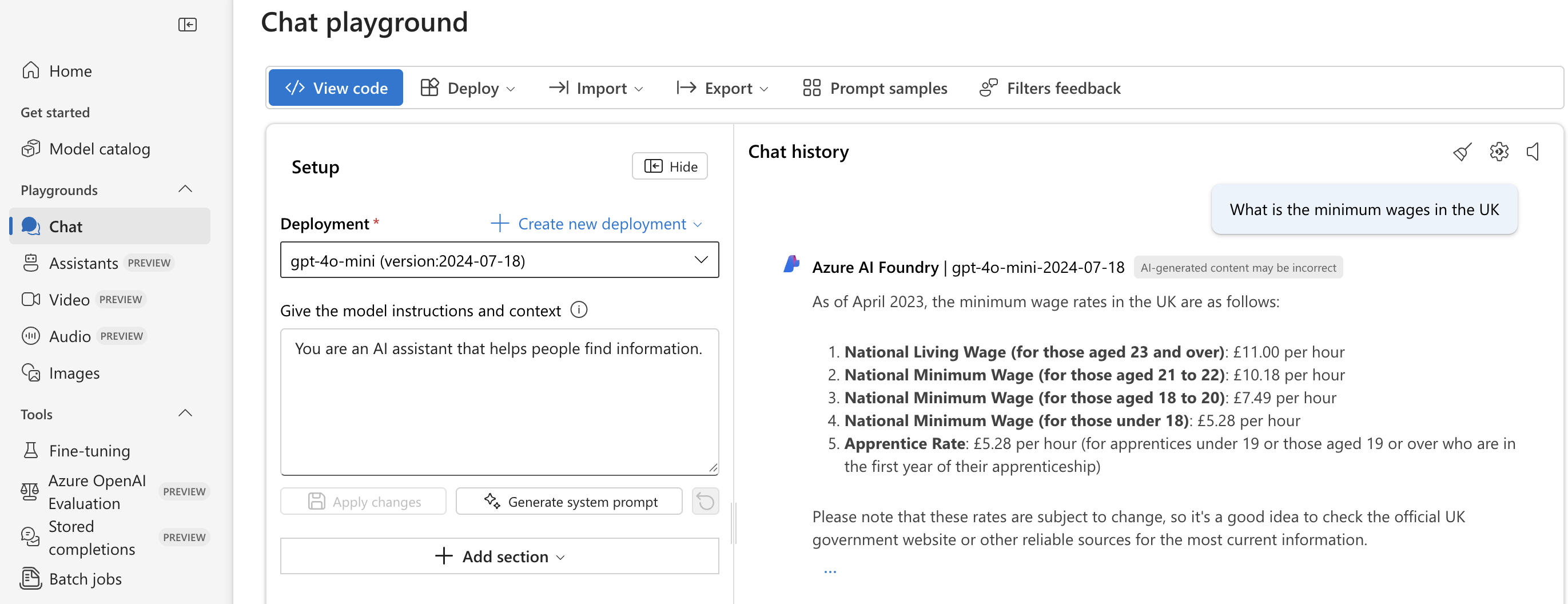


#### AZURE AI SEARCH RESOURCE CREATING



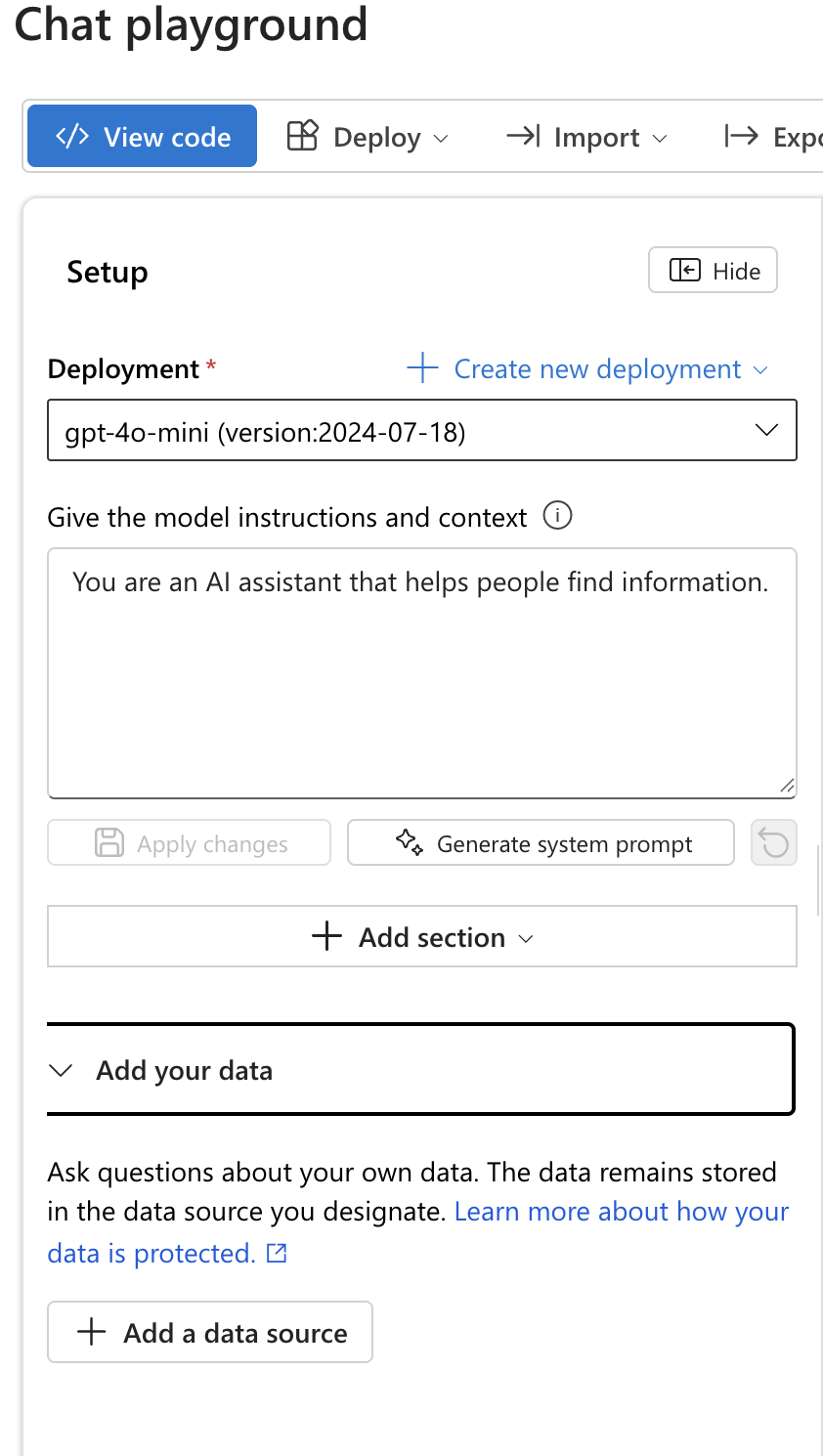
#### TESTING WITHOUT KNOWLEDGE

In the below use case – we are testing the chat model, without any knowledge source. Hence it is responding based in it old training data



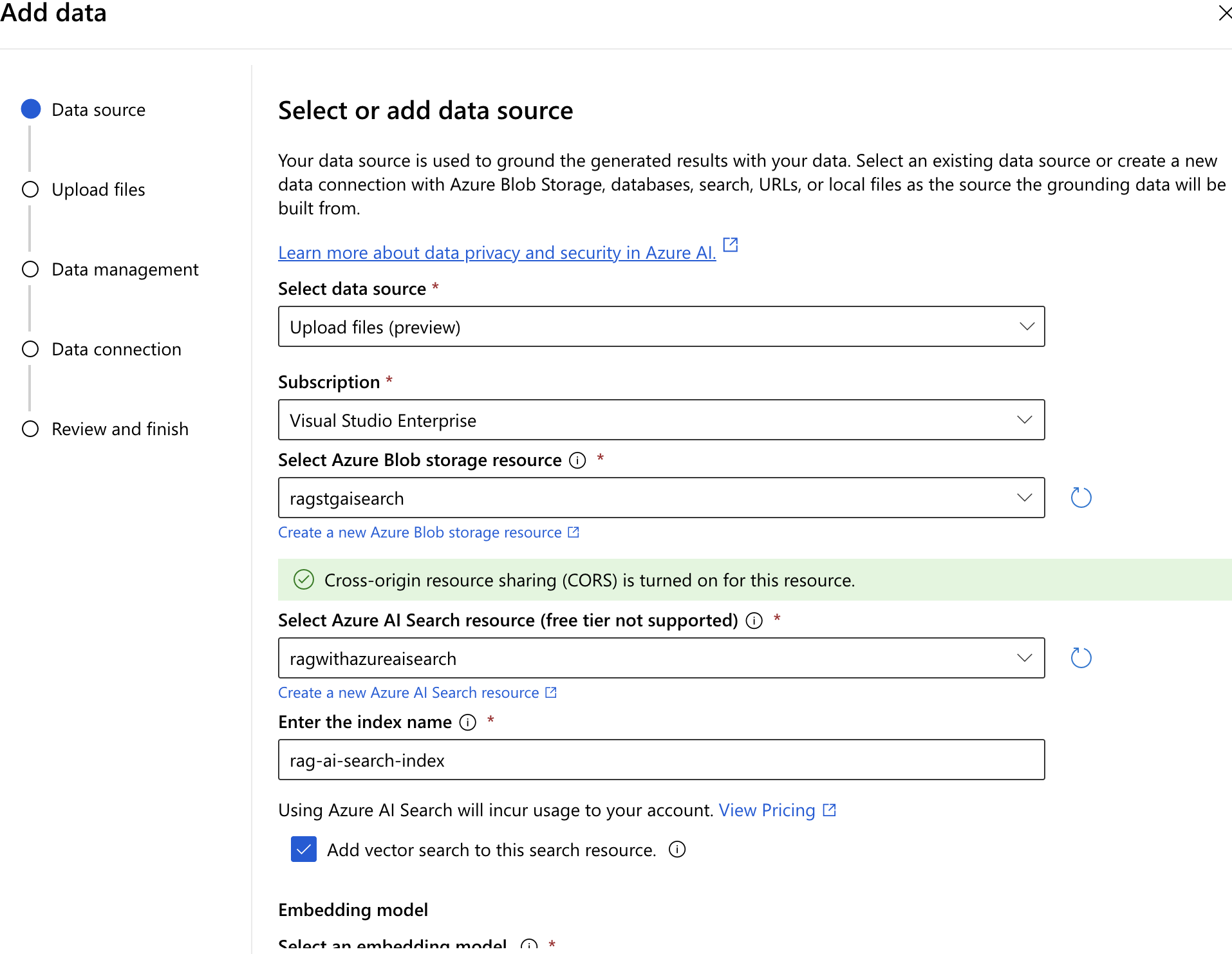
#### ADDING THE KNOWLEDGE

* From the chat playground select 🡪 Add your data



In the next step we need to map the following

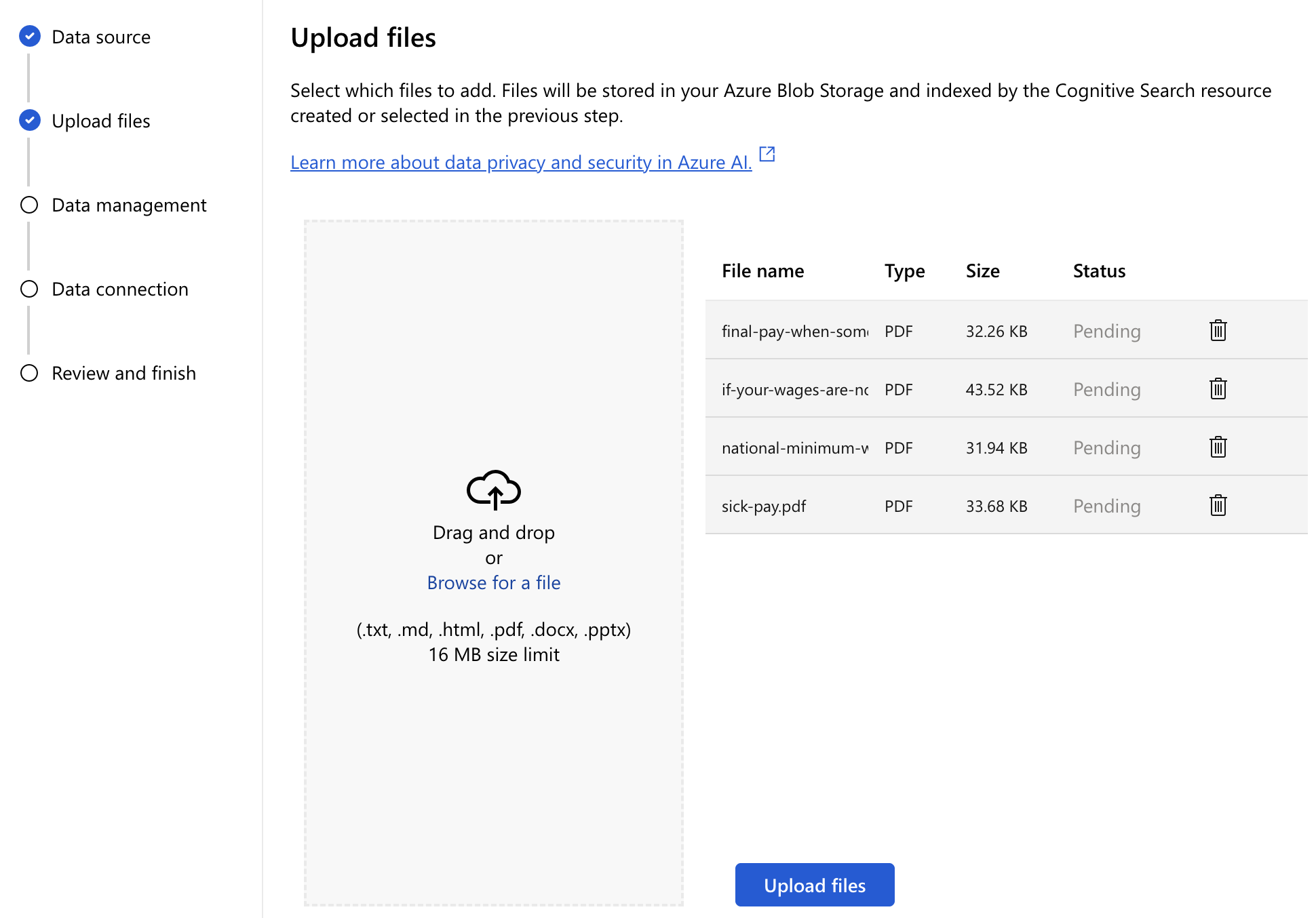
|  |  |
| --- | --- |
| STORAGE ACCOUNT | The document will get uploaded to the blob storage of the storage account |
| AI SEARCH RESOURCE | This is where reverse indexing and chunking of data will happen |
| EMBEDDING MODEL | The model responsible top create embedding of chunked data |



A white line with black lines

AI-generated content may be incorrect.

UPLOAD DOCUMENT(KNOWLEDGE)



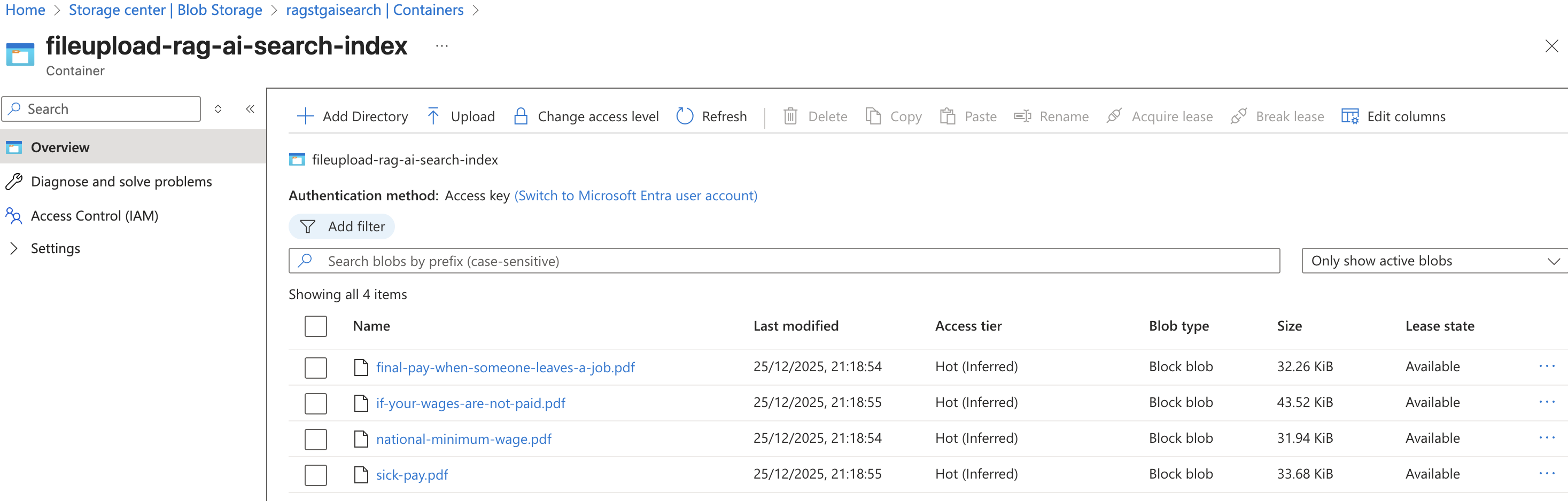
DATA CONNECTION

* This is how the azure resources connect to each other i.e, how AI search , Azure Open AI and Blob storage will be connected

A screenshot of a computer

AI-generated content may be incorrect.

This is here the file will be uploaded in Azure Storage Account

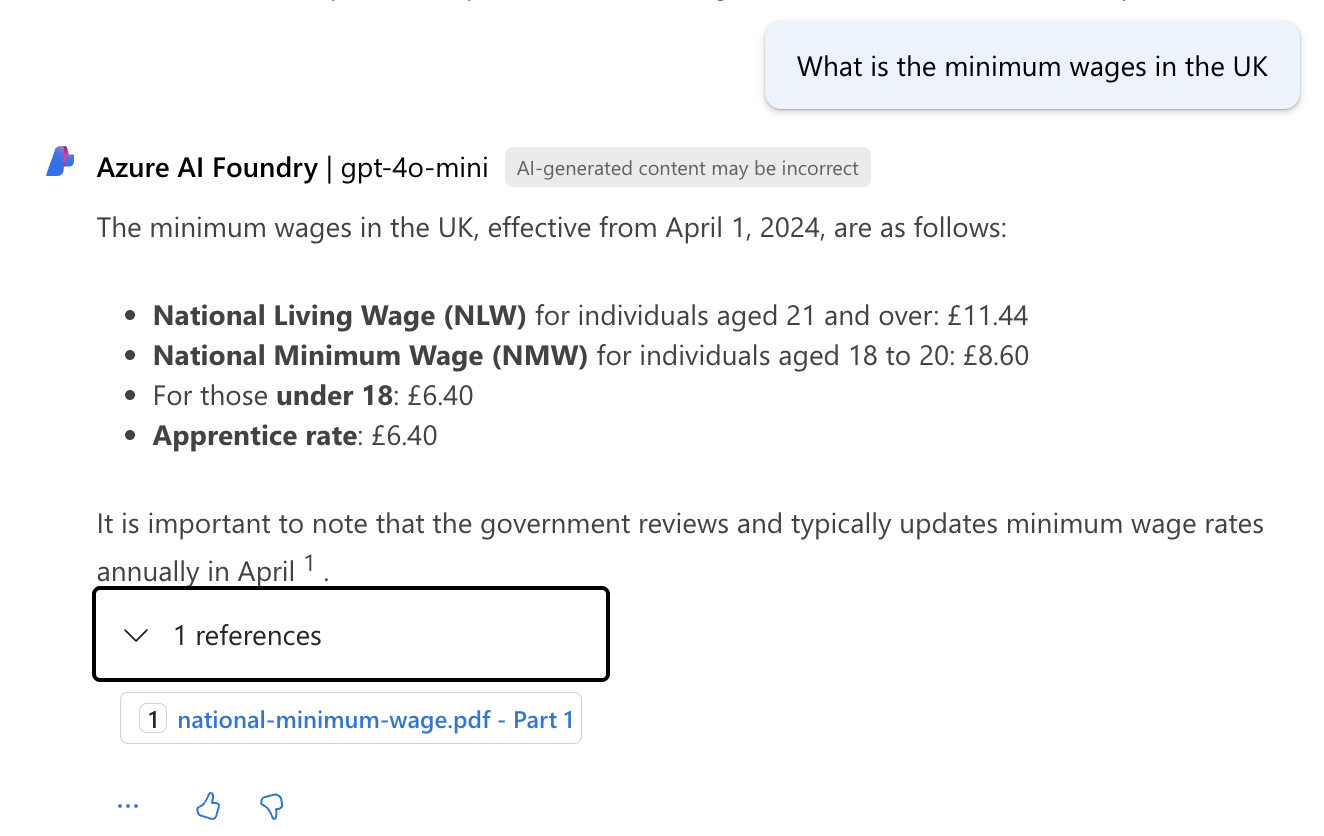


DATA INGESTION

* When we upload files in the Chat Playground, the system needs to make that data searchable and usable by AI models. Here’s what happens step by step:
* STEPS OF DATA INGESTION
* **Step 1: Upload the file -** We add PDFs, Word docs, or other files to the playground.
* **Step 2: Extract content -** The system reads the text from the files (including metadata like titles, authors).
* **Step 3: Create an index -** The extracted content is stored in a structured way (called an index) so it can be searched quickly.
* **Step 4: Enrich the data (optional) -** AI can add extra info like detecting language, extracting key phrases, or converting images to text (OCR).
* **Step 5: Ready for search and AI use -** Now, when we ask a question, the system can retrieve relevant parts of the uploaded files and combine them with the query for better answers.

|  |  |
| --- | --- |
|  |  |

* Once knowledge is added, the Chat can respond with up-to-date information and will include a reference to the source document used.



# BUILDING RAG WITH AZURE AI SEARCH, AZURE OPEN AI AND LANGCHAIN

A diagram of a cloud computing process

AI-generated content may be incorrect.

Summary of Components

1. **Azure Blob Storage:** Stores raw documents.
2. **Azure Search Indexer:** Prepares and indexes data.
3. **Azure AI Search:** Retrieves relevant chunks.
4. **LangChain:** Orchestrates retrieval and prompt building.
5. **Azure OpenAI:** Generates the final answer.

Step 1: User Interaction

* The user asks a question in your application (e.g., “What is the parental leave policy?”).
* This query starts the entire workflow.

Step 2: LangChain Orchestration  
LangChain acts as the **controller**.

* Receives the user query.
* Prepares to retrieve relevant data from Azure AI Search.
* Later, it will combine retrieved data with the query and send it to Azure OpenAI.

Step 3: Azure AI Search Retrieval

**Purpose:**Find the most relevant chunks of your private data.

**Process:**

* Azure AI Search looks into its **index** (created earlier from your documents).
* Performs **hybrid search**:
  + - Keyword search (using inverted index).
    - Vector search (using embeddings for semantic similarity).
* Returns **top N chunks** (e.g., paragraphs from HR policy PDFs).

Step 4: Data Source & Indexing (Behind the Scenes)

* **Before retrieval can happen:** The documents (PDFs, CSVs, etc.) are stored in **Azure Blob Storage**.
* **Azure Search Indexer** ingests these documents:
  + - Breaks them into **chunks**.
    - Applies **AI enrichment** (OCR, language detection, key phrase extraction).
    - Generates **embeddings** for each chunk.
    - Stores them in the **Azure AI Search index**.

Step 5: LangChain Builds the Prompt

LangChain takes:

* The user’s question.
* The retrieved chunks from Azure AI Search.
* Combines them into a **prompt** for the LLM:

A screenshot of a computer screen

AI-generated content may be incorrect.

Step 6: Azure OpenAI Generates the Answer

* Azure OpenAI (GPT model) processes the prompt.

**OUTPUT:**  
A natural language answer that is:

* Accurate.
* Contextual.
* Includes citations.

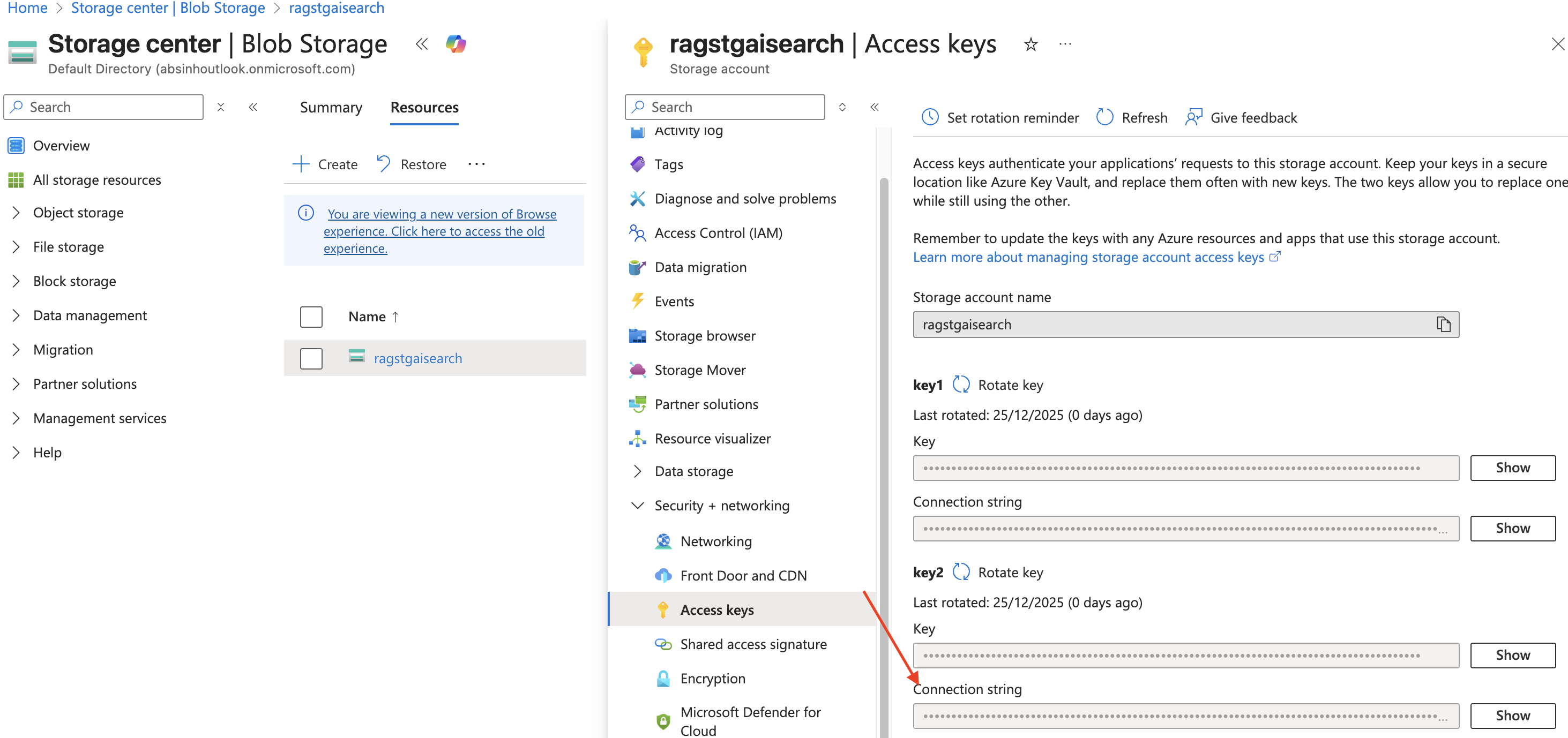
**Example Answer:**  
“Our 2026 policy offers 12 weeks paid parental leave for all employees [Source: HR Handbook 2026, Section 4.2].”

Step 7: Response to User

* **Final step:** LangChain sends the answer back to the application UI.
* **User sees:** A clear, grounded response with references.

## SETTING UP

|  |  |
| --- | --- |
| Setting up project | uv init |
| Create virtual env | uv venv |
| Install packages | * langchain-community * langchain-openapi * azure-data-tables * python-dotenv |
| * In this example – we will make use of csv file -<https://github.com/avishekhsinhaRepo/Docs/blob/master/Azure%20AI/RAG-Resources/bmw.csv> * This csv will be stored in table storage in Azure Blob . Hence “**azure-data-tables**” (package from Azure SDK)package will help in Reading/writing **structured data** in **Azure Table Storage** (rows with key‑value columns). | |



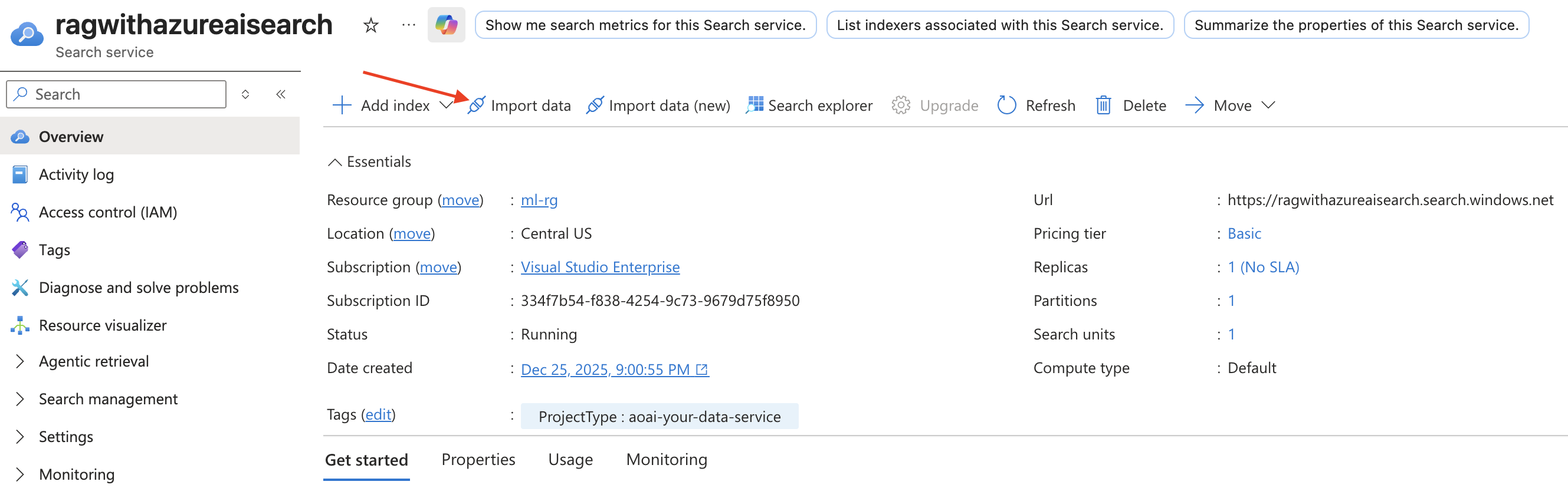
### STEP 1: CREATE AZURE TABLE BASED ON CSV FILE

|  |  |
| --- | --- |
| **TableServiceClient** | This is the **main entry point** for interacting with **Azure Table Storage**.  **What you can do with it**:   * Connect to **storage account** using a **connection string** or **SAS token**. * Create or delete tables. * Get a client for a specific table. |
| **TableEntity** | Represents a **single row (entity)** in an Azure Table.  **Structure**:  Every entity must have:   * PartitionKey (logical grouping for scalability) * RowKey (unique identifier within the partition) * Any custom properties (columns). |

|  |
| --- |
| import os  import csv  from azure.data.tables import TableServiceClient,TableEntity  from dotenv import load\_dotenv  load\_dotenv()  connection\_string = os.getenv("AZURE\_STORAGE\_CONNECTION\_STRING")  table\_name = os.getenv("AZURE\_TABLE\_NAME")  table\_service = TableServiceClient.from\_connection\_string(conn\_str=connection\_string)  #create table if not exists  try:  table\_service.create\_table(table\_name=table\_name)  except Exception as e:  print("Table already exists")  #Read the csv and insert into table  with open('resources/bmw.csv', mode ='r')as file:  csvFile = csv.DictReader(file)  for row in csvFile:  entity = TableEntity()  entity['PartitionKey'] = "BMW\_FAQ"  entity['RowKey'] = row['ID']  entity['Question'] = row['Question']  entity['Answer'] = row['Answer']  table\_service.get\_table\_client(table\_name=table\_name).create\_entity(entity=entity)  print("Data inserted successfully") |
|  |
|  |

### STEP 2: CREATE AZURE AI SEARCH SERVICE AND INDEXER

* Create Azure AI Search Service if not created and then click on import data



### STEP 3: CREATE AI SEARCH INDEXER

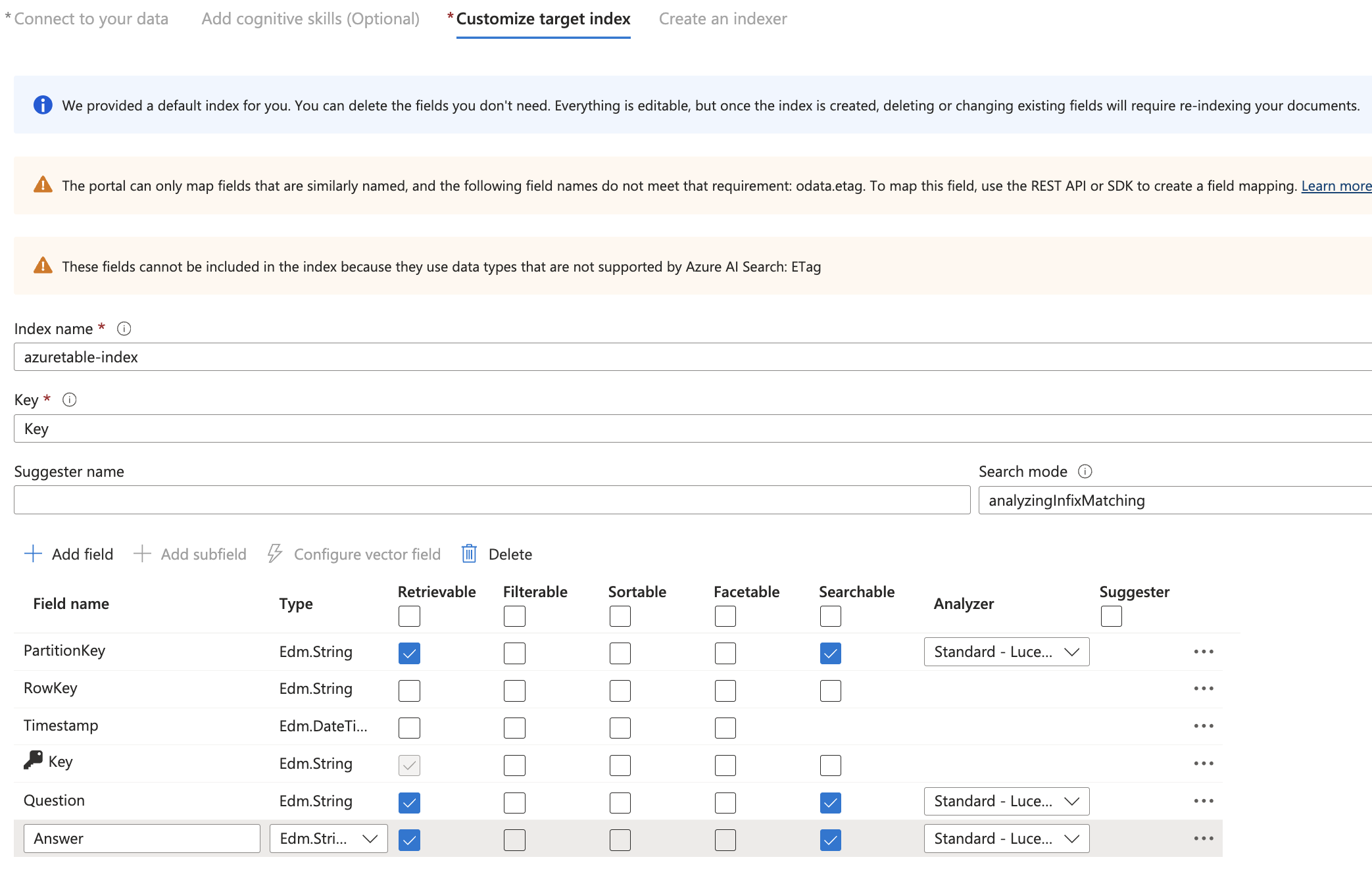
|  |
| --- |
| **WHAT IS AI SEARCH INDEXER?**   * An **indexer** in Azure AI Search is like a **data pipeline** that automatically **pulls data from your source** (e.g., Azure Blob Storage, Azure SQL Database, Cosmos DB) and **loads it into your search index** so it can be searched efficiently. * Think of it as a **robot that reads your documents, processes them, and organizes them for fast searching**.   **WHAT DOES IT DO?**   1. **Connects to your data source** (e.g., Blob Storage). 2. **Reads the content** (including OCR for scanned images). 3. **Breaks large documents into chunks** for better search. 4. **Extracts metadata** (like file name, date). 5. **Applies AI enrichment** (language detection, key phrase extraction, embeddings for vector search). 6. **Stores everything in the Azure AI Search index**.   **EXAMPLE**  Imagine you have HR policy PDFs in Azure Blob Storage:   * **Without indexer**: You’d have to manually upload and process each file. * **With indexer**:   + Reads all PDFs from Blob.   + Extracts text and metadata.   + Creates **chunks** and embeddings.   + Updates the **search index**.   + Now, when you ask: “What is parental leave policy?”, Azure AI Search can quickly find the right section.   **REAL-LIFE ANALOGY**   * Think of the indexer as a **librarian**: * Your Blob Storage is a pile of books. * The librarian (indexer) reads each book, summarizes key points, and organizes them in a **catalog (index)**. * When someone asks a question, the catalog helps find the right book section instantly. |

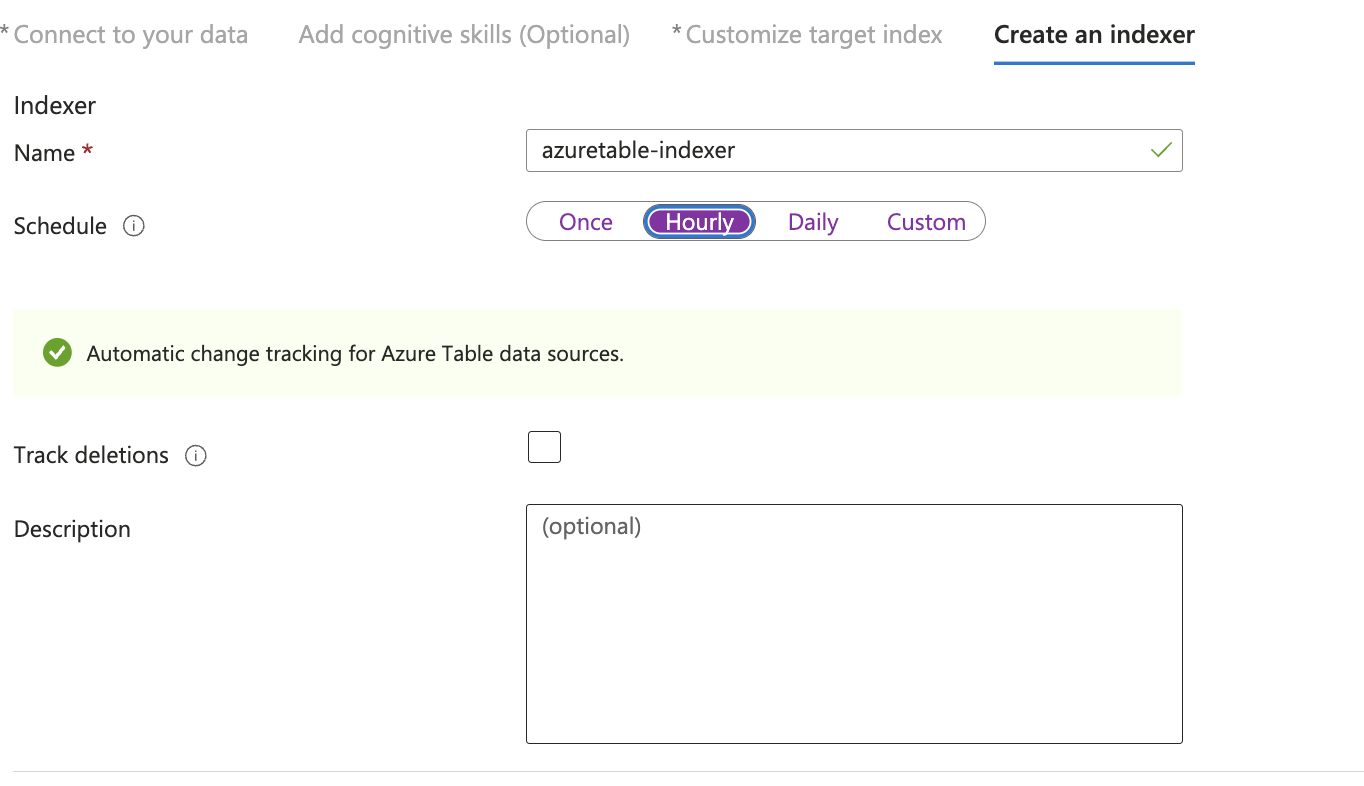
* This step uses Azure table storage to read PDFs, extract text and metadata, create **chunks** and embeddings, and update the **search index**.



We will select which are fields which are searchable and retrievable

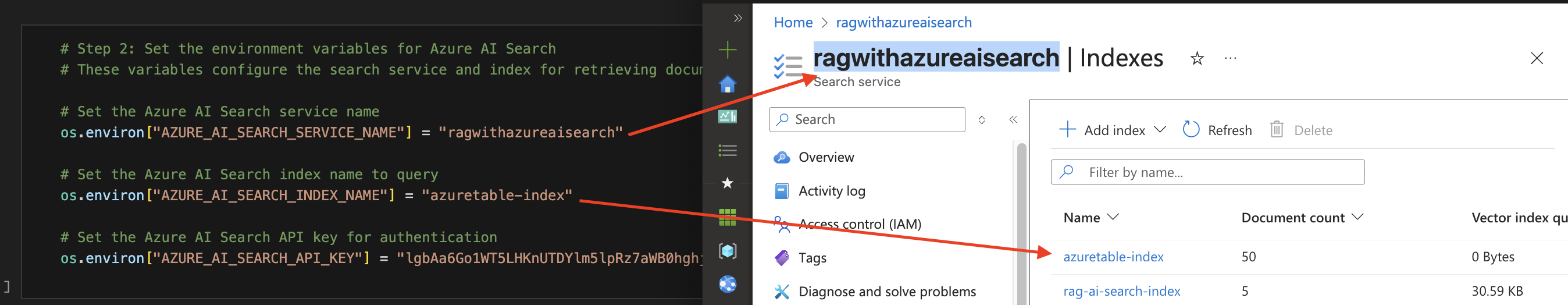
|  |
| --- |
| **WHAT IS A TARGET INDEX?**  The **target index** is the **search index** where the processed data will be stored.  This index contains fields like content, metadata, embedding, etc., which are used for keyword and vector search.  **What does “Custom Target Index” mean?**  Instead of letting Azure automatically create a default index for you, you **define your own index schema** (custom fields, data types, semantic settings).  This is useful when:  You need **specific fields** (e.g., department, effective\_date, source\_url).  You want to enable **vector search** (requires a vector field).  You want to configure **semantic ranking** or **filters**.  ✅ **Example**  Imagine you have HR policy PDFs in Blob Storage.\ You create a **custom target index** with fields like:  {    "name": "hr-policies-index",    "fields": [      {"name": "id", "type": "Edm.String", "key": true},      {"name": "title", "type": "Edm.String", "searchable": true},      {"name": "content", "type": "Edm.String", "searchable": true},      {"name": "department", "type": "Edm.String", "filterable": true},      {"name": "effective\_date", "type": "Edm.DateTimeOffset", "filterable": true, "sortable": true},      {"name": "embedding", "type": "Collection(Edm.Single)", "vectorSearchDimensions": 1536}    ]  }  When you configure the indexer, you select this **custom index** as the **target index**.  The indexer will map data from Blob → these fields → store in the index |





A screenshot of a computer

AI-generated content may be incorrect.





### INITIALIZE THE RETRIEVER, PROMPT, AND LLM

|  |
| --- |
| # Import necessary libraries and modules from Langchain  from langchain\_core.output\_parsers import StrOutputParser  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.runnables import RunnablePassthrough  from langchain\_openai import AzureChatOpenAI  from langchain\_community.retrievers import AzureAISearchRetriever  # Step 1: Initialize the AzureAI Search Retriever  # This retrieves relevant documents based on the user query from the Azure Search index  retriever = AzureAISearchRetriever(  content\_key="Answer", # The key for the content field in the search results change it accordingly as per your data  top\_k=1, # Number of top results to retrieve  index\_name="azuretable-index" # Name of the Azure Search index to query  )  # Step 2: Define the prompt template for the language model  # This sets up how the context and question will be formatted for the model  prompt = ChatPromptTemplate.from\_template(  """Answer the question based only on the context provided.  Context: {context} # Placeholder for the context from the retriever  Question: {question} # Placeholder for the user question"""  )  # Step 3: Initialize the Azure Chat OpenAI model  # This sets up the model to be used for generating responses  llm = AzureChatOpenAI(  azure\_endpoint=os.getenv("AZURE\_OPENAI\_ENDPOINT"), # Fetch the Azure OpenAI endpoint from environment variables  api\_key=os.getenv("AZURE\_OPENAI\_API\_KEY"), # Fetch the API key for Azure OpenAI from environment variables  api\_version=os.getenv("AZURE\_OPENAI\_API\_VERSION"), # Specify the API version to use  model="gpt-4o-mini" # Specify the model to use  ) |

### PROCESSING CHAIN AND USER INPUT LOOP

|  |
| --- |
| # Step 4: Create a processing chain  # This chain will process the retrieved context and the user question  chain = (  {"context": retriever , "question": RunnablePassthrough()} # Set context using the retriever and format it  | prompt # Pass the formatted context and question to the prompt  | llm # Generate a response using the language model  | StrOutputParser() # Parse the output to a string format  )  # Step 5: Infinite loop for user input  while True:  # Prompt the user to enter a question or type 'end' to exit  user\_question = input("Please enter your question (or type 'end' to exit): ")  # Check if the user wants to exit the loop  if user\_question.lower() == "end":  print("Exiting the loop. Goodbye!") # Inform the user that the loop is ending  break # Break the loop and exit  # Step 7: Invoke the processing chain with the user's question  response = chain.invoke(user\_question) # Get the response from the chain based on the user question  print("Response:", response) # Print the model's response to the console |