Lab 6: Hybrid Search with Azure AI Search

Lab 5 covered how to do vector search, using Azure Cosmos DB for MongoDB vCore. We will expand the environment to do hybrid search with Azure AI Search.

In this exercise, the Azure AI Search component has already been created.

Each team will create its own index within the context of this Azure AI search account

Table of Contents

Pre-requisites

Ensure that you have the following software installed on your system before proceeding with the lab:

- Visual Studio Code: A cross-platform code editor that supports Python development. You can download it from https://code.visualstudio.com/

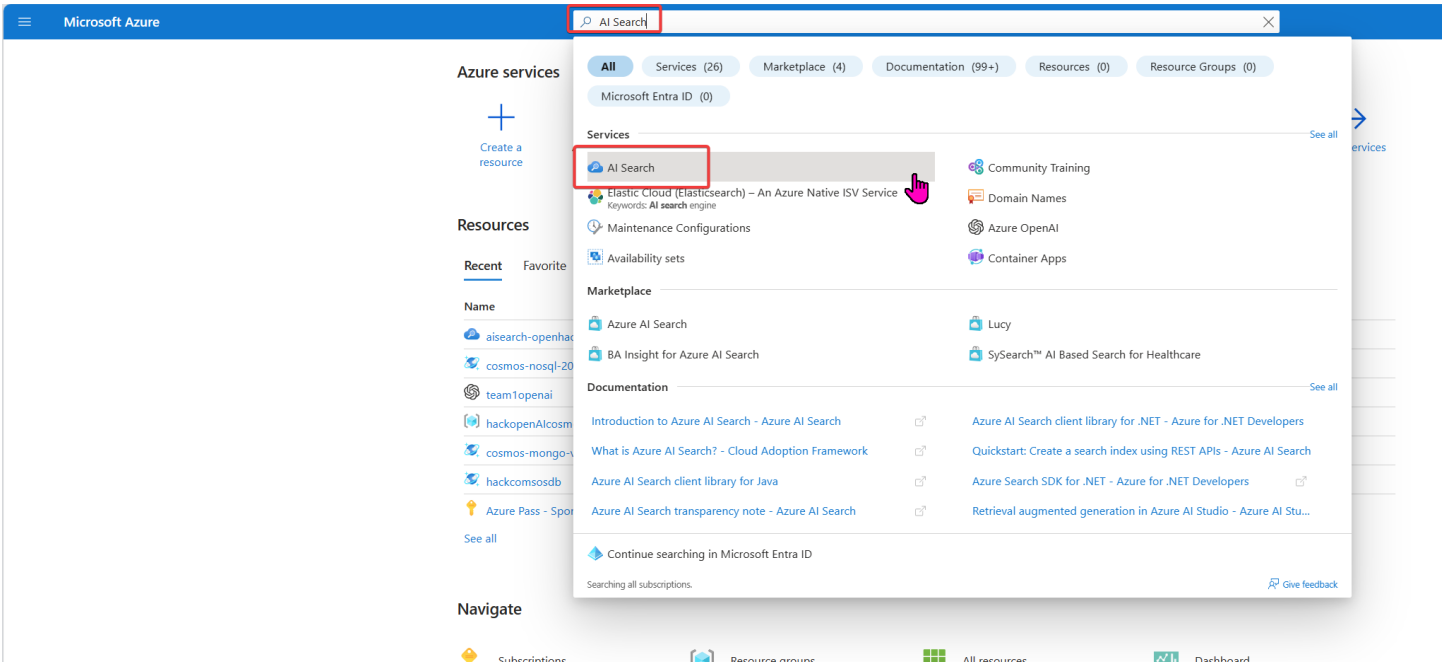
- Python 3.10.11: The latest version of the Python programming language. You can download it from https://www.python.org/downloads/release/python-31011/

Note: If you are using a different version of Python, make sure that it is compatible with the libraries and packages used in this lab.

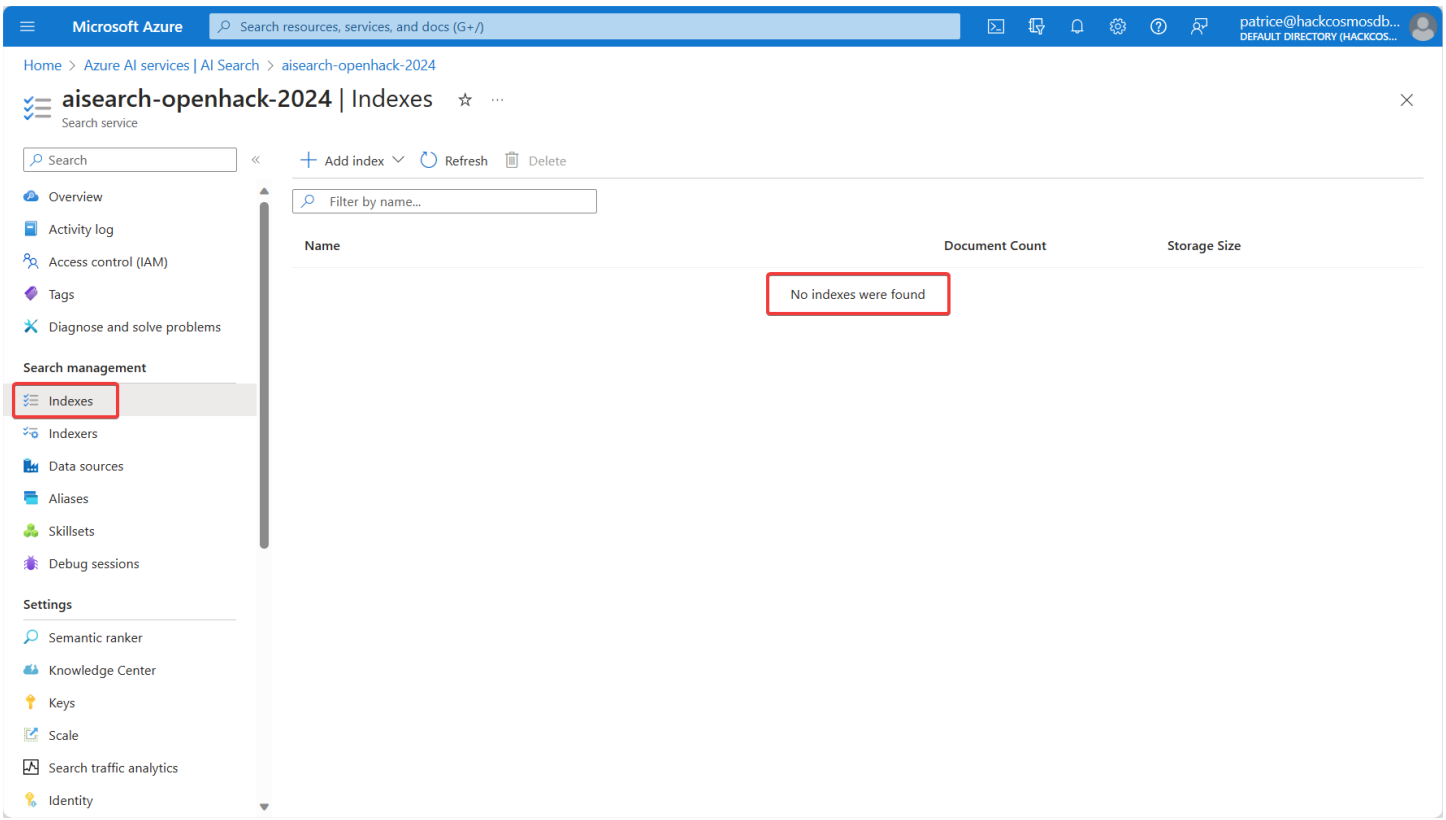
- Azure OpenAI account registered in the Azure subscription used for this lab
- Azure AI Search account registered in the Azure subscription

Connect to an existing Azure AI Search account

- In the Azure Portal, type "**`AI search`**" in the search bar at the top of the screen
- Select AI search in the list



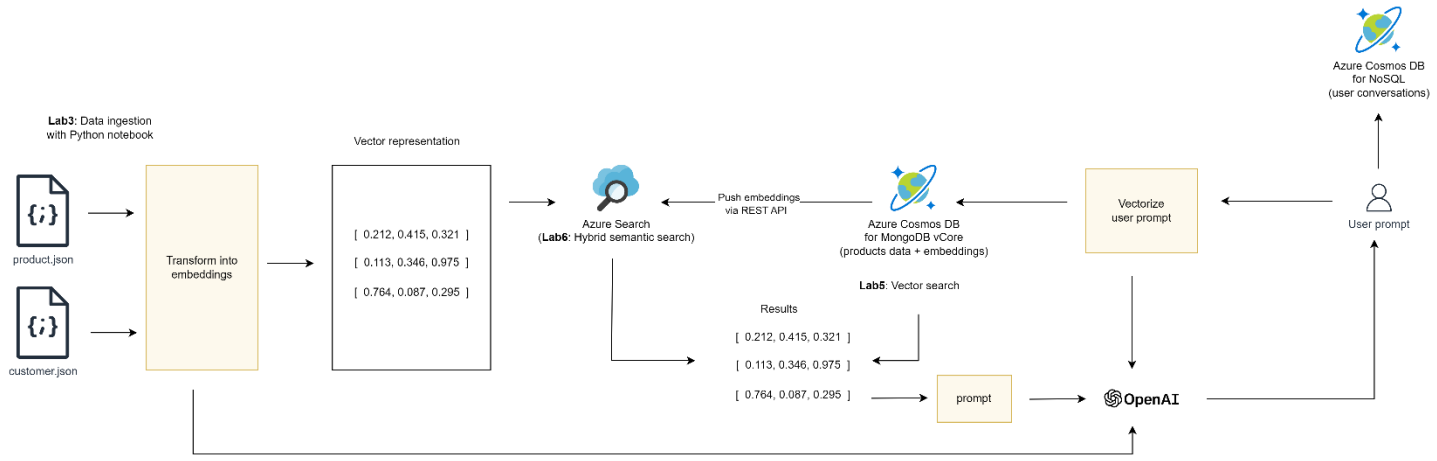
- Select **`aisearch-my-search`**
- In the left menu, select "Indexes"
- There are currently no indexes in the Azure AI search account



Install pre-requisites

In this chapter, we will create a Jupyter notebook and use product data and embeddings stored in our Azure Cosmos DB for MongoDB vCore "products" collection to populate an index in Azure AI search

As a reminder, here is an overview of the architecture:



Azure Cosmos DB for MongoDB vCore	Storage engine for data and product embeddings
Azure AI Search	Compute engine for product embeddings
Azure Cosmos DB for NoSQL	Storage for user conversations

We already have our product embeddings stored into Azure Cosmos DB for MongoDB vCore. In a Jupyter notebook, we will use the Azure Search REST API to push data into our AI Search account.

1. Create a new folder "`lab6`"
2. Open Visual Studio Code
3. Copy `.env` and `requirements.txt` from the lab3 folder to the lab6 folder
4. Open a new Powershell Teminal > New Terminal
5. Type this command to create a virtual environment: `python -m venv .venv`
6. Activate the virtual environment with `.venv\scripts\activate`
7. Install the required libraries with `pip install -r requirements.txt`

Note: we are recreating a new virtual environment for this lab. We could have created a common environment for all labs and shared .env and requirements.txt between all labs.

Create Azure AI Search index

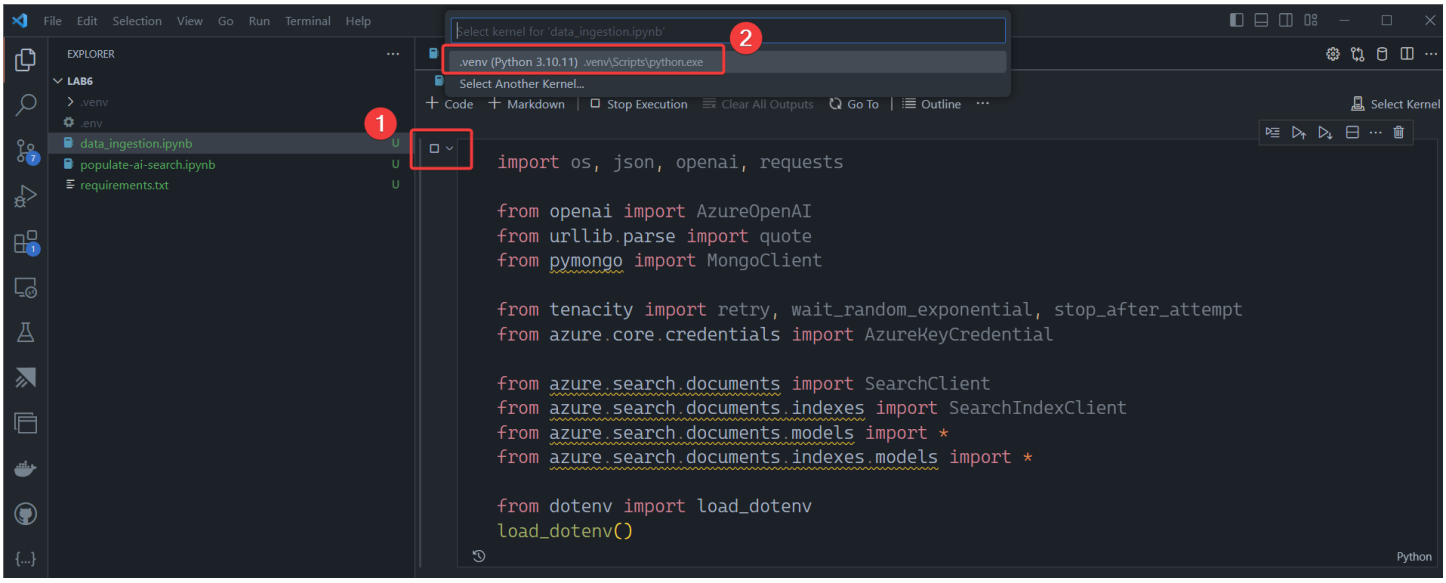
In visual Studio Code, create a new file called "**`data_ingestion.ipynb`**"

In the first cell, copy/paste this code to import the required libraries

```
import os, json, openai, requests

from openai import AzureOpenAI
from urllib.parse import quote
from pymongo import MongoClient

from tenacity import retry, wait_random_exponential, stop_after_attempt
from azure.core.credentials import AzureKeyCredential

from azure.search.documents import SearchClient
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.models import *
from azure.search.documents.indexes.models import *

from dotenv import load_dotenv
load_dotenv()
```

Press on the "play" button to execute the content of the cell

Select a virtual environment when prompted



Under the first cell, click on the "+ Code" button to add a new code cell

Copy/paste this code to set environment variables

```
openai.api_type = "azure"
openai.api_key = os.getenv("AZURE_OPENAI_API_KEY")
openai.api_version = os.getenv("AZURE_OPENAI_API_VERSION")
openai.api_base = os.getenv("AZURE_OPENAI_ENDPOINT")

AZURE_SEARCH_INDEX = os.getenv("COSMOSDB_MONGODB_PRODUCTS") + "_index"
AZURE_SEARCH_KEY = os.getenv("AZURE_SEARCH_KEY")
AZURE_SEARCH_SERVICE = os.getenv("AZURE_SEARCH_SERVICE")
AZURE_SEARCH_API_VERSION = os.getenv("AZURE_SEARCH_API_VERSION")

AZURE_SEARCH_SERVICE
```

Run the cell, it should display the name of the Azure AI search account

Add a new cell, copy/paste this code and execute the cell to create the Azure AI Search index, using the SDK

```
credential = AzureKeyCredential(AZURE_SEARCH_KEY)

print(f"Creating index {AZURE_SEARCH_INDEX}")

# Create a search index
index_client = SearchIndexClient(endpoint=AZURE_SEARCH_SERVICE, credential=credential)
fields = [
    SimpleField(name="id", type=SearchFieldDataType.String, key=True),
    SearchableField(name="categoryId", type=SearchFieldDataType.String),
    SearchableField(name="categoryName", type=SearchFieldDataType.String),
    SearchableField(name="sku", type=SearchFieldDataType.String),
    SearchableField(name="name", type=SearchFieldDataType.String),
    SearchableField(name="description", type=SearchFieldDataType.String),
    SimpleField(name="price", type=SearchFieldDataType.Double, filterable=True),
    SearchableField(name="tags", type=SearchFieldDataType.String),
    SearchField(name="vectorContent",
type=SearchFieldDataType.Collection(SearchFieldDataType.Single), searchable=True,
vector_search_dimensions=1536, vector_search_profile_name="myHnswProfile"),
]

# Configure the vector search configuration
```

```
vector_search = VectorSearch(
    algorithms=[
        HnswAlgorithmConfiguration(
            name="myHnsw",
            kind=VectorSearchAlgorithmKind.HNSW,
            parameters=HnswParameters(
                m=4,
                ef_construction=400,
                ef_search=500,
                metric=VectorSearchAlgorithmMetric.COSINE
            )
        ),
        ExhaustiveKnnAlgorithmConfiguration(
            name="myExhaustiveKnn",
            kind=VectorSearchAlgorithmKind.EXHAUSTIVE_KNN,
            parameters=ExhaustiveKnnParameters(
                metric=VectorSearchAlgorithmMetric.COSINE
            )
        )
    ],
    profiles=[
        VectorSearchProfile(
            name="myHnswProfile",
            algorithm_configuration_name="myHnsw",
        ),
        VectorSearchProfile(
            name="myExhaustiveKnnProfile",
            algorithm_configuration_name="myExhaustiveKnn",
        )
    ]
)

semantic_config = SemanticConfiguration(
    name="my-semantic-config",
    prioritized_fields=SemanticPrioritizedFields(
        title_field=SemanticField(field_name="id"),
```

```
        content_fields=[
            SemanticField(field_name="categoryId"),
            SemanticField(field_name="categoryName"),
            SemanticField(field_name="sku"),
            SemanticField(field_name="name"),
            SemanticField(field_name="description"),
            SemanticField(field_name="tags"),
        ]
    )
)

# Create the semantic settings with the configuration
semantic_search = SemanticSearch(configurations=[semantic_config])

# Create the search index with the semantic settings
index = SearchIndex(name=AZURE_SEARCH_INDEX, fields=fields,
                    vector_search=vector_search, semantic_search=semantic_search)
result = index_client.create_or_update_index(index)
print(f' {result.name} created')
```

It should display the following message:

```
...  Creating index products_team01_index
      products_team01_index created
```

Populate index from Azure Cosmos DB for MongoDB vCore collection

Initialize products collection

Add a new cell

Copy/paste this code to initialize the products collection

```
def init_collection():
    host = os.getenv('COSMOSDB_MONGODB_HOST')
    username = os.getenv('COSMOSDB_MONGODB_USERNAME')
    password = os.getenv('COSMOSDB_MONGODB_PASSWORD')
    database_name = os.getenv('COSMOSDB_MONGODB_DATABASE')
    products_collection_name = os.getenv('COSMOSDB_MONGODB_PRODUCTS')

    # Encode the password
    encoded_password = quote(password, safe='')

    connection_string =
f'mongodb+srv://{username}:{encoded_password}@{host}/?tls=true&authMechanism=SCRAM-SHA-
256&retrywrites=false&maxIdleTimeMS=120000'

    client = MongoClient(connection_string)

    database = client[database_name]
    products_collection = database[products_collection_name]

    return products_collection

# get products collection
products_collection = init_collection()
products_collection
```

Run the cell, you should get something similar to this:

```
Collection(Database(MongoClient(host=['c.cosmos-mongo-vcore-
2024.mongocluster.cosmos.azure.com:10260'], document_class=dict, tz_aware=False, connect=True,
```

```
authmechanism='SCRAM-SHA-256', retrywrites=False, maxidletimems=120000, ssl=True),
'database_team01'), 'products_team01')
```

Get products from Azure Cosmos DB for MongoDB vCore

Add a new cell and copy/paste this code to retrieve all products

```
def get_products(collection):
    products = []
    for product in collection.find():

        # serialized_product = { **product, "@search.action": "upload", "_id": str(product["_id"])}
        serialized_product = {
            "@search.action": "upload",
            "id": product["id"],
            "categoryId": product["categoryId"],
            "categoryName": product["categoryName"],
            "sku": product["sku"],
            "name": product["name"],
            "description": product["description"],
            "price": product["price"],
            "tags": json.dumps(product["tags"]),
            "vectorContent": product["vectorContent"]
        }
        products.append(serialized_product)
    return products

# retrieve products from Azure Cosmos DB for MongoDB vCore
products = get_products(products_collection)
products
```

Running the cell should output the first lines of the json result

```
[{'@search.action': 'upload',
  'id': '0A7E57DA-C73F-467F-954F-17B7AFD6227E',
  'categoryId': '4F34E180-384D-42FC-AC10-FEC30227577F',
  'categoryName': 'Components, Pedals',
  'sku': 'PD-R563',
  'name': 'ML Road Pedal',
  'description': 'The product called "ML Road Pedal"',
  'price': 62.09,
  'tags': '[{"id": "14CFF1D6-7749-4A57-85B3-783F47731F32", "name": "Tag-7"}, {"id": "319E277F-6B7A-483D-81B/
  'vectorContent': [0.02730783075094223,
   0.016781123355031013,
   -0.025580165907740593,
   -0.008691899478435516,
   -0.031097980216145515,
   0.013031152077019215,
   -0.006612673401832581,
   -0.002571409335359931,
   0.02241947501897812,
   -0.032892610877752304,
   -0.0045100105926394 46,
   0.013031152077019215,
   -0.026839084923267365,
   -0.006957537028938532,
   -0.018428433686494827,
   0.03516937792301178,
  ...
   0.006586225703358 65,
   0.00037235577474348247,
   -0.005593277048319578,
   -0.014202175661921501,
   ...]}]
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

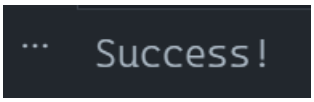
Populate the Azure AI Search index

Add a new cell and copy/paste this code to populate the Azure AI Search index

```
def populate_ai_index(products):
    # set Azure AI search header
    headers = {'Content-Type': 'application/json','api-key': AZURE_SEARCH_KEY}

    data = {"value": products}

    response = requests.post(
        f"{AZURE_SEARCH_SERVICE}/indexes/{AZURE_SEARCH_INDEX}/docs/index?api-
version={AZURE_SEARCH_API_VERSION}",
        data=json.dumps(data),
        headers=headers
    )

    if response.status_code != 200:
        print(f"Error: {response.text}")
```

```
    else:
        print("Success!")

# Populate the search index with products dat
populate_ai_index(products)
```

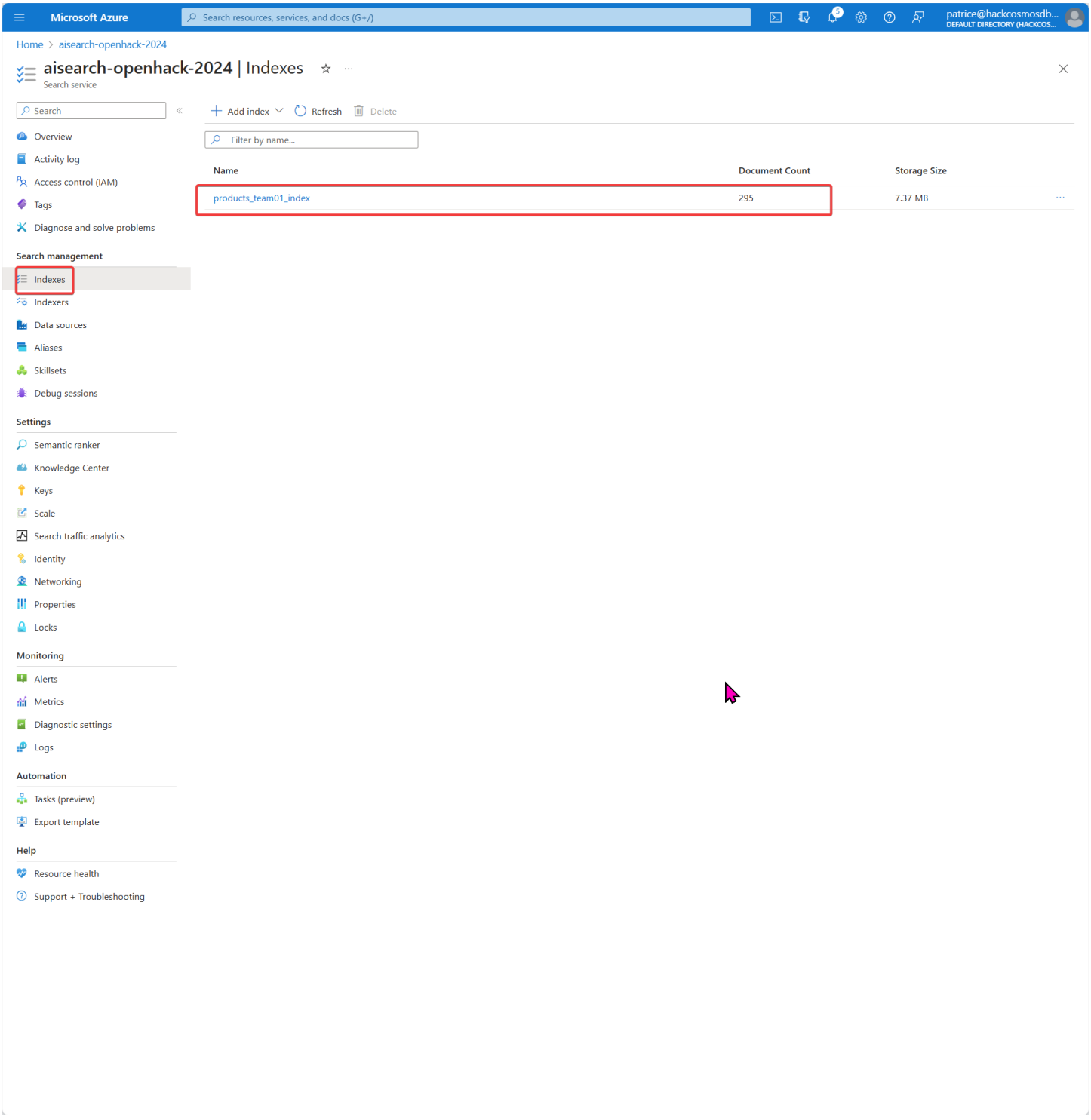
Running the cell should return a success message.



Verify Azure AI Search index

Wait a few minutes, and go to the Azure Portal to check if the index was properly populated

- Open Azure Portal
- Open the aisearch-openhack-2024 index
- Click on “Indexes” in the left navgiation menu
- Your index should show with 295 documents



Hybrid search with Jupyter notebook

In Visual Studio Code, open the file **`data_ingestion.ipynb`**

Add a new cell at the end and copy/paste this code to generate embeddings from a piece of text

```
@retry(wait=wait_random_exponential(min=1, max=20), stop=stop_after_attempt(10))
def generate_embeddings(openai_client, text):
    """
    Generates embeddings for a given text using the OpenAI API v1.x
    """
    response = openai_client.embeddings.create(
        input = text,
        model= os.getenv("AZURE_OPENAI_EMBEDDING_MODEL")
    )

    embeddings = response.data[0].embedding
    return embeddings
```

Run the cell

Add a new cell and copy/paste this code that runs a simple query (with a context)

```
def run_query(context, query):

    openai_client = AzureOpenAI(
        api_key = os.getenv("AZURE_OPENAI_API_KEY"),
        api_version = os.getenv("AZURE_OPENAI_API_VERSION"),
        azure_endpoint =os.getenv("AZURE_OPENAI_ENDPOINT")
        )

    prompt = f"""You are an AI chatbot having a conversation with a human.

    Context:
    {context}

    Human: {query}
    AI: """

    response = openai_client.chat.completions.create(
```

```
        model = os.getenv("AZURE_OPENAI_CHAT_MODEL"),
        messages = [{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content
```

Run the cell

Now, add a new cell and copy/paste this code that runs a hybrid search on the Azure AI search index we created earlier in this lab

```
# Semantic Hybrid Search
query = "Can you provide more details on Mountain-100?"

openai_client = AzureOpenAI(
  api_key = os.getenv("AZURE_OPENAI_API_KEY"),
  api_version = os.getenv("AZURE_OPENAI_API_VERSION"),
  azure_endpoint =os.getenv("AZURE_OPENAI_ENDPOINT")
)

search_client = SearchClient(AZURE_SEARCH_SERVICE, AZURE_SEARCH_INDEX,
AzureKeyCredential(AZURE_SEARCH_KEY))
vector_query = VectorizedQuery(vector=generate_embeddings(openai_client, query),
k_nearest_neighbors=10, fields="vectorContent")

results = search_client.search(
    search_text=query,
    vector_queries=[vector_query],
    select=[
        "id",
        "categoryId",
        "categoryName",
        "sku",
        "name",
        "description",
        "price",
        "tags",
```

```
    ],
    query_type=QueryType.SEMANTIC, semantic_configuration_name='my-semantic-config',
query_caption=QueryCaptionType.EXTRACTIVE, query_answer=QueryAnswerType.EXTRACTIVE,
    top=5
)

context = ""
for result in results:
    context += str(result) + "\n"

print(run_query(context, query))
```

Change the query (if you want to).

If running unchanged, this should return the following response:

```
Certainly! Mountain-100 is a popular mountain bike model, and we have a few
different variations available, such as the Mountain-100 Black in size 44 and 42,
and the Mountain-100 Silver in size 48 and 38. Is there anything specific you
would like to know about the bike?
```

Hybrid search with Streamlit application

In this exercise, we will modify our existing Streamlit application to use hybrid search over Azure AI search, instead of vector search over Azure Cosmos DB for MongoDB vCore.

Copy **app.py** from lab5 folder to our lab6 folder

Open **app.py** in Visual Studio Code

Replace the import statements with this code:

```
import streamlit as st
import os, uuid
from urllib.parse import quote
from datetime import datetime

from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
from langchain_openai import AzureChatOpenAI, AzureOpenAIEmbeddings
from langchain.schema import HumanMessage
from langchain.memory import ConversationBufferMemory, CosmosDBChatMessageHistory
from langchain.memory.chat_message_histories import StreamlitChatMessageHistory
from langchain.chains import ConversationalRetrievalChain
from langchain.callbacks.base import BaseCallbackHandler
from langchain.vectorstores import AzureCosmosDBVectorSearch

from dotenv import load_dotenv

from openai import AzureOpenAI
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from azure.search.documents.models import *
```

Copy/paste this code to add a new function hybrid_search() that will query our Azure AI search index, get all relevant documents and construct a prompt with these documents as context. This prompt will then be sent to Azure OpenAI to formulate a response

```
def hybrid_search(search_query):
    """
    Use Azure AI search to retrieve product information based on the given search query (Hybrid search).
```

```
Parameters:
    search_query (str): The search query

Returns:
    list: A list of documents containing product information
"""

AZURE_SEARCH_INDEX = os.getenv("COSMOSDB_MONGODB_PRODUCTS") + "_index"
AZURE_SEARCH_KEY = os.getenv("AZURE_SEARCH_KEY")
AZURE_SEARCH_SERVICE = os.getenv("AZURE_SEARCH_SERVICE")

# Query the semantic search index
search_client = SearchClient(
    AZURE_SEARCH_SERVICE,
    AZURE_SEARCH_INDEX,
    AzureKeyCredential(AZURE_SEARCH_KEY)
)

vector_query = VectorizedQuery(
    vector=calculate_embeddings(search_query),
    k_nearest_neighbors=10,
    fields="vectorContent"
)

results = search_client.search(
    search_text=search_query,
    vector_queries=[vector_query],
    select=[
        "id",
        "categoryId",
        "categoryName",
        "sku",
        "name",
        "description",
        "price",
        "tags",
```

```
    ],
    query_type=QueryType.SEMANTIC,
    semantic_configuration_name='my-semantic-config',
    query_caption=QueryCaptionType.EXTRACTIVE,
    query_answer=QueryAnswerType.EXTRACTIVE,
    top=5
)

products = []
for result in results:
    products.append({
        "id": result["id"],
        "categoryId": result["categoryId"],
        "categoryName": result["categoryName"],
        "sku": result["sku"],
        "name": result["name"],
        "description": result["description"],
        "price": result["price"],
        "tags": result["tags"],
        "@search.reranker_score": result["@search.reranker_score"]
    })

context = products

# formulate a response using Azure OpenAI
openai_client = AzureOpenAI(
    api_key = os.getenv("AZURE_OPENAI_API_KEY"),
    api_version = os.getenv("AZURE_OPENAI_API_VERSION"),
    azure_endpoint =os.getenv("AZURE_OPENAI_ENDPOINT")
    )

prompt = f"""You are an AI chatbot having a conversation with a human.
Context:
{context}

Human: {search_query}
```

```
    AI: """

    response = openai_client.chat.completions.create(
            model = os.getenv("AZURE_OPENAI_CHAT_MODEL"),
            messages = [{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content
```

All what's left to do now is to create a function that will call the hybrid_search() function we just created.

Copy/paste this code:

```
def hybrid_search_with_cosmos_history():

    cosmos_nosql = init_cosmos_nosql_history()

    msgs = StreamlitChatMessageHistory(key="langchain_messages")
    memory = ConversationBufferMemory(
        memory_key="chat_history",
        chat_memory=cosmos_nosql,
        return_messages=True
    )

    if len(msgs.messages) == 0:
        msgs.add_ai_message("How can I help you?")

    view_messages = st.expander("View the message contents in session state")

    # Render current messages
    for msg in msgs.messages:
        st.chat_message(msg.type).markdown(msg.content, unsafe_allow_html=True)

    # If user inputs a new prompt, generate and draw a new response
    if prompt := st.chat_input():
        st.chat_message("human").markdown(prompt, unsafe_allow_html=True)
        msgs.add_user_message(prompt)
```

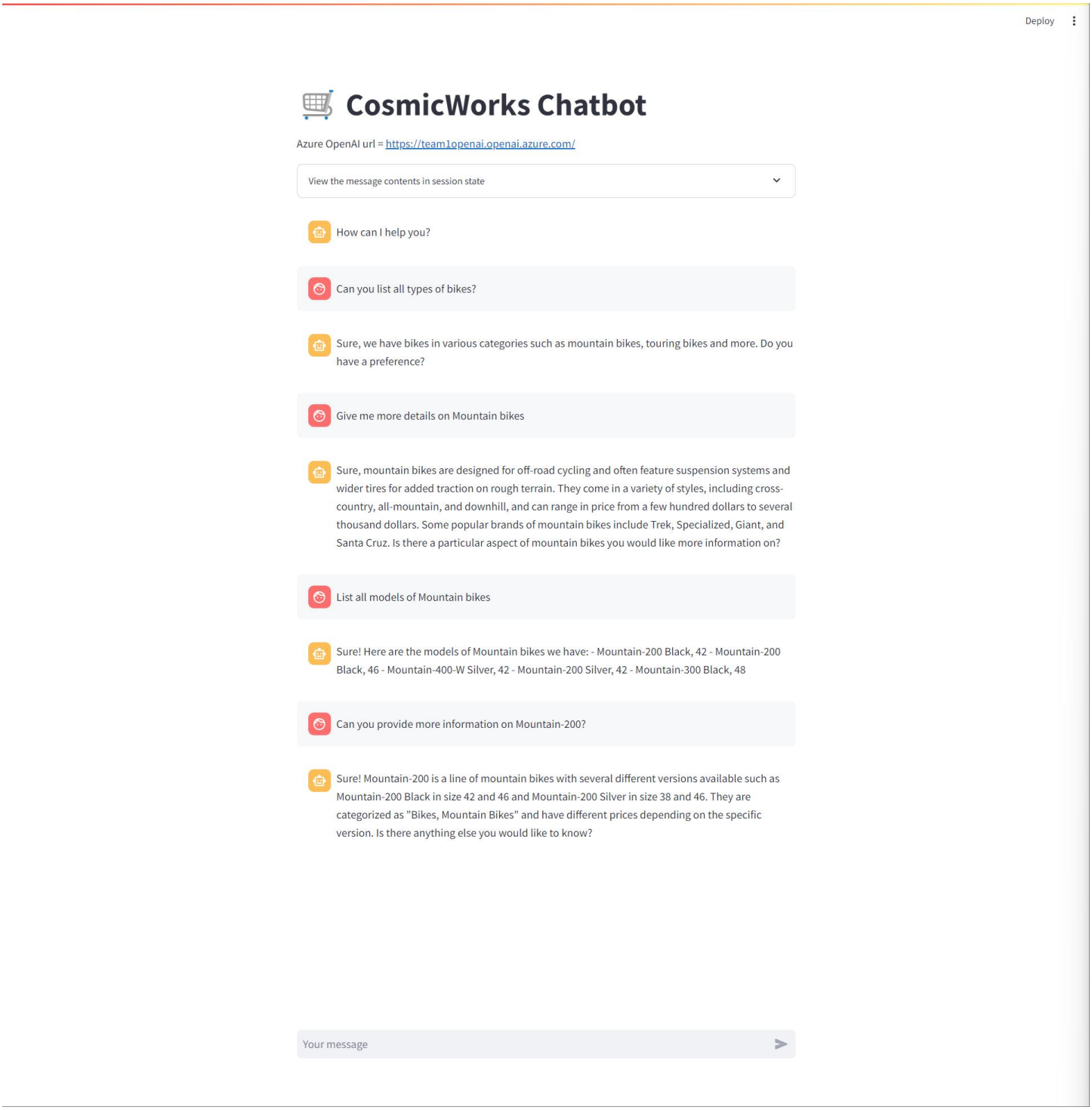
```
    with st.spinner("Please wait.."):
        response = hybrid_search(prompt)

        st.chat_message("ai").markdown(response, unsafe_allow_html=True)
        msgs.add_ai_message(response)

# Draw the messages at the end, so newly generated ones show up immediately
with view_messages:
    view_messages.json(st.session_state.langchain_messages)
```

Modify the application entry point to reference this new funcction:

```
if __name__ == "__main__":
    init_env()
    # main()
    # rag()
    # rag_with_cosmos_history()
    hybrid_search_with_cosmos_history()
```

Run the application with streamlit run app.py

You can now ask a couple of questions, that will be answered, using hybrid search on the Azure AI Search index



Complete code:

```
import streamlit as st
import os, uuid
from urllib.parse import quote
from datetime import datetime

from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
```

```
from langchain_openai import AzureChatOpenAI, AzureOpenAIEmbeddings
from langchain.schema import HumanMessage
from langchain.memory import ConversationBufferMemory, CosmosDBChatMessageHistory
from langchain.memory.chat_message_histories import StreamlitChatMessageHistory
from langchain.chains import ConversationalRetrievalChain
from langchain.callbacks.base import BaseCallbackHandler
from langchain.vectorstores import AzureCosmosDBVectorSearch

from dotenv import load_dotenv

from openai import AzureOpenAI
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from azure.search.documents.models import *


# Can you list all types of bikes?
# Can you provide more information on mountain bikes?
# List all types of mountain bikes
# Can you provide more details on the Mountain-100?

def init_env():

    load_dotenv()

    st.set_page_config(page_title="CosmicWorks Chatbot", page_icon="🛒")
    st.title("🛒 CosmicWorks Chatbot")

    os.environ["OPENAI_API_TYPE"] = "azure"
    os.environ["OPENAI_API_VERSION"] = os.getenv("AZURE_OPENAI_API_VERSION")
    os.environ["azure_endpoint"] = os.getenv("AZURE_OPENAI_ENDPOINT")
    os.environ["OPENAI_API_KEY"] = os.getenv("AZURE_OPENAI_API_KEY")
    os.environ["OPENAI_EMBEDDINGS_MODEL_NAME"] =
os.getenv("AZURE_OPENAI_EMBEDDING_MODEL")

    st.write(f"Azure OpenAI url = {os.getenv('AZURE_OPENAI_ENDPOINT')}")
def main():
```

```
    st.write("Product embeddings are stored in Azure Cosmos DB for MongoDB vCore")
    st.write("Conversations are stored in Azure Cosmos DB for NoSQL")

    # Set up the LLM
    llm = AzureChatOpenAI(
        deployment_name=os.getenv("AZURE_OPENAI_CHAT_MODEL"),
        temperature=0,
        max_tokens=1000
    )

    # Set up the LLMChain
    template = """You are an AI chatbot having a conversation with a human.

    Human: {human_input}
    AI: """
    prompt = PromptTemplate(input_variables=["human_input"], template=template)
    llm_chain = LLMChain(llm=llm, prompt=prompt)

    # Set up the conversation
    if prompt := st.chat_input():
        st.chat_message("human").write(prompt)

        with st.spinner("Please wait.."):
            response = llm_chain.run(prompt)

            st.chat_message("ai").write(response)

def get_cosmosdb_mongodb_connection_string():

    host = os.getenv('COSMOSDB_MONGODB_HOST')
    username = os.getenv('COSMOSDB_MONGODB_USERNAME')
    password = os.getenv('COSMOSDB_MONGODB_PASSWORD')
    encoded_password = quote(password, safe='')
```

```
    connection_string =
f'mongodb+srv://{username}:{encoded_password}@{host}/?tls=true&authMechanism=SCRAM-SHA-
256&retrywrites=false&maxIdleTimeMS=120000'

    return connection_string

def calculate_embeddings(query):
    embeddings = AzureOpenAIEmbeddings(
        azure_deployment=os.getenv("AZURE_OPENAI_EMBEDDING_MODEL"),
        openai_api_version=os.getenv("OPENAI_API_VERSION")
    )
    query_vector = embeddings.embed_query(query)
    return query_vector

def configure_retriever():

    connection_string = get_cosmosdb_mongodb_connection_string()

    embeddings = AzureOpenAIEmbeddings(
            azure_deployment=os.getenv("AZURE_OPENAI_EMBEDDING_MODEL"),
            openai_api_version=os.getenv("OPENAI_API_VERSION")
    )

    database_name = os.getenv('COSMOSDB_MONGODB_DATABASE')
    products_collection_name = os.getenv("COSMOSDB_MONGODB_PRODUCTS")
    namespace = f"{database_name}.{products_collection_name}"
    index_name = products_collection_name + "_vectorindex"

    vector_store = AzureCosmosDBVectorSearch.from_connection_string(
        connection_string,
        namespace,
        embeddings,
        index_name=index_name
    )

    return vector_store
```

```python
class StreamHandler(BaseCallbackHandler):
    def __init__(self, container: st.delta_generator.DeltaGenerator, initial_text: str = ""):
        self.container = container
        self.text = initial_text
        self.run_id_ignore_token = None
        self.complete = False  # Added flag to track completion

    def on_llm_start(self, serialized: dict, prompts: list, **kwargs):
        if prompts[0].startswith("Human"):
            self.run_id_ignore_token = kwargs.get("run_id")

    def on_llm_new_token(self, token: str, **kwargs) -> None:
        if self.run_id_ignore_token == kwargs.get("run_id", False):
            return
        self.text += token
        self.container.markdown(self.text)

    def on_llm_end(self, response, **kwargs):
        self.complete = True  # Mark completion

class PrintRetrievalHandler(BaseCallbackHandler):
    def __init__(self, container):
        self.status = container.status("**Context Retrieval**")

    def on_retriever_start(self, serialized: dict, query: str, **kwargs):
        self.status.write(f"**Question:** {query}")
        self.status.update(label=f"**Context Retrieval:** {query}")

    def on_retriever_end(self, documents, **kwargs):
        for doc in documents:
            self.status.markdown(doc.page_content)
        self.status.update(state="complete")

def rag():
    msgs = StreamlitChatMessageHistory(key="langchain_messages")
```

```
memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True
)
# Setup vector store, LLM and QA chain
vector_store = configure_retriever()

llm = AzureChatOpenAI(
    deployment_name=os.getenv("AZURE_OPENAI_CHAT_MODEL"),
    temperature=0,
    max_tokens=1000
)

# Setup the QA chain
qa_chain = ConversationalRetrievalChain.from_llm(
    llm,
    retriever=vector_store.as_retriever(),
    memory=memory,
    verbose=True
)

if len(msgs.messages) == 0:
    msgs.add_ai_message("How can I help you?")

view_messages = st.expander("View the message contents in session state")

# Render current messages
for msg in msgs.messages:
    st.chat_message(msg.type).markdown(msg.content, unsafe_allow_html=True)

# If user inputs a new prompt, generate and draw a new response
if prompt := st.chat_input():
    st.chat_message("human").markdown(prompt, unsafe_allow_html=True)
    msgs.add_user_message(prompt)

    with st.spinner("Please wait.."):
```

```
        retrieval_handler = PrintRetrievalHandler(st.container())
        stream_handler = StreamHandler(st.empty())

        response = qa_chain.run(
            prompt,
            callbacks=[retrieval_handler, stream_handler]
        )

        st.chat_message("ai").markdown(response, unsafe_allow_html=True)
        msgs.add_ai_message(response)

    # Draw the messages at the end, so newly generated ones show up immediately
    with view_messages:
        view_messages.json(st.session_state.langchain_messages)

def init_cosmos_nosql_history():
    cosmos_endpoint =
f"https://{os.getenv('COSMOSDB_NOSQL_ACCOUNT')}.documents.azure.com:443/"
    cosmos_key = os.getenv('COSMOSDB_NOSQL_KEY')
    cosmos_database = os.getenv('COSMOSDB_NOSQL_DATABASE_NAME')
    cosmos_container = os.getenv('COSMOSDB_NOSQL_CONTAINER_NAME')
    cosmos_connection_string = f"AccountEndpoint={cosmos_endpoint};AccountKey={cosmos_key}"

    current_dt = str(datetime.now().strftime("%Y%m%d_%H%M%S"))

    # get user_id from session_state
    if "session_id" not in st.session_state:
        st.session_state.session_id = str(uuid.uuid4())

    # get user_id from session_state (in a real app, we would read from authenticated user)
    if "user_id" not in st.session_state:
        st.session_state.user_id = str(uuid.uuid4())

    cosmos_nosql = CosmosDBChatMessageHistory(
        cosmos_endpoint=cosmos_endpoint,
        cosmos_database=cosmos_database,
```

```
        cosmos_container=cosmos_container,
        connection_string=cosmos_connection_string,
        session_id=current_dt,
        user_id=st.session_state.user_id
    )
    # prepare the cosmosdb instance
    cosmos_nosql.prepare_cosmos()
    return cosmos_nosql

def rag_with_cosmos_history():

    cosmos_nosql = init_cosmos_nosql_history()

    msgs = StreamlitChatMessageHistory(key="langchain_messages")
    memory = ConversationBufferMemory(
        memory_key="chat_history",
        chat_memory=cosmos_nosql,
        return_messages=True
    )
    # Setup vector store, LLM and QA chain
    vector_store = configure_retriever()

    llm = AzureChatOpenAI(
        deployment_name=os.getenv("AZURE_OPENAI_CHAT_MODEL"),
        temperature=0,
        max_tokens=1000
    )

    # Setup the QA chain
    qa_chain = ConversationalRetrievalChain.from_llm(
        llm,
        retriever=vector_store.as_retriever(),
        memory=memory,
        verbose=True
    )
```

```
    if len(msgs.messages) == 0:
        msgs.add_ai_message("How can I help you?")

    view_messages = st.expander("View the message contents in session state")

    # Render current messages
    for msg in msgs.messages:
        st.chat_message(msg.type).markdown(msg.content, unsafe_allow_html=True)

    # If user inputs a new prompt, generate and draw a new response
    if prompt := st.chat_input():
        st.chat_message("human").markdown(prompt, unsafe_allow_html=True)
        msgs.add_user_message(prompt)

        with st.spinner("Please wait.."):
            retrieval_handler = PrintRetrievalHandler(st.container())
            stream_handler = StreamHandler(st.empty())

            response = qa_chain.run(
                prompt,
                callbacks=[retrieval_handler, stream_handler]
            )

            st.chat_message("ai").markdown(response, unsafe_allow_html=True)
            msgs.add_ai_message(response)

    # Draw the messages at the end, so newly generated ones show up immediately
    with view_messages:
        view_messages.json(st.session_state.langchain_messages)

def hybrid_search(search_query):
    """
    Use Azure AI search to retrieve product information based on the given search query (Hybrid
search).

    Parameters:
```

```
    search_query (str): The search query

Returns:
    list: A list of documents containing product information
"""

AZURE_SEARCH_INDEX = os.getenv("COSMOSDB_MONGODB_PRODUCTS") + "_index"
AZURE_SEARCH_KEY = os.getenv("AZURE_SEARCH_KEY")
AZURE_SEARCH_SERVICE = os.getenv("AZURE_SEARCH_SERVICE")

# Query the semantic search index
search_client = SearchClient(
    AZURE_SEARCH_SERVICE,
    AZURE_SEARCH_INDEX,
    AzureKeyCredential(AZURE_SEARCH_KEY)
)

vector_query = VectorizedQuery(
    vector=calculate_embeddings(search_query),
    k_nearest_neighbors=10,
    fields="vectorContent"
)

results = search_client.search(
    search_text=search_query,
    vector_queries=[vector_query],
    select=[
        "id",
        "categoryId",
        "categoryName",
        "sku",
        "name",
        "description",
        "price",
        "tags",
    ],
```

```
    query_type=QueryType.SEMANTIC,
    semantic_configuration_name='my-semantic-config',
    query_caption=QueryCaptionType.EXTRACTIVE,
    query_answer=QueryAnswerType.EXTRACTIVE,
    top=5
)

products = []
for result in results:
    products.append({
        "id": result["id"],
        "categoryId": result["categoryId"],
        "categoryName": result["categoryName"],
        "sku": result["sku"],
        "name": result["name"],
        "description": result["description"],
        "price": result["price"],
        "tags": result["tags"],
        "@search.reranker_score": result["@search.reranker_score"]
    })

context = products

# formulate a response using Azure OpenAI
openai_client = AzureOpenAI(
    api_key = os.getenv("AZURE_OPENAI_API_KEY"),
    api_version = os.getenv("AZURE_OPENAI_API_VERSION"),
    azure_endpoint =os.getenv("AZURE_OPENAI_ENDPOINT")
    )

prompt = f"""You are an AI chatbot having a conversation with a human.
Context:
{context}

Human: {search_query}
AI: """
```

```
    response = openai_client.chat.completions.create(
            model = os.getenv("AZURE_OPENAI_CHAT_MODEL"),
            messages = [{"role": "user", "content": prompt}]
    )

    return response.choices[0].message.content

def hybrid_search_with_cosmos_history():

    cosmos_nosql = init_cosmos_nosql_history()

    msgs = StreamlitChatMessageHistory(key="langchain_messages")
    memory = ConversationBufferMemory(
        memory_key="chat_history",
        chat_memory=cosmos_nosql,
        return_messages=True
    )

    if len(msgs.messages) == 0:
        msgs.add_ai_message("How can I help you?")

    view_messages = st.expander("View the message contents in session state")

    # Render current messages
    for msg in msgs.messages:
        st.chat_message(msg.type).markdown(msg.content, unsafe_allow_html=True)

    # If user inputs a new prompt, generate and draw a new response
    if prompt := st.chat_input():
        st.chat_message("human").markdown(prompt, unsafe_allow_html=True)
        msgs.add_user_message(prompt)

        with st.spinner("Please wait.."):
            response = hybrid_search(prompt)
```

```
            st.chat_message("ai").markdown(response, unsafe_allow_html=True)
            msgs.add_ai_message(response)

    # Draw the messages at the end, so newly generated ones show up immediately
    with view_messages:
        view_messages.json(st.session_state.langchain_messages)


if __name__ == "__main__":
    init_env()
    # main()
    # rag()
    # rag_with_cosmos_history()
    hybrid_search_with_cosmos_history()
```