Labs Data & AI Innovation Day

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 2](#_Toc156040298)

[Connect to an existing Azure AI Search account 3](#_Toc156040299)

[Install pre-requisites 4](#_Toc156040300)

[Create Azure AI Search index 5](#_Toc156040301)

[Populate index from Azure Cosmos DB for MongoDB vCore collection 9](#_Toc156040302)

[Initialize products collection 9](#_Toc156040303)

[Get products from Azure Cosmos DB for MongoDB vCore 10](#_Toc156040304)

[Verify Azure AI Search index 13](#_Toc156040305)

[Hybrid search with Jupyter notebook 14](#_Toc156040306)

[Hybrid search with Streamlit application 17](#_Toc156040307)

# 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

A screenshot of a computer

Description automatically generated

* Select aisearch-openhack-2024
* In the left menu, select “Indexes”
* There are currently no indexes in the Azure AI search account

A screenshot of a computer

Description automatically generated

# 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:

A screenshot of a video game

Description automatically generated

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

A screenshot of a computer program

Description automatically generated

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:

A black background with white text

Description automatically generated

# 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

A screenshot of a computer

Description automatically generated

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.

A black background with white text

Description automatically generated

# 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

A screenshot of a computer

Description automatically generated

# 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:

A screenshot of a computer

Description automatically generated

# 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

A screenshot of a chatbot

Description automatically generated

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

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()