Lab 5: Build a Python chatbot

This document describes how to build a Python chatbot to query the Azure Cosmos DB for MongoDB vCore in natural language

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

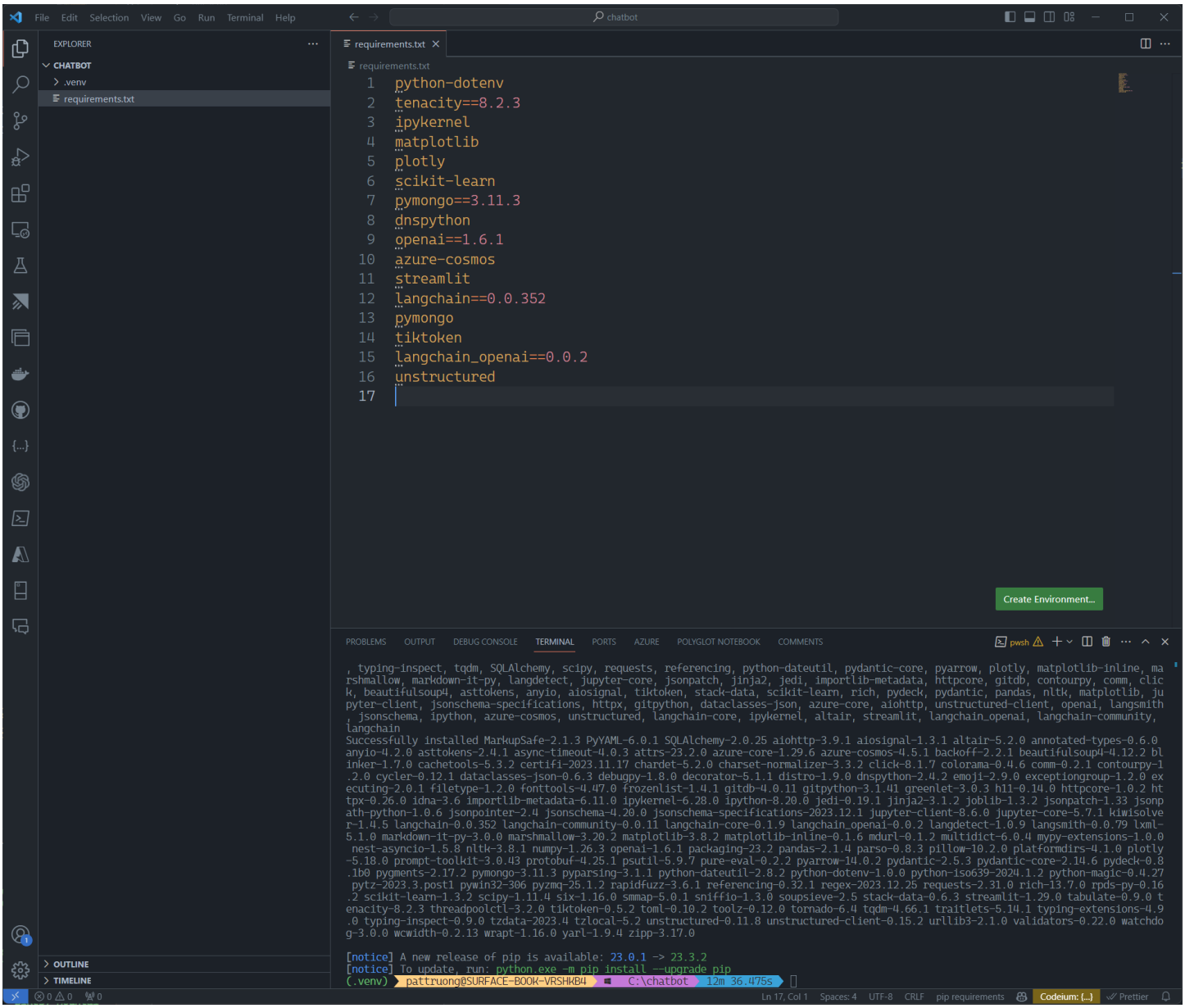
Create a new Python virtual environment

Follow these steps to create the environment required to build the Python chatbot

- Create a folder on your local machine e.g. C:\lab5
- Open Visual Studio Code
- Create a new file called **`requirements.txt`**
- Open requirements.txt and copy/paste the following code:

```
python-dotenv
tenacity==8.2.3
ipykernel
matplotlib
plotly
scikit-learn
pymongo==3.11.3
dnspython
openai==1.6.1
azure-cosmos
streamlit
langchain==0.0.352
pymongo
tiktoken
langchain_openai==0.0.2
unstructured
```

- Open a new terminal: Menu Terminal > New terminal
- Create a new environment by entering **`python -m venv .venv`**
- Activate this new environment with **`.venv\scripts\activate`**
- Install library dependencies with **`pip install -r requirements.txt`** (this can take several minutes)



- Create a new file called .env
- Copy/paste the following code:

```
COSMOSDB_MONGODB_HOST=yourclustername.mongocluster.cosmos.azure.com
COSMOSDB_MONGODB_USERNAME=sa
COSMOSDB_MONGODB_PASSWORD=your password
COSMOSDB_MONGODB_DATABASE=database_userXX
COSMOSDB_MONGODB_PRODUCTS=products_userXX
COSMOSDB_MONGODB_CUSTOMERS=customers_userXX

COSMOSDB_NOSQL_ACCOUNT=yournosqlaccount
COSMOSDB_NOSQL_DATABASE_NAME=database_teamXX
COSMOSDB_NOSQL_CONTAINER_NAME=conversations
COSMOSDB_NOSQL_KEY="xxxxxx"
```

```
AZURE_OPENAI_API_KEY=Xxxxxxx
AZURE_OPENAI_ENDPOINT=https://userXXopenai.openai.azure.com/
AZURE_OPENAI_EMBEDDING_MODEL=text-embedding-ada-002
AZURE_OPENAI_CHAT_MODEL=gpt-35-turbo
AZURE_OPENAI_API_VERSION=2023-12-01-preview

AZURE_SEARCH_SERVICE=https://yourserach.search.windows.net
AZURE_SEARCH_KEY=xxxxxxxxxxxx
AZURE_SEARCH_API_VERSION=2023-11-01
```

- Replace XX with your account for this lab

Create a simple chatbot

In this exercice, we will create a chatbot that can answer simple questions

- In Visual Studio Code, create a file call app.py
- Copy/paste the import statements

```
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
```

Let's make sure that we are able to read the content of the .env file

- Copy this code to app.py

```
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")
```

```
if __name__ == "__main__":
    init_env()
```

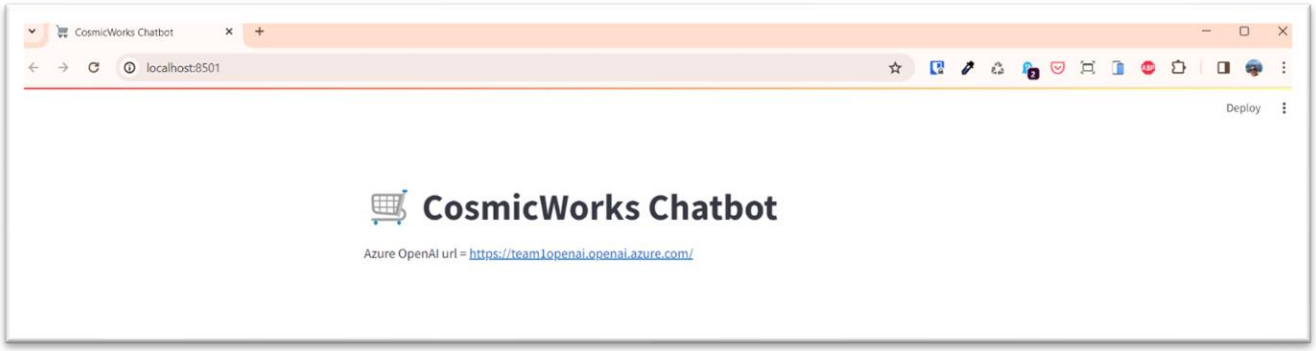
Save the app.py file

In the command prompt, let's run the application:

Makre sure you are in the correct virtual environment:

`.venv\scripts\activate`

Run the application with **`streamlit run app.py`**



Great!

Let's now add code to our application to run simple queries

Create a main() function

```
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)
```

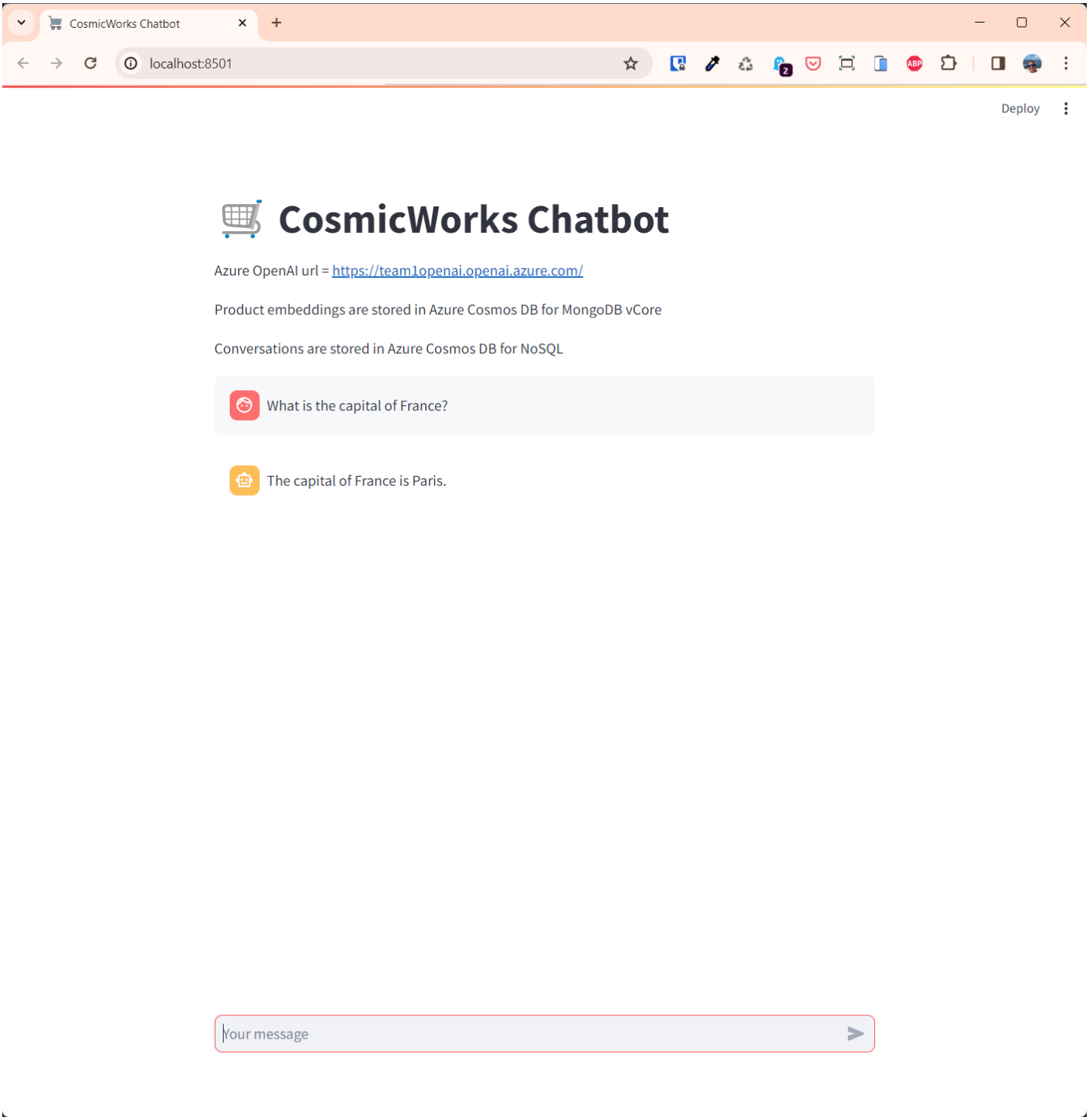
Modify the application entry point:

```
if __name__ == "__main__":
    init_env()
    main()
```

Run the application again with `streamlit run app.py`

In the textbox, enter "`What is the capital of France?`" and submit the query

You should get this answer:



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

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")

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


if __name__ == "__main__":
    init_env()
    main()
```

Ground the chatbot on your data (RAG)

Connection string

Create a new function get_ get_cosmosdb_mongodb_connection_string():

```
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
```

Embeddings

Copy/Paste this code to create a new function to calculate embeddings from a 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
```

Retriever

Copy/paste this code to create a retriever. The retriever uses the text-embedding-ada-002 model to calculate the embeddings and retrieve vectorized information from our Azure Cosmos DB for MongoDB vCore table (database_team02.products_team01)

```
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
```

Debug functions

We will now create two classes that will help us understand what is happening behind the scene

```
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")
```

Retrieval Augmented Generation (RAG)

Now that we have the environment ready, let's replace our simple chatbot with a chatbot grounded on the Contoso data, stored in Azure Cosmos DB for MongoDB vCore.

Create a rag() function:

```
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)
```

Modify the application entry point to call our rag() function

```
if __name__ == "__main__":
    init_env()
    rag()
```

Run the application again with **`streamlit run app.py`**

Here are some questions that you can ask:

```
What types of bikes do you have?
Can you provide more information on mountain bikes?
List all types of mountain bikes
Can you provide more details on the Mountain-100?
```

Deploy

CosmicWorks Chatbot

Azure OpenAI url = https://team1openai.openai.azure.com/

View the message contents in session state

How can I help you?

What types of bikes do you have?

We have Mountain Bikes, Road Bikes, and Touring Bikes.

Can you provide more information on mountain bikes?

Sure! Mountain bikes are designed for off-road cycling and are typically characterized by their sturdy frames, wide tires with knobby tread, and suspension systems to absorb shock. They are great for riding on rough terrain, such as dirt trails, rocky paths, and steep hills. Mountain bikes come in a variety of styles and price ranges, from entry-level models for beginners to high-end bikes for experienced riders.

List all types of mountain bikes

Based on the given context, all the products belong to the category "Bikes, Mountain Bikes". Therefore, the types of mountain bikes available are:

- Mountain-400-W Silver, 46
- Mountain-300 Black, 48
- Mountain-500 Silver, 44
- Mountain-100 Silver, 42

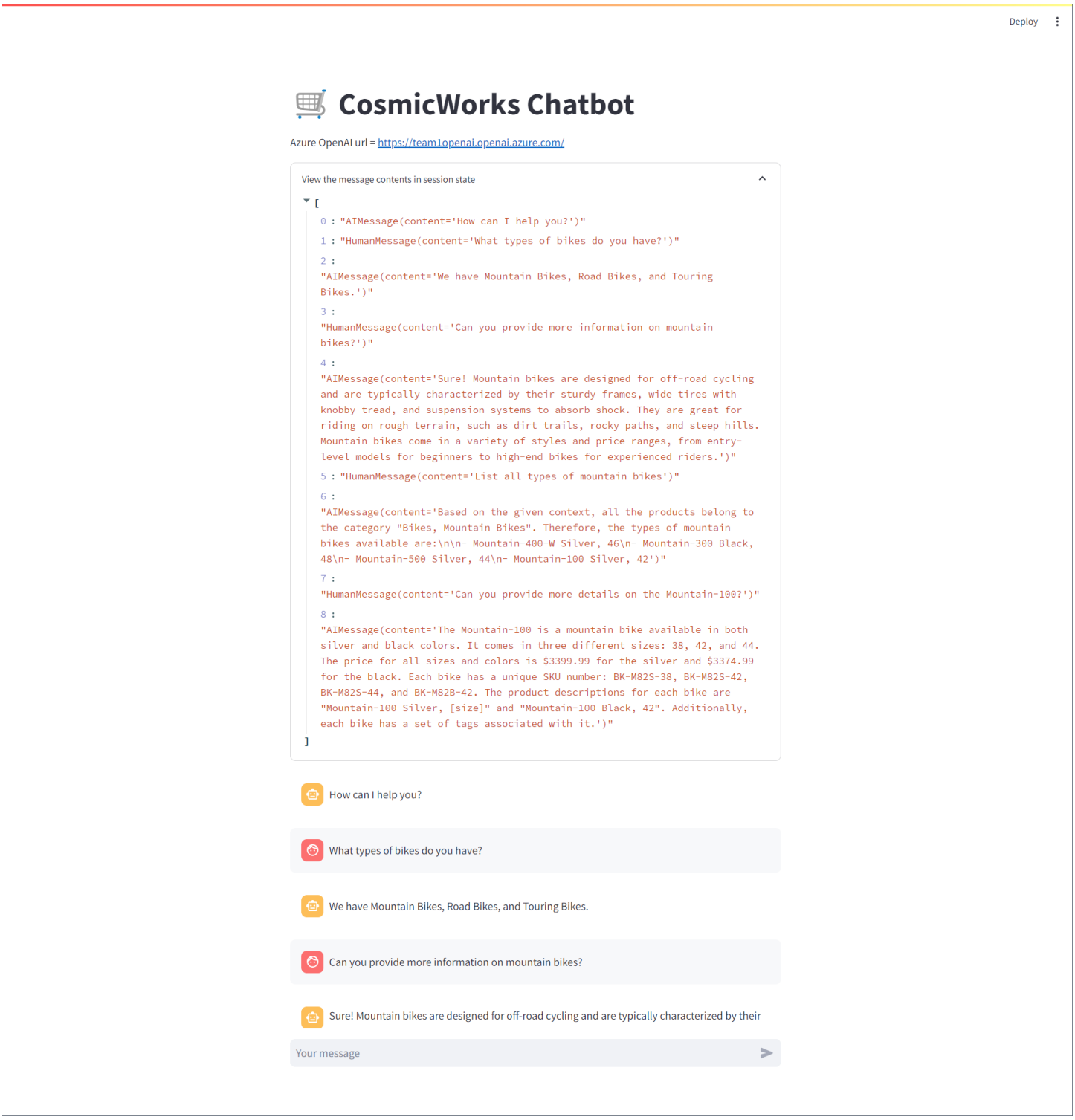
Can you provide more details on the Mountain-100?

Context Retrieval: Can you provide more details on the Mountain-100?

The Mountain-100 is a mountain bike available in both silver and black colors. It comes in three different sizes: 38, 42, and 44. The price for all sizes and colors is $3399.99 for the silver and 3374.99 for the black. Each bike has a unique SKU number: BK-M82S-38, BK-M82S-42, BK-M82S-44, and BK-M82B-42. The product descriptions for each bike are "Mountain-100 Silver, [size]" and "Mountain-100 Black, 42". Additionally, each bike has a set of tags associated with it.

Your message

You can expand the dropdown to view the history of messages



Full 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

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")

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)


if __name__ == "__main__":
    init_env()
    rag()
```

Store conversation history in Azure Cosmos DB for NoSQL

Let’s store the conversations history into an Azure Cosmos DB for NoSQL container.

Create a new method to initialize the NoSQL instance:

```
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
```

Duplicate the rag() function and rename the new function to `rag_with_cosmos_history()`

```python
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)
```

Modify the application entry point to call our rag_with_cosmos_history() function

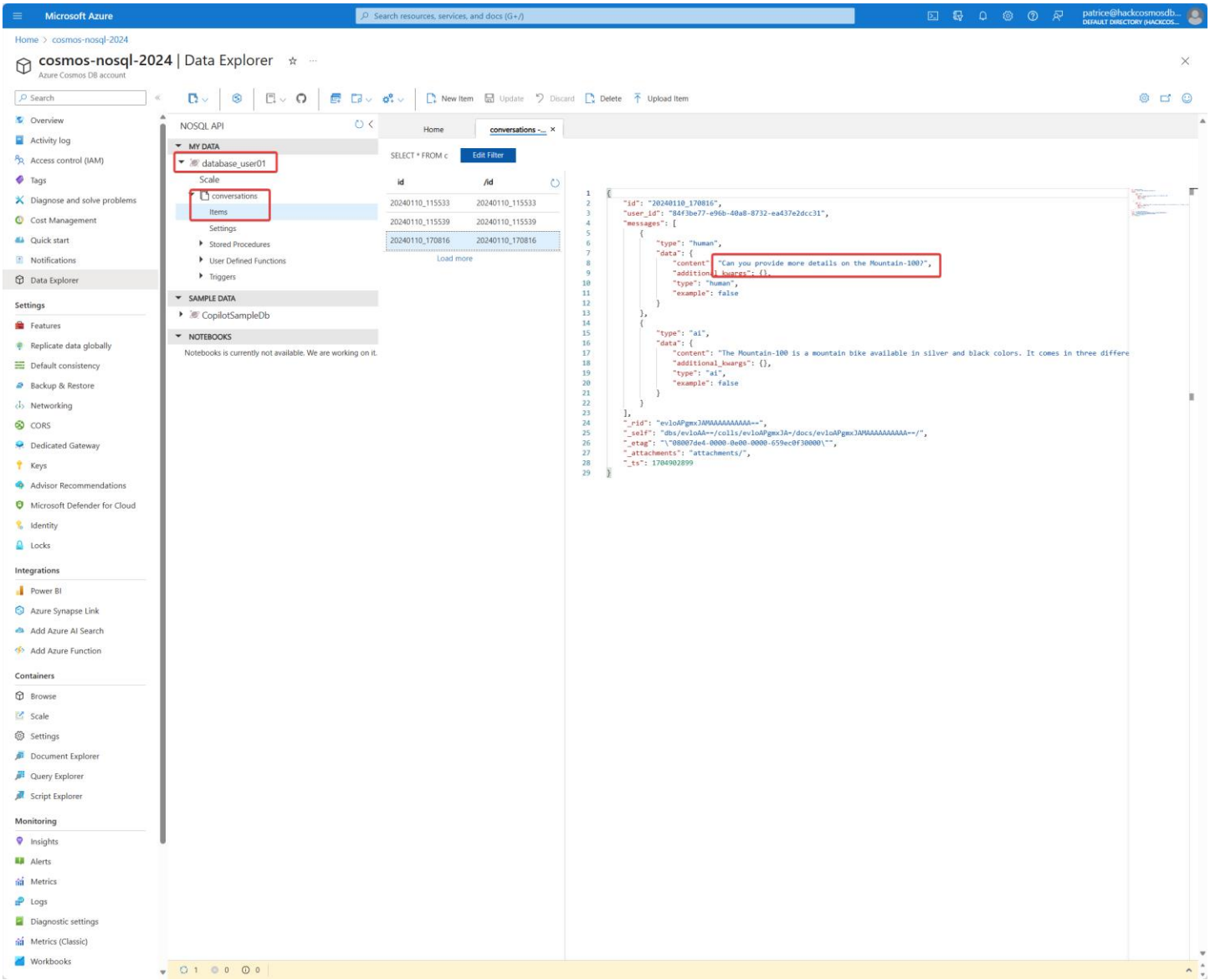
```
if __name__ == "__main__":
    init_env()
    rag_with_cosmos_history ()
```

Run the application again with **`streamlit run.app.py`**

Ask a question, such as **“`Can you provide more details on the Mountain-100?`”**

Check history in Azure Cosmos DB for NoSQL

- Go into the Azure Cosmos DB for NoSQL account in the Azure Portal: **`cosmos-nosql-2024`**
- Click on “Data Explorer”
- Expand database_user01 > conversations > Items
- Select the last document



Complete code for this lab

```
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

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")

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)


if __name__ == "__main__":
    init_env()
    rag_with_cosmos_history()
```