

# 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)

The screenshot shows a code editor interface with the following details:

- File Explorer:** Shows a project structure with a **CHATBOT** folder containing **.venv** and **requirements.txt**.
- Code Editor:** Displays the contents of **requirements.txt**:

```

1 python-dotenv
2 tenacity==8.2.3
3 ipykernel
4 matplotlib
5 plotly
6 scikit-learn
7 pymongo==3.11.3
8 dnspython
9 openai==1.6.1
10 azure-cosmos
11 streamlit
12 langchain==0.0.352
13 pymongo
14 tiktoken
15 langchain_openai==0.0.2
16 unstructured
17

```
- Terminal:** Shows the output of pip install requirements.txt:

```

typing-inspect tqdm SQLAlchemy scipy requests referencing python-dateutil pydantic-core pyarrow plotly matplotlib-inline marshmallow markdown-it-py langdetect jupyter-core jsonpatch jinja2 jedi importlib-metadata httpcore gitdb contourpy comm clic k beautifulsoup4 asttokens aiosignal tiktoken stack-data scikit-learn rich pydeck pydantic pandas nltk matplotlib ju unstructured-client openai langsmith ipython azure-cosmos unstructured langchain-core ipykernel altair streamlit langchain.openai langchain-community langchain
Successfully installed MarkupSafe-2.1.3 PyYAML-6.0.1 SQLAlchemy-2.0.25 aiohttp-3.9.1 aiosignal-1.3.1 altair-5.2.0 annotated-types-0.6.0 aiohttp-3.0.0 asttokens-2.4.1 async-timeout-0.4.3 attrs-23.2.0 azure-core-1.29.6 azure-cosmos-4.5.1 backoff-2.2.1 beautifulsoup4-4.12.2 blinker-1.7.0 cachetools-5.3.2 certifi-2023.11.17 charset-normalizer-3.3.2 click-8.1.7 colorama-0.4.6 comm-0.2.1 contourpy-1.2.0 cycler-0.12.1 dataclasses-json-0.6.3 debugpy-1.8.0 decorator-5.1.1 distro-1.9.0 dnspython-2.4.2 emoji-2.9.0 exceptiongroup-1.2.0 execting-2.0.1 filetype-1.2.0 fonttools-4.47.0 frozenlist-1.4.1 gitdb-4.0.1 gitpython-3.1.41 greenlet-3.0.3 h11-0.14.0 httpcore-1.0.2 httx-0.36.0 idna-3.6 importlib-metadata-11.0 ipykernel-6.28.0 ipython-8.20.0 jedi-0.19.1 jinja2-3.1.2 joblib-1.3.2 jsonpatch-1.33 jsonpath-python-1.0.6 jsonpointer-2.4.1 jsonschema-4.20.0 jsonschema-specifications-2023.12.1 jupyter-client-8.6.0 jupyter-core-5.7.1 kiwisolve-1-4.5 langchain-0.0.352 langchain-community-0.0.11 langchain-core-0.1.9 langchain.openai-0.0.2 langdetect-1.0.9 langsmith-0.0.79 lxml-5.1.0 markdown-it-py-3.0.0 marshmallow-3.20.2 matplotlib-3.8.2 matplotlib-inline-0.1.6 multidict-6.0.4 mypy-extensions-1.0.0 nose-asyncio-1.5.8 nntplib-2.8.1 numpy-1.26.3 openai-1.6.1 packaging-23.3 pandas-2.1.4 parso-0.8.3 pillow-10.2.0 platformdirs-4.1.0 plotly-5.18.0 prompt-toolkit-3.0.43 protobuf-4.26.1 psutil-5.9.7 pure-eval-0.2.2 pyarrow-14.0.2 pydantic-2.5.3 pydantic-core-2.14.6 pydeck-0.8 .lbb pygments-2.17.2 pymongo-3.11.3 pyarsing-3.1.1 python-dateutil-2.8.2 python-dotenv-1.6.0 python-isotp-2024.1.2 python-magic-0.4.27 pytz-2023.3.post1 pywin32-306 pymq-25.1.2 rapidfuzz-3.6.1 referencing-0.32.1 regex-2023.12.28 requests-2.31.0 rich-13.7.0 rpds-py-0.16 .2 scikit-learn-1.3.2 scipy-1.11.4 six-1.16.0 smmap-5.0.1 snffio-1.3.0 soupsieve-2.5 stack-data-0.6.3 streamlit-1.29.0 tabulate-0.9.0 t encility-8.2.3 threadpoolctl-3.2.0 tiktoken-0.5.2 toml-0.10.2 toolz-0.12.0 tornado-6.4 tqdm-4.66.1 traitlets-5.14.1 typing-extensions-4.9 .0 typing-Inspect-0.9.0 tzdata-2023.4 tzlocal-5.2 unstructured-0.11.8 unstructured-client-0.15.2 urlib3-2.1.0 validators-0.22.0 watchdog-0.10.0 wcwidth-0.2.13 wrapt-1.16.0 yarl-1.9.4 zipp-3.17.0
[notice] A new release of pip is available: 23.0.1 -> 23.3.2
[notice] To update, run: python -m pip install --upgrade pip
(.venv) > pattruong@SURFACE-BOOK-VRSHKB4 C:\chatbot 12m 36.475s

```
- Bottom Status Bar:** Shows file path (C:\chatbot), line count (17), column count (Col 1), spaces (Spaces 4), encoding (UTF-8), line endings (CRLF), pip requirements, Codeium icon, and Prettier icon.

- 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=xxxxxxxxxxxxxx  
AZURE_SEARCH_API_VERSION=2023-11-01
```

- Replace XX with your account for this lab

# Create a simple chatbot

---

In this exercise, 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:

The screenshot shows a web browser window titled "CosmicWorks Chatbot" at the URL "localhost:8501". The page features a header with a shopping cart icon and the text "CosmicWorks Chatbot". Below the header, it says "Azure OpenAI url = <https://team1openai.openai.azure.com/>". It also mentions that "Product embeddings are stored in Azure Cosmos DB for MongoDB vCore" and "Conversations are stored in Azure Cosmos DB for NoSQL". A conversation log is displayed, starting with a user message "What is the capital of France?" followed by a bot response "The capital of France is Paris.". At the bottom, there is a text input field labeled "Your message" with a red border and a send button icon.

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

Product embeddings are stored in Azure Cosmos DB for MongoDB vCore

Conversations are stored in Azure Cosmos DB for NoSQL

What is the capital of France?

The capital of France is Paris.

Your message >

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?*



## 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

Deploy :

## CosmicWorks Chatbot

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

```
View the message contents in session state ^  
[  
  0 : "AIMessage(content='How can I help you?')"  
  1 : "HumanMessage(content='What types of bikes do you have?')"  
  2 :  
    "AIMessage(content='We have Mountain Bikes, Road Bikes, and Touring  
    Bikes.')"  
  3 :  
    "HumanMessage(content='Can you provide more information on mountain  
    bikes?')"  
  4 :  
    "AIMessage(content='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.')"  
  5 : "HumanMessage(content='List all types of mountain bikes')"  
  6 :  
    "AIMessage(content='Based on the given context, all the products belong to  
    the category "Bikes, Mountain Bikes". Therefore, the types of mountain  
    bikes available are:\n- Mountain-400-W Silver, 46\n- Mountain-300 Black,  
    48\n- Mountain-500 Silver, 44\n- Mountain-100 Silver, 42')"  
  7 :  
    "HumanMessage(content='Can you provide more details on the Mountain-100?')"  
  8 :  
    "AIMessage(content='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.')"  
]  
]
```

 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

Your message 

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()`

```
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

The screenshot shows the Microsoft Azure Data Explorer interface for the database **cosmos-nosql-2024**. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Cost Management, Quick start, Notifications, and Data Explorer. Under Data Explorer, there are sections for Settings, Features, Replicate data globally, Default consistency, Backup & Restore, Networking, CORS, Dedicated Gateway, Keys, Advisor Recommendations, Microsoft Defender for Cloud, Identity, Locks, Integrations (Power BI, Azure Synapse Link, Add Azure AI Search, Add Azure Function), Containers (Browse, Scale, Settings, Document Explorer, Query Explorer, Script Explorer), and Monitoring (Insights, Alerts, Metrics, Logs, Diagnostic settings, Metrics (Classic), Workbooks). The main area shows the NOSQL API Data Explorer for the database **database\_user01**. The 'conversations' item is selected under the 'Items' section. A table lists three documents with IDs: 20240110\_115533, 20240110\_115539, and 20240110\_170816. The document with ID 20240110\_170816 is selected, and its content is displayed in a code editor:

```
1  {
2      "id": "20240110_170816",
3      "user_id": "84f3be77-e96b-40a8-8732-ea437e2dcc31",
4      "messages": [
5          {
6              "type": "human",
7              "data": {
8                  "content": "Can you provide more details on the Mountain-100?",
9                  "additional_args": {}
10             },
11             "type": "human",
12             "example": false
13         },
14         {
15             "type": "ai",
16             "data": {
17                 "content": "The Mountain-100 is a mountain bike available in silver and black colors. It comes in three differen",
18                 "additional_args": {},
19                 "type": "ai",
20                 "example": false
21             }
22         }
23     ],
24     "_rid": "evloApGmxAJAAAAAAA=--",
25     "_self": "dbs/evloApGmxA=/colls/evloApGmxA/_docs/evloApGmxAJAAAAAAA=--/",
26     "_ts": 1784902894000,
27     "_attachments": "attachments/",
28     "_ts": 1784902894
29 }
```

## 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()
```