Application development with Azure OpenAI Service

With the Azure OpenAI Service, developers can create chatbots and other applications that excel at understanding natural human language through the use of REST APIs or language specific SDKs. When working with these language models, how developers shape their prompt greatly impacts how the generative AI model will respond. Azure OpenAI models are able to tailor and format content, if requested in a clear and concise way. In this exercise, you'll learn how to connect your application to Azure OpenAI and see how different prompts for similar content help shape the AI model's response to better satisfy your requirements.

In the scenario for this exercise, you will perform the role of a software developer working on a wildlife marketing campaign. You are exploring how to use generative AI to improve advertising emails and categorize articles that might apply to your team. The prompt engineering techniques used in the exercise can be applied similarly for a variety of use cases.

This exercise will take approximately **30** minutes.

Clone the repository for this course

If you have not already done so, you must clone the code repository for this course:

1. Start Visual Studio Code.
2. Open the command palette (SHIFT+CTRL+P or **View** > **Command Palette...**) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLearning/mslearn-openai` repository to a local folder (it doesn't matter which folder).
3. When the repository has been cloned, open the folder in Visual Studio Code.
4. Wait while additional files are installed to support the C# code projects in the repo.

> **Note**: If you are prompted to add required assets to build and debug, select **Not Now**.

Provision an Azure OpenAI resource

If you don't already have one, provision an Azure OpenAI resource in your Azure subscription.

1. Sign into the **Azure portal** at `https://portal.azure.com`.
2. Create an **Azure OpenAI** resource with the following settings:
    - **Subscription**: *Select an Azure subscription that has been approved for access to the Azure OpenAI service*
    - **Resource group**: *Choose or create a resource group*
    - **Region**: *Make a **random** choice from any of the following regions**
        - East US
        - East US 2
        - North Central US
        - South Central US
        - Sweden Central
        - West US
        - West US 3
    - **Name**: *A unique name of your choice*
    - **Pricing tier**: Standard S0

> * Azure OpenAI resources are constrained by regional quotas. The listed regions include default quota for the model type(s) used in this exercise. Randomly choosing a region reduces the risk of a single region reaching its quota limit in scenarios where you are sharing a subscription with other users. In the event of a quota limit being reached later in the exercise, there's a possibility you may need to create another resource in a different region.

3. Wait for deployment to complete. Then go to the deployed Azure OpenAI resource in the Azure portal.

Deploy a model

Next, you will deploy an Azure OpenAI model resource from Cloud Shell.

1. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***Bash*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *PowerShell* environment, switch it to ***Bash***.

2. Refer to this example and replace the following variables with your own values from above:

   Code Copy

   ```
   az cognitiveservices account deployment create \
      -g <your_resource_group> \
      -n <your_OpenAI_service> \
      --deployment-name gpt-4o \
      --model-name gpt-4o \
      --model-version 2024-05-13 \
      --model-format OpenAI \
      --sku-name "Standard" \
      --sku-capacity 5
   ```

> **Note**: Sku-capacity is measured in thousands of tokens per minute. A rate limit of 5,000 tokens per minute is more than adequate to complete this exercise while leaving capacity for other people using the same subscription.

Configure your application

Applications for both C# and Python have been provided, and both apps feature the same functionality. First, you'll complete some key parts of the application to enable using your Azure OpenAI resource with asynchronous API calls.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **Labfiles/01-app-develop** folder and expand the **CSharp** or **Python** folder depending on your language preference. Each folder contains the language-specific files for an app into which you're you're going to integrate Azure OpenAI functionality.

2. Right-click the **CSharp** or **Python** folder containing your code files and open an integrated terminal. Then install the Azure OpenAI SDK package by running the appropriate command for your language preference:

   C#:

   Code Copy

   ```
   dotnet add package Azure.AI.OpenAI --version 2.1.0
   ```

   Python:

   Code Copy

```
pip install openai==1.65.2
```

3. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the configuration file for your preferred language
    - **C#**: appsettings.json
    - **Python**: .env
4. Update the configuration values to include:
    - The **endpoint** and a **key** from the Azure OpenAI resource you created (available on the **Keys and Endpoint** page for your Azure OpenAI resource in the Azure portal)
    - The **deployment name** you specified for your model deployment.
5. Save the configuration file.

Add code to use the Azure OpenAI service

Now you're ready to use the Azure OpenAI SDK to consume your deployed model.

1. In the **Explorer** pane, in the **CSharp** or **Python** folder, open the code file for your preferred language, and replace the comment ***Add Azure OpenAI package*** with code to add the Azure OpenAI SDK library:

C#: Program.cs

C# Copy

```
// Add Azure OpenAI packages
using Azure.AI.OpenAI;
using OpenAI.Chat;
```

Python: application.py

Code Copy

```
# Add Azure OpenAI package
from openai import AsyncAzureOpenAI
```

2. In the code file, find the comment ***Configure the Azure OpenAI client***, and add code to configure the Azure OpenAI client:

C#: Program.cs

C# Copy

```
// Configure the Azure OpenAI client
AzureOpenAIClient azureClient = new (new Uri(oaiEndpoint), new ApiKeyCredential(oaiKey));
ChatClient chatClient = azureClient.GetChatClient(oaiDeploymentName);
```

Python: application.py

Code Copy

```
# Configure the Azure OpenAI client
client = AsyncAzureOpenAI(
    azure_endpoint = azure_oai_endpoint,
    api_key=azure_oai_key,
    api_version="2024-02-15-preview"
)
```

3. In the function that calls the Azure OpenAI model, under the comment ***Get response from Azure OpenAI***, add the code to format and send the request to the model.

C#: Program.cs

C# Copy

```
// Get response from Azure OpenAI
ChatCompletionOptions chatCompletionOptions = new ChatCompletionOptions()
{
    Temperature = 0.7f,
    MaxOutputTokenCount = 800
};

ChatCompletion completion = chatClient.CompleteChat(
    [
        new SystemChatMessage(systemMessage),
        new UserChatMessage(userMessage)
    ],
    chatCompletionOptions
);

Console.WriteLine($"{completion.Role}: {completion.Content[0].Text}");
```

Python: application.py

Code Copy

```
# Get response from Azure OpenAI
messages =[
    {"role": "system", "content": system_message},
    {"role": "user", "content": user_message},
]

print("\nSending request to Azure OpenAI model...\n")

# Call the Azure OpenAI model
response = await client.chat.completions.create(
    model=model,
    messages=messages,
    temperature=0.7,
    max_tokens=800
)
```

4. Save the changes to the code file.

Run your application

Now that your app has been configured, run it to send your request to your model and observe the response. You'll notice the only difference between the different options is the content of the prompt, all other parameters (such as token count and temperature) remain the same for each request.

1. In the folder of your preferred language, open `system.txt` in Visual Studio Code. For each of the interactions, you'll enter the **System message** in this file and save it. Each iteration will pause first for you to change the system message.

2. In the interactive terminal pane, ensure the folder context is the folder for your preferred language. Then enter the following command to run the application.

- **C#**: `dotnet run`
- **Python**: `python application.py`

> **Tip**: You can use the **Maximize panel size** (^) icon in the terminal toolbar to see more of the console text.

3. For the first iteration, enter the following prompts:

System message

```
You are an AI assistant
```

User message:

```
Write an intro for a new wildlife Rescue
```

4. Observe the output. The AI model will likely produce a good generic introduction to a wildlife rescue.

5. Next, enter the following prompts which specify a format for the response:

System message

```
You are an AI assistant helping to write emails
```

User message:

```
Write a promotional email for a new wildlife rescue, including the following:
- Rescue name is Contoso
- It specializes in elephants
- Call for donations to be given at our website
```

> **Tip**: You may find the automatic typing in the VM doesn't work well with multiline prompts. If that is your issue, copy the entire prompt then paste it into Visual Studio Code.

6. Observe the output. This time, you'll likely see the format of an email with the specific animals included, as well as the call for donations.

7. Next, enter the following prompts that additionally specify the content:

System message

```
You are an AI assistant helping to write emails
```

User message:

```
Write a promotional email for a new wildlife rescue, including the following:
- Rescue name is Contoso
- It specializes in elephants, as well as zebras and giraffes
- Call for donations to be given at our website
Include a list of the current animals we have at our rescue after the signature, in the form of a table. These animals include elephants, zebras, gorillas, lizards, and jackrabbits.
```

8. Observe the output, and see how the email has changed based on your clear instructions.
9. Next, enter the following prompts where we add details about tone to the system message:

System message

Code Copy

```
You are an AI assistant that helps write promotional emails to generate interest in a new business. Your tone is light, chit-chat oriented and you always include at least two jokes.
```

User message:

Code Copy

```
Write a promotional email for a new wildlife rescue, including the following:
- Rescue name is Contoso
- It specializes in elephants, as well as zebras and giraffes
- Call for donations to be given at our website
Include a list of the current animals we have at our rescue after the signature, in the form of a table. These animals include elephants, zebras, gorillas, lizards, and jackrabbits.
```

10. Observe the output. This time you'll likely see the email in a similar format, but with a much more informal tone. You'll likely even see jokes included!

Use grounding context and maintain chat history

1. For the final iteration, we're deviating from email generation and exploring *grounding context* and maintaining chat history. Here you provide a simple system message, and change the app to provide the grounding context as the beginning of the chat history. The app will then append the user input, and extract information from the grounding context to answer our user prompt.
2. Open the file `grounding.txt` and briefly read the grounding context you'll be inserting.
3. In your app immediately after the comment ***Initialize messages list*** and before any existing code, add the following code snippet to read text in from `grounding.txt` and to initialize the chat history with the grounding context.

C#: Program.cs

C# Copy

```
// Initialize messages list
Console.WriteLine("\nAdding grounding context from grounding.txt");
string groundingText = System.IO.File.ReadAllText("grounding.txt");
var messagesList = new List<ChatMessage>()
{
    new UserChatMessage(groundingText),
};
```

Python: application.py

Code Copy

```
# Initialize messages array
print("\nAdding grounding context from grounding.txt")
grounding_text = open(file="grounding.txt", encoding="utf8").read().strip()
messages_array = [{"role": "user", "content": grounding_text}]
```

4. Under the comment ***Format and send the request to the model***, replace the code from the comment to the end of the **while** loop with the following code. The code is mostly the same, but now using the messages array to send the request to the model.

   C#: Program.cs

   C# Copy

   ```
   // Format and send the request to the model
   messagesList.Add(new SystemChatMessage(systemMessage));
   messagesList.Add(new UserChatMessage(userMessage));
   GetResponseFromOpenAI(messagesList);
   ```

   Python: application.py

   Code Copy

   ```
   # Format and send the request to the model
   messages_array.append({"role": "system", "content": system_text})
   messages_array.append({"role": "user", "content": user_text})
   await call_openai_model(messages=messages_array,
       model=azure_oai_deployment,
       client=client
   )
   ```

5. Under the comment ***Define the function that will get the response from Azure OpenAI endpoint***, replace the function declaration with the following code to use the chat history list when calling the function `GetResponseFromOpenAI` for C# or `call_openai_model` for Python.

   C#: Program.cs

   C# Copy

   ```
   // Define the function that gets the response from Azure OpenAI endpoint
   private static void GetResponseFromOpenAI(List<ChatMessage> messagesList)
   ```

   Python: application.py

   Code Copy

   ```
   # Define the function that will get the response from Azure OpenAI endpoint
   async def call_openai_model(messages, model, client):
   ```

6. Lastly, replace all the code under ***Get response from Azure OpenAI***. The code is mostly the same, but now using the messages array to store the conversation history.

   C#: Program.cs

   C# Copy

```
// Get response from Azure OpenAI
ChatCompletionOptions chatCompletionOptions = new ChatCompletionOptions()
{
    Temperature = 0.7f,
    MaxOutputTokenCount = 800
};

ChatCompletion completion = chatClient.CompleteChat(
    messagesList,
    chatCompletionOptions
);

Console.WriteLine($"{completion.Role}: {completion.Content[0].Text}");
messagesList.Add(new AssistantChatMessage(completion.Content[0].Text));
```

Python: application.py

Code Copy

```
# Get response from Azure OpenAI
print("\nSending request to Azure OpenAI model...\n")

# Call the Azure OpenAI model
response = await client.chat.completions.create(
    model=model,
    messages=messages,
    temperature=0.7,
    max_tokens=800
)

print("Response:\n" + response.choices[0].message.content + "\n")
messages.append({"role": "assistant", "content": response.choices[0].message.content})
```

7. Save the file and rerun your app.
8. Enter the following prompts (with the **system message** still being entered and saved in `system.txt`).

   System message

   Code Copy

   ```
   You're an AI assistant who helps people find information. You'll provide answers from the text provided in the prompt, and respond concisely.
   ```

   User message:

   Code Copy

   ```
   What animal is the favorite of children at Contoso?
   ```

   Notice that the model uses the grounding text information to answer your question.
9. Without changing the system message, enter the following prompt for the user message:

   User message:

   Code Copy

   ```
   How can they interact with it at Contoso?
   ```

Notice that the model recognizes “they” as the children and “it” as their favorite animal, since now it has access to your previous question in the chat history.

Clean up

When you’re done with your Azure OpenAI resource, remember to delete the deployment or the entire resource in the **Azure portal** at `https://portal.azure.com` .