Use Azure OpenAI APIs in your app

With the Azure OpenAI Service, developers can create chatbots, language models, and other applications that excel at understanding natural human language. The Azure OpenAI provides access to pre-trained AI models, as well as a suite of APIs and tools for customizing and fine-tuning these models to meet the specific requirements of your application. In this exercise, you'll learn how to deploy a model in Azure OpenAI and use it in your own application.

In the scenario for this exercise, you will perform the role of a software developer who has been tasked to implement an app that can use generative AI to help provide hiking recommendations. The techniques used in the exercise can be applied to any app that wants to use Azure OpenAI APIs.

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**
        - Australia East
        - Canada East
        - East US
        - East US 2
        - France Central
        - Japan East
        - North Central US
        - Sweden Central
        - Switzerland North
        - UK South
    - **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

Azure OpenAI provides a web-based portal named **Azure OpenAI Studio**, that you can use to deploy, manage, and explore models. You'll start your exploration of Azure OpenAI by using Azure OpenAI Studio to deploy a model.

1. On the **Overview** page for your Azure OpenAI resource, use the **Go to Azure OpenAI Studio** button to open Azure OpenAI Studio in a new browser tab.
2. In Azure OpenAI Studio, on the **Deployments** page, view your existing model deployments. If you don't already have one, create a new deployment of the **gpt-35-turbo-16k** model with the following settings:
    - **Model**: gpt-35-turbo-16k *(if the 16k model isn't available, choose gpt-35-turbo)*
    - **Model version**: Auto-update to default
    - **Deployment name**: *A unique name of your choice. You'll use this name later in the lab.*
    - **Advanced options**
        - **Content filter**: Default
        - **Deployment type**: Standard
        - **Tokens per minute rate limit**: 5K*
        - **Enable dynamic quota**: Enabled

    * 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.

Prepare to develop an app in Visual Studio Code

You'll develop your Azure OpenAI app using Visual Studio Code. The code files for your app have been provided in a GitHub repo.

Tip: If you have already cloned the **mslearn-openai** repo, open it in Visual Studio code. Otherwise, follow these steps to clone it to your development environment.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/parveenkrraina/azure-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.

    Note: If Visual Studio Code shows you a pop-up message to prompt you to trust the code you are opening, click on **Yes, I trust the authors** option in the pop-up.

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**.

Configure your application

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

1. In Visual Studio Code, in the **Explorer** pane, browse to the **Labfiles/02-azure-openai-api** 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 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#:

   ```
   dotnet add package Azure.AI.OpenAI --version 1.0.0-beta.14
   ```

   Python:

   ```
   pip install openai==1.13.3
   ```

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 (available in the **Deployments** page in Azure OpenAI Studio).
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

   ```
       csharp
       // Add Azure OpenAI package
   using Azure.AI.OpenAI;
   ```

Python: test-openai-model.py

```
python
# Add Azure OpenAI package
from openai import AzureOpenAI
```

2. In the application code for your language, replace the comment ***Initialize the Azure OpenAI client...*** with the following code to initialize the client and define our system message.

   C#: Program.cs

```
csharp
// Initialize the Azure OpenAI client
OpenAIClient client = new OpenAIClient(new Uri(oaiEndpoint), new
AzureKeyCredential(oaiKey));

// System message to provide context to the model
string systemMessage = "I am a hiking enthusiast named Forest who helps
people discover hikes in their area. If no area is specified, I will
default to near Rainier National Park. I will then provide three
suggestions for nearby hikes that vary in length. I will also share an
interesting fact about the local nature on the hikes when making a
recommendation.";
```

   Python: test-openai-model.py

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

# Create a system message
system_message = """I am a hiking enthusiast named Forest who helps
people discover hikes in their area.
    If no area is specified, I will default to near Rainier National
Park.
    I will then provide three suggestions for nearby hikes that vary in
length.
    I will also share an interesting fact about the local nature on the
hikes when making a recommendation.
    """
```

3. Replace the comment ***Add code to send request...*** with the necessary code for building the request; specifying the various parameters for your model such as `messages` and `temperature`.

   C#: Program.cs

```
csharp
// Add code to send request...
// Build completion options object
```

```
    ChatCompletionsOptions chatCompletionsOptions = new
    ChatCompletionsOptions()
    {
        Messages =
        {
            new ChatRequestSystemMessage(systemMessage),
            new ChatRequestUserMessage(inputText),
        },
        MaxTokens = 400,
        Temperature = 0.7f,
        DeploymentName = oaiDeploymentName
    };

    // Send request to Azure OpenAI model
    ChatCompletions response =
    client.GetChatCompletions(chatCompletionsOptions);

    // Print the response
    string completion = response.Choices[0].Message.Content;
Console.WriteLine("Response: " + completion + "\n");
```

Python: test-openai-model.py

```
    python
    # Add code to send request...
    # Send request to Azure OpenAI model
    response = client.chat.completions.create(
        model=azure_oai_deployment,
        temperature=0.7,
        max_tokens=400,
        messages=[
            {"role": "system", "content": system_message},
            {"role": "user", "content": input_text}
        ]
    )
    generated_text = response.choices[0].message.content

    # Print the response
print("Response: " + generated_text + "\n")
```

4. Save the changes to your code file.

Test your application

Now that your app has been configured, run it to send your request to your model and observe the response.

1. 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 test-openai-model.py`

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

2. When prompted, enter the text `What hike should I do near Rainier?`.
3. Observe the output, taking note that the response follows the guidelines provided in the system message you added to the *messages* array.
4. Provide the prompt `Where should I hike near Boise? I'm looking for something of easy difficulty, between 2 to 3 miles, with moderate elevation gain.` and observe the output.
5. In the code file for your preferred language, change the *temperature* parameter value in your request to **1.0** and save the file.
6. Run the application again using the prompts above, and observe the output.

Increasing the temperature often causes the response to vary, even when provided the same text, due to the increased randomness. You can run it several times to see how the output may change. Try using different values for your temperature with the same input.

Maintain conversation history

In most real-world applications, the ability to reference previous parts of the conversation allows for a more realistic interaction with an AI agent. The Azure OpenAI API is stateless by design, but by providing a history of the conversation in your prompt you enable the AI model to reference past messages.

1. Run the app again and provide the prompt `Where is a good hike near Boise?`.
2. Observe the output, and then prompt `How difficult is the second hike you suggested?`.
3. The response from the model will likely indicate can't understand the hike you're referring to. To fix that, we can enable the model to have the past conversation messages for reference.
4. In your application, we need to add the previous prompt and response to the future prompt we are sending. Below the definition of the **system message**, add the following code.

   C#: Program.cs

   ```
   csharp
   // Initialize messages list
   var messagesList = new List<ChatRequestMessage>()
   {
       new ChatRequestSystemMessage(systemMessage),
   };
   ```

   Python: test-openai-model.py

   ```
   python
   # Initialize messages array
   messages_array = [{"role": "system", "content": system_message}]
   ```

5. Under the comment ***Add code to send request...***, replace all the code from the comment to the end of the **while** loop with the following code then save the file. The code is mostly the same, but now using the messages array to store the conversation history.

C#: Program.cs

```
csharp
// Add code to send request...
// Build completion options object
messagesList.Add(new ChatRequestUserMessage(inputText));

ChatCompletionsOptions chatCompletionsOptions = new
ChatCompletionsOptions()
{
    MaxTokens = 1200,
    Temperature = 0.7f,
    DeploymentName = oaiDeploymentName
};

// Add messages to the completion options
foreach (ChatRequestMessage chatMessage in messagesList)
{
    chatCompletionsOptions.Messages.Add(chatMessage);
}

// Send request to Azure OpenAI model
ChatCompletions response =
client.GetChatCompletions(chatCompletionsOptions);

// Return the response
string completion = response.Choices[0].Message.Content;

// Add generated text to messages list
messagesList.Add(new ChatRequestAssistantMessage(completion));

Console.WriteLine("Response: " + completion + "\n");
```

Python: test-openai-model.py

```
python
# Add code to send request...
# Send request to Azure OpenAI model
messages_array.append({"role": "user", "content": input_text})

response = client.chat.completions.create(
    model=azure_oai_deployment,
    temperature=0.7,
    max_tokens=1200,
    messages=messages_array
)
generated_text = response.choices[0].message.content
# Add generated text to messages array
messages_array.append({"role": "assistant", "content": generated_text})

# Print generated text
print("Summary: " + generated_text + "\n")
```

6. Save the file. In the code you added, notice we now append the previous input and response to the prompt array which allows the model to understand the history of our conversation.
7. In the terminal pane, enter the following command to run the application.

    - **C#**: `dotnet run`
    - **Python**: `python test-openai-model.py`
8. Run the app again and provide the prompt `Where is a good hike near Boise?`.
9. Observe the output, and then prompt `How difficult is the second hike you suggested?`.
10. You'll likely get a response about the second hike the model suggested, which provides a much more realistic conversation. You can ask additional follow up questions referencing previous answers, and each time the history provides context for the model to answer.