6.2 - Build natural language solutions with Azure OpenAI Service

Overview

This module provides engineers with the skills to begin building apps that integrate with the Azure OpenAI Service.

By the end of this module, you'll be able to:

- Integrate Azure OpenAI into your application
- Differentiate between different endpoints available to your application
- Generate completions to prompts using the REST API and language specific SDKs

Introduction

Azure OpenAI provides a platform for developers to add artificial intelligence functionality to their applications with the help of both language specific SDKs and REST APIs. The platform has various AI models available, each specializing in different tasks, which can be deployed through the Azure OpenAI Service.

This module guides you through how to build Azure OpenAI into your own application, giving you a starting point for developing solutions with generative AI.

Integrate Azure OpenAI into your app

Azure OpenAI offers both language specific SDKs and a REST API that developers can use to add AI functionality to their applications. Generative AI capabilities in Azure OpenAI are provided through *models*. The models available in the Azure OpenAI service belong to different families, each with their own focus. To use one of these models, you need to deploy through the Azure OpenAI Service.

Create an Azure OpenAI resource

An Azure OpenAI resource can be deployed through both the Azure command line interface (CLI) and the Azure portal. Creating the Azure OpenAI resource through the Azure portal is similar to deploying individual Azure AI Services resources, and is part of the Azure AI Services services.

1. Navigate to the Azure portal
2. Search for **Azure OpenAI**, select it, and click **Create**
3. Enter the appropriate values for the empty fields, and create the resource.

The possible regions for Azure OpenAI are currently limited. Choose the region closest to your physical location, or the closest one that has the availability for the model(s) you want to use.

Once the resource has been created, you'll have keys and an endpoint that you can use in your app.

Choose and deploy a model

Each model family excels at different tasks, and there are different capabilities of the models within each family. Model families break down into three main families:

- **Generative Pre-trained Transformer (GPT)** - Models that understand and generate natural language and some code. These models are best at general tasks, conversations, and chat formats.
- **Code** (`gpt-3` and earlier) - Code models are built on top of GPT models, and trained on millions of lines of code. These models can understand and generate code, including interpreting comments or natural language to generate code. `gpt-35-turbo` and later models have this code functionality included without the need for a separate code model.
- **Embeddings** - These models can understand and use embeddings, which are a special format of data that can be used by machine learning models and algorithms.

This module focuses on general GPT models, with other models being covered in other modules.

For older models, the model family and capability is indicated in the name of the base model, such as `text-davinci-003` , which specifies that it's a text model, with `davinci` level capability, and identifier `3` . Details on models, capability levels, and naming conventions can be found on the Azure OpenAI Models documentation page.

More recent models specify which `gpt` generation, and if they are the `turbo` version, such as `gpt-35-turbo` representing the *GPT 3.5 Turbo* model.

To deploy a model for you to use, navigate to the Azure AI Studio and go to the **Deployments** page. The lab later in this module covers exactly how to do that.

Authentication and specification of deployed model

When you deploy a model in Azure OpenAI, you choose a deployment name to give it. When configuring your app, you need to specify your resource endpoint, key, and deployment name to specify which deploy model to send your request to. This enables you to deploy various models within the same resource, and make requests to the appropriate model depending on the task.

Prompt engineering

How the input prompt is written plays a large part in how the AI model will respond. For example, if prompted with a simple request such as "What is Azure OpenAI", you often get a generic answer similar to using a search engine.

However, if you give it more details about what you want in your response, you get a more specific answer. For example, given this prompt:

```
Classify the following news headline into 1 of the following categories: Business, Tech,
Politics, Sport, Entertainment

Headline 1: Donna Steffensen Is Cooking Up a New Kind of Perfection. The Internet's most
beloved cooking guru has a buzzy new book and a fresh new perspective
Category: Entertainment

Headline 2: Major Retailer Announces Plans to Close Over 100 Stores
Category:
```

You'll likely get the "Category:" under headline filled out with "Business".

Several examples similar to this one can be found in the Azure AI Studio Playground, under the **Prompt samples** dropdown. Try to be as specific as possible about what you want in response from the model, and you may be surprised at how insightful it can be!

> Note: It is never safe to assume that answers from an AI model are factual or correct. Teams or individuals tasked with developing and deploying AI systems should work to identify, measure, and mitigate harm. It is your responsibility to verify any responses from an AI model, and to use AI responsibly. Check out Microsoft's Transparency Notes on Azure OpenAI for further guidelines on how to use Azure OpenAI models responsibly.

Further details can be found at the Prompt engineering documentation page.

Available endpoints

Azure OpenAI can be accessed via a REST API or an SDK available for Python, C#, JavaScript, and more. The endpoints available for interacting with a deployed model are used differently, and certain endpoints can only use certain models. The available endpoints are:

- **Completion** - model takes an input prompt, and generates one or more predicted completions. You'll see this playground in the studio, but won't be covered in depth in this module.
- **ChatCompletion** - model takes input in the form of a chat conversation (where roles are specified with the message they send), and the next chat completion is generated.
- **Embeddings** - model takes input and returns a vector representation of that input.

For example, the input for `ChatCompletion` is a conversation with clearly defined roles for each message:

```
{"role": "system", "content": "You are a helpful assistant, teaching people about AI."},
{"role": "user", "content": "Does Azure OpenAI support multiple languages?"},
{"role": "assistant", "content": "Yes, Azure OpenAI supports several languages, and can
translate between them."},
{"role": "user", "content": "Do other Azure AI Services support translation too?"}
```

When you give the AI model a real conversation, it can generate a better response with more accurate tone, phrasing, and context. The `ChatCompletion` endpoint enables the ChatGPT model to have a more realistic conversation by sending the history of the chat with the next user message.

`ChatCompletion` also allows for non-chat scenarios, such as summarization or entity extraction. This can be accomplished by providing a short conversation, specifying the system information and what you want, along with the user input. For example, if you want to generate a job description, provide `ChatCompletion` with something like the following conversation input.

```
{"role": "system", "content": "You are an assistant designed to write intriguing job
descriptions. "},
{"role": "user", "content": "Write a job description for the following job title:
'Business Intelligence Analyst'. It should include responsibilities, required
qualifications, and highlight benefits like time off and flexible hours."}
```

> Note: `Completion` is available for all `gpt-3` generation models, while `ChatCompletion` is the only supported option for `gpt-4` models and is the preferred endpoint when using the `gpt-35-turbo` model. The lab in this module uses `gpt-35-turbo` with the `ChatCompletion` endpoint.

Use Azure OpenAI REST API

Azure OpenAI offers a REST API for interacting and generating responses that developers can use to add AI functionality to their applications. This unit covers example usage, input and output from the API.

> Note: Before interacting with the API, you must create an Azure OpenAI resource in the Azure portal, deploy a model in that resource, and retrieve your endpoint and keys. Check out the Getting started with Azure OpenAI Service to learn how to do that.

For each call to the REST API, you need the **endpoint and a key** from your Azure OpenAI resource, and the **name you gave for your deployed model.** In the following examples, the following placeholders are used:

Placeholder name	Value
`YOUR_ENDPOINT_NAME`	This base endpoint is found in the **Keys & Endpoint** section in the Azure portal. It's the base endpoint of your resource, such

Placeholder name	Value
	as `https://sample.openai.azure.com/`.
`YOUR_API_KEY`	Keys are found in the **Keys & Endpoint** section in the Azure portal. You can use either key for your resource.
`YOUR_DEPLOYMENT_NAME`	This deployment name is the name provided when you deployed your model in the Azure AI Studio.

Chat completions

Once you've deployed a model in your Azure OpenAI resource, you can send a prompt to the service using a `POST` request.

```
curl
https://YOUR_ENDPOINT_NAME.openai.azure.com/openai/deployments/YOUR_DEPLOYMENT_NAME/chat
/completions?api-version=2023-03-15-preview \
  -H "Content-Type: application/json" \
  -H "api-key: YOUR_API_KEY" \
  -d '{"messages":[{"role": "system", "content": "You are a helpful assistant, teaching
people about AI."},
{"role": "user", "content": "Does Azure OpenAI support multiple languages?"},
{"role": "assistant", "content": "Yes, Azure OpenAI supports several languages, and can
translate between them."},
{"role": "user", "content": "Do other Azure AI Services support translation too?"}]}'
```

The response from the API will be similar to the following JSON:

```
{
    "id": "chatcmpl-6v7mkQj980V1yBec6ETrKPRqFjNw9",
    "object": "chat.completion",
    "created": 1679001781,
    "model": "gpt-35-turbo",
    "usage": {
        "prompt_tokens": 95,
        "completion_tokens": 84,
        "total_tokens": 179
    },
    "choices": [
        {
            "message":
                {
                    "role": "assistant",
                    "content": "Yes, other Azure AI Services also support translation.
Azure AI Services offer translation between multiple languages for text, documents, or
custom translation through Azure AI Services Translator."
                },
            "finish_reason": "stop",
            "index": 0
        }
    ]
}
```

REST endpoints allow for specifying other optional input parameters, such as `temperature`, `max_tokens` and more. If you'd like to include any of those parameters in your request, add them to the input data with the request.

Embeddings

Embeddings are helpful for specific formats that are easily consumed by machine learning models. To generate embeddings from the input text, `POST` a request to the `embeddings` endpoint.

```
curl
https://YOUR_ENDPOINT_NAME.openai.azure.com/openai/deployments/YOUR_DEPLOYMENT_NAME/embeddings?api-version=2022-12-01 \
  -H "Content-Type: application/json" \
  -H "api-key: YOUR_API_KEY" \
  -d "{\"input\": \"The food was delicious and the waiter...\"}"
```

When generating embeddings, be sure to use a model in Azure OpenAI meant for embeddings. Those models start with `text-embedding` or `text-similarity`, depending on what functionality you're looking for.

The response from the API will be similar to the following JSON:

```
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "embedding": [
        0.0172990688066482523,
        -0.0291879814639389515,
        ....,
        0.0134544348834753042
      ],
      "index": 0
    }
  ],
  "model": "text-embedding-ada:002"
}
```

Use Azure OpenAI SDK

In addition to REST APIs covered in the previous unit, users can also access Azure OpenAI models through C# and Python SDKs. The same functionality is available through both REST and these SDKs.

> Note: Before interacting with the API using either SDK, you must create an Azure OpenAI resource in the Azure portal, deploy a model in that resource, and retrieve your endpoint and keys. Check out the Getting started with Azure OpenAI Service to learn how to do that.

For both SDKs covered in this unit, you need the endpoint and a key from your Azure OpenAI resource, and the name you gave for your deployed model. In the following code snippets, the following placeholders are used:

Placeholder name	Value
`YOUR_ENDPOINT_NAME`	This base endpoint is found in the **Keys & Endpoint** section in the Azure portal. It's the base endpoint of your resource, such as `https://sample.openai.azure.com/`.
`YOUR_API_KEY`	Keys are found in the **Keys & Endpoint** section in the Azure portal. You can use either key for your resource.
`YOUR_DEPLOYMENT_NAME`	This deployment name is the name provided when you deployed your model in the Azure AI Studio.

Install libraries

First, install the client library for your preferred language. The C# SDK is a .NET adaptation of the REST APIs and built specifically for Azure OpenAI, however it can be used to connect to Azure OpenAI resources or non-Azure OpenAI endpoints. The Python SDK is built and maintained by OpenAI: `pip install openai`

Configure app to access Azure OpenAI resource

Configuration for each language varies slightly, but both require the same parameters to be set. The necessary parameters are `endpoint`, `key`, and the name of your deployment, which is called the `engine` when sending your prompt to the model.

Add the library to your app, and set the required parameters for your client.

```
# Add OpenAI library
from openai import AzureOpenAI

deployment_name = '<YOUR_DEPLOYMENT_NAME>'

# Initialize the Azure OpenAI client
client = AzureOpenAI(
        azure_endpoint = '<YOUR_ENDPOINT_NAME>',
        api_key='<YOUR_API_KEY>',
        api_version="20xx-xx-xx" #  Target version of the API, such as 2024-02-15-
preview
        )
```

Call Azure OpenAI resource

Once you've configured your connection to Azure OpenAI, send your prompt to the model.

```
response = client.chat.completions.create(
    model=deployment_name,
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "What is Azure OpenAI?"}
    ]
)
generated_text = response.choices[0].message.content
```

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

The response object contains several values, such as `total_tokens` and `finish_reason`. The completion from the response object will be similar to the following completion:

```
"Azure OpenAI is a cloud-based artificial intelligence (AI) service that offers a range
of tools and services for developing and deploying AI applications. Azure OpenAI
provides a variety of services for training and deploying machine learning models,
including a managed service for training and deploying deep learning models, a managed
service for deploying machine learning models, and a managed service for managing and
deploying machine learning models."
```

In both C# and Python, your call can include optional parameters including `temperature` and `max_tokens`. Examples of using those parameters are included in this module's lab.

Exercise - Integrate Azure OpenAI into 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.

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

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

> **Note**: As you use Azure AI Foundry portal, message boxes suggesting tasks for you to perform may be displayed. You can close these and follow the steps in this exercise.

1. In the Azure portal, on the **Overview** page for your Azure OpenAI resource, scroll down to the **Get Started** section and select the button to go to **AI Foundry portal** (previously AI Studio).
2. In Azure AI Foundry portal, in the pane on the left, select the **Deployments** page and 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:
    - **Deployment name**: *A unique name of your choice*
    - **Model**: gpt-35-turbo-16k *(if the 16k model isn't available, choose gpt-35-turbo)*
    - **Model version**: *Use default version*
    - **Deployment type**: Standard
    - **Tokens per minute rate limit**: 5K*
    - **Content filter**: Default
    - **Enable dynamic quota**: Disabled

    > 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/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.

   > **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==1.55.3
   ```

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 AI Foundry portal).
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
   `// 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
    `// 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
   `// 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
# 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}
    ]
)

# Get the generated text from the response
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
   `// Initialize messages list var messagesList = new List<ChatRequestMessage>() { new ChatRequestSystemMessage(systemMessage), };`

   Python: test-openai-model.py

   ```
   # 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
   `// 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

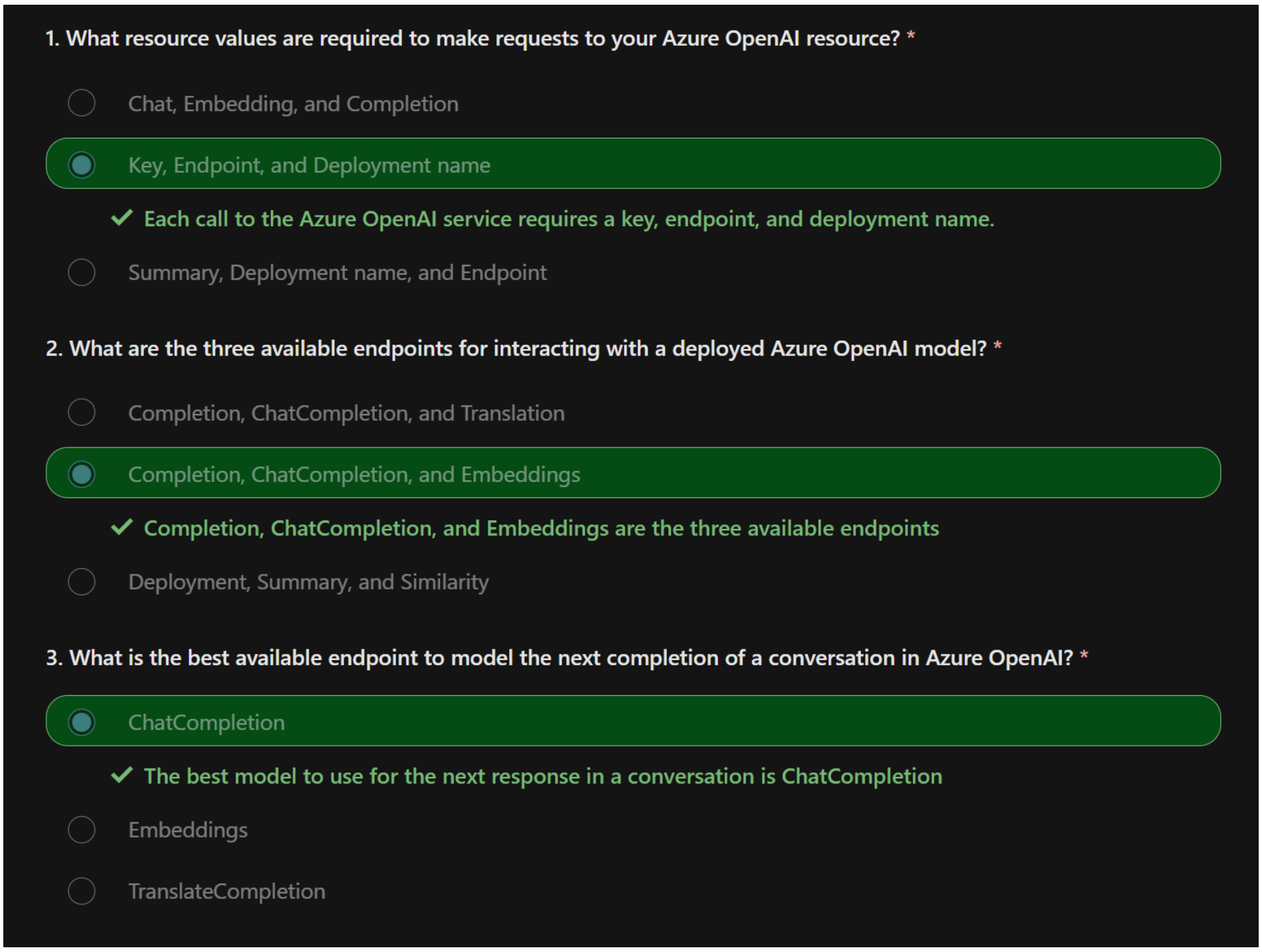
   ```
   # Add code to send request...

   # Append user input to messages array
   messages_array.append({"role": "user", "content": input_text})

   # Send request to Azure OpenAI model
   response = client.chat.completions.create(
       model=azure_oai_deployment,
       temperature=0.7,
       max_tokens=1200,
       messages=messages_array
   ```

```
)

# Get the generated text from the response
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.

> **Tip**: The token count is only set to 1200, so if the conversation continues too long the application will run out of available tokens, resulting in an incomplete prompt. In production uses, limiting the length of the history to the most recent inputs and responses will help control the number of required tokens.

Knowledge Check


1. What resource values are required to make requests to your Azure OpenAI resource? *

- Chat, Embedding, and Completion
- [✓ Key, Endpoint, and Deployment name]
  ✓ Each call to the Azure OpenAI service requires a key, endpoint, and deployment name.
- Summary, Deployment name, and Endpoint

2. What are the three available endpoints for interacting with a deployed Azure OpenAI model? *

- Completion, ChatCompletion, and Translation
- [✓ Completion, ChatCompletion, and Embeddings]
  ✓ Completion, ChatCompletion, and Embeddings are the three available endpoints
- Deployment, Summary, and Similarity

3. What is the best available endpoint to model the next completion of a conversation in Azure OpenAI? *

- [✓ ChatCompletion]
  ✓ The best model to use for the next response in a conversation is ChatCompletion
- Embeddings
- TranslateCompletion

Summary

Azure OpenAI offers models for text and embeddings, available through REST API or SDKs. This module covered how to use the models in your own application. By using the specific model deployed in your resource, you can utilize Azure OpenAI generative AI models to add intelligence to your application.

In this module, you learned how to:

- Integrate Azure OpenAI into your application
- Differentiate between different endpoints available to your application
- Generate completions to prompts using the REST API and language specific SDKs

For more detail on integrating Azure OpenAI into your application, check out the quick-starts on the Azure OpenAI Service documentation site.

✍️ Compiled by Kenneth Leung (2025)