**Develop applications with Azure OpenAI Service**

**Introduction**

Suppose you want to build a support application that summarizes text and suggest code. To build this app, you want to utilize the capabilities you see in ChatGPT, a chatbot built by the OpenAI research company that takes in natural language input from a user and returns a machine-created, human-like response.

Generative AI models power ChatGPT's ability to produce new content, such as text, code, and images, based on natural language prompts. Many generative AI models are a subset of [deep learning algorithms](https://learn.microsoft.com/en-us/dotnet/machine-learning/deep-learning-overview). These algorithms support various workloads across vision, speech, language, decision, search, and more.

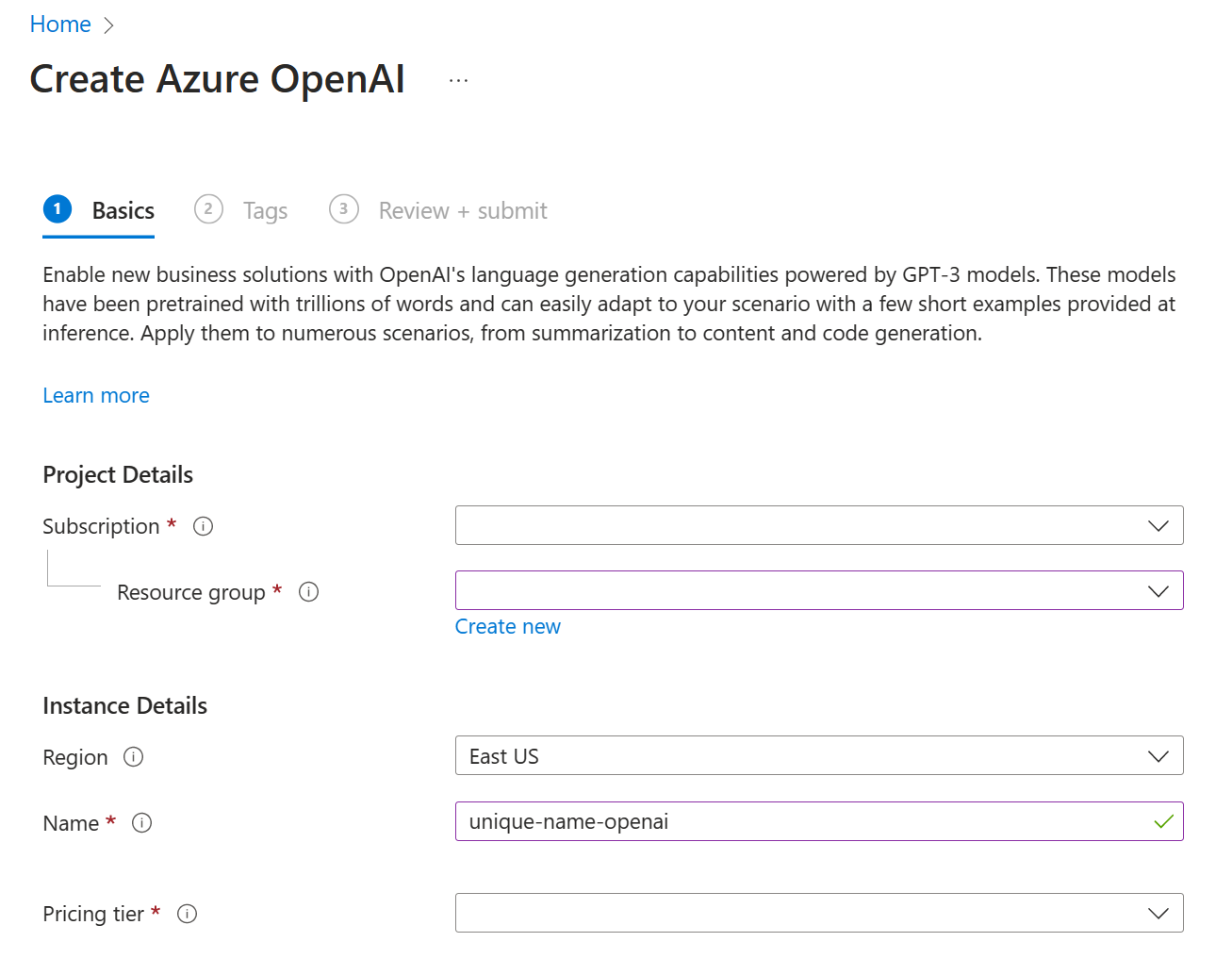
Azure OpenAI Service brings these generative AI models to the Azure platform, enabling you to develop powerful AI solutions that benefit from the security, scalability, and integration of other services provided by the Azure cloud platform. These models are available for building applications through a REST API, various SDKs, and a Studio interface. This module guides you through the Azure AI Foundry experience, giving you the foundation to further develop solutions with generative AI.

**Access Azure OpenAI Service**

The first step in building a generative AI solution with Azure OpenAI is to provision an Azure OpenAI resource in your Azure subscription. You can get started by creating a resource in the [Azure portal](https://portal.azure.com/) or with the Azure command line interface (CLI).

**Create an Azure OpenAI Service resource in the Azure portal**

When you create an Azure OpenAI Service resource, you need to provide a subscription name, resource group name, region, unique instance name, and select a pricing tier.



**Create an Azure OpenAI Service resource in Azure CLI**

To create an Azure OpenAI Service resource from the CLI, refer to this example and replace the following variables with your own:

* MyOpenAIResource: *replace with a unique name for your resource*
* OAIResourceGroup: *replace with your resource group name*
* eastus: *replace with the region to deploy your resource*
* subscriptionID: *replace with your subscription ID*

az cognitiveservices account create \

-n MyOpenAIResource \

-g OAIResourceGroup \

-l eastus \

--kind OpenAI \

--sku s0 \

--subscription subscriptionID

**Note**

You can find the regions available for a service through the CLI command az account list-locations. To see how to sign into Azure and create an Azure group via the CLI, you can refer to the [**documentation here**](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/how-to/create-resource?pivots=cli#sign-in-to-the-cli?azure-portal=true). Azure OpenAI Service provides access to many types of models. Certain models are only available in select regions. Consult the [**Azure OpenAI model availability guide**](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/models#model-summary-table-and-region-availability/?azure-portal=true) for region availability. You can create two Azure OpenAI resources per region.

**Use Azure OpenAI Foundry**

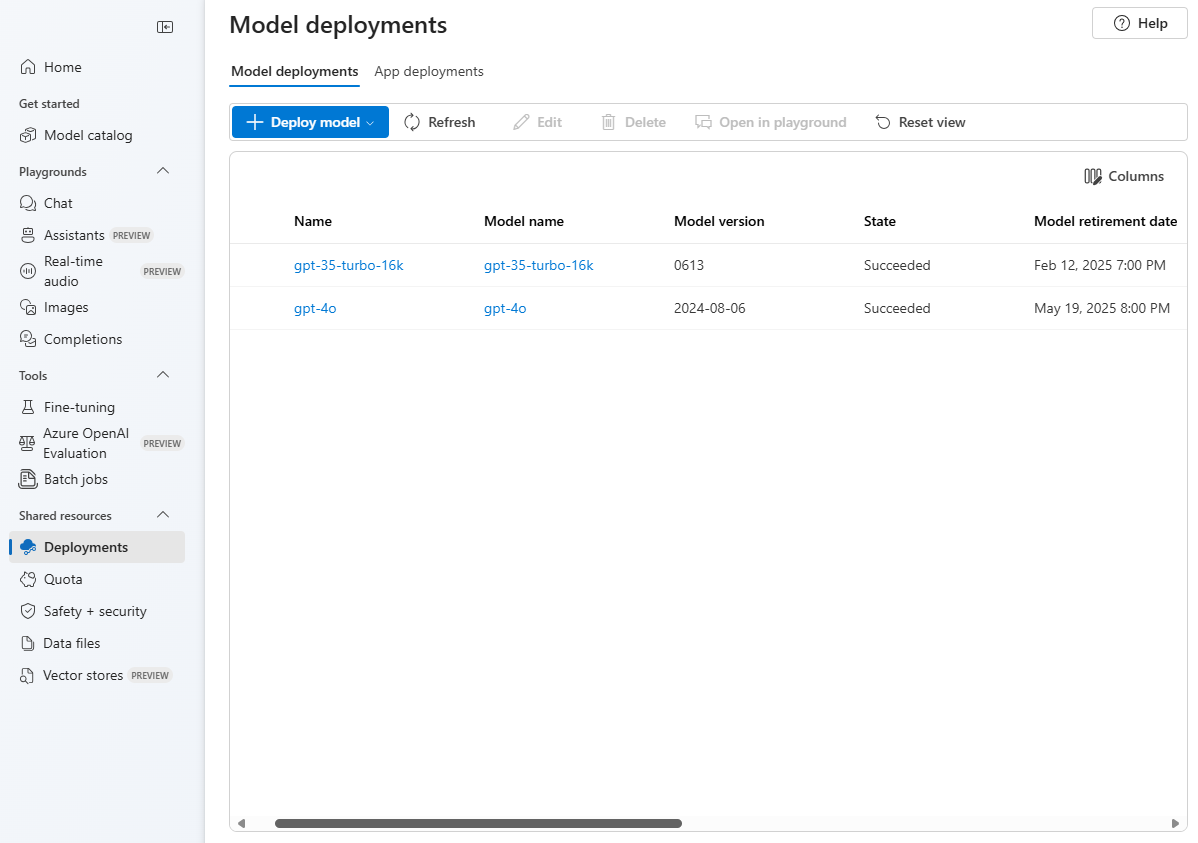
Azure AI Foundry provides access to model management, deployment, experimentation, customization, and learning resources.

You can access the Azure AI Foundry through the Azure portal after creating a resource, or at <https://ai.azure.com/> by signing in to your Azure account. During the sign-in workflow, select the appropriate directory, Azure subscription, and Azure OpenAI resource.

When you first open Azure AI Foundry, you'll want to navigate to the **Azure OpenAI** page (where you focus on only Azure OpenAI Service models), select your resource if you haven't already, and deploy your first model. To do so, select the **Deployments** page, from where you can deploy a base model and start experimenting with it.

**Note**

If you are not the resource owner, you will need the following role-based access controls: 1. *Cognitive Services OpenAI User*: This role allows viewing resources and using the chat playground. 2. *Cognitive Services OpenAI Contributor*: This role allows the user to create new deployments.



**Types of OpenAI models**

To begin building with Azure OpenAI, you need to choose a base model and deploy it. Microsoft provides base models and the option to create customized base models. This unit covers the currently available base models.

Azure OpenAI includes several types of model:

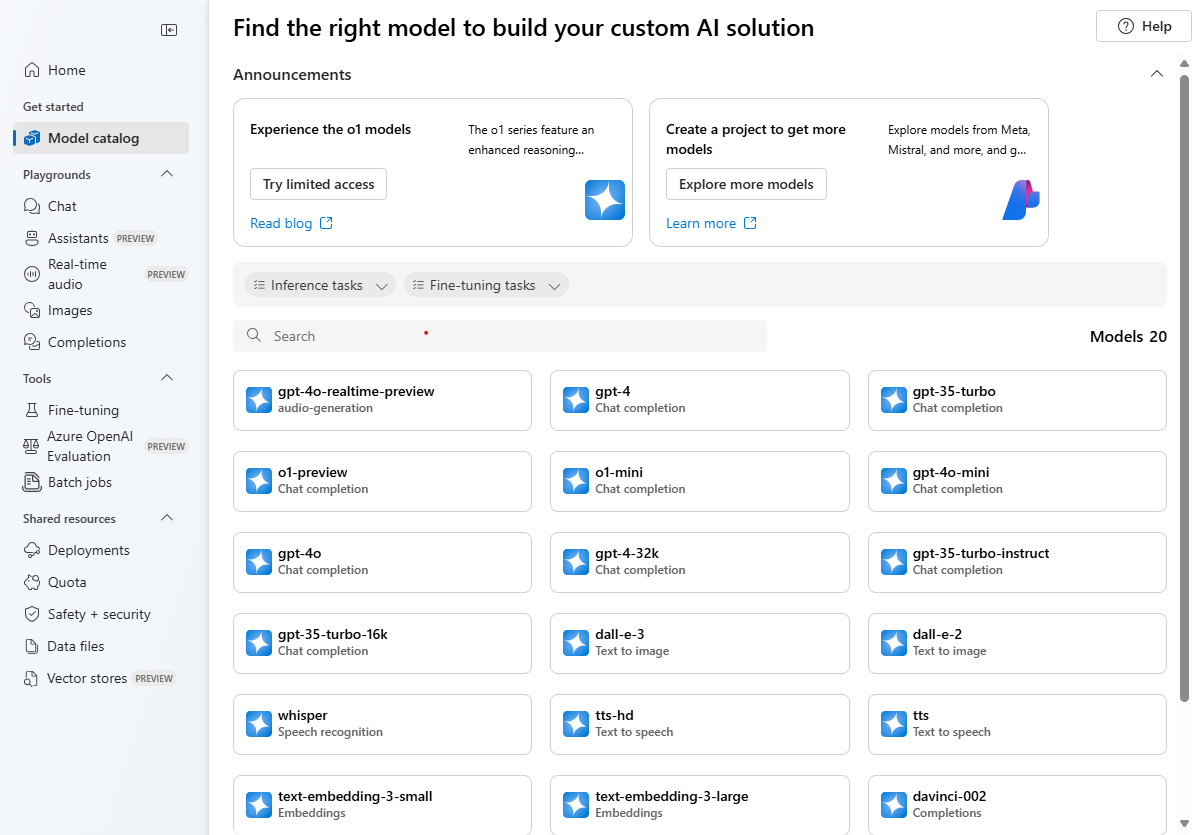
* **GPT-4 models** are the latest generation of *generative pretrained* (GPT) models that can generate natural language and code completions based on natural language prompts.
* **GPT 3.5 models** can generate natural language and code completions based on natural language prompts. In particular, **GPT-35-turbo** models are optimized for chat-based interactions and work well in most generative AI scenarios.
* **Embeddings models** convert text into numeric vectors, and are useful in language analytics scenarios such as comparing text sources for similarities.
* **DALL-E models** are used to generate images based on natural language prompts. Currently, DALL-E models are in preview.
* **Whisper models** are used to convert speech to text.
* **Text to speech models** are used to convert text to speech.

**Note**

Pricing is determined by tokens and by model type. Learn more about the latest [**pricing here**](https://azure.microsoft.com/pricing/details/cognitive-services/openai-service/).

In the Azure AI Foundry portal, the **Model Catalog** page lists the available base models and provides an option to create additional customized models by fine-tuning the base models. The models that have a *Succeeded* status mean they're successfully trained and can be selected for deployment.

You'll notice that there are various models beyond OpenAI available in the Model Catalog, including models from Microsoft, Meta, Mistral, and more. Azure AI Foundry enables you to deploy any of these models for your use case. This module will focus on Azure OpenAI models.



**Deploying OpenAI models**

You first need to deploy a model to chat with or make API calls to receive responses to prompts. When you create a new deployment, you need to indicate which base model to deploy. You can deploy any number of deployments as long as their Tokens Per Minute (TPM) stays within the deployment quota.

**Deploy using Azure AI Foundry**

In Azure AI Foundry portal's **Model catalog** page, you can create a new deployment by selecting a model name from the list.

**Deploy using Azure CLI**

You can also deploy a model using the console. Using this example, replace the following variables with your own resource values:

* OAIResourceGroup: *replace with your resource group name*
* MyOpenAIResource: *replace with your resource name*
* MyModel: *replace with a unique name for your model*
* gpt-35-turbo: *replace with the base model you wish to deploy*

az cognitiveservices account deployment create \

-g OAIResourceGroup \

-n MyOpenAIResource \

--deployment-name MyModel \

--model-name gpt-35-turbo \

--model-version "0125" \

--model-format OpenAI \

--sku-name "Standard" \

--sku-capacity 1

**Deploy using the REST API**

You can deploy a model using the REST API. In the request body, you specify the base model you wish to deploy. See an example in the [Azure OpenAI documentation](https://learn.microsoft.com/en-us/azure/ai-services/openai/).

**Explore OpenAI prompts**

Once the model is deployed, you can test how it completes prompts. A prompt is the text portion of a request that is sent to the deployed model's completions endpoint. Responses are referred to as *completions*, which can come in form of text, code, or other formats.

**Prompt types**

Prompts can be grouped into types of requests based on task.

| **Task type** | **Prompt example** | **Completion example** |
| --- | --- | --- |
| **Classifying content** | Tweet: I enjoyed the trip. Sentiment: | Positive |
| **Generating new content** | List ways of traveling | 1. Bike 2. Car ... |
| **Holding a conversation** | A friendly AI assistant | [See examples](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/how-to/completions#conversation?portal=true) |
| **Transformation** (translation and symbol conversion) | English: Hello French: | bonjour |
| **Summarizing content** | Provide a summary of the content {text} | The content shares methods of machine learning. |
| **Picking up where you left off** | One way to grow tomatoes | is to plant seeds. |
| **Giving factual responses** | How many moons does Earth have? | One |

**Completion quality**

Several factors affect the quality of completions you'll get from a generative AI solution.

* The way a prompt is engineered. Learn more about prompt engineering in the *Prompt engineering* unit, later in this module.
* The model parameters (covered below).
* The data the model is trained on, which can be adapted through [model fine-tuning with customization](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/how-to/fine-tuning?pivots=programming-language-studio?portal=true).

You have more control over the completions returned by training a custom model than through prompt engineering and parameter adjustment.

**Making API calls**

You can start making calls to your deployed model via the REST API, Python, or C#. If your deployed model has a GPT-3.5 or GPT-4 model base, use the [Chat completions documentation](https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chatgpt), which has endpoints and variables required for these base models.

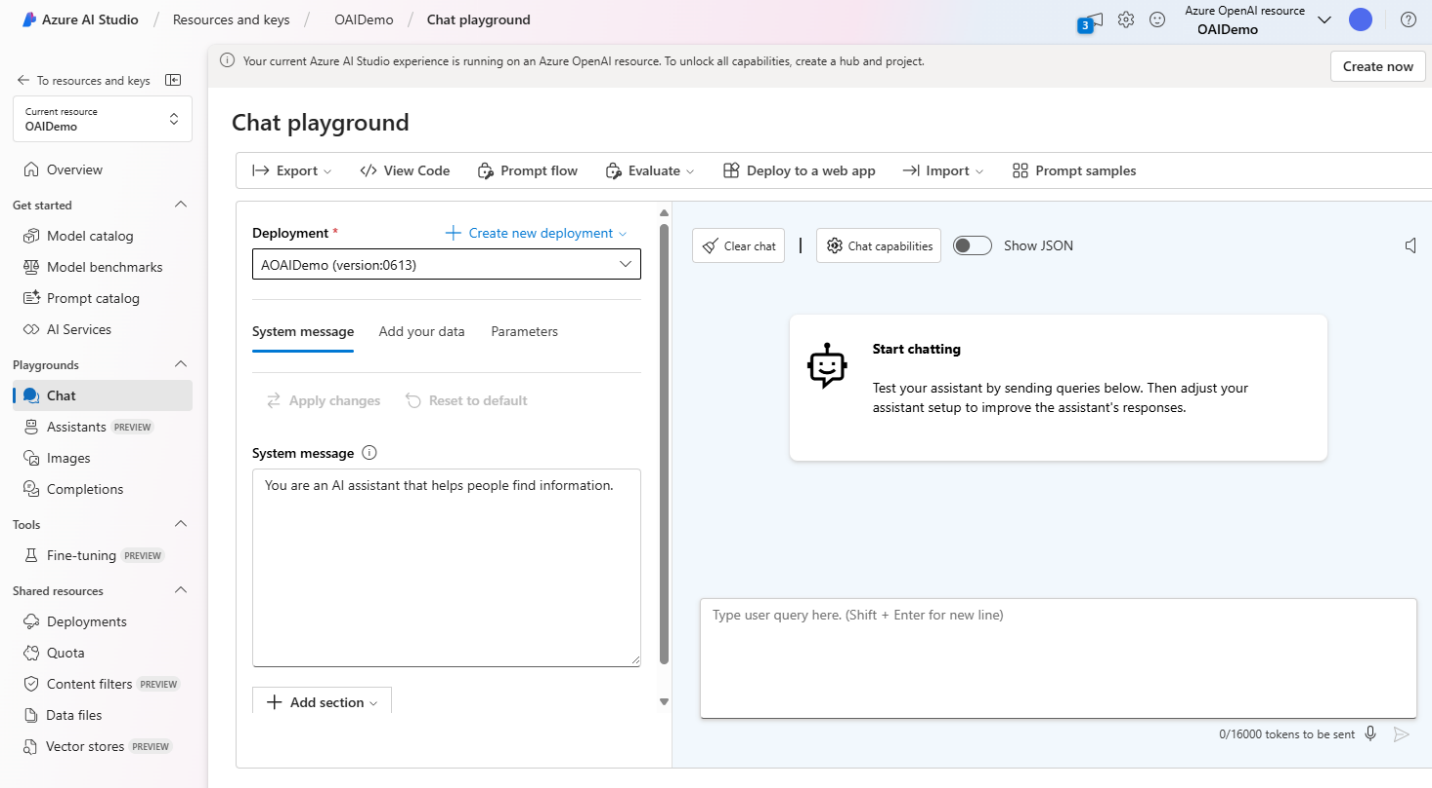
**Use the Azure Studio playgrounds**

Playgrounds are useful interfaces in Azure AI Foundry that you can use to experiment with your deployed models without needing to develop your own client application. Azure AI Foundry portal offers multiple playgrounds with different parameter tuning options.

**Chat playground**

The Chat playground is based on a conversation-in, message-out interface. You can initialize the session with a system message to set up the chat context.

In the Chat playground, you're able to use prompt samples, adjust parameters, and add *few-shot examples*. The term few-shot refers to providing a few of examples to help the model learn what it needs to do. You can think of it in contrast to zero-shot, which refers to providing no examples.



**Integrate OpenAI into an 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.

Once you have created an Azure OpenAI resource and deployed a model, you can configure your app.

**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 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 earlier 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.

**Use the 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.

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 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 Foundry. |

**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 with SDKs**

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

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. |

**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 deployment name.

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.

**OpenAI prompt engineering**

Completed100 XP

* 6 minutes

The quality of the input prompts we send to an AI model, like those available in Azure OpenAI, directly influences the quality of what we get back. By carefully constructing the prompts we send to the model, the model can provide better and more interesting responses.

**What is prompt engineering**

Prompt engineering is the process of designing and optimizing prompts to better utilize AI models. Designing effective prompts is critical to the success of prompt engineering, and it can significantly improve the AI model's performance on specific tasks. Providing relevant, specific, unambiguous, and well structured prompts can help the model better understand the context and generate more accurate responses.

For example, if we want an OpenAI model to generate product descriptions, we can provide it with a detailed description that describes the features and benefits of the product. By providing this context, the model can generate more accurate and relevant product descriptions.

Prompt engineering can also help mitigate bias and improve fairness in AI models. By designing prompts that are diverse and inclusive, we can ensure that the model isn't biased towards a particular group or perspective.

**Important**

No matter how good of a prompt you can design, responses from AI models should never be taken as fact or completely free from bias. Always use AI responsibly. For more information, see Microsoft's [**transparency note**](https://learn.microsoft.com/en-us/legal/cognitive-services/openai/transparency-note) on Azure OpenAI and [**Microsoft's AI principles**](https://www.microsoft.com/ai/responsible-ai).

In addition, prompt engineering can help us understand which references the model uses to generate its response. Generative AI models have a ton of parameters and the logic it follows is largely unknown to users, so it can be confusing how it arrives at the response it gives. By designing prompts that are easy to understand and interpret, we can help humans better understand how the model is generating its responses. This can be particularly important in domains such as healthcare, where it's critical to understand how the model is making decisions.

There are different methods to utilize when engineering your own prompts. These include providing instructions, contextual content, cues or few-shot examples, and correctly ordering content in your prompt. The methods covered here aren't exhaustive as this area is a nuanced and fluid topic.

**Considerations for API endpoints**

The examples in the rest of this unit will be focused on ChatCompletion. It's worth noting that ChatCompletion can also be used for non chat scenarios, where any instructions are included in the system message and user content is provided in the user role message. Most of these examples can be altered to use the Completion endpoint, if desired.

In terms of model availability, the Completion endpoint can be used with gpt-3 and earlier, and ChatCompletion can be used with gpt-35-turbo and later models.

**Adjusting model parameters**

In addition to techniques discussed in this unit, adjusting parameters of the model can have a significant impact on the response. In particular, temperature and top\_p (top\_probability) are the most likely to impact a model's response as they both control randomness in the model, but in different ways.

Higher values produce more creative and random responses, but will likely be less consistent or focused. Responses expected to be fictional or unique benefit from higher values for these parameters, whereas content desired to be more consistent and concrete should use lower values.

Specifically, high temperature allows for more variation in sentence structure and high top\_p allows for more variation in words that are used (using a variety of synonyms).

Try adjusting these parameters with the same prompt to see how they impact the response. It's recommended to change either temperature or top\_p at a time, but not both.

**Write effective prompts**

Azure OpenAI models are capable of generating responses to natural language queries with remarkable accuracy. However, the quality of the responses depends largely on how well the prompt is written. Developers can optimize the performance of Azure OpenAI models by using different techniques in their prompts, resulting in more accurate and relevant responses.

**Provide clear instructions**

Asking the Azure OpenAI model clearly for what you want is one way to get desired results. By being as descriptive as possible, the model can generate a response that most closely matches what you're looking for.

For example, say you want to create a product description for a new water bottle. Consider the answer associated with each prompt:

**Prompt**

code

Write a product description for a new water bottle

**Response**

code

Introducing the latest addition to our product line - the innovative and eco-friendly water bottle.

Made from high-quality, BPA-free materials, this bottle is designed to keep your beverages at the

perfect temperature for hours. Its sleek and stylish design makes it the perfect accessory for any

outdoor activity or workout session. With its convenient carrying handle and leak-proof lid, you

can take it with you wherever you go, without worrying about spills or leaks. Plus, its easy-to-clean

design ensures that you can use it again and again, making it the perfect choice for anyone looking

for a reliable and sustainable water bottle.

Versus a similar prompt, with clear instructions.

**Prompt**

code

Write a product description for a new water bottle that is 100% recycled. Be sure to include that

it comes in natural colors with no dyes, and each purchase removes 10 pounds of plastic from our

oceans

**Response**

code

Introducing our newest water bottle, the eco-friendly choice for the conscious consumer. Made from

100% recycled materials and available in natural colors with no dyes, this bottle is not only

stylish but also sustainable. Each purchase of this bottle removes 10 pounds of plastic from

our oceans, making it a responsible choice for those who care about the environment. Its durable

and leak-proof design makes it perfect for outdoor activities, while its easy-to-carry handle

makes it convenient to take on the go. Join us in our mission to reduce plastic waste and choose

this eco-friendly water bottle for your daily hydration needs.

Including specifics that need to be included lets the model know what information to add to the description, making it more accurate to the new product.

This method can be extrapolated to include complex instructions, such as a bulleted list of details to include, length of response, or desired formats to be included in the output. Try asking for exactly what you want to see in the result, and you may be surprised at how well the model satisfies your request.

**Format of instructions**

How instructions are formatted can impact how the model interprets the prompt. Recency bias can affect models, where information located towards the end of the prompt can have more influence on the output than information at the beginning. You may get better responses by repeating the instructions at the end of the prompt and assessing how that affects the generated response.

This recency bias can also come into play when using ChatCompletion in a chat scenario, where more recent messages in the conversation included in the prompt have a greater impact on the response.

**Primary, supporting, and grounding content**

Including content for the model to use in its response allows the model to answer with greater accuracy. This content can be thought of in two ways: primary and supporting content.

Primary content refers to content that is the subject of the query, such as a sentence to translate or an article to summarize. This content is often included at the beginning or end of the prompt (as an instruction and differentiated by --- blocks), with instructions explaining what to do with it.

For example, say we have a long article that we want to summarize. We could put it in a --- block in the prompt, then end with the instruction.

---

<insert full article here, as primary content>

---

Summarize this article and identify three takeaways in a bulleted list

Supporting content is content that may alter the response, but isn't the focus or subject of the prompt. Examples of supporting content include things like names, preferences, future date to include in the response, and so on. Providing supporting content allows the model to respond more completely, accurately, and be more likely to include the desired information.

For example, given a very long promotional email, the model is able to extract key information. If you then add supporting content to the prompt specifying something specific you're looking for, the model can provide a more useful response. In this case the email is the primary content, with the specifics of what you're interested in as the supporting content

---

<insert full email here, as primary content>

---

<the next line is the supporting content>

Topics I'm very interested in: AI, webinar dates, submission deadlines

Extract the key points from the above email, and put them in a bulleted list:

Grounding content allows the model to provide reliable answers by providing content for the model to draw answers from. Grounding content could be an essay or article that you then ask questions about, a company FAQ document, or information that is more recent than the data the model was trained on. If you need more reliable and current responses, or you need to reference unpublished or specific information, grounding content is highly recommended.

Grounding content differs from primary content as it's the source of information to answer the prompt query, instead of the content being operated on for things like summarization or translation. For example, when provided an unpublished research paper on the history of AI, it can then answer questions using that grounding content.

---

<insert unpublished paper on the history of AI here, as grounding content>

---

Where and when did the field of AI start?

This grounding data allows the model to give more accurate and informed answers that may not be part of the dataset it was trained on.

**Cues**

Cues are leading words for the model to build upon, and often help shape the response in the right direction. They are often used with instructions, but not always. Cues are particularly helpful if prompting the model for code generation. Code generation is covered in more depth in a subsequent unit.

**Provide context with prompt engineering**

Bottom of Form

By providing context to the AI model, it allows the model to better understand what you are asking for or what it should know to provide the best answer. Context can be provided in several ways.

**Request output composition**

Specifying the structure of your output can have a large impact on your results. This could include something like asking the model to cite their sources, write the response as an email, format the response as a SQL query, classify sentiment into a specific structure, and so on. For example:

**Prompt**

Write a table in markdown with 6 animals in it, with their genus and species

**Response**

| Animal | Genus | Species |

| ---------------- |:--------------:|:--------------:|

| Red Panda | Ailurus | fulgens |

| African Elephant | Loxodonta | africana |

| Snow Leopard | Panthera | uncia |

| Arctic Fox | Vulpes | lagopus |

| Chimpanzee | Pan | troglodytes |

| Siberian Tiger | Panthera | tigris altaica |

This technique can be used with custom formats, such as a JSON structure:

**Prompt**

Put two fictional characters into JSON of the following format

{

firstNameFictional:

jobFictional:

}

**Response**

Here's an example of how you can put two fictional characters into JSON format:

{

"firstNameFictional": "Henry",

"jobFictional": "Wizard"

},

{

"firstNameFictional": "Frank",

"jobFictional": "Hero"

}

**System message**

The system message is included at the beginning of a prompt and is designed to give the model instructions, perspective to answer from, or other information helpful to guide the model's response. This system message might include tone or personality, topics that shouldn't be included, or specifics (like formatting) of how to answer.

For example, you could give it some of the following system messages:

* "I want you to act like a command line terminal. Respond to commands exactly as cmd.exe would, in one unique code block, and nothing else."
* "I want you to be a translator, from English to Spanish. Don't respond to anything I say or ask, only translate between those two languages and reply with the translated text."
* "Act as a motivational speaker, freely giving out encouraging advice about goals and challenges. You should include lots of positive affirmations and suggested activities for reaching the user's end goal."

Other example system messages are available at the top of the chat window in [Azure AI Foundry](https://ai.azure.com/portal) by selecting the **Prompt samples** button. Try defining your own system prompt that specifies a unique response, and chat with the model to see how responses differ.

The ChatCompletion endpoint enables including the system message by using the System chat role.

response = openai.ChatCompletion.create(

model="gpt-35-turbo",

messages=[

{"role": "system", "content": "You are a casual, helpful assistant. You will talk like an American old western film character."},

{"role": "user", "content": "Can you direct me to the library?"}

]

)

**Response**

{

"choices": [

{

"finish\_reason": "stop",

"index": 0,

"message": {

"content": "Well howdy there, stranger! The library, huh?

Y'all just head down the main road till you hit the town

square. Once you're there, take a left and follow the street

for a spell. You'll see the library on your right, can’t

miss it. Happy trails!",

"role": "assistant"

}

}

],

...

}

System messages can significantly change the response, both in format and content. Try defining a clear system message for the model that explains exactly what kind of response you expect, and what you do or don't want it to include.

**Conversation history**

Along with the system message, other messages can be provided to the model to enhance the conversation. Conversation history enables the model to continue responding in a similar way (such as tone or formatting) and allow the user to reference previous content in subsequent queries. This history can be provided in two ways: from an actual chat history, or from a user defined example conversation.

Chat interfaces that use OpenAI models, such as ChatGPT and the chat playground in [Azure AI Foundry](https://ai.azure.com/portal/chat), include conversation history automatically which results in a richer, more meaningful conversation. In the **Parameters** section of the chat playground, you can specify how many past messages you want included. Try reducing that to 1 or increasing to max to see how different amounts of history impact the conversation.

**Note**

More conversation history included in the prompt means a larger number of input tokens are used. You will have to determine what the correct balance is for your use case, considering the token limit of the model you are using.

Chat systems can also utilize the summarization capabilities of the model to save on input tokens. An app can choose to summarize past messages and include that summary in the conversation history, then provide only the past couple messages verbatim to the model.

**Few shot learning**

Using a user defined example conversation is what is called *few shot learning*, which provides the model examples of how it should respond to a given query. These examples serve to train the model how to respond.

For example, by providing the model a couple of prompts and the expected responses, it continues in the same pattern without instructing it what to do:

User: That was an awesome experience

Assistant: positive

User: I won't do that again

Assistant: negative

User: That was not worth my time

Assistant: negative

User: You can't miss this

Assistant:

If the model is provided with just You can't miss this with no additional context from few shot learning, the response isn't likely to be useful.

In practical terms, conversation history and few shot learning are sent to the model in the same way; each user message and assistant response is a discrete message in the message object. The ChatCompletion endpoint is optimized to include message history, regardless of if this message history is provided as few shot learning, or actual conversation history.

response = openai.ChatCompletion.create(

model="gpt-35-turbo",

messages=[

{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": "That was an awesome experience"},

{"role": "assistant", "content": "positive"},

{"role": "user", "content": "I won't do that again"},

{"role": "assistant", "content": "negative"},

{"role": "user", "content": "That was not worth my time"},

{"role": "assistant", "content": "negative"},

{"role": "user", "content": "You can't miss this"}

]

)

**Break down a complex task**

Another technique for improved interaction is to divide complex prompts into multiple queries. This allows the model to better understand each individual part, and can improve the overall accuracy. Dividing your prompts also allows you to include the response from a previous prompt in a future prompt, and using that information in addition to the capabilities of the model to generate interesting responses.

For example, you could ask the model Doug can ride down the zip line in 30 seconds, and takes 5 minutes to climb back up to the top. How many times can Doug ride the zip line in 17 minutes?. The result is likely 3, which if Doug starts at the top of the zip line is because he doesn't need to return to the top after the final ride.

A more informative answer could come from asking it multiple questions, about the round trip time to get back to the top of the zip line, and how to account for the fact that Doug starts at the top. Breaking down this problem reveals that Doug can, in fact, ride the zip line four times.

**Chain of thought**

One useful method to help you break down your task effectively is to ask the model to explain its chain of thought.

Asking a model to respond with the step by step process by which it determined the response is a helpful way to understand how the model is interpreting the prompt. By doing so, you can see where the model made an incorrect logical turn and better understand how to change your prompt to avoid the error. This technique can include asking it to cite its sources, like Bing chat does (which uses a GPT-4 generation model), and giving reasoning for why it determined its answer.

The chain of thought prompting technique is best used to help you iterate and improve on your prompts to get the highest quality answer from the model.

For example, asking the model What sport is easiest to learn but hardest to master? results in response with an answer, and a small explanation of why. However, when prompted with What sport is easiest to learn but hardest to master? Give a step by step approach of your thoughts, ending in your answer, the response is a complete explanation of how it arrived at its answer.

Sure, here's a step-by-step approach to answering the question:

1. Identify sports that are popular and have a large following.

2. Consider the basic mechanics of each sport and how easy they are to learn.

3. Think about the various techniques and skills required to play each sport at a high level, and how difficult they are to master.

4. Compare the relative ease of learning each sport to the difficulty of mastering it.

5. Based on the above analysis, determine which sport is easiest to learn but hardest to master.

Using this approach, I would say that the sport of golf is easiest to learn but hardest to master. Here's why:

1. Golf is a popular sport with a large following.

2. The basic mechanics of hitting a golf ball are relatively simple and can be learned quickly with some practice.

3. However, achieving consistency and mastering the various shots and techniques required to play at a high level can take years of practice and dedication.

4. Compared to other sports, the difference between a beginner and an intermediate golfer is relatively small, while the difference between an intermediate and an advanced golfer is much larger.

5. Therefore, I would say that golf is the sport that is easiest to learn but hardest to master.

Knowing the steps of reason the model used to form its response allows us to identify where we need to split the prompt, provide more information, or clarify instructions.

**Construct code from natural language**

One of the capabilities of Azure OpenAI models is to generate code from natural language prompts. Tasks can range from a simple one line command to a full application. The AI models can also edit and update provided code or previous responses to complete the requested task.

The examples in this unit are using gpt-35-turbo.

**Write functions**

Azure OpenAI models can create functions and apps in several languages by just describing what you want. For example, given the prompt write a function for binary search in python, you likely receive a response with the function and an explanation of the code.

Sure, here's an example implementation of binary search algorithm in Python:

def binary\_search(array, target):

low = 0

high = len(array) - 1

while low <= high:

mid = (low + high) // 2

if array[mid] == target:

return mid

elif array[mid] < target:

low = mid + 1

else:

high = mid - 1

return -1

This function takes in two arguments - an array of integers and a target integer that we want to search

for in the array. The function initializes two pointers - low and high - to the first and last indices

of the array, respectively.

Inside the while loop, the function calculates the middle index using integer division (//). If the

middle element is equal to the target, the function returns the index of the middle element. If the

middle element is less than the target, we update the low pointer to mid + 1. Otherwise, we update the

high pointer to mid - 1. We continue these steps until we find the target or the pointers cross each

other (i.e., low > high).

If the target is not found in the array, the function returns -1.

Note that the binary search algorithm assumes that the input array is sorted in ascending order. If

the array is not sorted, the algorithm may not return the correct result.

The request can go further than functions, and request a series of functions or an entire app. The response provides a good starting point for the task you're trying to achieve.

**Change coding language**

If you have code in one language, but need it in another, Azure OpenAI can translate that for you. For example, say you have this print\_squares function in Python.

def print\_squares(n):

for i in range(1, n+1):

print(i\*\*2)

If you provide that code, along with the prompt convert this code to C#, you likely get something similar to the following function.

public void PrintSquares(int n)

{

for (int i = 1; i <= n; i++)

{

Console.WriteLine(i \* i);

}

}

Notice the model understood the differences between how for loops and functions are defined in the different languages (including access modifiers in C#), and changed the code appropriately.

**Understand unknown code**

Azure OpenAI models are helpful for understanding code that doesn't make sense, or may be in a language you aren't familiar with. For example, say you were given the following function (in a fictitious coding language!) and didn't know how to understand it.

fn qzplv(n: i32, m: i32) -> i32 {

if n == 0 {

return m + 1;

} else if m == 0 {

return qzplv(n - 1, 1);

} else {

return qzplv(n - 1, qzplv(n, m - 1));

}

}

When provided to the model, along with the prompt could you explain what this code is doing?, you get a response similar to the following:

Sure, I'd be happy to explain what this code is doing!

This code is implementing the Ackermann function, which is a well-known mathematical function that

grows very quickly in terms of computation time as its inputs increase. The function takes two integer

arguments (n and m) and returns another integer.

Here's how the function works:

- If n is equal to 0, the function returns m + 1.

- If m is equal to 0, the function calls itself recursively with n - 1 and 1 as arguments.

- Otherwise, the function calls itself recursively with n - 1 and the result of calling itself with n

and m - 1 as arguments.

The function is implemented in a way that is easy to understand and follow, but it has a very high

computational complexity. In fact, it grows so quickly that it can't be computed for values of n and m

greater than about 4 or 5.

Combining the explanation and translation into a language you're familiar with makes this functionality even more useful.

**Complete code and assist the development process**

Azure OpenAI can assist developers in common software development tasks such as writing unit tests, completing partial code, commenting code, and generating documentation. Using AI assistance can enable developers to spend more time on complex programming and problem solving tasks.

Azure OpenAI models are capable of generating code based on comments, function names, and partially written code. The more context you can provide the model, the more accurate the response likely is.

For example, when given the following prompt:

complete the following function

"""

# calculate the average of the numbers in an array, but only if they're even

Def

The model takes the comments and start of the function definition, and complete it from whatever context it has.

Similar functionality can be seen by providing partial code to the model, without any comments. Take the following prompt for example.

complete the following code

"""

def func1(n)

if n==0:

The AI model does its best to complete the function with the most likely completion of the function.

def func1(n):

if n == 0:

return 1

else:

return n \* func1(n-1)

The response likely also includes a natural language explanation of what that function is doing. In this case, it responded with a recursive implementation of the factorial function.

**Write unit tests**

Azure OpenAI models can generate unit tests for functions you write to help make your code more robust. Take for example the binary search function.

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

Provide that function along with the prompt write three unit tests for this function, and you get a response similar to the following.

def test\_binary\_search():

# Test case 1: Target element is in the middle of the array

arr1 = [1, 2, 3, 4, 5, 6]

target1 = 4

assert binary\_search(arr1, target1) == 3

# Test case 2: Target element is at the beginning of the array

arr2 = [1, 2, 3, 4, 5, 6]

target2 = 1

assert binary\_search(arr2, target2) == 0

# Test case 3: Target element is not in the array

arr3 = [1, 2, 3, 4, 5, 6]

target3 = 7

assert binary\_search(arr3, target3) == -1

This functionality is even more useful if you specify the cases you want included in your prompt, which reduces the time it takes for you to write unit tests for your code.

**Add comments and generate documentation**

To further improve your code, AI models can add comments and documentation for you. Take the following function as an example, which can be a little hard to understand when first reading it without any code comments.

def permutations(lst):

if len(lst) == 0:

return []

elif len(lst) == 1:

return [lst]

else:

result = []

for i in range(len(lst)):

temp = lst[i]

remaining = lst[:i] + lst[i+1:]

for p in permutations(remaining):

result.append([temp] + p)

return result

Provide that function to the model, along with a prompt requesting that it add comments to the code, and you get a response similar to the following.

def permutations(lst):

# Check if the input list is empty

if len(lst) == 0:

return []

# Check if the input list has only one element

elif len(lst) == 1:

return [lst]

else:

# Initialize an empty list to store permutations

result = []

# Iterate through each element of the input list

for i in range(len(lst)):

# Remove the current element from the list and store it in a variable

temp = lst[i]

remaining = lst[:i] + lst[i+1:]

# Recursively call the function on the remaining list

for p in permutations(remaining):

# Append the current element to the beginning of each permutation and add it to the result list

result.append([temp] + p)

# Return the final list of permutations

return result

If you take it one step further, and request documentation, OpenAI will add a description including inputs and outputs.

**Fix bugs and improve your code**

Developers sometimes can write code that mostly works, but could be improved by fixing bugs, syntax, performance, or modularity. Azure OpenAI models can help identify ways to improve and provide suggestions on how to write better code.

**Fix bugs in your code**

Azure OpenAI models can help fix bugs in code by analyzing the code and suggesting changes that can potentially fix the issue. This can help developers identify and resolve bugs faster and more efficiently.

For example, say you have the following function that isn't working for you.

def calculate\_average(numbers):

total = 0

for i in range(len(numbers)):

number = numbers[i]

total += number

average = total

rerun averave

Provide that function to the model, along with the prompt Fix the bugs in this function, and you get a response with the fixed code and an explanation of what was fixed.

def calculate\_average(numbers):

total = 0

for i in range(len(numbers)):

number = numbers[i]

total += number

average = total / len(numbers)

return average

AI:

There were a few issues with the original code:

1. The variable "average" was assigned the value of "total" instead of the actual average.

2. There was a typo in the last line - "rerun" should be "return".

3. The average was not actually calculated - the corrected code divides the total by the length of

the "numbers" list to get the average.

**Improve performance**

While code developers write may work, there might be a more efficient way to accomplish the task. Here's an example of a function that calculates the sum of the first n positive integers, possibly inefficiently:

def sum\_of\_n(n):

result = 0

for i in range(1, n+1):

result += i

return result

This version works correctly, but its time complexity is O(n). When provided to the model, here's the response:

AI

We can improve this function it by using a mathematical formula instead.

Here's the more efficient version:

def sum\_of\_n(n):

return n \* (n + 1) // 2

This version still returns the correct result, but its time complexity is now O(1), which makes it much more efficient.

**Refactor inefficient code**

Better code is less prone to bugs and is easier to maintain, and the Azure OpenAI models can help guide developers on how to refactor their code.

Consider the following function.

def calculateTotalPrice(item, quantity):

if item == 'apple':

return quantity \* 0.5

elif item == 'banana':

return quantity \* 0.75

elif item == 'orange':

return quantity \* 0.6

else:

return 0

This code calculates the total price of a particular item based on its name and quantity. However, the code isn't modular and can be difficult to maintain. When provided to the model with the request to refactor it, here's the response:

item\_prices = {

'apple': 0.5,

'banana': 0.75,

'orange': 0.6

}

def calculate\_total\_price(item, quantity):

price = item\_prices.get(item, 0)

return quantity \* price

Along with the code, the model provides an explanation of the refactoring.

In this refactored code, we've created an object called itemPrices that maps item names to their

prices. We've also simplified the calculateTotalPrice function by using the itemPrices object to look

up the price of the item, rather than using a series of if/else statements. This makes the code more

modular and easier to maintain, and also reduces the risk of introducing bugs in the future.

**Knowledge check**

Top of Form

**1. What Azure OpenAI base model can you deploy to access the capabilities of ChatGPT?**

text-davinci-003

gpt-35-turbo

**Correct. Only the gpt-3.5-turbo and later models can be used to access the chat capabilities.**

text-embedding-ada-002 (Version 2)

**2. Which parameter could you adjust to change the randomness or creativeness of the completions returned?**

Temperature

**Correct. The temperature parameter can be adjusted to change the randomness or creativeness of the completions returned.**

Frequency penalty

Stop sequence

**3. Which Azure AI Foundry playground is able to support conversation-in, message-out scenarios?**

Images

Chat

**Correct. The Chat playground is able to support conversation-in, message-out scenarios.**

Bot

**Implement Retrieval Augmented Generation (RAG) with Azure OpenAI Service**

Bottom of Form**Introduction**

Azure OpenAI enables developers to implement Retrieval Augmented Generation (RAG) by connecting supporting AI chat models to your own data. Those models can reference specific sources of data to ground the response, augmenting the capabilities of the AI model when it creates a response.

In this module, you'll learn how to implement RAG by adding your own data with Azure OpenAI and generate responses based on both your data and the Azure OpenAI pretrained knowledge.

**Understand Retrieval Augmented Generation (RAG) with Azure OpenAI Service**

RAG with Azure OpenAI allows developers to use supported AI chat models that can reference specific sources of information to ground the response. Adding this information allows the model to reference both the specific data provided and its pretrained knowledge to provide more effective responses.

**RAG Pattern or Retrieval Augmented Generation**

🡪RAG combines Information retrieval + content generation to generate responses based on your on data

🡪Ex:

You can think it as of giving the model a Library of books to read. The books are indexed to the same semantic space as the model. And then the most relevant part are selected and added as the context to answer the questions

🡪The RAG pattern research and retrieve the best response

🡪RAG patterns are versatile across many applns including question answering ,summarization, and text generation

🡪RAG patterns are especially useful, bcoz they allows them to customize the large languages model outputs without finetune or retrain it with new data which can be costly and time consuming.It enables services like bing chat by providing access to current domain specific data and incorporating FACT checking component Into the LLM response generation for ability

Azure OpenAI enables RAG by connecting pretrained models to your own data sources. Azure OpenAI on your data utilizes the search ability of Azure AI Search to add the relevant data chunks to the prompt. Once your data is in an AI Search index, Azure OpenAI on your data goes through the following steps:

1. Receive user prompt.
2. Determine relevant content and intent of the prompt.
3. Query the search index with that content and intent.
4. Insert search result chunk into the Azure OpenAI prompt, along with system message and user prompt.
5. Send entire prompt to Azure OpenAI.
6. Return response and data reference (if any) to the user.

By default, Azure OpenAI on your data encourages, but doesn't require, the model to respond only using your data. This setting can be unselected when connecting your data, which may result in the model choosing to use its pretrained knowledge over your data.

**Fine-tuning vs. RAG**

Fine-tuning is a technique used to create a custom model by training an existing foundational model such as gpt-35-turbo with a dataset of additional training data. Fine-tuning can result in higher quality requests than prompt engineering alone, customize the model on examples larger than can fit in a prompt, and allow the user to provide fewer examples to get the same high quality response. However, the process for fine-tuning is both costly and time intensive, and should only be used for use cases where it's necessary.

RAG with Azure OpenAI on your data still uses the stateless API to connect to the model, which removes the requirement of training a custom model with your data and simplifies the interaction with the AI model. AI Search first finds the useful information to answer the prompt, it then adds that information to the prompt as grounding data, and Azure OpenAI then forms the response based on that information.

**Add your own data source**

Adding your data can be done through the Azure AI Studio, in the **Chat** playground, or by specifying your data source in an API call. The data source you add is then used to augment the prompt sent to the model. When setting up your data in the studio, you can choose to upload your data files, use data in a blob storage account, or connect to an existing AI Search index.

If you're uploading or using files already in a storage account, Azure OpenAI on your data supports .md, .txt, .html, .pdf, and Microsoft Word or PowerPoint files. If any of these files contain graphics or images, the response quality depends on how well text can be extracted from the visual content.

When uploading data or connecting to files in a storage account, it's recommended to use the Azure AI Studio to create the search resource and index. Adding data this way allows the appropriate chunking to happen when inserting into the index, yielding better responses. If you're using large text files or forms, you should use the available [data preparation script](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/use-your-data#ingesting-your-data-into-azure-cognitive-search?azure-portal=true) to improve the AI model's accuracy.

Enabling [semantic search](https://learn.microsoft.com/en-us/azure/search/semantic-search-overview) for your AI Search service can improve the result of searching your data index and you're likely to receive higher quality responses and citations. However, enabling semantic search may increase the cost of the search service.

You can also use the wizard in your AI Search resource to vectorize your data appropriately, which you'll walk through in this module's exercise. It takes a few extra steps compared to doing so in AI Studio, however serves as a good example of using the RAG pattern with an existing dataset.

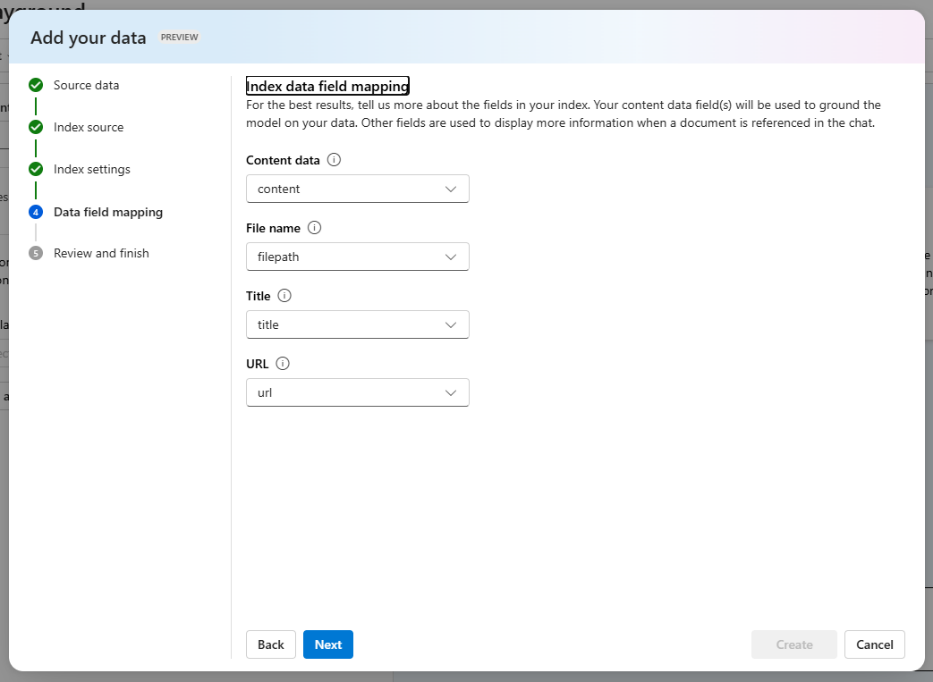
**Connect your data in Azure AI Studio**

To connect your data, navigate to the **Chat** playground in Azure AI Studio and select the **Add your data** tab. Select the **Add a data source** button to get your data connected. The prompts guide you through setting up the connection to each data source, and getting that data into a search index.

**Tip**

If you are using the wizard in AI Studio to create and connect your data source, you will need to create a hub and a project, which aren't covered here. AI Studio will walk you through doing that, or refer to the [**AI Studio documentation**](https://learn.microsoft.com/en-us/azure/ai-studio/concepts/ai-resources).

If you're using your own index that wasn't created through Azure AI Studio, one of the pages allows you to specify your column mapping. It's important to provide accurate fields, to enable the model to provide a better response, especially for **Content data**.



**Chat with your model using your own data**

RAG with Azure OpenAI on your own data can be used in Azure AI Studio with the **Chat** playground, or by using the API.

**Token considerations and recommended settings**

Since RAG with Azure OpenAI on your data includes search results on your index in the prompt, it's important to understand how that impacts your token allotment. Each call to the model includes tokens for the system message, the user prompt, conversation history, retrieved search documents, internal prompts, and the model's response.

The system message, for example, is a useful reference for instructions for the model and is included with every call. While there's no token limit for the system message, when using your own data the system message gets truncated if it exceeds the model's token limit (which varies per model, from 400 to 4000 tokens). The response from the model is also limited when using your own data to 1500 tokens.

Due to these token limitations, it's recommended that you limit both the question length and the conversation history length in your call. [Prompt engineering techniques](https://learn.microsoft.com/en-us/azure/cognitive-services/openai/concepts/advanced-prompt-engineering) such as breaking down the task and chain of thought prompting can help the model respond more effectively.

**Using the API**

Using the API with your own data, you need to specify the data source where your data is stored. With each call you need to include the endpoint, key, and indexName for your AI Search resource.

Your request body will be similar to the following JSON.

{

"dataSources": [

{

"type": "AzureCognitiveSearch",

"parameters": {

"endpoint": "<your\_search\_endpoint>",

"key": "<your\_search\_endpoint>",

"indexName": "<your\_search\_index>"

}

}

],

"messages":[

{

"role": "system",

"content": "You are a helpful assistant assisting users with travel recommendations."

},

{

"role": "user",

"content": "I want to go to New York. Where should I stay?"

}

]

}

The call when using your own data needs to be sent to a different endpoint than is used when calling a base model, which includes extensions. Your call will be sent to a URL similar to the following.

<your\_azure\_openai\_resource>/openai/deployments/<deployment\_name>/chat/completions?api-version=<version>

The request will also need to include the Content-Type and api-key.

**Knowledge check**

Top of Form

**1. What does RAG with Azure OpenAI enable developers to do?**

Create their own AI chat models

Access Azure OpenAI without an approved subscription

Use supported AI chat models that can reference specific sources of data

**Correct. Azure OpenAI on your data allows developers to use supported AI chat models that can reference specific sources of data to ground the response.**

**2. What is the recommended way to add data when using Azure OpenAI on your data?**

Using any data source option available for Azure OpenAI on your data.

Using Azure AI Studio to create the search resource and index.

**Correct. Using Azure AI Studio allows the appropriate chunking to happen when inserting into the index, yielding better responses.**

Connecting to files in a storage account without using Azure AI Studio.

**3. What are some recommended prompt engineering techniques when using RAG with Azure OpenAI on your own data?**

Break down the task and use chain of thought prompting.

**Correct. Breaking down the task and using chain of thought prompting can help the model respond more effectively within the token limit.**

Include as much conversation history as possible in your call.

Use a single long prompt to provide all necessary information.

Bottom of Form