

Get started with Azure OpenAI service

Azure OpenAI Service brings the generative AI models developed by OpenAI to the Azure platform, enabling you to develop powerful AI solutions that benefit from the security, scalability, and integration of services provided by the Azure cloud platform. In this exercise, you'll learn how to get started with Azure OpenAI by provisioning the service as an Azure resource and using Azure OpenAI Studio to deploy and explore generative AI models.

In the scenario for this exercise, you will perform the role of a software developer who has been tasked to implement an AI agent that can use generative AI to help a marketing organization improve its effectiveness at reaching customers and advertising new products. The techniques used in the exercise can be applied to any scenario where an organization wants to use generative AI models to help employees be more effective and productive.

Exercise-1

Provision an Azure OpenAI resource

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

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

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

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

Deploy a model

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

Note: As you use Azure OpenAI Studio, 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, use the **Go to Azure OpenAI Studio** button to open Azure OpenAI Studio in a new browser tab.

After the new tab opens, you can close any banner notifications for new preview services that are displayed at the top of the Azure OpenAI Studio page.

2. In Azure OpenAI Studio, 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:
 - **Model:** gpt-35-turbo-16k (*if the 16k model isn't available, choose gpt-35-turbo*)
 - **Model version:** Auto-update to default
 - **Deployment name:** A unique name of your choice
 - **Advanced options**
 - **Content filter:** Default
 - **Deployment type:** Standard
 - **Tokens per minute rate limit:** 5K*
 - **Enable dynamic quota:** Enabled

* A rate limit of 5,000 tokens per minute is more than adequate to complete this exercise while leaving capacity for other people using the same subscription.

Use the Chat playground

Now that you've deployed a model, you can use it to generate responses based on natural language prompts. The *Chat* playground in Azure OpenAI Studio provides a chatbot interface for GPT 3.5 and higher models.

Note: The *Chat* playground uses the *ChatCompletions* API rather than the older *Completions* API that is used by the *Completions* playground. The *Completions* playground is provided for compatibility with older models.

1. In the **Playground** section, select the **Chat** page. The **Chat** playground page consists of three main panels (which may be arranged right-to-left horizontally, or top-to-bottom vertically depending on your screen resolution):
 - **Setup** - used to set the context for the model's responses.
 - **Chat session** - used to submit chat messages and view responses.
 - **Configuration** - used to configure settings for the model deployment.
2. In the **Configuration** panel, ensure that your gpt-35-turbo-16k model deployment is selected.
3. In the **Setup** panel, review the default **System message**, which should be *You are an AI assistant that helps people find information*. The system message is included in prompts submitted to the model, and provides context for the model's responses; setting expectations about how an AI agent based on the model should interact with the user.
4. In the **Chat session** panel, enter the user query `How can I use generative AI to help me market a new product?`

Note: You may receive a response that the API deployment is not yet ready. If so, wait for a few minutes and try again.

5. Review the response, noting that the model has generated a cohesive natural language answer that is relevant to the query with which it was prompted.
6. Enter the user query `What skills do I need if I want to develop a solution to accomplish this?`.
7. Review the response, noting that the chat session has retained the conversational context (so "this" is interpreted as a generative AI solution for marketing). This contextualization is achieved by including the recent conversation history in each successive prompt submission, so the prompt sent to the model for the second query included the original query and response as well as the new user input.
8. In the **Chat session** panel toolbar, select **Clear chat** and confirm that you want to restart the chat session.
9. Enter the query `Can you help me find resources to learn those skills?` and review the response, which should be a valid natural language answer, but since the previous chat history has been lost, the answer is likely to be about finding generic skilling resources rather than being related to the specific skills needed to build a generative AI marketing solution.

Experiment with system messages, prompts, and few-shot examples

So far, you've engaged in a chat conversation with your model based on the default system message. You can customize the system setup to have more control over the kinds of responses generated by your model.

1. In the **Setup** panel, under **Use a system message template**, select the **Marketing Writing Assistant** template and confirm that you want to update the system message.
2. Review the new system message, which describes how an AI agent should use the model to respond.
3. In the **Chat session** panel, enter the user query `Create an advertisement for a new scrubbing brush`.
4. Review the response, which should include advertising copy for a scrubbing brush. The copy may be quite extensive and creative.

In a real scenario, a marketing professional would likely already know the name of the scrubbing brush product as well as have some ideas about key features that should be highlighted in an advert. To get the most useful results from a generative AI model, users need to design their prompts to include as much pertinent information as possible.

5. Enter the prompt `Revise the advertisement for a scrubbing brush named "Scrubadub 2000", which is made of carbon fiber and reduces cleaning times by half compared to ordinary scrubbing brushes`.
6. Review the response, which should take into account the additional information you provided about the scrubbing brush product.

The response should now be more useful, but to have even more control over the output from the model, you can provide one or more *few-shot* examples on which responses should be based.

7. In the **Setup** panel, under **Examples**, select **Add**. Then type the following message and response in the designated boxes:

User:

Code

Write an advertisement for the lightweight "Ultramop" mop, which uses patented absorbent materials to clean floors.

Assistant:

Code

Welcome to the future of cleaning!

The Ultramop makes light work of even the dirtiest of floors. Thanks to its patented absorbent materials, it ensures a brilliant shine. Just look at these features:

- Lightweight construction, making it easy to use.
- High absorbency, enabling you to apply lots of clean soapy water to the floor.
- Great low price.

Check out this and other products on our website at www.contoso.com.

8. Use the **Apply changes** button to save the examples and start a new session.
9. In the **Chat session** section, enter the user query `Create an advertisement for the Scrubadub 2000 - a new scrubbing brush made of carbon fiber that reduces cleaning time by half.`
10. Review the response, which should be a new advert for the "Scrubadub 2000" that is modeled on the "Ultramop" example provided in the system setup.

Experiment with parameters

You've explored how the system message, examples, and prompts can help refine the responses returned by the model. You can also use parameters to control model behavior.

1. In the **Configuration** panel, select the **Parameters** tab and set the following parameter values:
 - **Max response:** 1000
 - **Temperature:** 1
2. In the **Chat session** section, use the **Clear chat** button to reset the chat session. Then enter the user query `Create an advertisement for a cleaning sponge` and review the response. The resulting advertisement copy should include a maximum of 1000 text tokens, and include some creative elements - for example, the model may have invented a product name for the sponge and made some claims about its features.
3. Use the **Clear chat** button to reset the chat session again, and then re-enter the same query as before (`Create an advertisement for a cleaning sponge`) and review the response. The response may be different from the previous response.

4. In the **Configuration** panel, on the **Parameters** tab, change the **Temperature** parameter value to 0.
5. In the **Chat session** section, use the **Clear chat** button to reset the chat session again, and then re-enter the same query as before (**Create an advertisement for a cleaning sponge**) and review the response. This time, the response may not be quite so creative.
6. Use the **Clear chat** button to reset the chat session one more time, and then re-enter the same query as before (**Create an advertisement for a cleaning sponge**) and review the response; which should be very similar (if not identical) to the previous response.

The **Temperature** parameter controls the degree to which the model can be creative in its generation of a response. A low value results in a consistent response with little random variation, while a high value encourages the model to add creative elements its output; which may affect the accuracy and realism of the response.

Deploy your model to a web app

Now that you've explored some of the capabilities of a generative AI model in the Azure OpenAI Studio playground, you can deploy an Azure web app to provide a basic AI agent interface through which users can chat with the model.

1. At the top right of the **Chat** playground page, in the **Deploy to** menu, select **A new web app**.
2. In the **Deploy to a web app** dialog box, create a new web app with the following settings:
 - **Name:** *A unique name*
 - **Subscription:** *Your Azure subscription*
 - **Resource group:** *The resource group in which you provisioned your Azure OpenAI resource*
 - **Locations:** *The region where you provisioned your Azure OpenAI resource*
 - **Pricing plan:** *Free (F1) - If this is not available, select Basic (B1)*
 - **Enable chat history in the web app:** Unselected
 - **I acknowledge that web apps will incur usage to my account:** Selected
3. Deploy the new web app and wait for deployment to complete (which may take 10 minutes or so)
4. After your web app has deployed successfully, use the button at the top right of the **Chat** playground page to launch the web app. The app may take a few minutes to launch. If prompted, accept the permissions request.
5. In the web app, enter the following chat message:

Code

```
Write an advertisement for the new "WonderWipe" cloth that attracts dust  
particulates and can be used to clean any household surface.
```

6. Review the response.

Note: You deployed the *model* to a web app, but this deployment doesn't include the system settings and parameters you set in the playground; so the response may not reflect the examples you specified in the playground. In a real scenario, you would add logic to your application to modify the prompt so that it includes the appropriate contextual data for the kinds of response you want to generate. This kind of customization is beyond the scope of this introductory-level exercise, but you can learn about prompt engineering techniques and Azure OpenAI APIs in other exercises and product documentation.

7. When you have finished experimenting with your model in the web app, close the web app tab in your browser to return to Azure OpenAI Studio.

Clean up

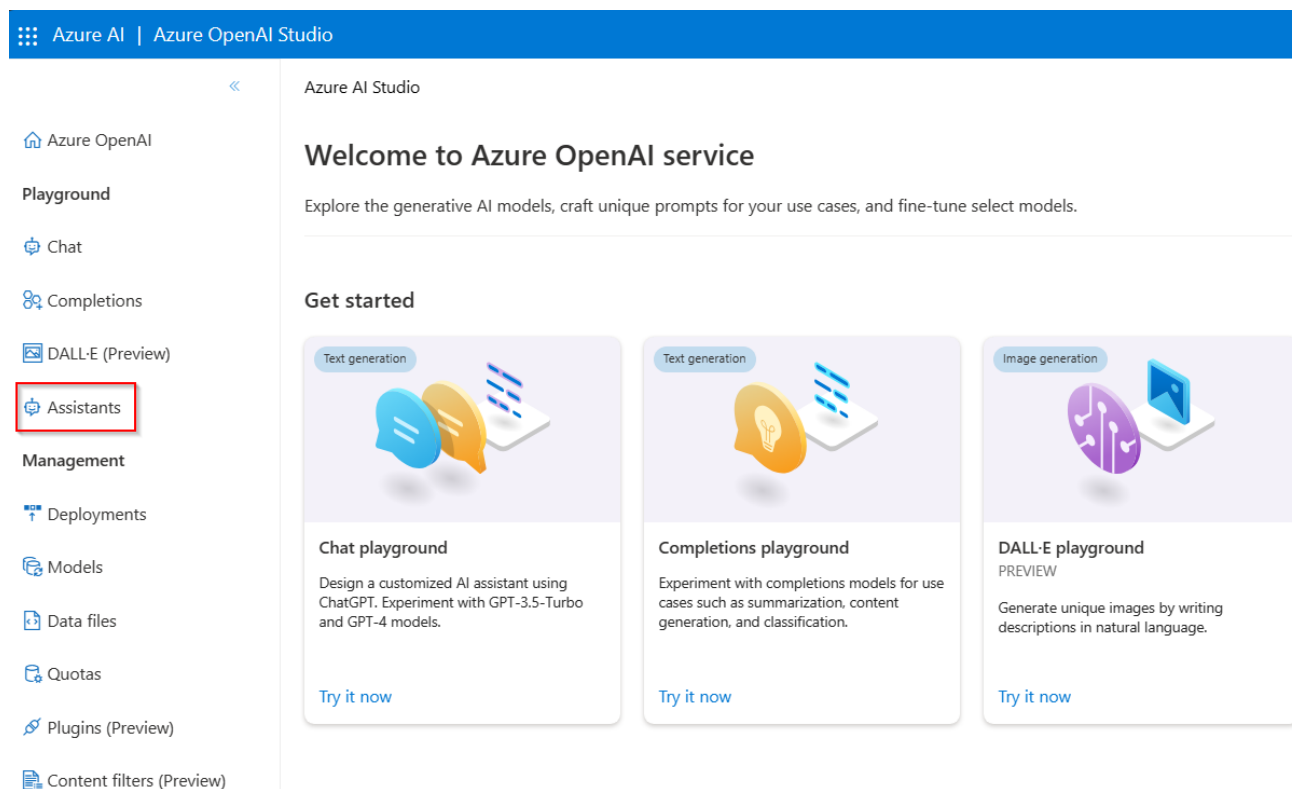
When you're done with your Azure OpenAI resource, remember to delete the deployment or the entire resource in the **Azure portal** at <https://portal.azure.com>.

Exercise-2

Go to the Azure OpenAI Studio

Navigate to Azure OpenAI Studio at <https://oai.azure.com/> and sign-in with credentials that have access to your OpenAI resource. During or after the sign-in workflow, select the appropriate directory, Azure subscription, and Azure OpenAI resource.


From the Azure OpenAI Studio landing page launch the Assistant's playground from the left-hand navigation **Playground** > **Assistants (Preview)**



Playground

The Assistants playground allows you to explore, prototype, and test AI Assistants without needing to run any code. From this page, you can quickly iterate and experiment with new ideas.

Assistants playground

 Show panels

Assistant setup

Assistant

+ New

Save

Open

Delete

Assistant name

Instructions

Deployment

gpt-4-1106-preview

Tools

Functions

+ Add function

Code interpreter

Files

+ Add files

Chat session

Clear chat

View code

Start chatting

Test your assistant by sending queries below. Then adjust your assistant setup to improve the assistant's responses.

Type user query here. (Shift + Enter for new line)

+

➤

Logs

Assistant setup

Use the **Assistant setup** pane to create a new AI assistant or to select an existing assistant.

Name	Description
Assistant name	Your deployment name that is associated with a specific model.
Instructions	Instructions are similar to system messages this is where you give the model guidance about how it should behave and any context it should reference when generating a response. You can describe the assistant's personality, tell it what it should and shouldn't answer, and tell it how to format responses. You can also provide examples of the steps it should take when answering responses.
Deployment	This is where you set which model deployment to use with your assistant.
Functions	Create custom function definitions for the models to formulate API calls and structure data outputs based on your specifications
Code interpreter	Code interpreter provides access to a sandboxed Python environment that can be used to allow the model to test and execute code.

Name	Description
Files	You can upload up to 20 files, with a max file size of 512 MB to use with tools.

Tools

An individual assistant can access up to 128 tools including `code interpreter`, as well as any custom tools you create via [functions](#).

Chat session

Chat session also known as a *thread* within the Assistant's API is where the conversation between the user and assistant occurs. Unlike traditional chat completion calls there is no limit to the number of messages in a thread. The assistant will automatically compress requests to fit the input token limit of the model.

This also means that you are not controlling how many tokens are passed to the model during each turn of the conversation. Managing tokens is abstracted away and handled entirely by the Assistants API.

Select the **Clear chat** button to delete the current conversation history.

Underneath the text input box there are two buttons:

- Add a message without run.
- Add and run.

Logs

Logs provide a detailed snapshot of what the assistant API activity.

Show panels

By default there are three panels: assistant setup, chat session, and Logs. **Show panels** allows you to add, remove, and rearrange the panels. If you ever close a panel and need to get it back, use **Show panels** to restore the lost panel.

Create your first assistant

1. From the assistant setup drop-down, select **New**

2. Give your Assistant a name
3. Enter the following instructions "You are an AI assistant that can write code to help answer math questions"
4. Select a deployment of gpt-4 (1106-preview). The dropdown will display your custom deployment names so if your model/deployment names are different you should choose a deployment where the underlying model is equivalent to the 1106-preview model.
5. Select the toggle enabling code interpreter.

6. Select Save.

Assistant setup



Assistant

New Save Open Delete

Selected assistant id: `asst_3hOfhqSLjbbKwqPvNtmJ3gYz`

Assistant name ⓘ

Math Assist

Instructions ⓘ

You are an AI assistant that can write code to help answer math questions.

Deployment ⓘ

gpt-4-1106-preview

Tools

Functions ⓘ

Add function

Code interpreter ⓘ



Files ⓘ

Add files

7. Enter a question for the assistant to answer: "I need to solve the equation $3x + 11 = 14$. Can you help me?"

8. Select the **Add and run button**



Output

The solution to the equation ($3x + 11 = 14$) is ($x = 1$).

While we can see that answer is correct, to confirm that the model used code interpreter to get to this answer, and that the code it wrote is valid rather than just repeating an answer from the model's training data we'll ask another question.

9. Enter the follow-up question: "Show me the code you ran to get this solution."

OutputCopy

Certainly! Here's the Python code I ran to find the solution for the equation ($3x + 11 = 14$):

PythonCopy

```
from sympy import symbols, Eq, solve
```

```
# Define the variable
```

```
x = symbols('x')
```

```
# Define the equation
```

```
equation = Eq(3*x + 11, 14)
```

```
# Solve the equation
```

```
solution = solve(equation, x)
```

```
solution
```

Chat session

 Clear chat  View code

New thread started
thread_pbpoaldrTWUriyrt0kukChsp

I need to solve the equation $3x + 11 = 14$. Can you help me?



The solution to the equation $(3x + 11 = 14)$ is $(x = 1)$.

Show me the code you ran to get this solution.



Certainly! Here's the Python code I ran to find the solution for the equation $(3x + 11 = 14)$:

```
from sympy import symbols, Eq, solve

# Define the variable
x = symbols('x')

# Define the equation
equation = Eq(3*x + 11, 14)

# Solve the equation
solution = solve(equation, x)
solution
```

I used the [sympy](#) library in Python, which is a library for symbolic mathematics. I defined a symbol x to represent the unknown in the equation, then I set up the equation itself using [Eq](#). The [solve](#) function was used to analytically solve the equation for x , which yielded the solution $(x = 1)$.

Type user query here. (Shift + Enter for new line)



Logs

- List assistants >
- Create a thread >
- Update assistant >
- Add message >
- Run thread >
- Get run steps >
- Add message >
- Run thread >
- Get run steps >

You could also consult the logs in the right-hand panel to confirm that code interpreter was used and to validate the code that was run to generate the response. It is important to remember that while code interpreter gives the model the capability to respond to more complex math questions by converting the questions into code and running in a sandboxed Python environment, you still need to validate

the response to confirm that the model correctly translated your question into a valid representation in code.

Exercise-3

C# Code

Create a new .NET Core application

In a console window (such as cmd, PowerShell, or Bash), use the `dotnet new` command to create a new console app with the name `azure-openai-quickstart`. This command creates a simple "Hello World" project with a single C# source file: *Program.cs*.

```
.NET CLI
dotnet new console -n azure-openai-assistants-quickstart
```

Change your directory to the newly created app folder. You can build the application with:

```
.NET CLI
dotnet build
```

The build output should contain no warnings or errors.

Output

```
...
Build succeeded.
  0 Warning(s)
  0 Error(s)
...
```

Install the OpenAI .NET client library with:

```
Console
dotnet add package Azure.AI.OpenAI.Assistants --prerelease
```

Retrieve key and endpoint

To successfully make a call against Azure OpenAI, you need an **endpoint** and a **key**.

Variable name	Value
ENDPOINT	This value can be found in the Keys & Endpoint section when examining your resource from the Azure portal. Alternatively, you can find the value in the Azure OpenAI Studio > Playground > Code View . An example endpoint is: https://docs-test-001.openai.azure.com/ .

Variable name	Value
API-KEY	This value can be found in the Keys & Endpoint section when examining your resource from the Azure portal. You can use either KEY1 or KEY2.

Go to your resource in the Azure portal. The **Keys & Endpoint** section can be found in the **Resource Management** section. Copy your endpoint and access key as you'll need both for authenticating your API calls. You can use either KEY1 or KEY2. Always having two keys allows you to securely rotate and regenerate keys without causing a service disruption.

Home > docs-test-001 | Keys and Endpoint ☆ ...

Cognitive Service | Directory: Microsoft

Search (Ctrl+/) << Regenerate Key1 Regenerate Key2

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource Management

Keys and Endpoint

Deployments

Pricing tier

Networking

Identity

Cost analysis

Properties

Locks

These keys are used to access your Cognitive Service API. Do not share your keys. Store them securely– for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

Show Keys

KEY 1

KEY 2

Location/Region ⓘ

eastus

Endpoint

https:// docs-test-001.openai.azure.com/

Environment variables

Create and assign persistent environment variables for your key and endpoint.

Windows Command Prompt

```
setx AZURE_OPENAI_API_KEY "REPLACE_WITH_YOUR_KEY_VALUE_HERE"
setx AZURE_OPENAI_ENDPOINT "REPLACE_WITH_YOUR_ENDPOINT_HERE"
```

Create an assistant

In our code we are going to specify the following values:

Name	Description
Assistant name	Your deployment name that is associated with a specific model.
Instructions	Instructions are similar to system messages this is where you give the model guidance about how it should behave and any context it should reference when generating a response. You can describe the assistant's personality, tell it what it should and shouldn't answer, and tell it how to format responses. You can also provide examples of the steps it should take when answering responses.
Model	This is where you set which model deployment name to use with your assistant. The retrieval tool requires gpt-35-turbo (1106) Or gpt-4 (1106-preview) model. Set this value to your deployment name, not the model name unless it is the same.
Code interpreter	Code interpreter provides access to a sandboxed Python environment that can be used to allow the model to test and execute code.

Tools

An individual assistant can access up to 128 tools including code interpreter, as well as any custom tools you create via [functions](#).

Create and run an assistant with the following:

```
C#
using Azure;
using Azure.AI.OpenAI.Assistants;

string endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
throw new ArgumentNullException("AZURE_OPENAI_ENDPOINT");
string key = Environment.GetEnvironmentVariable("AZURE_OPENAI_API_KEY") ?? throw
new ArgumentNullException("AZURE_OPENAI_API_KEY");
AssistantsClient client = new AssistantsClient(new Uri(endpoint), new
AzureKeyCredential(key));

// Create an assistant
Assistant assistant = await client.CreateAssistantAsync(
    new AssistantCreationOptions("gpt-4-1106-preview") // Replace this with the
name of your model deployment
    {
        Name = "Math Tutor",
        Instructions = "You are a personal math tutor. Write and run code to
answer math questions.",
        Tools = { new CodeInterpreterToolDefinition() }
    });

// Create a thread
AssistantThread thread = await client.CreateThreadAsync();

// Add a user question to the thread
```

```

ThreadMessage message = await client.CreateMessageAsync(
    thread.Id,
    MessageRole.User,
    "I need to solve the equation `3x + 11 = 14`. Can you help me?");

// Run the thread
ThreadRun run = await client.CreateRunAsync(
    thread.Id,
    new CreateRunOptions(assistant.Id)
);

// Wait for the assistant to respond
do
{
    await Task.Delay(TimeSpan.FromMilliseconds(500));
    run = await client.GetRunAsync(thread.Id, run.Id);
}
while (run.Status == RunStatus.Queued
    || run.Status == RunStatus.InProgress);

// Get the messages
PageableList<ThreadMessage> messagesPage = await
client.GetMessagesAsync(thread.Id);
IReadOnlyList<ThreadMessage> messages = messagesPage.Data;

// Note: messages iterate from newest to oldest, with the messages[0] being the
most recent
foreach (ThreadMessage threadMessage in messages.Reverse())
{
    Console.WriteLine($"{threadMessage.CreatedAt:yyyy-MM-dd HH:mm:ss} -
{threadMessage.Role,10}: ");
    foreach (MessageContent contentItem in threadMessage.ContentItems)
    {
        if (contentItem is MessageTextContent textItem)
        {
            Console.WriteLine(textItem.Text);
        }
        Console.WriteLine();
    }
}

```

This will print an output as follows:

```

2024-03-05 03:38:17 -      user: I need to solve the equation `3x + 11 = 14`. Can
you help me?
2024-03-05 03:38:25 -  assistant: The solution to the equation \((3x + 11 = 14)\) is
\((x = 1)\).

```

New messages can be created on the thread before re-running, which will see the assistant use the past messages as context within the thread.

Exercise -3

Python Code

Install the OpenAI Python client library with:

Console
`pip install openai==v1.20.0`

Note

Azure OpenAI does not yet support Assistants V2. Please use the v1.20.0 release of the OpenAI Python library until V2 support is available.

Note

This library is maintained by OpenAI. Refer to the [release history](#) to track the latest updates to the library.


Retrieve key and endpoint

To successfully make a call against the Azure OpenAI service, you'll need the following:

Variable name	Value
ENDPOINT	This value can be found in the Keys and Endpoint section when examining your resource from the Azure portal. Alternatively, you can find the value in Azure OpenAI Studio > Playground > View code . An example endpoint is: <code>https://docs-test-001.openai.azure.com/</code> .
API-KEY	This value can be found in the Keys and Endpoint section when examining your resource from the Azure portal. You can use either KEY1 or KEY2.
DEPLOYMENT-NAME	This value will correspond to the custom name you chose for your deployment when you deployed a model. This value can be found under Resource Management > Model Deployments in the Azure portal or alternatively under Management > Deployments in Azure OpenAI Studio.

Go to your resource in the Azure portal. The **Keys and Endpoint** can be found in the **Resource Management** section. Copy your endpoint and access key as you'll need both for authenticating your API calls. You can use either KEY1 or KEY2. Always having two keys allows you to securely rotate and regenerate keys without causing a service disruption.

Home >

 docs-test-001 | Keys and Endpoint ☆ ...

Cognitive Service Directory: Microsoft

Search (Ctrl+/) << Regenerate Key1 Regenerate Key2

Overview


Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Resource Management

 Keys and Endpoint

Deployments

Pricing tier


Networking

Identity

Cost analysis

Properties

Locks

 These keys are used to access your Cognitive Service API. Do not share your keys. Store them securely– for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

Show Keys

KEY 1

.....

KEY 2

.....

Location/Region ⓘ

eastus

Endpoint

https:// docs-test-001.openai.azure.com/

Create and assign persistent environment variables for your key and endpoint.

Environment variables

Create and assign persistent environment variables for your key and endpoint.

Windows Command Prompt

```
setx AZURE_OPENAI_API_KEY "REPLACE_WITH_YOUR_KEY_VALUE_HERE"
setx AZURE_OPENAI_ENDPOINT "REPLACE_WITH_YOUR_ENDPOINT_HERE"
```

Create an assistant

In our code we are going to specify the following values:

Name	Description
Assistant name	Your deployment name that is associated with a specific model.
Instructions	Instructions are similar to system messages this is where you give the model guidance about how it should behave and any context it should reference when generating a response. You can describe the assistant's personality, tell it what it should and shouldn't answer, and tell it how to format responses. You can also provide examples of the steps it should take when answering responses.

Name	Description
Model	This is where you set which model deployment name to use with your assistant. The retrieval tool requires gpt-35-turbo (1106) or gpt-4 (1106-preview) model. Set this value to your deployment name, not the model name unless it is the same.
Code interpreter	Code interpreter provides access to a sandboxed Python environment that can be used to allow the model to test and execute code.

Tools

An individual assistant can access up to 128 tools including code interpreter, as well as any custom tools you create via [functions](#).

Create and run an assistant with the following:

```
Python
import os
import time
import json
from openai import AzureOpenAI

client = AzureOpenAI(
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="2024-02-15-preview",
    azure_endpoint = os.getenv("AZURE_OPENAI_ENDPOINT")
)

# Create an assistant
assistant = client.beta.assistants.create(
    name="Math Assist",
    instructions="You are an AI assistant that can write code to help answer math questions.",
    tools=[{"type": "code_interpreter"}],
    model="gpt-4-1106-preview" #You must replace this value with the deployment name for your model.
)

# Create a thread
thread = client.beta.threads.create()

# Add a user question to the thread
message = client.beta.threads.messages.create(
    thread_id=thread.id,
    role="user",
    content="I need to solve the equation `3x + 11 = 14`. Can you help me?"
)

# Run the thread
run = client.beta.threads.runs.create(
    thread_id=thread.id,
    assistant_id=assistant.id,
)
```

```

# Retrieve the status of the run
run = client.beta.threads.runs.retrieve(
    thread_id=thread.id,
    run_id=run.id
)

status = run.status

# Wait till the assistant has responded
while status not in ["completed", "cancelled", "expired", "failed"]:
    time.sleep(5)
    run = client.beta.threads.runs.retrieve(thread_id=thread.id, run_id=run.id)
    status = run.status

messages = client.beta.threads.messages.list(
    thread_id=thread.id
)

print(messages.model_dump_json(indent=2))

```

Output

JSON

```

{
  "data": [
    {
      "id": "msg_XOL8597uuV6zIEgaqZtI0KD3",
      "assistant_id": "asst_WKF0CDJ42Ld1bVUfS8w2pt6E",
      "content": [
        {
          "text": {
            "annotations": [],
            "value": "The solution to the equation  $(3x + 11 = 14)$  is  $(x = 1)$ ."
          },
          "type": "text"
        }
      ],
      "created_at": 1705892759,
      "file_ids": [],
      "metadata": {},
      "object": "thread.message",
      "role": "assistant",
      "run_id": "run_TSmF4LoU6bX4SD3xp5xDr1ey",
      "thread_id": "thread_hCOKdEZy1diZAAzwDudRqGRc"
    },
    {
      "id": "msg_F25tb90W5xTPqSn4KgU4aMsb",
      "assistant_id": null,
      "content": [
        {
          "text": {
            "annotations": [],
            "value": "I need to solve the equation  $3x + 11 = 14$ . Can you help me?"
          },
          "type": "text"
        }
      ],
      "created_at": 1705892759,
      "file_ids": [],
      "metadata": {},
      "object": "thread.message",
      "role": "user",
      "run_id": "run_TSmF4LoU6bX4SD3xp5xDr1ey",
      "thread_id": "thread_hCOKdEZy1diZAAzwDudRqGRc"
    }
  ]
}

```

```

    }
  ],
  "created_at": 1705892751,
  "file_ids": [],
  "metadata": {},
  "object": "thread.message",
  "role": "user",
  "run_id": null,
  "thread_id": "thread_hCOKdEZy1diZAAzwDudRqGRc"
}
],
"object": "list",
"first_id": "msg_XOL8597uuV6zIEgaqZtI0KD3",
"last_id": "msg_F25tb90W5xTPqSn4KgU4aMsb",
"has_more": false
}

```

Understanding your results

In this example we create an assistant with code interpreter enabled. When we ask the assistant a math question it translates the question into python code and executes the code in sandboxed environment in order to determine the answer to the question. The code the model creates and tests to arrive at an answer is:

Python

```

from sympy import symbols, Eq, solve

# Define the variable
x = symbols('x')

# Define the equation
equation = Eq(3*x + 11, 14)

# Solve the equation
solution = solve(equation, x)
solution

```

It is important to remember that while code interpreter gives the model the capability to respond to more complex queries by converting the questions into code and running that code iteratively in the Python sandbox until it reaches a solution, you still need to validate the response to confirm that the model correctly translated your question into a valid representation in code.