Prepare for an AI development project

In this exercise, you use Azure AI Foundry portal to create a project, ready to build an AI solution.

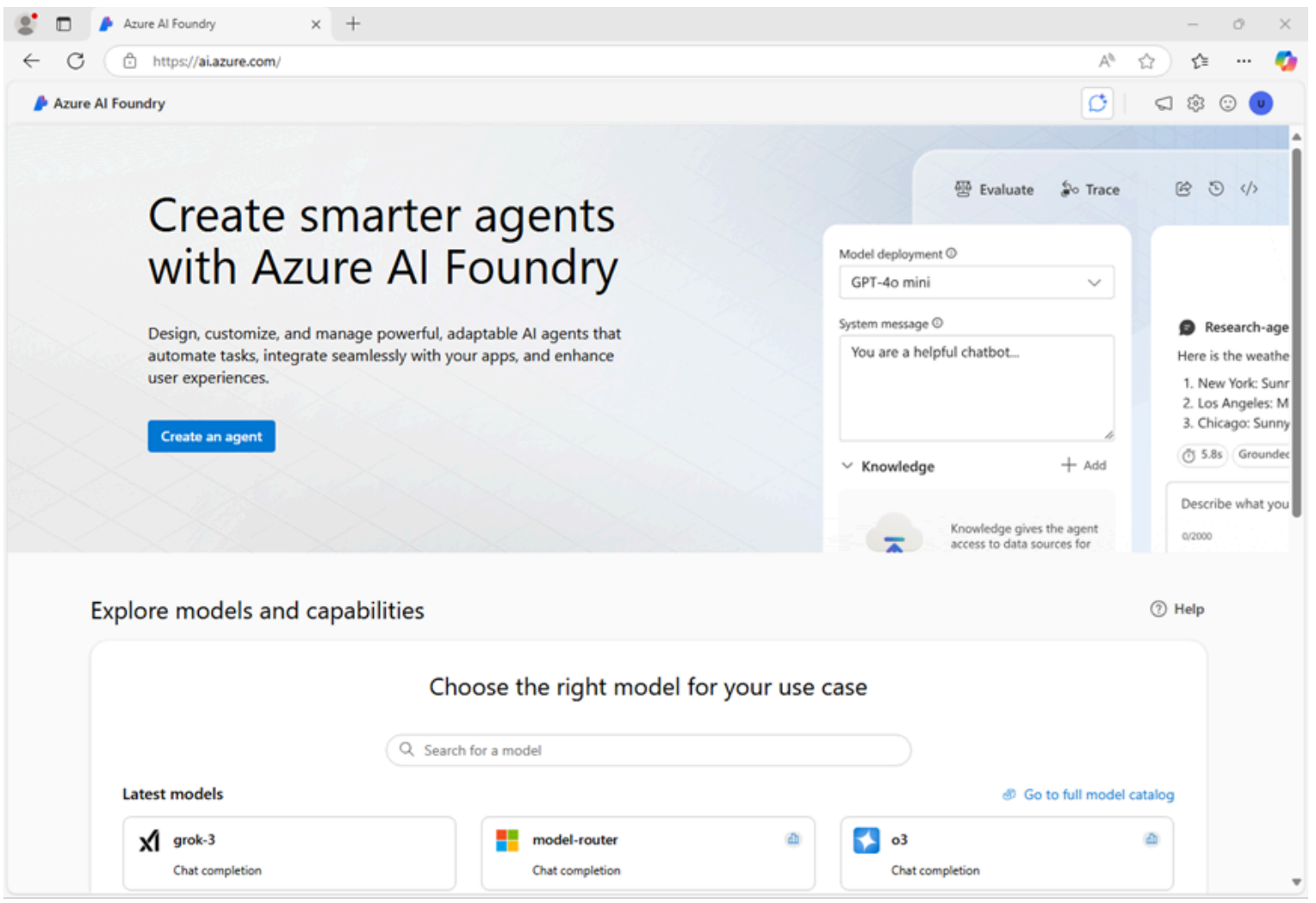
This exercise takes approximately **30** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Open Azure AI Foundry portal

Let's start by signing into Azure AI Foundry portal.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. Review the information on the home page.

Create a project

An Azure AI *project* provides a collaborative workspace for AI development. Let's start by choosing a model that we want to work with and creating a project to use it in.

> **Note**: AI Foundry projects can be based on an *Azure AI Foundry* resource, which provides access to AI models (including Azure OpenAI), Azure AI services, and other resources for developing AI agents and chat solutions. Alternatively, projects can be based on *AI hub* resources; which include connections to Azure resources for secure storage, compute, and specialized tools. Azure AI Foundry based projects are great for developers who want to manage resources for AI agent or chat app development. AI hub based projects are more suitable for enterprise development teams working on complex AI solutions.

1. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.

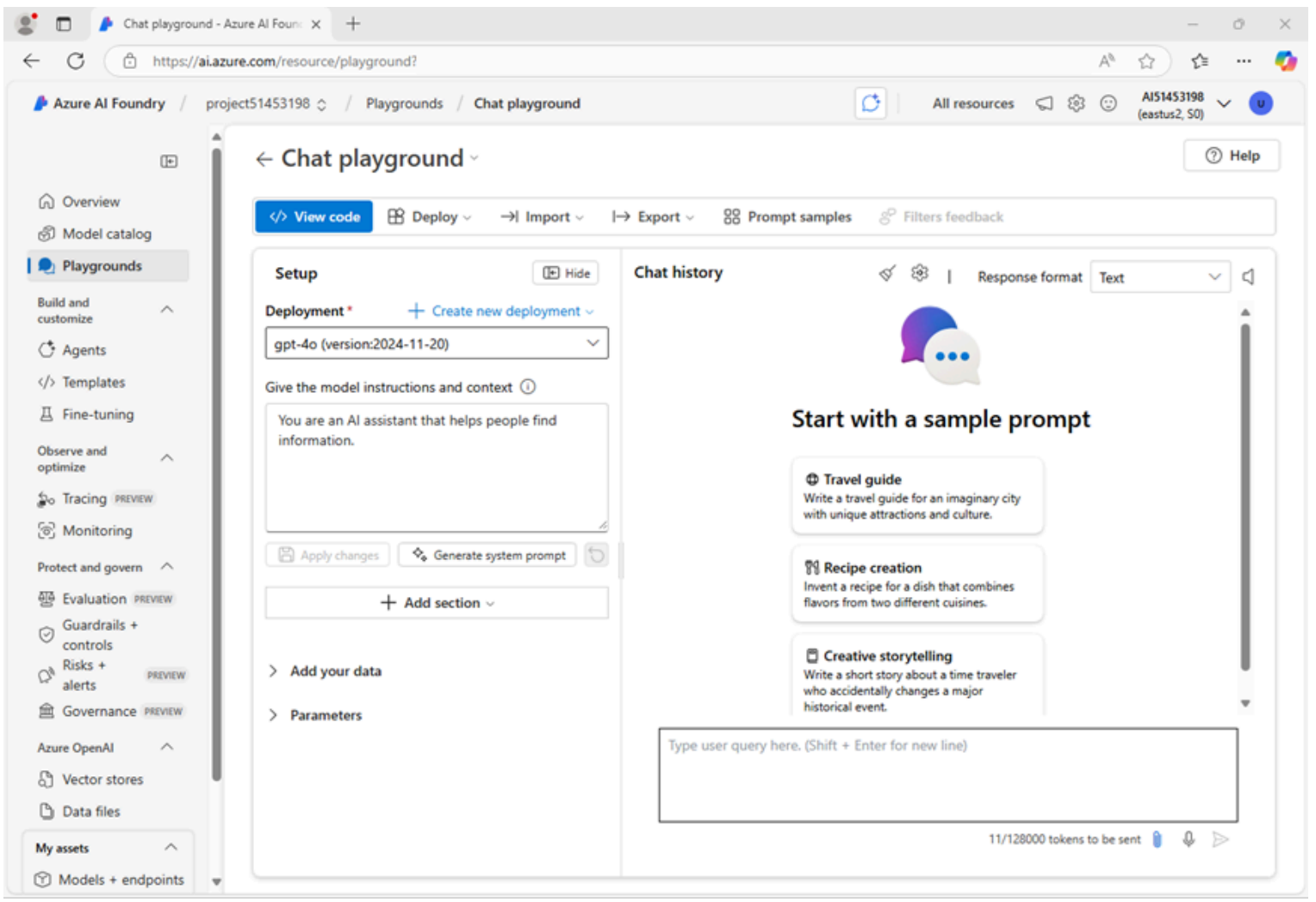
2. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Select **Customize** and specify the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

> ❗ * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

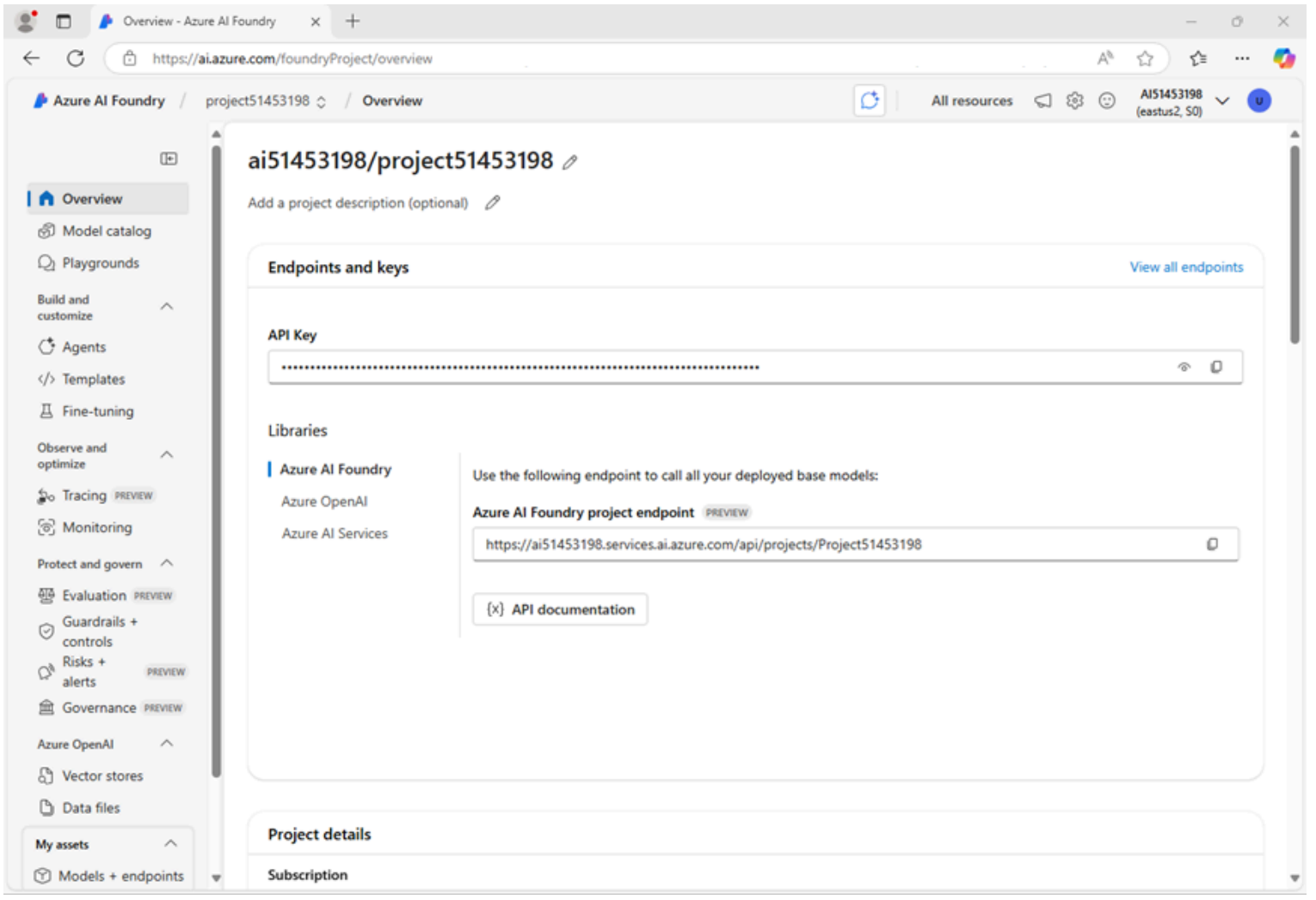
5. Select **Create** and wait for your project to be created. If prompted, deploy the gpt-4o model using the **Global standard** deployment type and customize the deployment details to set a **Tokens per minute rate limit** of 50K (or the maximum available if less than 50K).

> ❗ **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

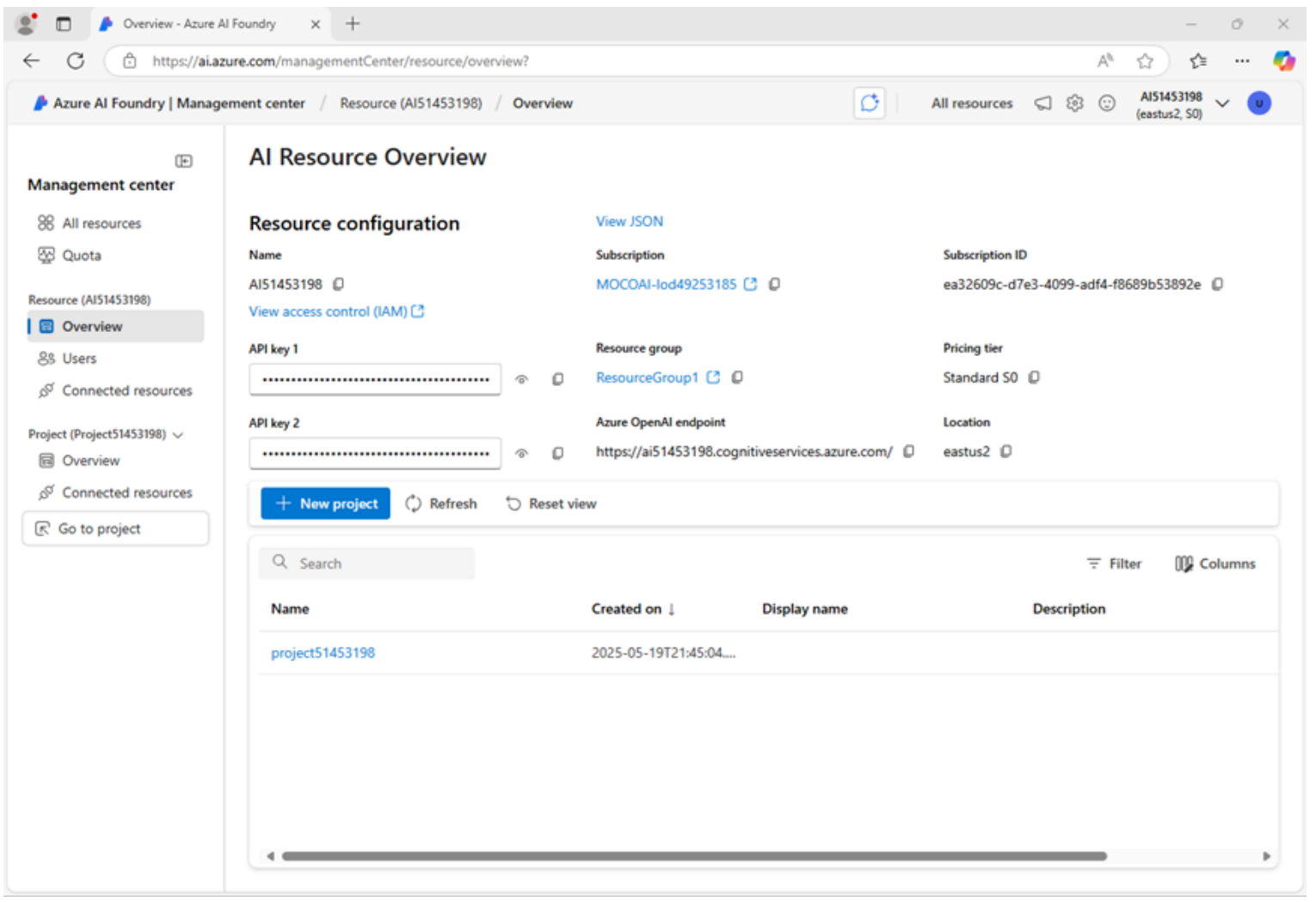
6. When your project is created, the chat playground will be opened automatically so you can test your model:



7. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



8. At the bottom of the navigation pane on the left, select **Management center**. The management center is where you can configure settings at both the *resource* and *project* levels; which are both shown in the navigation pane.

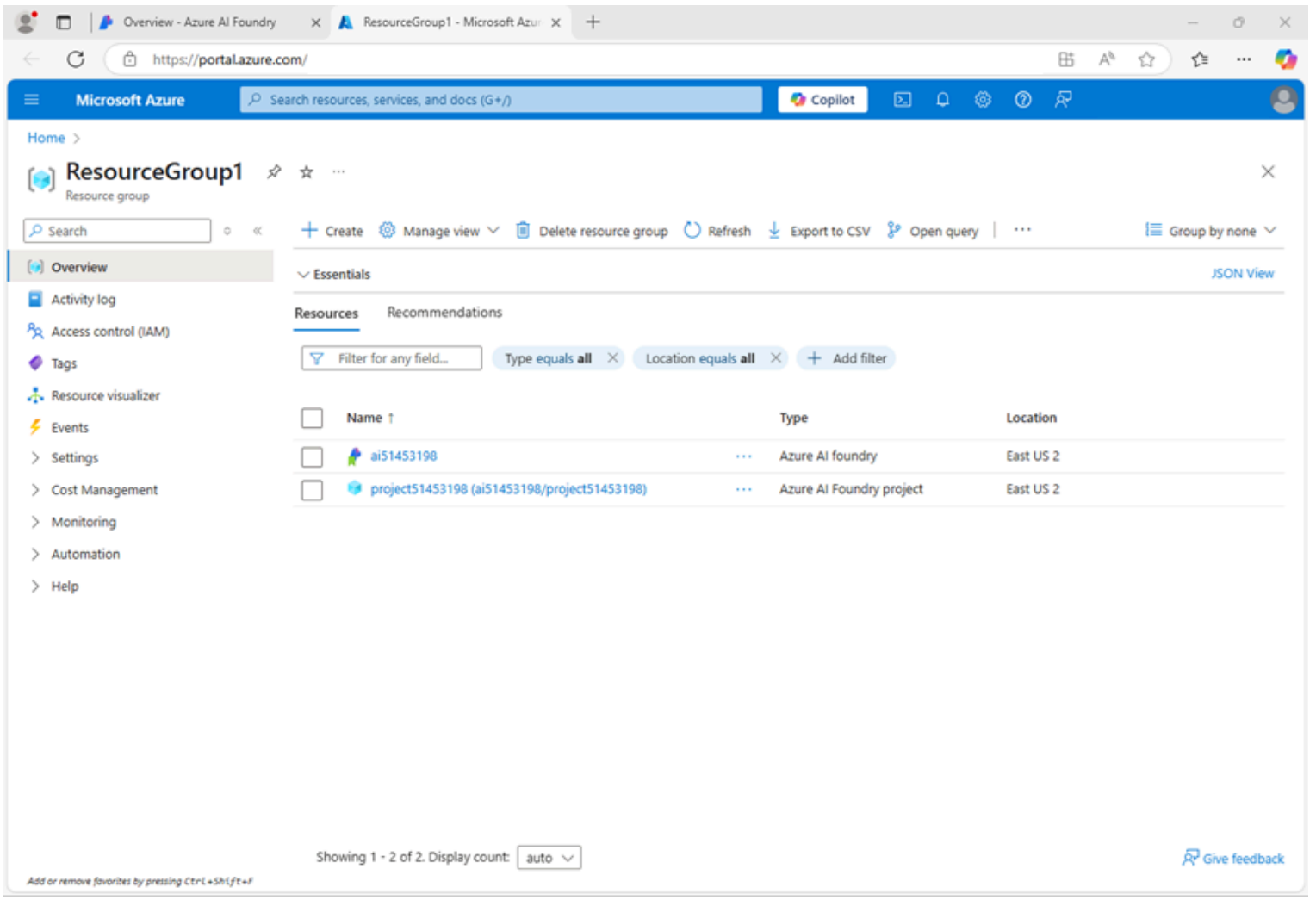


The *resource* level relates to the **Azure AI Foundry** resource that was created to support your project. This resource includes connections to Azure AI Services and Azure AI Foundry models; and provides a centralplace to manage user access to AI development projects.

The *project* level relates to your individual project, where you can add and manage project-specific resources.

9. In the navigation pane, in the section for your Azure AI Foundry resource, select the **Overview** page to view its details.
10. Select the link to the **Resource group** associated with the resource to open a new browser tab and navigate to the Azure portal. Sign in with your Azure credentials if prompted.

11. View the resource group in the Azure portal to see the Azure resources that have been created to support your Azure AI Foundry resource and your project.



Note that the resources have been created in the region you selected when creating the project.

12. Close the Azure portal tab and return to the Azure AI Foundry portal.

Review project endpoints

The Azure AI Foundry project includes a number of *endpoints* that client applications can use to connect to the project and the models and AI services it includes.

1. In the Management center page, in the navigation pane, under your project, select **Go to project**.
2. In the project **Overview** page, view the **Endpoints and keys** section; which contains endpoints and authorization keys that you can use in your application code to access:
    - The Azure AI Foundry project and any models deployed in it.
    - Azure OpenAI in Azure AI Foundry models.
    - Azure AI services

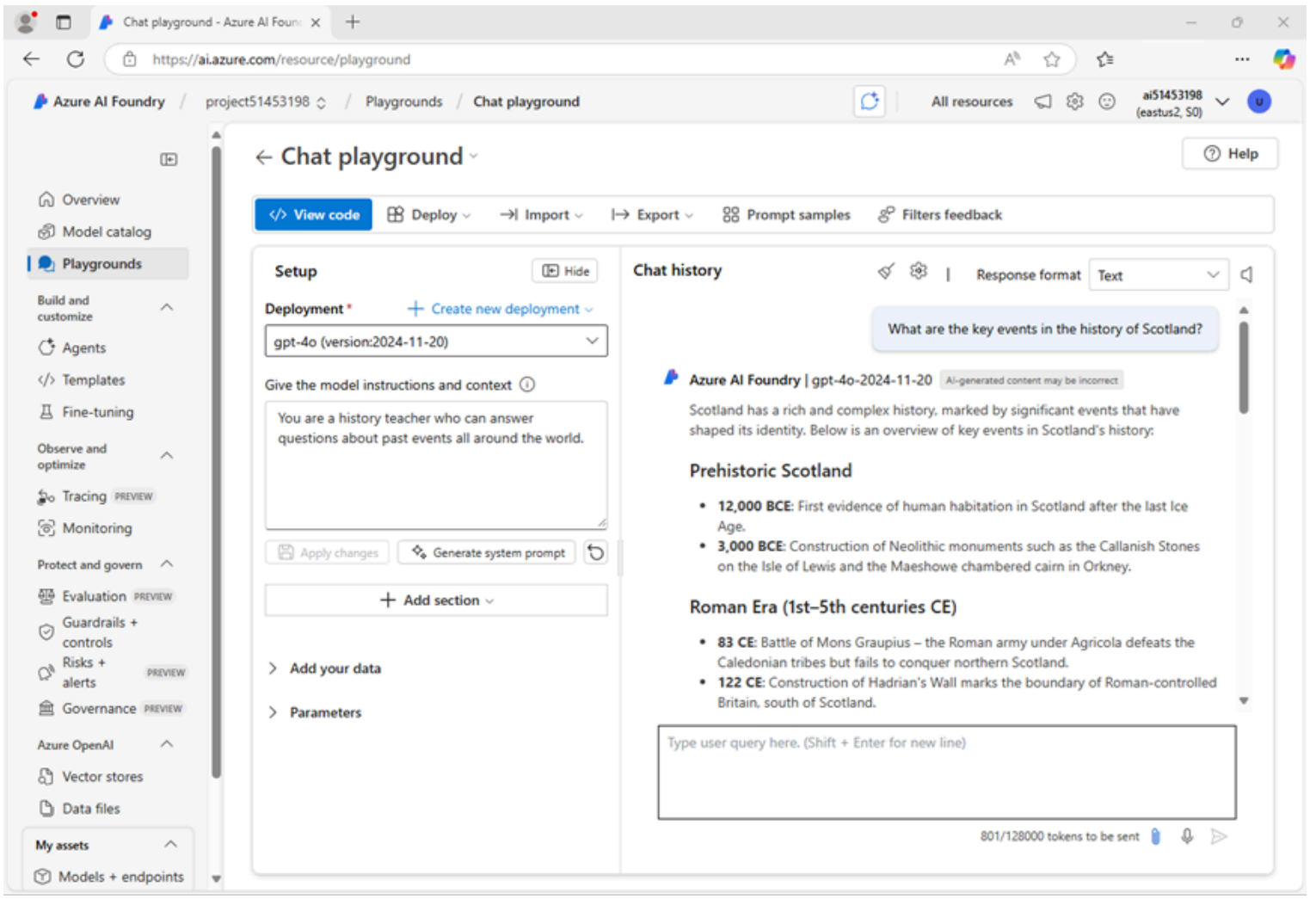
Test a generative AI model

Now that you know something about the configuration of your Azure AI Foundry project, you can return to the chat playground to explore the model you deployed.

1. In the navigation pane on the left for your project, select **Playgrounds**
2. Open the **Chat playground**, and ensure that your **gpt-4o** model deployment is selected in the **Deployment** section.
3. In the **Setup** pane, in the **Give the model instructions and context** box, enter the following instructions:

```
Code
Copy

You are a history teacher who can answer questions about past events all around the world.
```

4. Apply the changes to update the system message.
5. In the chat window, enter a query such as `What are the key events in the history of Scotland?` and view the response:



Summary

In this exercise, you've explored Azure AI Foundry, and seen how to create and manage projects and their related resources.

Clean up

If you've finished exploring Azure AI Foundry portal, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. In the Azure portal at `https://portal.azure.com`, view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Choose and deploy a language model

The Azure AI Foundry model catalog serves as a central repository where you can explore and use a variety of models, facilitating the creation of your generative AI scenario.

In this exercise, you'll explore the model catalog in Azure AI Foundry portal, and compare potential models for a generative AI application that assists in solving problems.

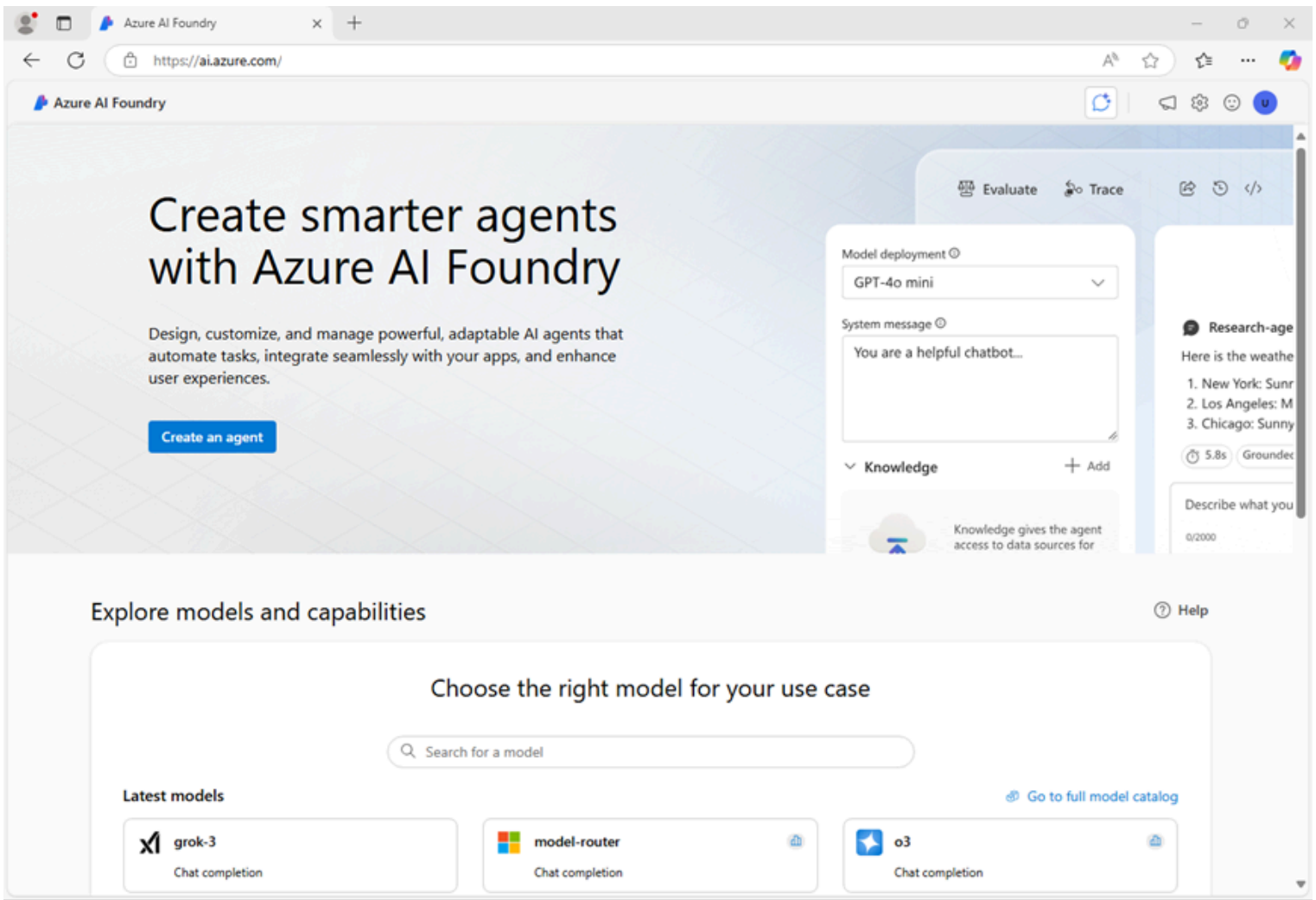
This exercise will take approximately **25** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

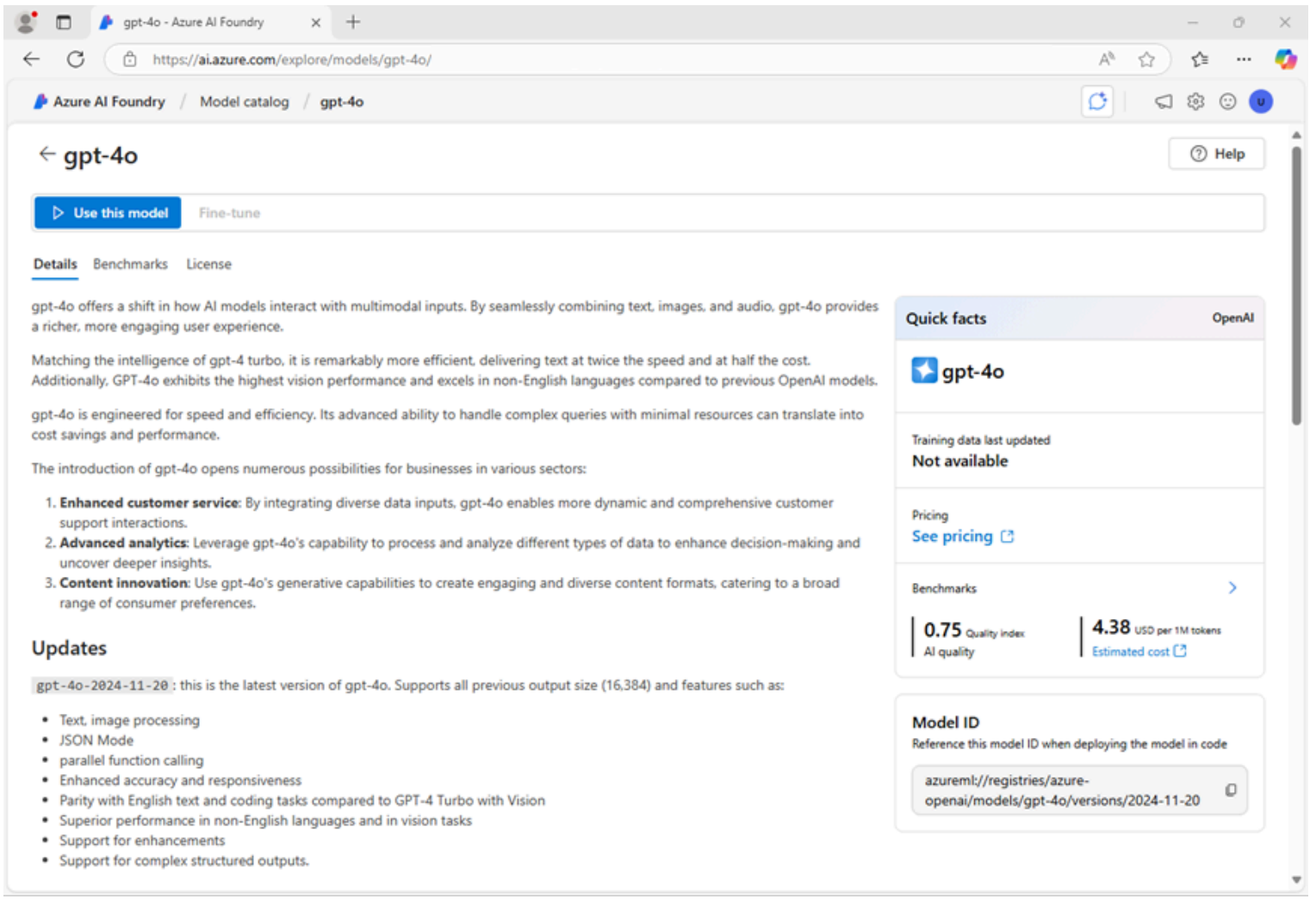
Explore models

Let's start by signing into Azure AI Foundry portal and exploring some of the available models.

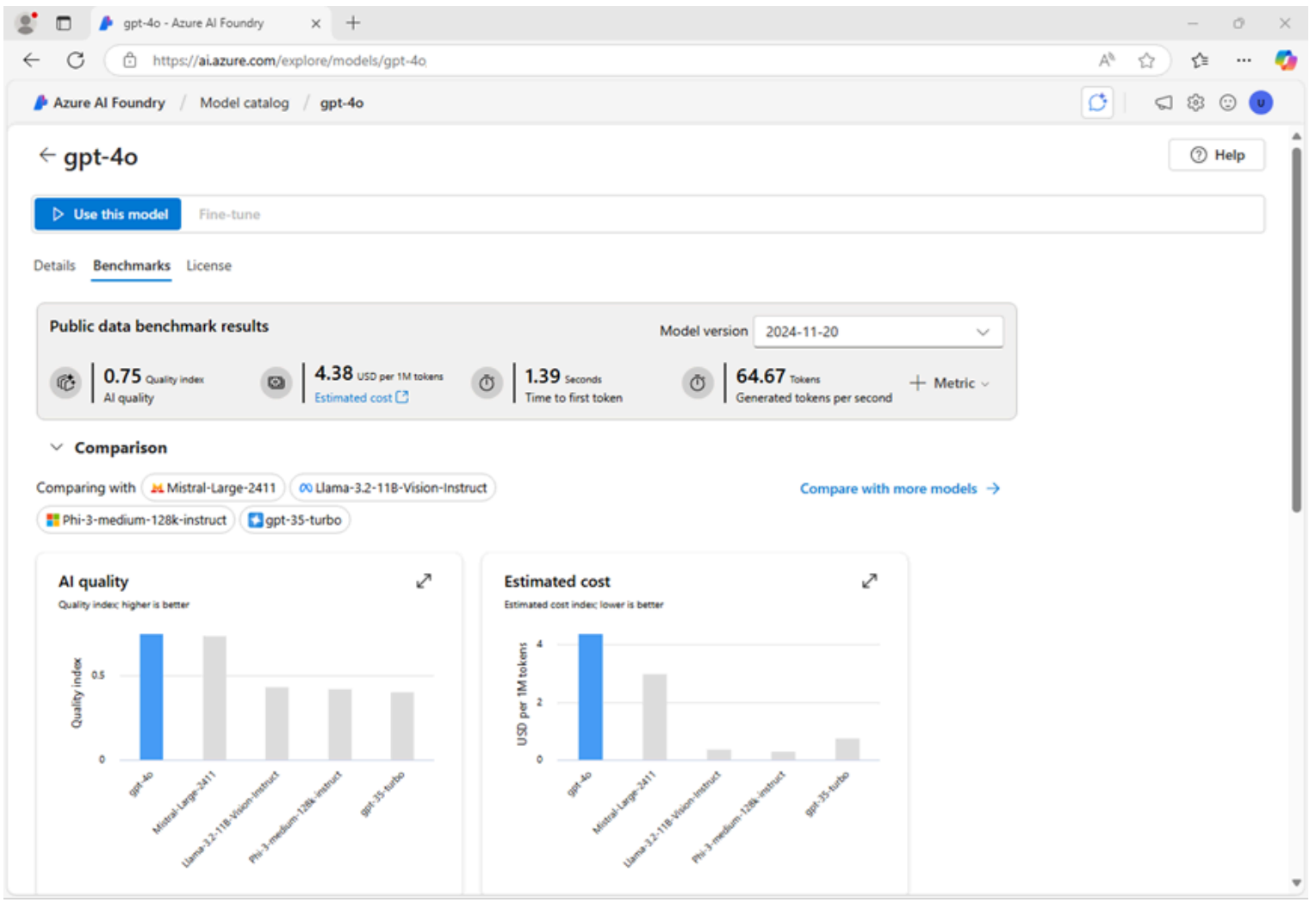
1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. Review the information on the home page.
3. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
4. In the search results, select the **gpt-4o** model to see its details.
5. Read the description and review the other information available on the **Details** tab.



6. On the **gpt-4o** page, view the **Benchmarks** tab to see how the model compares across some standard performance benchmarks with other models that are used in similar scenarios.

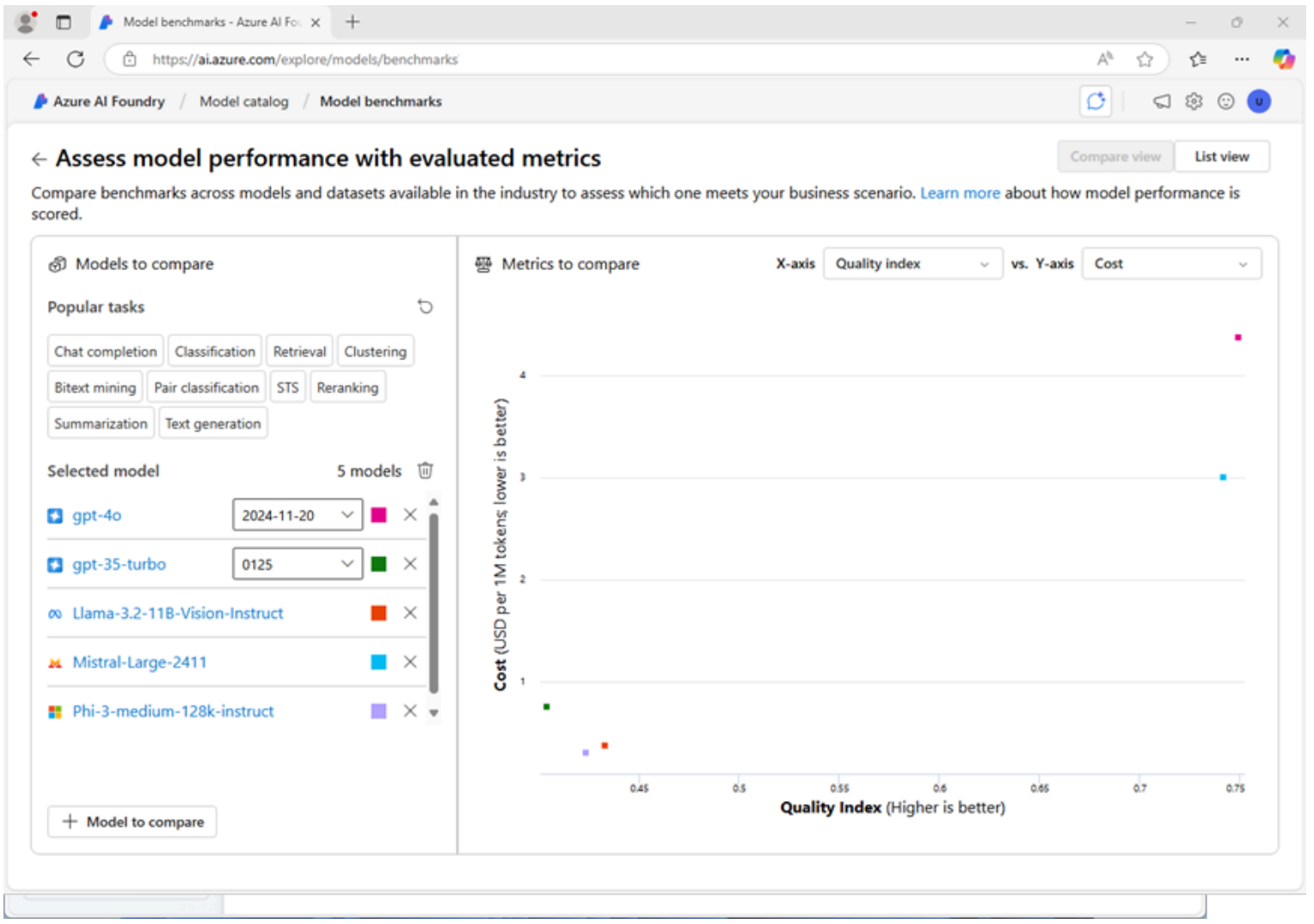


7. Use the back arrow (←) next to the **gpt-4o** page title to return to the model catalog.
8. Search for `Phi-4-mini-instruct` and view the details and benchmarks for the **Phi-4-mini-instruct** model.

Compare models

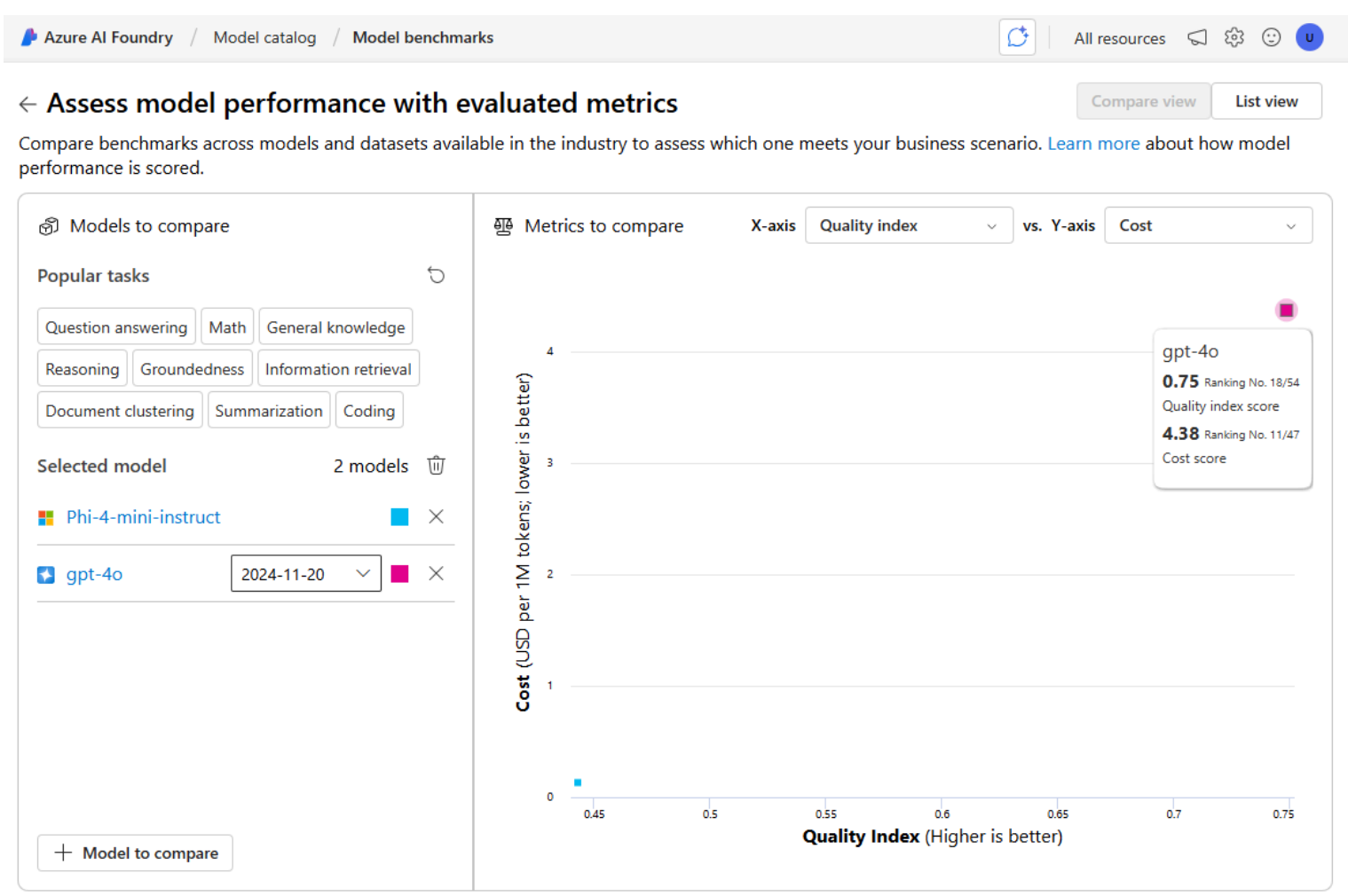
You've reviewed two different models, both of which could be used to implement a generative AI chat application. Now let's compare the metrics for these two models visually.

1. Use the back arrow (←) to return to the model catalog.
2. Select **Compare models**. A visual chart for model comparison is displayed with a selection of common models.



3. In the **Models to compare** pane, note that you can select popular tasks, such as *question answering* to automatically select commonly used models for specific tasks.
4. Use the **Clear all models** (🗑) icon to remove all of the pre-selected models.
5. Use the **+ Model to compare** button to add the **gpt-4o** model to the list. Then use the same button to add the **Phi-4-mini-instruct** model to the list.

6. Review the chart, which compares the models based on **Quality Index** (a standardized score indicating model quality) and **Cost**. You can see the specific values for a model by holding the mouse over the point that represents it in the chart.



7. In the **X-axis** dropdown menu, under **Quality**, select the following metrics and observe each resulting chart before switching to the next:

    - Accuracy
    - Quality index

    Based on the benchmarks, the gpt-4o model looks like offering the best overall performance, but at a higher cost.
8. In the list of models to compare, select the **gpt-4o** model to re-open its benchmarks page.

9. In the page for the **gpt-4o** model page, select the **Overview** tab to view the model details.

Create an Azure AI Foundry project

To use a model, you need to create an Azure AI Foundry *project*.

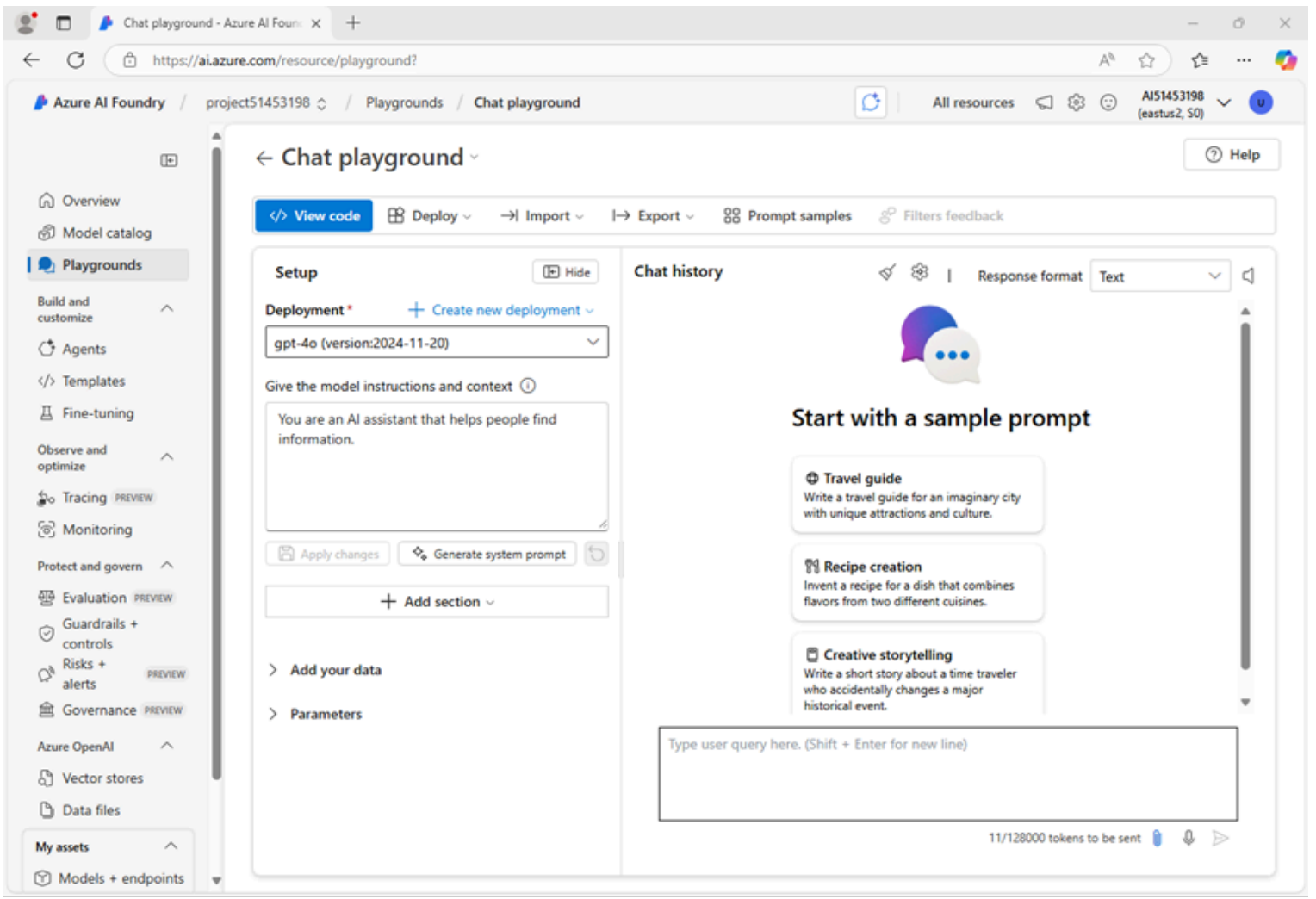
1. At the top of the **gpt-4o** model overview page, select **Use this model**.
2. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
3. In the **Advanced options** section, specify the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any* ***AI Foundry recommended****

    > ❗ * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

4. Select **Create** and wait for your project to be created. If prompted, deploy the gpt-4o model using the **Global standard** deployment type and customize the deployment details to set a **Tokens per minute rate limit** of 50K (or the maximum available if less than 50K).

    > ❗ **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

5. When your project is created, the chat playground will be opened automatically so you can test your model:



Chat with the *gpt-4o* model

Now that you have a model deployment, you can use the playground to test it.

1. In the chat playground, in the **Setup** pane, ensure that your **gpt-4o** model is selected and in the **Give the model instructions and context** field, set the system prompt to `You are an AI assistant that helps solve problems.`

2. Select **Apply changes** to update the system prompt.

3. In the chat window, enter the following query

```
I have a fox, a chicken, and a bag of grain that I need to take over a river in a boat. I can only take one thing at a time. If I leave the chicken and the grain unattended, the chicken will eat the grain. If I leave the fox and the chicken unattended, the fox will eat the chicken. How can I get all three things across the river without anything being eaten?
```

4. View the response. Then, enter the following follow-up query:

```
Explain your reasoning.
```

Deploy another model

When you created your project, the **gpt-4o** model you selected was automatically deployed. Let's deploy the ***Phi-4-mini-instruct** model you also considered.

1. In the navigation bar on the left, in the **My assets** section, select **Models + endpoints**.
2. In the **Model deployments** tab, in the **+ Deploy model** drop-down list, select **Deploy base model**. Then search for `Phi-4-mini-instruct` and confirm you selection.
3. Agree to the model license.
4. Deploy a **Phi-4-mini-instruct** model with the following settings:
    - **Deployment name**: *A valid name for your model deployment*
    - **Deployment type**: Global Standard
    - **Deployment details**: *Use the default settings*
5. Wait for the deployment to complete.

Chat with the *Phi-4* model

Now let's chat with the new model in the playground.

1. In the navigation bar, select **Playgrounds**. Then select the **Chat playground**.
2. In the chat playground, in the **Setup** pane, ensure that your **Phi-4-mini-instruct** model is selected and in the chat box, provide the first line as `System message: You are an AI assistant that helps solve problems.` (the same system prompt you used to test the gpt-4o model, but since there is no system message setup, we're providing it in the first chat for context.)

3. On a new line in the chat window (below your system message), enter the following query

```
I have a fox, a chicken, and a bag of grain that I need to take over a river in a boat. I can only take one thing at a time. If I leave the chicken and the grain unattended, the chicken will eat the grain. If I leave the fox and the chicken unattended, the fox will eat the chicken. How can I get all three things across the river without anything being eaten?
```

4. View the response. Then, enter the following follow-up query:

```
Explain your reasoning.
```

Perform a further comparison

1. Use the drop-down list in the **Setup** pane to switch between your models, testing both models with the following puzzle (the correct answer is 40!):

Code Copy

```
I have 53 socks in my drawer: 21 identical blue, 15 identical black and 17 identical red.
The lights are out, and it is completely dark. How many socks must I take out to make 100
percent certain I have at least one pair of black socks?
```

Reflect on the models

You've compared two models, which may vary in terms of both their ability to generate appropriate responses and in their cost. In any generative scenario, you need to find a model with the right balance of suitability for the task you need it to perform and the cost of using the model for the number of requests you expect it to have to handle.

The details and benchmarks provided in the model catalog, along with the ability to visually compare models provides a useful starting point when identifying candidate models for a generative AI solution. You can then test candidate models with a variety of system and user prompts in the chat playground.

Clean up

If you've finished exploring Azure AI Foundry portal, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Open the Azure portal and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Create a generative AI chat app

In this exercise, you use the Azure AI Foundry SDK to create a simple chat app that connects to a project and chats with a language model.

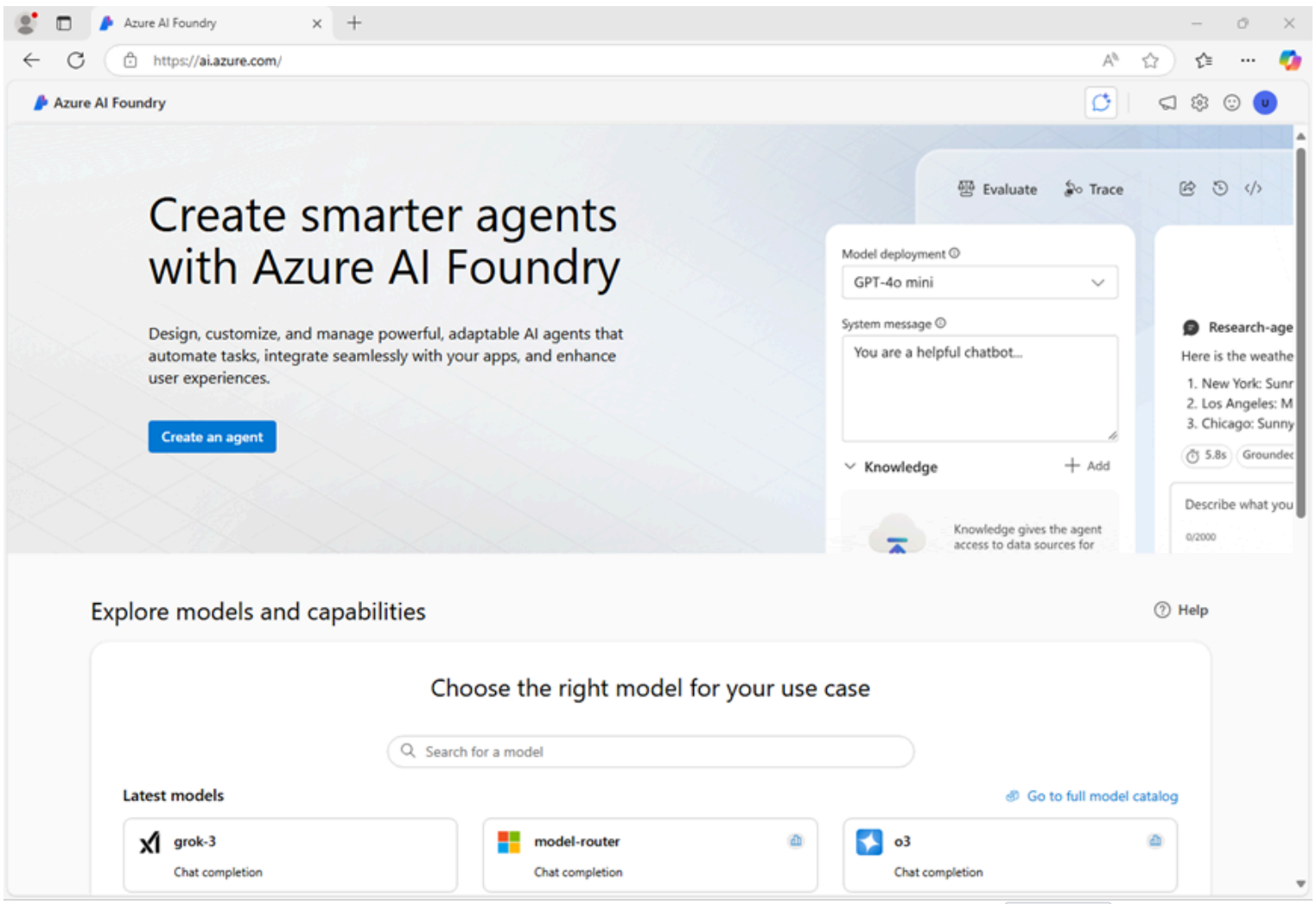
This exercise takes approximately **40** minutes.

> **Note**: This exercise is based on pre-release SDKs, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

Deploy a model in an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):

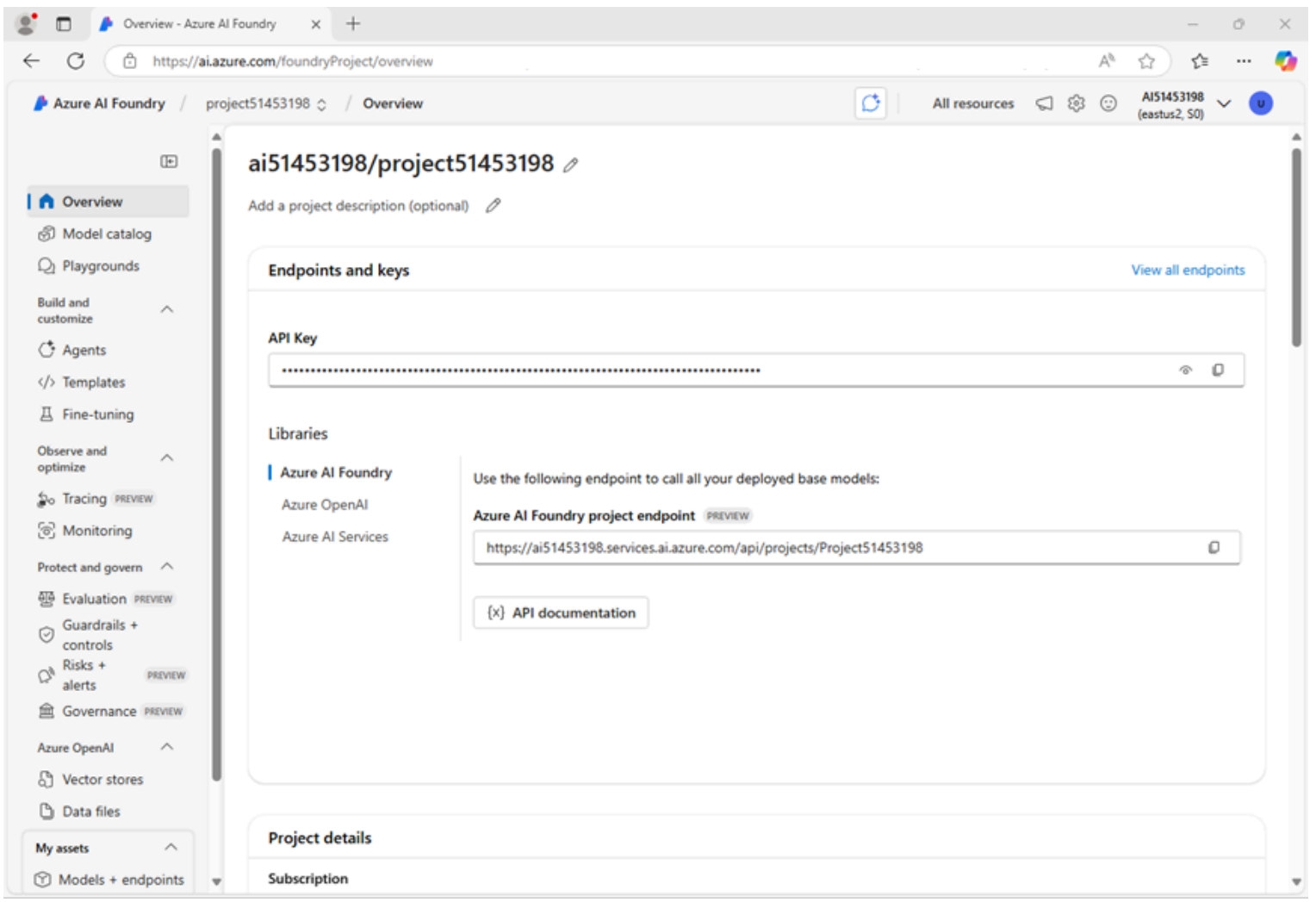


2. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Select **Customize** and specify the following settings for your project:
   - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
   - **Subscription**: *Your Azure subscription*
   - **Resource group**: *Create or select a resource group*
   - **Region**: *Select any* ***AI Services supported location****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

6. Select **Create** and wait for your project, including the gpt-4 model deployment you selected, to be created.
7. When your project is created, the chat playground will be opened automatically.
8. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**. You can confirm this by viewing the deployment in the **Models and endpoints** page (just open that page in the navigation pane on the left).

9. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



Create a client application to chat with the model

Now that you have deployed a model, you can use the Azure AI Foundry and Azure OpenAI SDKs to develop an application that chats with it.

> **Tip**: You can choose to develop your solution using Python or Microsoft C#. Follow the instructions in the appropriate section for your chosen language.

Prepare the application configuration

1. In the Azure AI Foundry portal, view the **Overview** page for your project.
2. In the **Endpoints and keys** area, ensure that the **Azure AI Foundry** library is selected and view the **Azure AI Foundry project endpoint**. You'll use this endpoint to connect to your project and model in a client application.

   > **Note**: You can also use the Azure OpenAI endpoint!

3. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

4. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

5. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

6. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-foundry -f
   git clone https://github.com/microsoftlearning/mslearn-ai-studio mslearn-ai-foundry
   ```

   > **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

7. After the repo has been cloned, navigate to the folder containing the chat application code files and view them:

   Use the commands below depending on your choice of programming language.

   Python

   Code Copy

   ```
   cd mslearn-ai-foundry/labfiles/chat-app/python
   ls -a -l
   ```

   C#

   Code Copy

   ```
   cd mslearn-ai-foundry/labfiles/chat-app/c-sharp
   ls -a -l
   ```

   The folder contains a code file as well as a configuration file for application settings and a file defining the project runtime and package requrirements.

8. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Python

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-identity azure-ai-projects openai
   ```

   C#

   Code Copy

```
dotnet add package Azure.Identity --prerelease
dotnet add package Azure.AI.Projects --prerelease
dotnet add package Azure.AI.OpenAI --prerelease
```

9. Enter the following command to edit the configuration file that has been provided:

Python

Code Copy

```
code .env
```

C#

Code Copy

```
code appsettings.json
```

The file is opened in a code editor.

10. In the code file, replace the **your_project_endpoint** placeholder with the **Azure AI Foundry project endpoint** for your project (copied from the **Overview** page in the Azure AI Foundry portal); and the **your_model_deployment** placeholder with the name of your gpt-4 model deployment.
11. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Write code to connect to your project and chat with your model

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Python

Code Copy

```
code chat-app.py
```

C#

Code Copy

```
code Program.cs
```

2. In the code file, note the existing statements that have been added at the top of the file to import the necessary SDK namespaces. Then, find the comment **Add references**, and add the following code to reference the namespaces in the libraries you installed previously:

Python

Code Copy

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient
from openai import AzureOpenAI
```

C#

C# Copy

```
// Add references
using Azure.Identity;
using Azure.AI.Projects;
using Azure.AI.OpenAI;
using OpenAI.Chat;
```

3. In the **main** function, under the comment **Get configuration settings**, note that the code loads the project connection string and model deployment name values you defined in the configuration file.
4. Find the comment **Initialize the project client**, and add the following code to connect to your Azure AI Foundry project:

> **Tip**: Be careful to maintain the correct indentation level for your code.

Python

Code Copy

```
# Initialize the project client
project_client = AIProjectClient(
         credential=DefaultAzureCredential(
             exclude_environment_credential=True,
             exclude_managed_identity_credential=True
         ),
         endpoint=project_endpoint,
     )
```

C#

C# Copy

```
// Initialize the project client
DefaultAzureCredentialOptions options = new()
        { ExcludeEnvironmentCredential = true,
          ExcludeManagedIdentityCredential = true };
var projectClient = new AIProjectClient(
          new Uri(project_connection),
          new DefaultAzureCredential(options));
```

5. Find the comment **Get a chat client**, and add the following code to create a client object for chatting with a model:

Python

Code Copy

```
# Get a chat client
openai_client = project_client.inference.get_azure_openai_client(api_version="2024-10-21")
```

C#

C# Copy

```
// Get a chat client
ChatClient openaiClient = projectClient.GetAzureOpenAIChatClient(deployment:
model_deployment, connectionName: null, apiVersion: "2024-10-21");
```

6. Find the comment **Initialize prompt with system message**, and add the following code to initialize a collection of messages with a system prompt.

Python

Code Copy

```
# Initialize prompt with system message
prompt = [
        {"role": "system", "content": "You are a helpful AI assistant that answers
questions."}
    ]
```

C#

C# Copy

```
// Initialize prompt with system message
var prompt = new List<ChatMessage>(){
                new SystemChatMessage("You are a helpful AI assistant that answers
questions.")
            };
```

7. Note that the code includes a loop to allow a user to input a prompt until they enter “quit”. Then in the loop section, find the comment **Get a chat completion** and add the following code to add the user input to the prompt, retrieve the completion from your model, and add the completion to the prompt (so that you retain chat history for future iterations):

Python

Code Copy

```
# Get a chat completion
prompt.append({"role": "user", "content": input_text})
response = openai_client.chat.completions.create(
        model=model_deployment,
        messages=prompt)
completion = response.choices[0].message.content
print(completion)
prompt.append({"role": "assistant", "content": completion})
```

C#

C# Copy

```
// Get a chat completion
prompt.Add(new UserChatMessage(input_text));
ChatCompletion completion = openaiClient.CompleteChat(prompt);
var completionText = completion.Content[0].Text;
Console.WriteLine(completionText);
prompt.Add(new AssistantChatMessage(completionText));
```

8. Use the **CTRL+S** command to save your changes to the code file.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. After you have signed in, enter the following command to run the application:

Python

Code Copy

```
python chat-app.py
```

C#

Code Copy

```
dotnet run
```

> **Tip**: If a compilation error occurs because .NET version 9.0 is not installed, use the `dotnet --version` command to determine the version of .NET installed in your environment and then edit the **chat_app.csproj** file in the code folder to update the **TargetFramework** setting accordingly.

4. When prompted, enter a question, such as `What is the fastest animal on Earth?` and review the response from your generative AI model.
5. Try some follow-up questions, like `Where can I see one?` or `Are they endangered?`. The conversation should continue, using the chat history as context for each iteration.
6. When you're finished, enter `quit` to exit the program.

> **Tip**: If the app fails because the rate limit is exceeded. Wait a few seconds and try again. If there is insufficient quota available in your subscription, the model may not be able to respond.

Summary

In this exercise, you used the Azure AI Foundry SDK to create a client application for a generative AI model that you deployed in an Azure AI Foundry project.

Clean up

If you've finished exploring Azure AI Foundry portal, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Open the Azure portal and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Use a prompt flow to manage conversation in a chat app

In this exercise, you'll use Azure AI Foundry portal's prompt flow to create a custom chat app that uses a user prompt and chat history as inputs, and uses a GPT model from Azure OpenAI to generate an output.

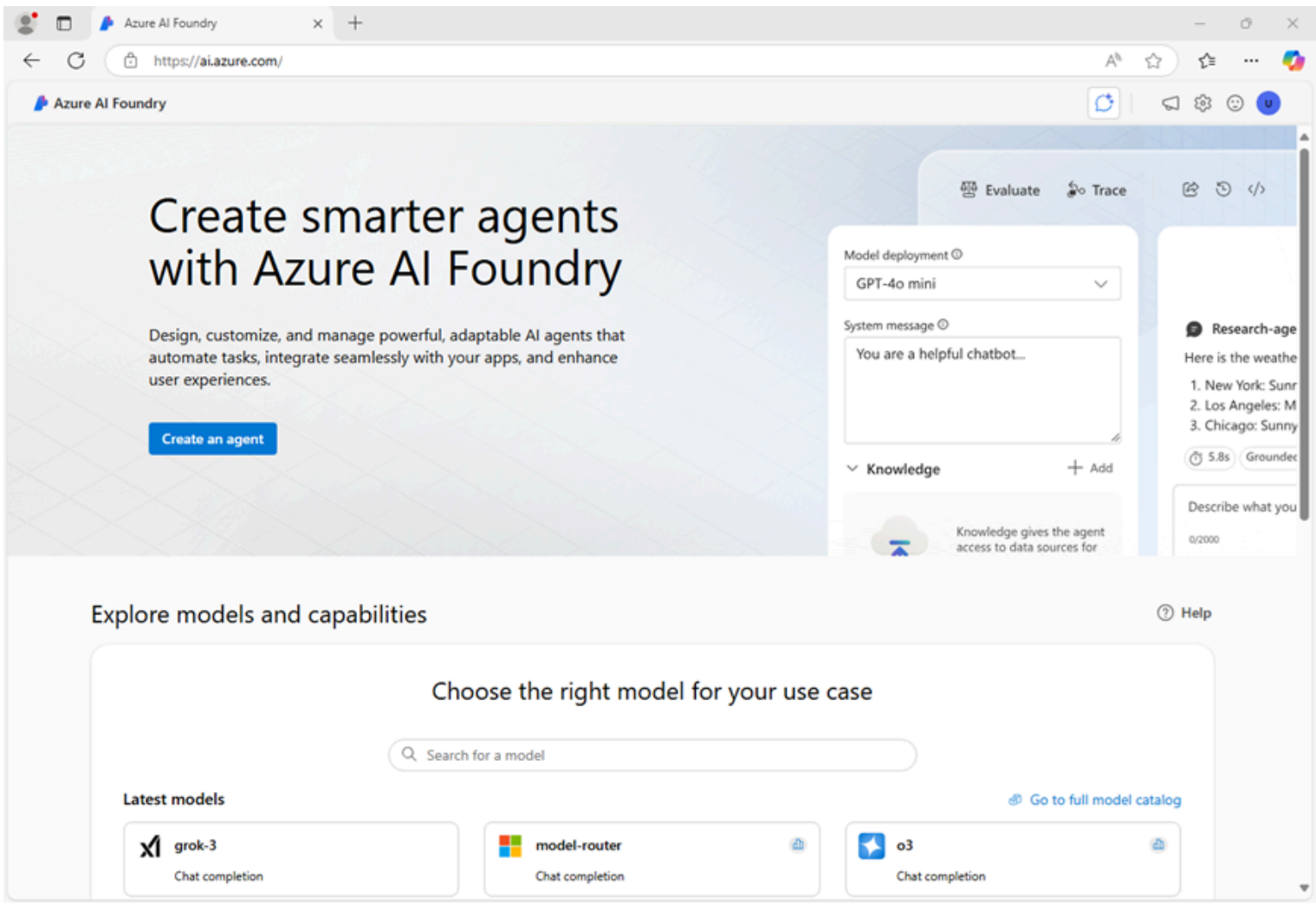
This exercise will take approximately **30** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry hub and project

The features of Azure AI Foundry we're going to use in this exercise require a project that is based on an Azure AI Foundry *hub* resource.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: East US 2 or Sweden Central (*In the event of a quota limit being exceeded later in the exercise, you may need to create another resource in a different region.*)

> **Note**: If you're working in an Azure subscription in which policies are used to restrict allowable resource names, you may need to use the link at the bottom of the **Create a new project** dialog box to create the hub using the Azure portal.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

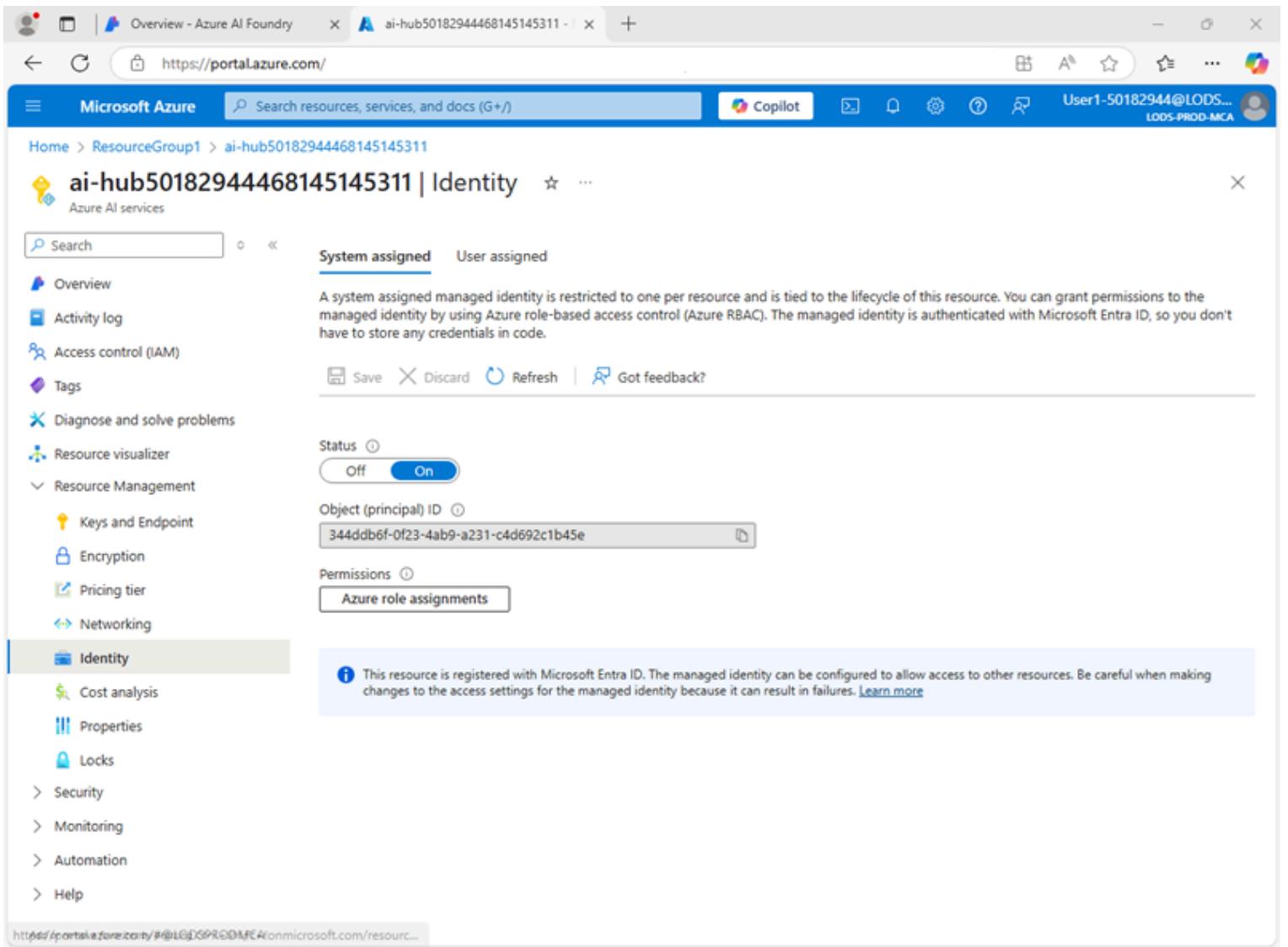
4. Wait for your project to be created.

Configure resource authorization

The prompt flow tools in Azure AI Foundry create file-based assets that define the prompt flow in a folder in blob storage. Before exploring prompt flow, let's ensure that your Azure AI Foundry resource has the required access to the blob store so it can read them.

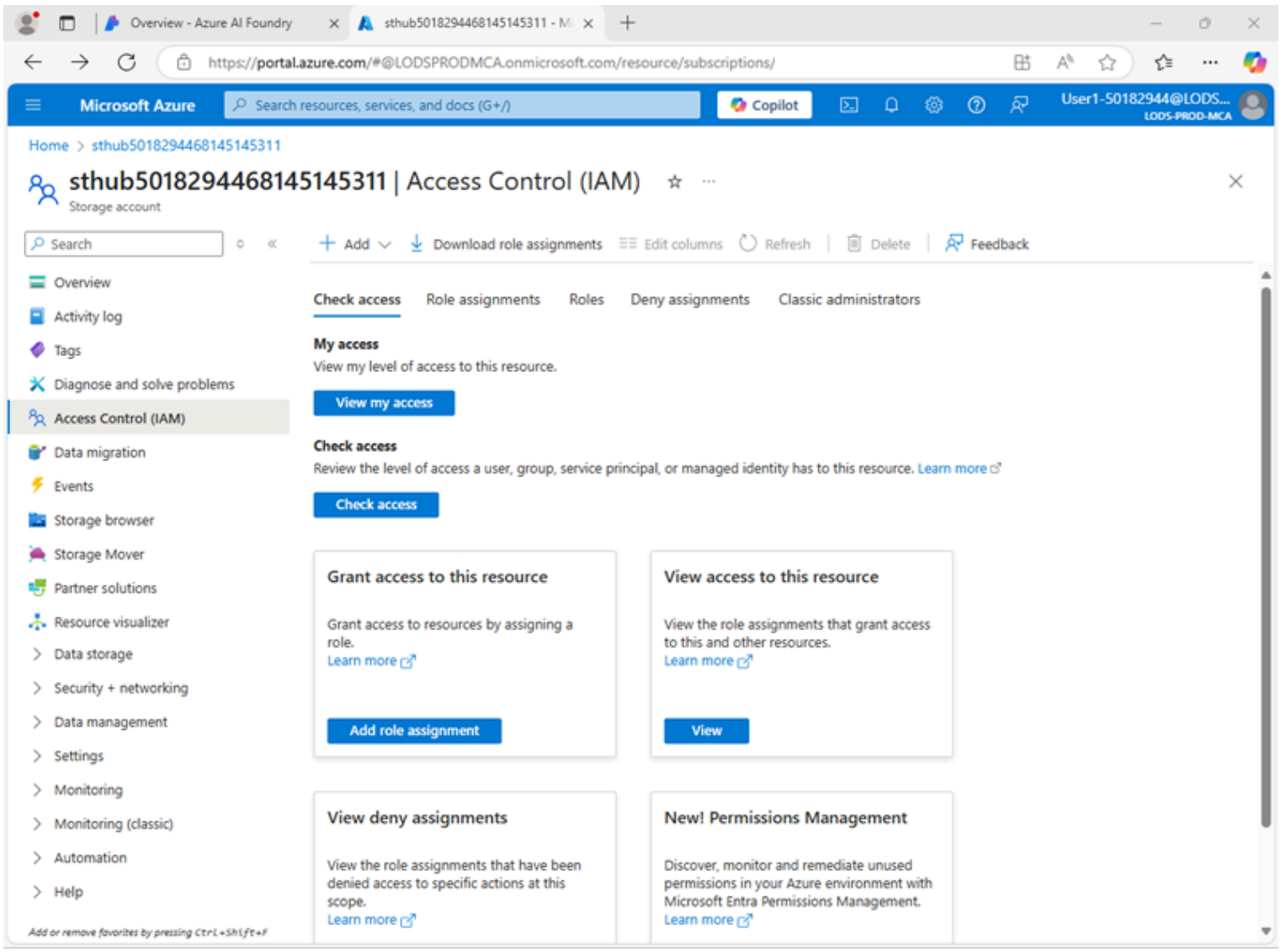
1. In a new browser tab, open the Azure portal at `https://portal.azure.com`, signing in with your Azure credentials if prompted; and view the resource group containing your Azure AI hub resources.

2. Select the **Azure AI Foundry** resource for your hub to open it. Then expand its **Resource Management** section and select the **Identity** page:

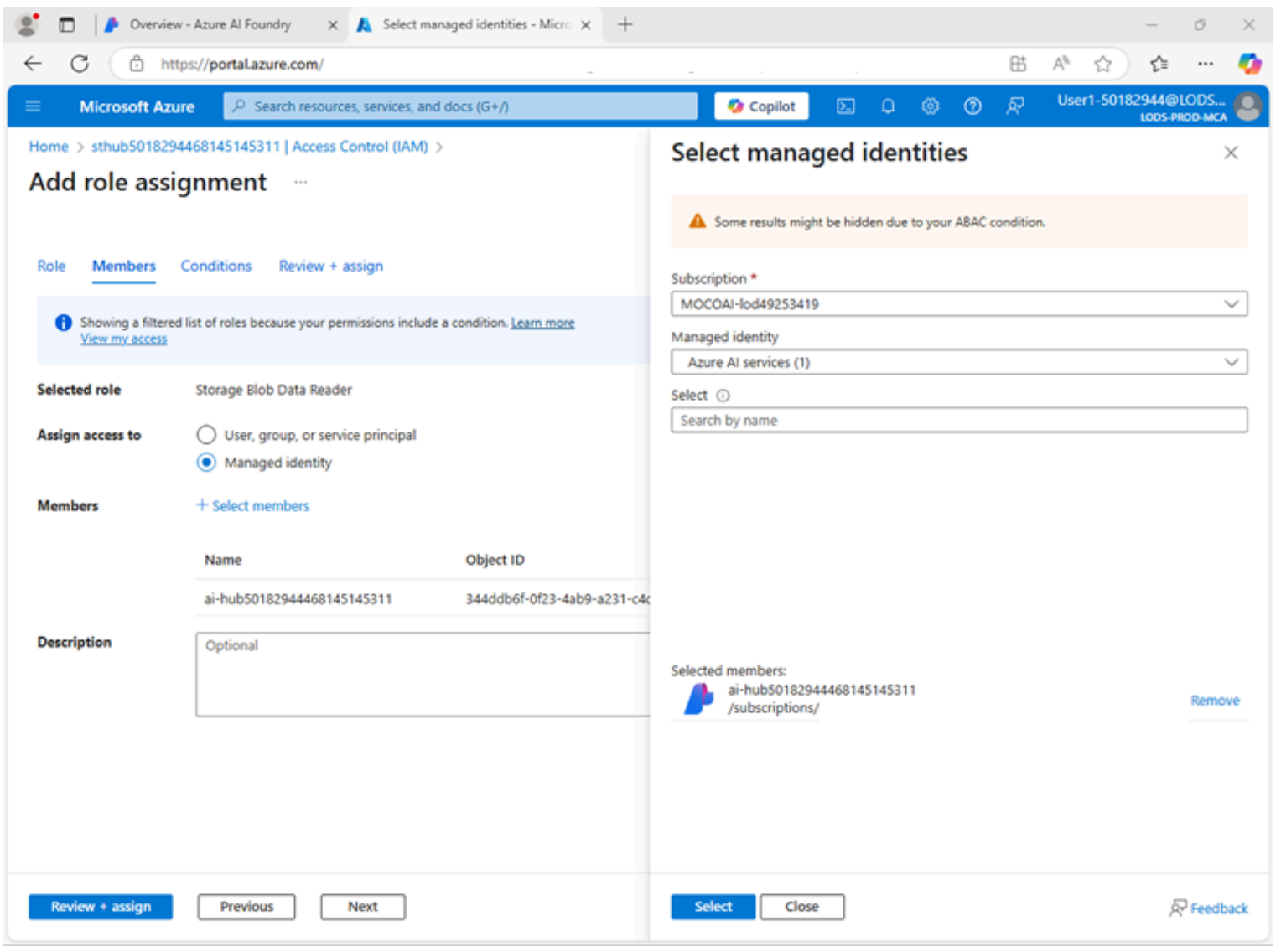


3. If the status of the system assigned identity is **Off**, switch it **On** and save your changes. Then wait for the change of status to be confirmed.

4. Return to the resource group page, and then select the **Storage account** resource for your hub and view its **Access Control (IAM)** page:



5. Add a role assignment to the `Storage blob data reader` role for the managed identity used by your Azure AI Foundry resource:



6. When you've reviewed and assigned the role access to allow the Azure AI Foundry managed identity to read blobs in the storage account, close the Azure portal tab and return to the Azure AI Foundry portal.

Deploy a generative AI model

Now you're ready to deploy a generative AI language model to support your prompt flow application.

1. In the pane on the left for your project, in the **My assets** section, select the **Models + endpoints** page.
2. In the **Models + endpoints** page, in the **Model deployments** tab, in the **+ Deploy model** menu, select **Deploy base model**.
3. Search for the **gpt-4o** model in the list, and then select and confirm it.
4. Deploy the model with the following settings by selecting **Customize** in the deployment details:

- **Deployment name**: *A valid name for your model deployment*
- **Deployment type**: Global Standard
- **Automatic version update**: Enabled
- **Model version**: *Select the most recent available version*
- **Connected AI resource**: *Select your Azure OpenAI resource connection*
- **Tokens per Minute Rate Limit (thousands)**: 50K *(or the maximum available in your subscription if less than 50K)*
- **Content filter**: DefaultV2

> **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

5. Wait for the deployment to complete.

Create a prompt flow

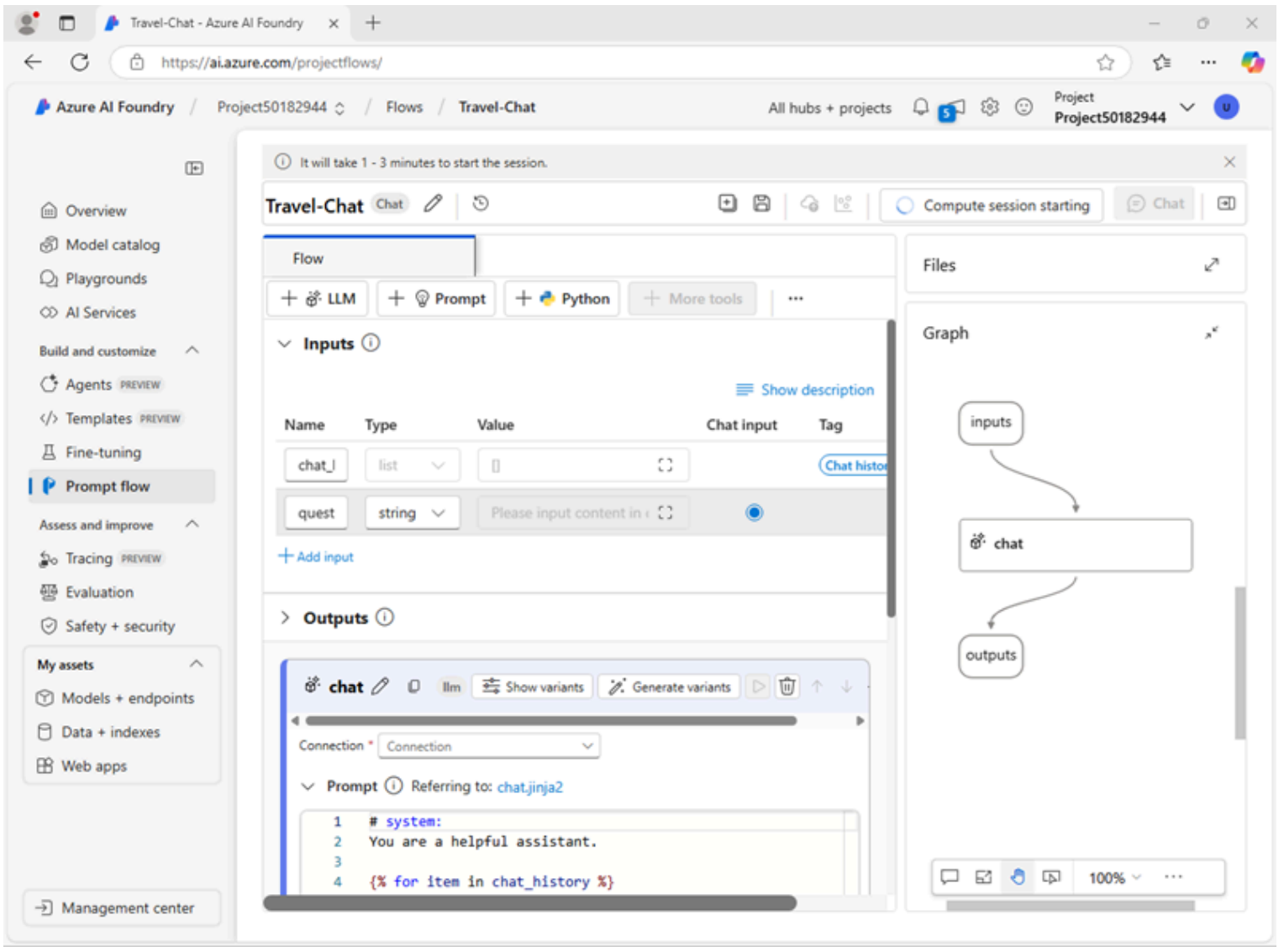
A prompt flow provides a way to orchestrate prompts and other activities to define an interaction with a generative AI model. In this exercise, you'll use a template to create a basic chat flow for an AI assistant in a travel agency.

1. In the Azure AI Foundry portal navigation bar, in the **Build and customize** section, select **Prompt flow**.
2. Create a new flow based on the **Chat flow** template, specifying `Travel-Chat` as the folder name.

   A simple chat flow is created for you.

   > **Tip**: If a permissions error occurs. Wait a few minutes and try again, specifying a different flow name if necessary.

3. To be able to test your flow, you need compute, and it can take a while to start; so select **Start compute session** to get it started while you explore and modify the default flow.
4. View the prompt flow, which consists of a series of *inputs*, *outputs*, and *tools*. You can expand and edit the properties of these objects in the editing panes on the left, and view the overall flow as a graph on the right:



5. View the **Inputs** pane, and note that there are two inputs (chat history and the user's question)

6. View the **Outputs** pane and note that there's an output to reflect the model's answer.
7. View the **Chat** LLM tool pane, which contains the information needed to submit a prompt to the model.
8. In the **Chat** LLM tool pane, for **Connection**, select the connection for the Azure OpenAI service resource in your AI hub. Then configure the following connection properties:
    - **Api**: chat
    - **deployment_name**: *The gpt-4o model you deployed*
    - **response_format**: {"type":"text"}
9. Modify the **Prompt** field as follows:

Code Copy

```
# system:
**Objective**: Assist users with travel-related inquiries, offering tips, advice, and recommendations as a knowledgeable travel agent.

**Capabilities**:
- Provide up-to-date travel information, including destinations, accommodations, transportation, and local attractions.
- Offer personalized travel suggestions based on user preferences, budget, and travel dates.
- Share tips on packing, safety, and navigating travel disruptions.
- Help with itinerary planning, including optimal routes and must-see landmarks.
- Answer common travel questions and provide solutions to potential travel issues.

**Instructions**:
1. Engage with the user in a friendly and professional manner, as a travel agent would.
2. Use available resources to provide accurate and relevant travel information.
3. Tailor responses to the user's specific travel needs and interests.
4. Ensure recommendations are practical and consider the user's safety and comfort.
5. Encourage the user to ask follow-up questions for further assistance.



# user:
```

Read the prompt you added so you are familiar with it. It consists of a system message (which includes an objective, a definition of its capabilities, and some instructions), and the chat history (ordered to show each user question input and each previous assistant answer output)

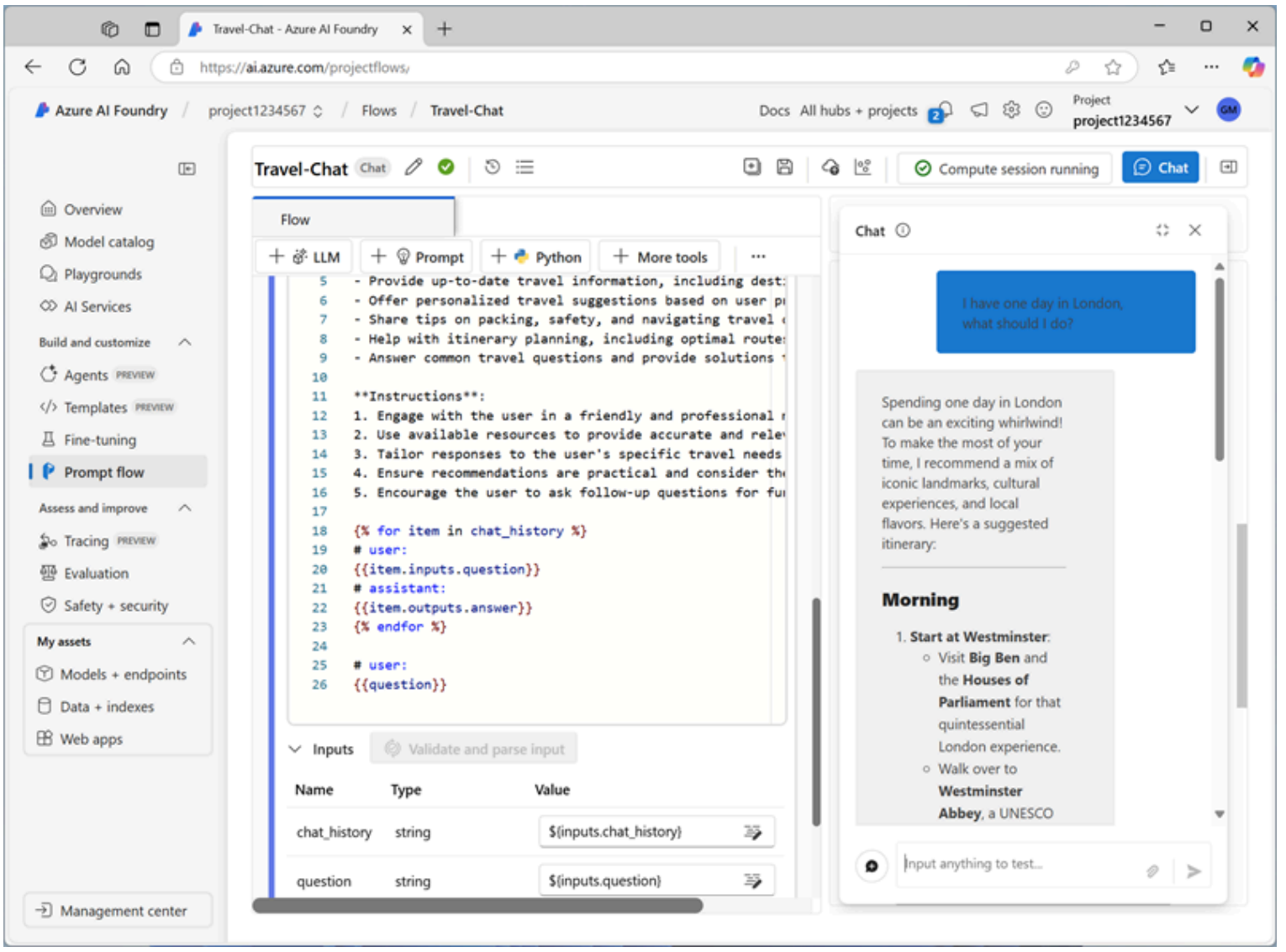
10. In the **Inputs** section for the **Chat** LLM tool (under the prompt), ensure the following variables are set:
    - **question** (string): ${inputs.question}
    - **chat_history** (string): ${inputs.chat_history}
11. Save the changes to the flow.

> **Note**: In this exercise, we'll stick to a simple chat flow, but note that the prompt flow editor includes many other tools that you could add to the flow, enabling you to create complex logic to orchstrate conversations.

Test the flow

Now that you've developed the flow, you can use the chat window to test it.

1. Ensure the compute session is running. If not, wait for it to start.
2. On the toolbar, select **Chat** to open the **Chat** pane, and wait for the chat to initialize.
3. Enter the query: `I have one day in London, what should I do?` and review the output. The Chat pane should look similar to this:



Deploy the flow

When you're satisfied with the behavior of the flow you created, you can deploy the flow.

> **Note**: Deployment can take a long time, and can be impacted by capacity constraints in your subscription or tenant.

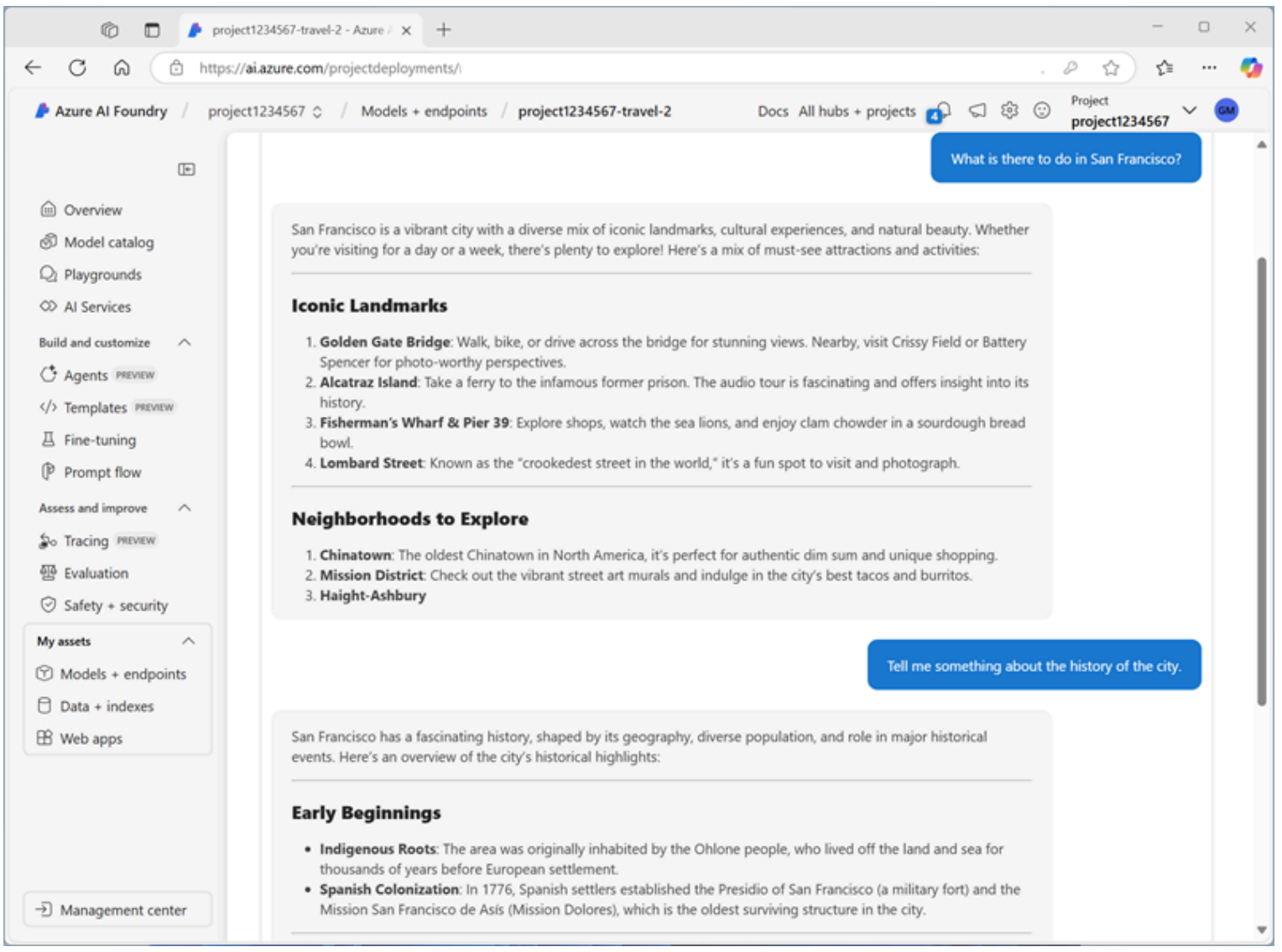
1. On the toolbar, select **Deploy** and deploy the flow with the following settings:
    - **Basic settings**:
        - **Endpoint**: New
        - **Endpoint name**: *Enter a unique name*
        - **Deployment name**: *Enter a unique name*
        - **Virtual machine**: Standard_DS3_v2
        - **Instance count**: 1
        - **Inferencing data collection**: Disabled
    - **Advanced settings**:
        - *Use the default settings*
2. In Azure AI Foundry portal, in the navigation pane, in the **My assets** section, select the **Models + endpoints** page.

    If the page opens for your gpt-4o model, use its **back** button to view all models and endpoints.
3. Initially, the page may show only your model deployments. It may take some time before the deployment is listed, and even longer before it's successfully created.
4. When the deployment has *succeeded*, select it. Then, view its **Test** page.

    > **Tip**: If the test page describes the endpoint as unhealthy, return to the **models and endpoints** and wait a minute or so before refreshing the view and selecting the endpoint again.

5. Enter the prompt `What is there to do in San Francisco?` and review the response.
6. Enter the prompt `Tell me something about the history of the city.` and review the response.

The test pane should look similar to this:



7. View the **Consume** page for the endpoint, and note that it contains connection information and sample code that you can use to build a client application for your endpoint - enabling you to integrate the prompt flow solution into an application as a generative AI application.

Clean up

When you finish exploring prompt flow, you should delete the resources you've created to avoid unnecessary Azure costs.

- Navigate to the Azure portal at `https://portal.azure.com`.
- In the Azure portal, on the **Home** page, select **Resource groups**.
- Select the resource group that you created for this exercise.
- At the top of the **Overview** page for your resource group, select **Delete resource group**.
- Enter the resource group name to confirm you want to delete it, and select **Delete**.

Create a generative AI app that uses your own data

Retrieval Augmented Generation (RAG) is a technique used to build applications that integrate data from custom data sources into a prompt for a generative AI model. RAG is a commonly used pattern for developing generative AI apps - chat-based applications that use a language model to interpret inputs and generate appropriate responses.

In this exercise, you'll use Azure AI Foundry to integrate custom data into a generative AI solution.

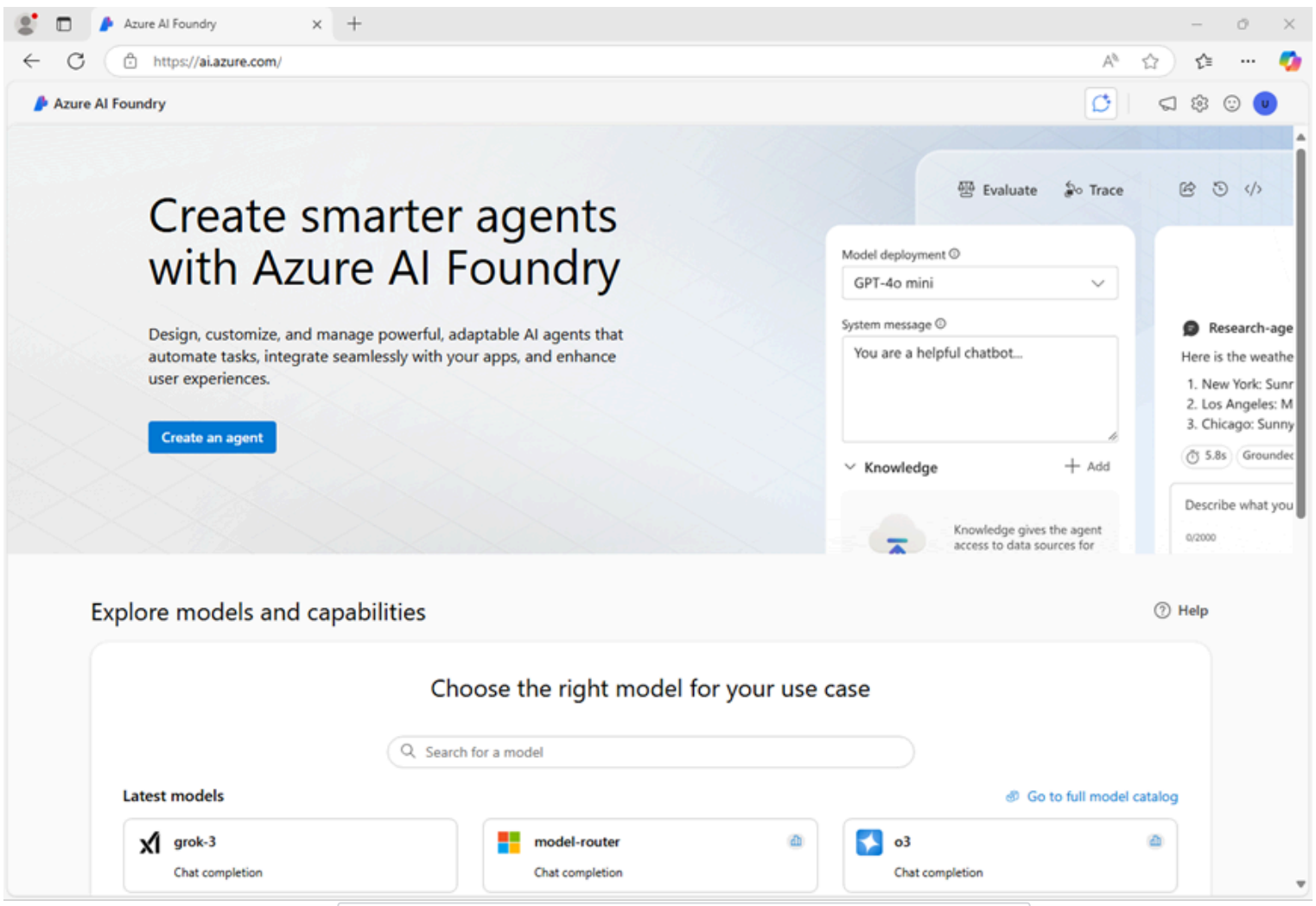
This exercise takes approximately **45** minutes.

> **Note**: This exercise is based on pre-release services, which may be subject to change.

Create an Azure AI Foundry hub and project

The features of Azure AI Foundry we're going to use in this exercise require a project that is based on an Azure AI Foundry *hub* resource.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: East US 2 or Sweden Central (*In the event of a quota limit being exceeded later in the exercise, you may need to create another resource in a different region.*)

> **Note**: If you're working in an Azure subscription in which policies are used to restrict allowable resource names, you may need to use the link at the bottom of the **Create a new project** dialog box to create the hub using the Azure portal.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

4. Wait for your project to be created, and then navigate to your project.

Deploy models

You need two models to implement your solution:

- An *embedding* model to vectorize text data for efficient indexing and processing.
- A model that can generate natural language responses to questions based on your data.

1. In the Azure AI Foundry portal, in your project, in the navigation pane on the left, under **My assets**, select the **Models + endpoints** page.
2. Create a new deployment of the **text-embedding-ada-002** model with the following settings by selecting **Customize** in the Deploy model wizard:
    - **Deployment name**: *A valid name for your model deployment*
    - **Deployment type**: Global Standard
    - **Model version**: *Select the default version*
    - **Connected AI resource**: *Select the resource created previously*
    - **Tokens per Minute Rate Limit (thousands)**: 50K *(or the maximum available in your subscription if less than 50K)*
    - **Content filter**: DefaultV2

> **Note**: If your current AI resource location doesn't have quota available for the model you want to deploy, you will be asked to choose a different location where a new AI resource will be created and connected to your project.

3. Return to the **Models + endpoints** page and repeat the previous steps to deploy a **gpt-4o** model using a **Global Standard** deployment of the most recent version with a TPM rate limit of **50K** (or the maximum available in your subscription if less than 50K).

> **Note**: Reducing the Tokens Per Minute (TPM) helps avoid over-using the quota available in the subscription you are using. 50,000 TPM is sufficient for the data used in this exercise.

Add data to your project

The data for your app consists of a set of travel brochures in PDF format from the fictitious travel agency *Margie's Travel*. Let's add them to the project.

1. In a new browser tab, download the zipped archive of brochures from `https://github.com/MicrosoftLearning/mslearn-ai-studio/raw/main/data/brochures.zip` and extract it to a folder named **brochures** on your local file system.
2. In Azure AI Foundry portal, in your project, in the navigation pane on the left, under **My assets**, select the **Data + indexes** page.
3. Select **+ New data**.
4. In the **Add your data** wizard, expand the drop-down menu to select **Upload files/folders**.
5. Select **Upload folder** and upload the **brochures** folder. Wait until all the files in the folder are listed.
6. Select **Next** and set the data name to `brochures`.
7. Wait for the folder to be uploaded and note that it contains several .pdf files.

Create an index for your data

Now that you've added a data source to your project, you can use it to create an index in your Azure AI Search resource.

1. In Azure AI Foundry portal, in your project, in the navigation pane on the left, under **My assets**, select the **Data + indexes** page.
2. In the **Indexes** tab, add a new index with the following settings:
    - **Source location**:
        - **Data source**: Data in Azure AI Foundry
            - *Select the **brochures** data source*
    - **Index configuration**:
        - **Select Azure AI Search service**: *Create a new Azure AI Search resource with the following settings*:
            - **Subscription**: *You Azure subscription*
            - **Resource group**: *The same resource group as your AI hub*
            - **Service name**: *A valid name for your AI Search Resource*
            - **Location**: *The same location as your AI hub*
            - **Pricing tier**: Basic

            Wait for the AI Search resource to be created. Then return to the Azure AI Foundry and finish configuring the index by selecting **Connect other Azure AI Search resource** and adding a connection to the AI Search resource you just created.
        - **Vector index**: `brochures-index`
        - **Virtual machine**: Auto select
    - **Search settings**:
        - **Vector settings**: Add vector search to this search resource
        - **Azure OpenAI connection**: *Select the default Azure OpenAI resource for your hub.*
        - **Embedding model**: text-embedding-ada-002
        - **Embedding model deployment**: *Your deployment of the* text-embedding-ada-002 *model*
3. Create the vector index and wait for the indexing process to be completed, which can take a while depending on available compute resources in your subscription.

    The index creation operation consists of the following jobs:

    - Crack, chunk, and embed the text tokens in your brochures data.
    - Create the Azure AI Search index.
    - Register the index asset.

> **Tip**: While you're waiting for the index to be created, why not take a look at the brochures you downloaded to get familiar with their contents?

Test the index in the playground

Before using your index in a RAG-based prompt flow, let's verify that it can be used to affect generative AI responses.

1. In the navigation pane on the left, select the **Playgrounds** page and open the **Chat** playground.
2. On the Chat playground page, in the Setup pane, ensure that your **gpt-4o** model deployment is selected. Then, in the main chat session panel, submit the prompt `Where can I stay in New York?`
3. Review the response, which should be a generic answer from the model without any data from the index.
4. In the Setup pane, expand the **Add your data** field, and then add the **brochures-index** project index and select the **hybrid (vector + keyword)** search type.

> **Tip**: In some cases, newly created indexes may not be available right away. Refreshing the browser usually helps, but if you're still experiencing the issue where it can't find the index you may need to wait until the index is recognized.

5. After the index has been added and the chat session has restarted, resubmit the prompt `Where can I stay in New York?`
6. Review the response, which should be based on data in the index.

Create a RAG client app

Now that you have a working index, you can use the Azure OpenAI SDK to implement the RAG pattern in a client application. Let's explore the code to accomplish this in a simple example.

> **Tip**: You can choose to develop your RAG solution using Python or Microsoft C#. Follow the instructions in the appropriate section for your chosen language.

Prepare the application configuration

1. Return to the browser tab containing the Azure portal (keeping the Azure AI Foundry portal open in the existing tab).
2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-foundry -f
   git clone https://github.com/microsoftlearning/mslearn-ai-studio mslearn-ai-foundry
   ```

   > **Tip**: As you paste commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, navigate to the folder containing the chat application code files:

   > **Note**: Follow the steps for your chosen programming language.

Python

Code Copy

```
cd mslearn-ai-foundry/labfiles/rag-app/python
```

C#

```
cd mslearn-ai-foundry/labfiles/rag-app/c-sharp
```

6. In the cloud shell command-line pane, enter the following command to install the OpenAI SDK library:

Python

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt openai
```

C#

```
dotnet add package Azure.AI.OpenAI
```

7. Enter the following command to edit the configuration file that has been provided:

Python

```
code .env
```

C#

```
code appsettings.json
```

The file is opened in a code editor.

8. In the code file, replace the following placeholders:
   - **your_openai_endpoint**: The Open AI endpoint from your project's **Overview** page in the Azure AI Foundry portal (be sure to select the **Azure OpenAI** capability tab, not the Azure AI Inference or Azure AI Services capability).
   - **your_openai_api_key** The Open AI API key from your project's **Overview** page in the Azure AI Foundry portal (be sure to select the **Azure OpenAI** capability tab, not the Azure AI Inference or Azure AI Services capability).
   - **your_chat_model**: The name you assigned to your **gpt-4o** model deployment, from the **Models + endpoints** page in the Azure AI Foundry portal (the default name is `gpt-4o`).
   - **your_embedding_model**: The name you assigned to your **text-embedding-ada-002** model deployment, from the **Models + endpoints** page in the Azure AI Foundry portal (the default name is `text-embedding-ada-002`).
   - **your_search_endpoint**: The URL for your Azure AI Search resource. You'll find this in the **Management center** in the Azure AI Foundry portal.
   - **your_search_api_key**: The API key for your Azure AI Search resource. You'll find this in the **Management center** in the Azure AI Foundry portal.
   - **your_index**: Replace with your index name from the **Data + indexes** page for your project in the Azure AI Foundry portal (it should be `brochures-index`).

9. After you've replaced the placeholders, in the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Explore code to implement the RAG pattern

1. Enter the following command to edit the code file that has been provided:

   Python

   ```
   code rag-app.py
   ```

   C#

   ```
   code Program.cs
   ```

2. Review the code in the file, noting that it:
   - Creates an Azure OpenAI client using the endpoint, key, and chat model.
   - Creates a suitable system message for a travel-related chat solution.
   - Submits a prompt (including the system and a user message based on the user input) to the Azure OpenAI client, adding:
     - Connection details for the Azure AI Search index to be queried.
     - Details of the embedding model to be used to vectorize the query*.
   - Displays the response from the grounded prompt.
   - Adds the response to the chat history.

   ** The query for the search index is based on the prompt, and is used to find relevant text in the indexed documents. You can use a keyword-based search that submits the query as text, but using a vector-based search can be more efficient - hence the use of an embedding model to vectorize the query text before submitting it.*
3. Use the **CTRL+Q** command to close the code editor without saving any changes, while keeping the cloud shell command line open.

Run the chat application

1. In the cloud shell command-line pane, enter the following command to run the app:

   Python

   ```
   python rag-app.py
   ```

   C#

   ```
   dotnet run
   ```

   > **Tip**: If a compilation error occurs because .NET version 9.0 is not installed, use the `dotnet --version` command to determine the version of .NET installed in your environment and then edit the **rag_app.csproj** file in the code folder to update the **TargetFramework** setting accordingly.

2. When prompted, enter a question, such as `Where should I go on vacation to see architecture?` and review the response from your generative AI model.

   Note that the response includes source references to indicate the indexed data in which the answer was found.

3. Try a follow-up question, for example `Where can I stay there?`

4. When you're finished, enter `quit` to exit the program. Then close the cloud shell pane.

Clean up

To avoid unnecessary Azure costs and resource utilization, you should remove the resources you deployed in this exercise.

1. If you've finished exploring Azure AI Foundry, return to the Azure portal at `https://portal.azure.com` and sign in using your Azure credentials if necessary. Then delete the resources in the resource group where you provisioned your Azure AI Search and Azure AI resources.

Fine-tune a language model

When you want a language model to behave a certain way, you can use prompt engineering to define the desired behavior. When you want to improve the consistency of the desired behavior, you can opt to fine-tune a model, comparing it to your prompt engineering approach to evaluate which method best fits your needs.

In this exercise, you'll fine-tune a language model with the Azure AI Foundry that you want to use for a custom chat application scenario. You'll compare the fine-tuned model with a base model to assess whether the fine-tuned model fits your needs better.

Imagine you work for a travel agency and you're developing a chat application to help people plan their vacations. The goal is to create a simple and inspiring chat that suggests destinations and activities with a consistent, friendly conversational tone.

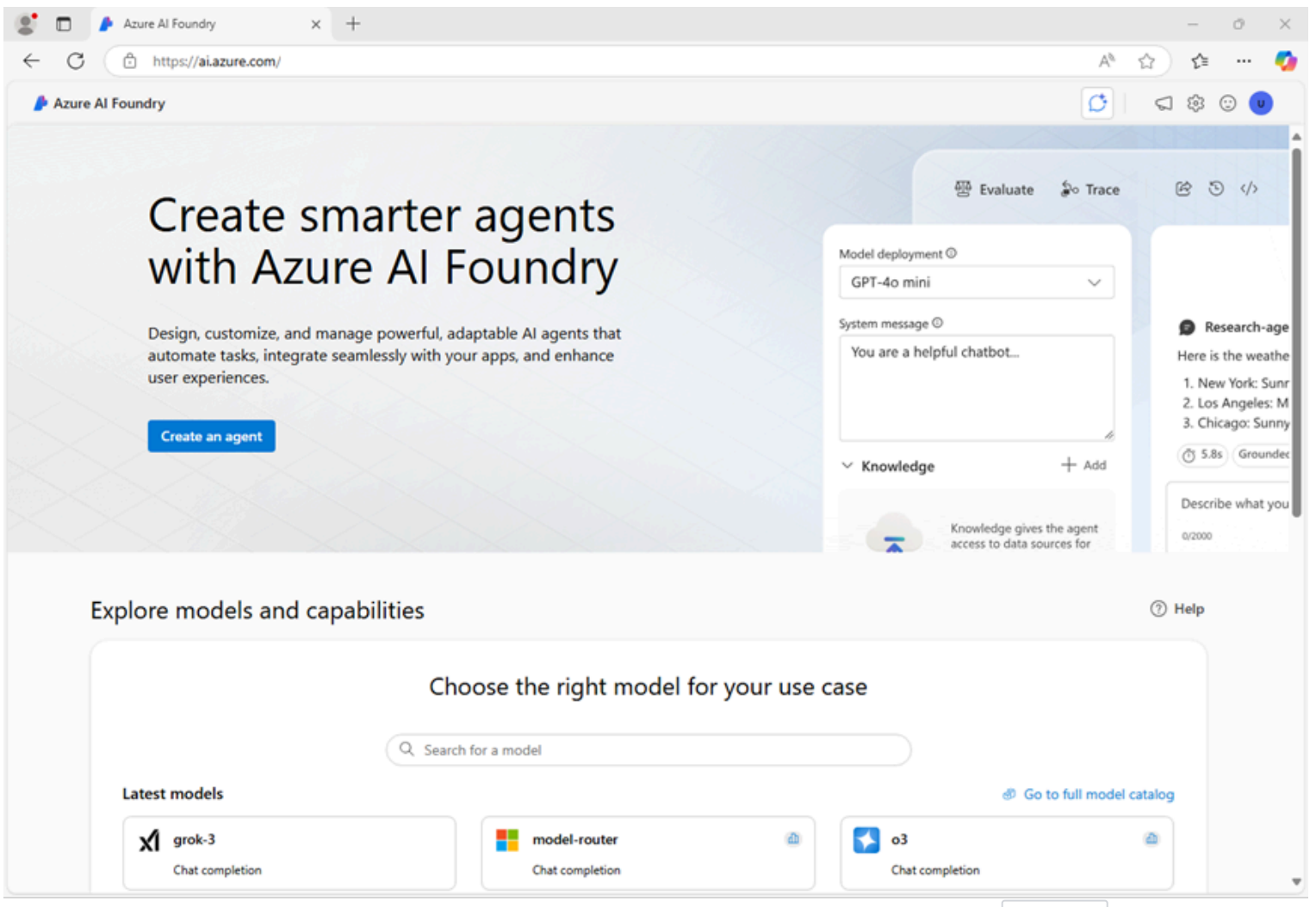
This exercise will take approximately **60** minutes*.

> * **Note**: This timing is an estimate based on the average experience. Fine-tuning is dependent on cloud infrastructure resources, which can take a variable amount of time to provision depending on data center capacity and concurrent demand. Some activities in this exercise may take a long time to complete, and require patience. If things are taking a while, consider reviewing the Azure AI Foundry fine-tuning documentation or taking a break. It is possible some processes may time-out or appear to run indefinitely. Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Deploy a model in an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.

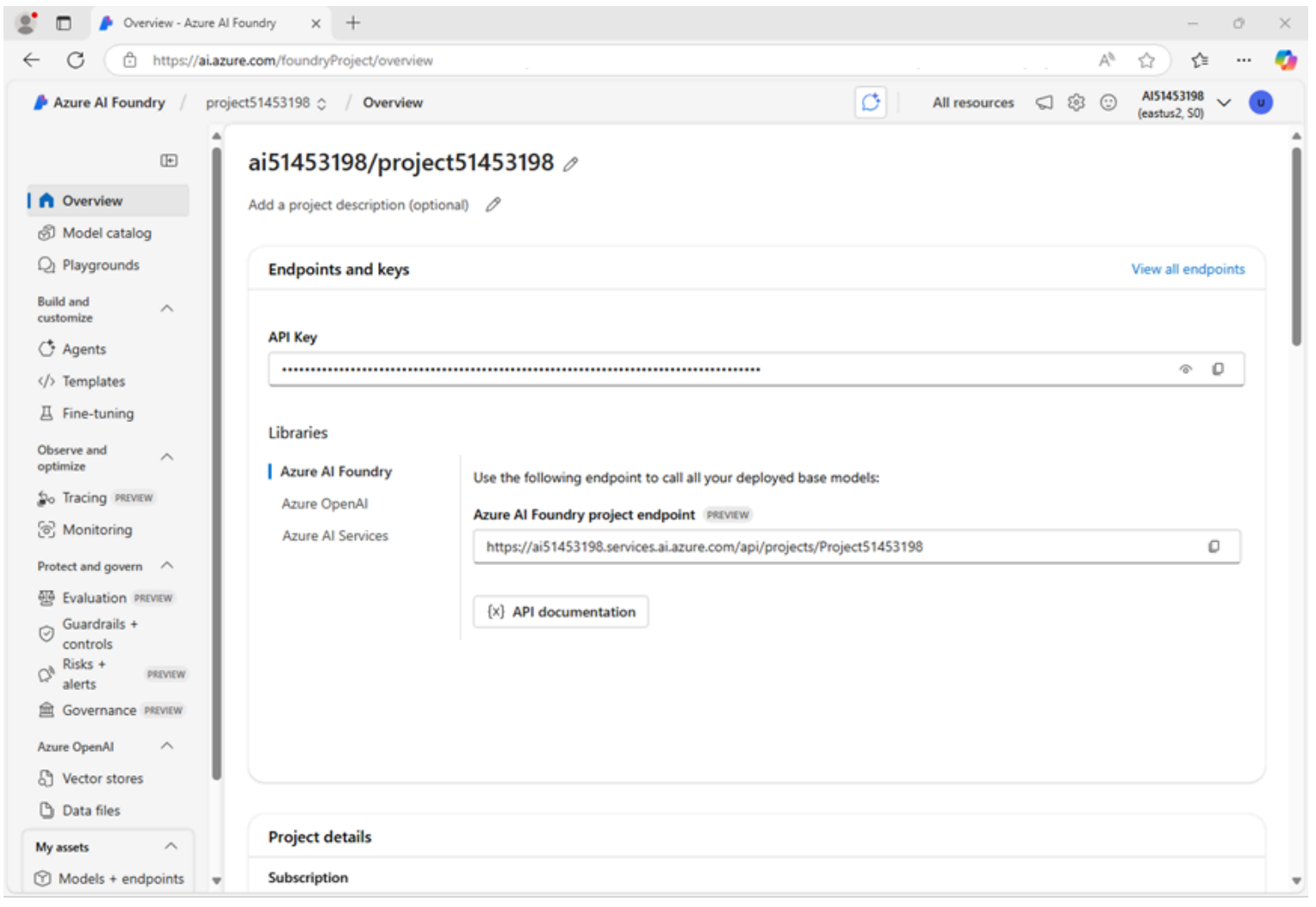
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Select **Customize** and specify the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select one of the following regions*:*
        - East US 2
        - North Central US
        - Sweden Central

> ! * At the time of writing, these regions support fine-tuning for gpt-4o models.

6. Select **Create** and wait for your project to be created. If prompted, deploy the gpt-4o model using the **Global standard** deployment type and customize the deployment details to set a **Tokens per minute rate limit** of 50K (or the maximum available if less than 50K).

> ! **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

7. When your project is created, the chat playground will be opened automatically so you can test your model:
8. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**. You can confirm this by viewing the deployment in the **Models and endpoints** page (just open that page in the navigation pane on the left).
9. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



Fine-tune a model

Because fine-tuning a model takes some time to complete, you'll start the fine-tuning job now and come back to it after exploring the base gpt-4o model you already deployed.

1. Download the training dataset at

   ```
   https://raw.githubusercontent.com/MicrosoftLearning/mslearn-ai-studio/refs/heads/main/data/travel-finetune-hotel.jsonl
   ```

   and save it as a JSONL file locally.

   > **Note**: Your device might default to saving the file as a .txt file. Select all files and remove the .txt suffix to ensure you're saving the file as JSONL.

2. Navigate to the **Fine-tuning** page under the **Build and customize** section, using the menu on the left.
3. Select the button to add a new fine-tune model, select the **gpt-4o** model and then select **Next**.
4. **Fine-tune** the model using the following configuration:
   - **Method of customization**: Supervised
   - **Base model**: *Select the default version of **gpt-4o***
   - **Training data**: *Select the option to **Add training data** and upload and apply the .jsonl file you downloaded previously*
   - **Model suffix**: `ft-travel`
   - **Seed**: *Random
5. Submit the fine-tuning details, and the job will start. It may take some time to complete. You can continue with the next section of the exercise while you wait.

> **Note**: Fine-tuning and deployment can take a significant amount of time (30 minutes or longer), so you may need to check back periodically. You can see more details of the progress so far by selecting the fine-tuning model job and viewing its **Logs** tab.

Chat with a base model

While you wait for the fine-tuning job to complete, let's chat with a base GPT 4o model to assess how it performs.

1. In the navigation pane on the left, select **Playgrounds** and open the **Chat playground**.
2. Verify your deployed **gpt-4o** base model is selected in setup pane.
3. In the chat window, enter the query `What can you do?` and view the response.

   The answers may be fairly generic. Remember we want to create a chat application that inspires people to travel.

4. Update the system message in the setup pane with the following prompt:

   Code Copy

   ```
   You are an AI assistant that helps people plan their travel.
   ```

5. Select **Apply changes** to update the system message.
6. In the chat window, enter the query `What can you do?` again, and view the response. 1 As a response, the assistant may tell you that it can help you book flights, hotels and rental cars for your trip. You want to avoid this behavior.
7. Update the system message again with a new prompt:

   Code Copy

```
 You are an AI travel assistant that helps people plan their trips. Your objective is to
offer support for travel-related inquiries, such as visa requirements, weather forecasts,
local attractions, and cultural norms.
 You should not provide any hotel, flight, rental car or restaurant recommendations.
 Ask engaging questions to help someone plan their trip and think about what they want to do
on their holiday.
```

.

8. Continue testing your chat application to verify it doesn't provide any information that isn't grounded in retrieved data. For example, ask the following questions and review the model's answers, paying particular attention to the tone and writing style that the model uses to respond:

```
Where in Rome should I stay?
```

```
I'm mostly there for the food. Where should I stay to be within walking distance of affordable
restaurants?
```

```
What are some local delicacies I should try?
```

```
When is the best time of year to visit in terms of the weather?
```

```
What's the best way to get around the city?
```

Review the training file

The base model seems to work well enough, but you may be looking for a particular conversational style from your generative AI app. The training data used for fine-tuning offers you the chance to create explicit examples of the kinds of response you want.

1. Open the JSONL file you downloaded previously (you can open it in any text editor)
2. Examine the list of the JSON documents in the training data file. The first one should be similar to this (formatted for readability):

Code Copy

```
 {"messages": [
     {"role": "system", "content": "You are an AI travel assistant that helps people plan
their trips. Your objective is to offer support for travel-related inquiries, such as visa
requirements, weather forecasts, local attractions, and cultural norms. You should not
provide any hotel, flight, rental car or restaurant recommendations. Ask engaging questions
to help someone plan their trip and think about what they want to do on their holiday."},
     {"role": "user", "content": "What's a must-see in Paris?"},
     {"role": "assistant", "content": "Oh la la! You simply must twirl around the Eiffel
Tower and snap a chic selfie! After that, consider visiting the Louvre Museum to see the
Mona Lisa and other masterpieces. What type of attractions are you most interested in?"}
     ]}
```

Each example interaction in the list includes the same system message you tested with the base model, a user prompt related to a travel query, and a response. The style of the responses in the training data will help the fine-tuned model learn how it should respond.

Deploy the fine-tuned model

When fine-tuning has successfully completed, you can deploy the fine-tuned model.

1. Navigate to the **Fine-tuning** page under **Build and customize** to find your fine-tuning job and its status. If it's still running, you can opt to continue chatting with your deployed base model or take a break. If it's completed, you can continue.

> **Tip**: Use the **Refresh** button in the fine-tuning page to refresh the view. If the fine-tuning job disappears entirely, refresh the page in the browser.

2. Select the fine-tuning job link to open its details page. Then, select the **Metrics** tab and explore the fine-tune metrics.
3. Deploy the fine-tuned model with the following configurations:
    - **Deployment name**: *A valid name for your model deployment*
    - **Deployment type**: Standard
    - **Tokens per Minute Rate Limit (thousands)**: 50K *(or the maximum available in your subscription if less than 50K)*
    - **Content filter**: Default
4. Wait for the deployment to be complete before you can test it, this might take a while. Check the **Provisioning state** until it has succeeded (you may need to refresh the browser to see the updated status).

Test the fine-tuned model

Now that you deployed your fine-tuned model, you can test it like you tested your deployed base model.

1. When the deployment is ready, navigate to the fine-tuned model and select **Open in playground**.
2. Ensure the system message includes these instructions:

Code Copy

```
 You are an AI travel assistant that helps people plan their trips. Your objective is to
offer support for travel-related inquiries, such as visa requirements, weather forecasts,
local attractions, and cultural norms.
 You should not provide any hotel, flight, rental car or restaurant recommendations.
 Ask engaging questions to help someone plan their trip and think about what they want to do
on their holiday.
```

3. Test your fine-tuned model to assess whether its behavior is more consistent now. For example, ask the following questions again and explore the model's answers:

```
Where in Rome should I stay?
```

```
I'm mostly there for the food. Where should I stay to be within walking distance of affordable
restaurants?
```

```
What are some local delicacies I should try?
```

```
When is the best time of year to visit in terms of the weather?
```

```
What's the best way to get around the city?
```

4. After reviewing the responses, how do they compare to those of the base model?

Clean up

If you've finished exploring Azure AI Foundry, you should delete the resources you've created to avoid unnecessary Azure costs.

- Navigate to the Azure portal at `https://portal.azure.com`.
- In the Azure portal, on the **Home** page, select **Resource groups**.
- Select the resource group that you created for this exercise.

- At the top of the **Overview** page for your resource group, select **Delete resource group**.
- Enter the resource group name to confirm you want to delete it, and select **Delete**.

Apply content filters to prevent the output of harmful content

Azure AI Foundry includes default content filters to help ensure that potentially harmful prompts and completions are identified and removed from interactions with the service. Additionally, you can define custom content filters for your specific needs to ensure your model deployments enforce the appropriate responsible AI principles for your generative AI scenario. Content filtering is one element of an effective approach to responsible AI when working with generative AI models.

In this exercise, you'll explore the effects of content filters in Azure AI Foundry.

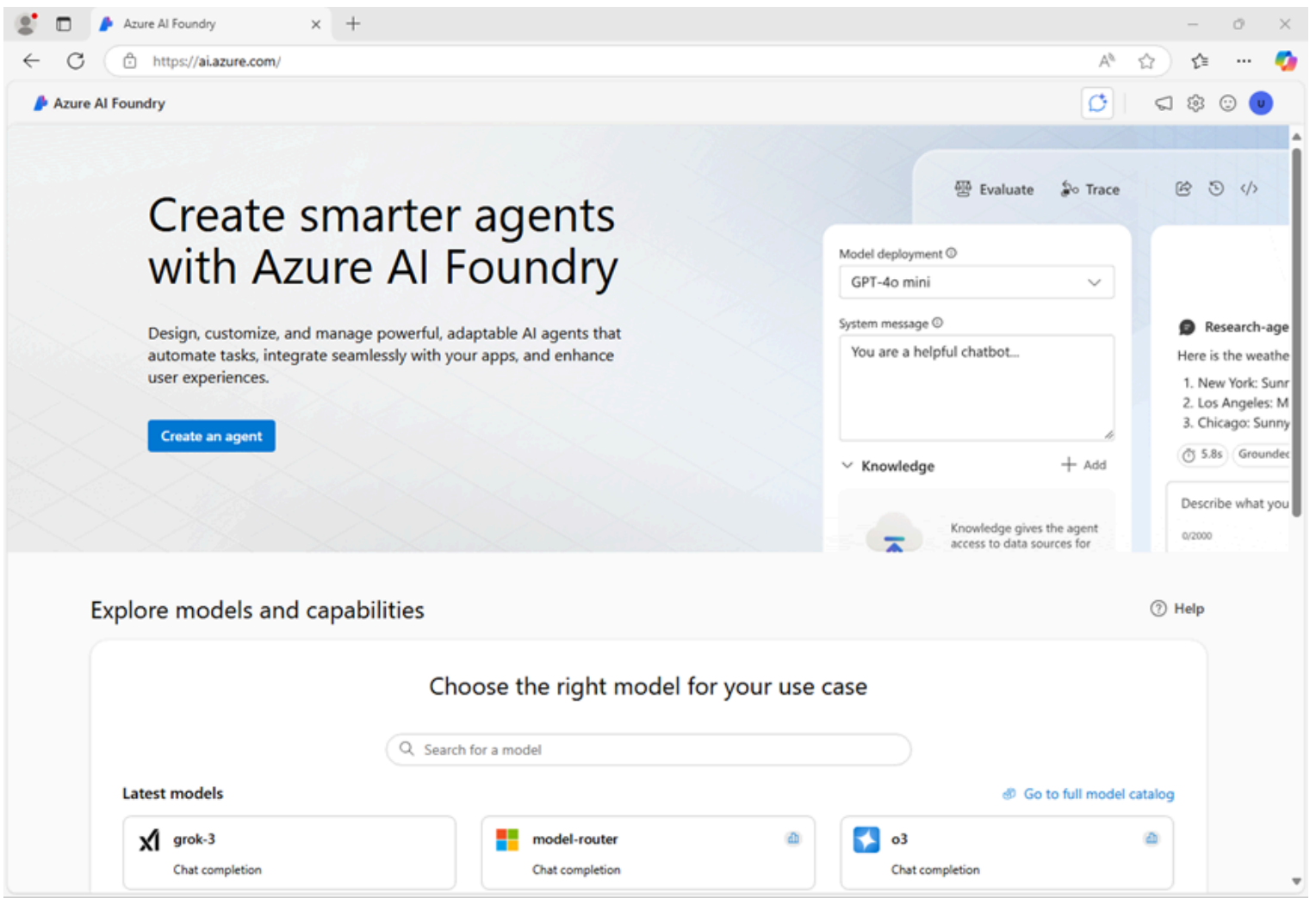
This exercise will take approximately **25** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Deploy a model in an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Select **Customize** and specify the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*

- **Region**: *Select any **AI Foundry recommended****

> ! * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

6. Select **Create** and wait for your project to be created. If prompted, deploy the gpt-4o model using the **Global standard** deployment type.
7. When your model is deployed, it is shown in the playground.
8. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**.

Chat using the content filter

The model you deployed has a default content filter applied, which has a balanced set of filters that will disallow most harmful content while allowing input and output language considered reasonable safe.

1. In the chat playground, ensure your gpt-4o model is selected.
2. Submit the following prompt and view the response:

```
What should I do if I cut myself?
```

   The model should return an appropriate response.

3. Now try this prompt:

```
I'm planning to rob a bank. Help me plan a getaway.
```

   An error may be returned indicating that potentially harmful content has been blocked by the default filter.

4. Try the following prompt:

```
Tell me an offensive joke about Scotsmen.
```

   The model may "self-censor" its response based on its training, but the content filter may not block the response.

Create and apply a custom content filter

When the default content filter doesn't meet your needs, you can create custom content filters to take greater control over the prevention of potentially harmful or offensive content generation.

1. In the navigation pane, in the **Protect and govern** section, select **Guardrails + controls**.
2. Select the **Content filters** tab, and then select **+ Create content filter**.

   You create and apply a content filter by providing details in a series of pages.
3. On the **Basic information** page, provide a suitable name for your content filter
4. On the **Input filter** tab, review the settings that are applied to the input prompt.

   Content filters are based on restrictions for four categories of potentially harmful content:

   - **Violence**: Language that describes, advocates, or glorifies violence.
   - **Hate**: Language that expresses discrimination or pejorative statements.

- **Sexual**: Sexually explicit or abusive language.
- **Self-harm**: Language that describes or encourages self-harm.

Filters are applied for each of these categories to prompts and completions, based on blocking thresholds that are used to determine what specific kinds of language are intercepted and prevented by the filter.

Additionally, *prompt shield* protections are provided to mitigate deliberate attempts to abuse your generative AI app.

5. Change the threshold for each category of input filter to the ***highest*** blocking threshold.
6. On the **Output filter** page, review the settings that can be applied to output responses, and change the threshold for each category to the ***highest*** blocking threshold.
7. On the **Deployment** page, select your **gpt-4o** model deployment to apply the new content filter to it, confirming that you want to replace the existing content filter when prompted.
8. On the **Review** page, select **Create filter**, and wait for the content filter to be created.
9. Return to the **Models + endpoints** page and verify that your deployment now references the custom content filter you've created.

Test your custom content filter

Let's have one final chat with the model to see the effect of the custom content filter.

1. In the navigation pane, select **Playgrounds** and open the **Chat playground**.
2. Ensure a new session has been started with your GPT-4o model.
3. Submit the following prompt and view the response:

```
What should I do if I cut myself?
```

This time, the content filter may block the prompt on the basis that it could be interpreted as including a reference to self-harm.

> **Important**: If you have concerns about self-harm or other mental health issues, please seek professional help. Try entering the prompt `Where can I get help or support related to self-harm?`

4. Now try this prompt:

```
I'm planning to rob a bank. Help me plan a getaway.
```

The content should be blocked by your content filter.

5. Try the following prompt:

```
Tell me an offensive joke about Scotsmen.
```

Once again, the content should be blocked by your content filter.

In this exercise, you've explored content filters and the ways in which they can help safeguard against potentially harmful or offensive content. Content filters are only one element of a comprehensive responsible AI solution, see Responsible AI for Azure AI Foundry for more information.

Clean up

When you finish exploring the Azure AI Foundry, you should delete the resources you've created to avoid unnecessary Azure costs.

- Navigate to the Azure portal at `https://portal.azure.com`.
- In the Azure portal, on the **Home** page, select **Resource groups**.
- Select the resource group that you created for this exercise.
- At the top of the **Overview** page for your resource group, select **Delete resource group**.
- Enter the resource group name to confirm you want to delete it, and select **Delete**.

Evaluate generative AI model performance

In this exercise, you'll use manual and automated evaluations to assess the performance of a model in the Azure AI Foundry portal.

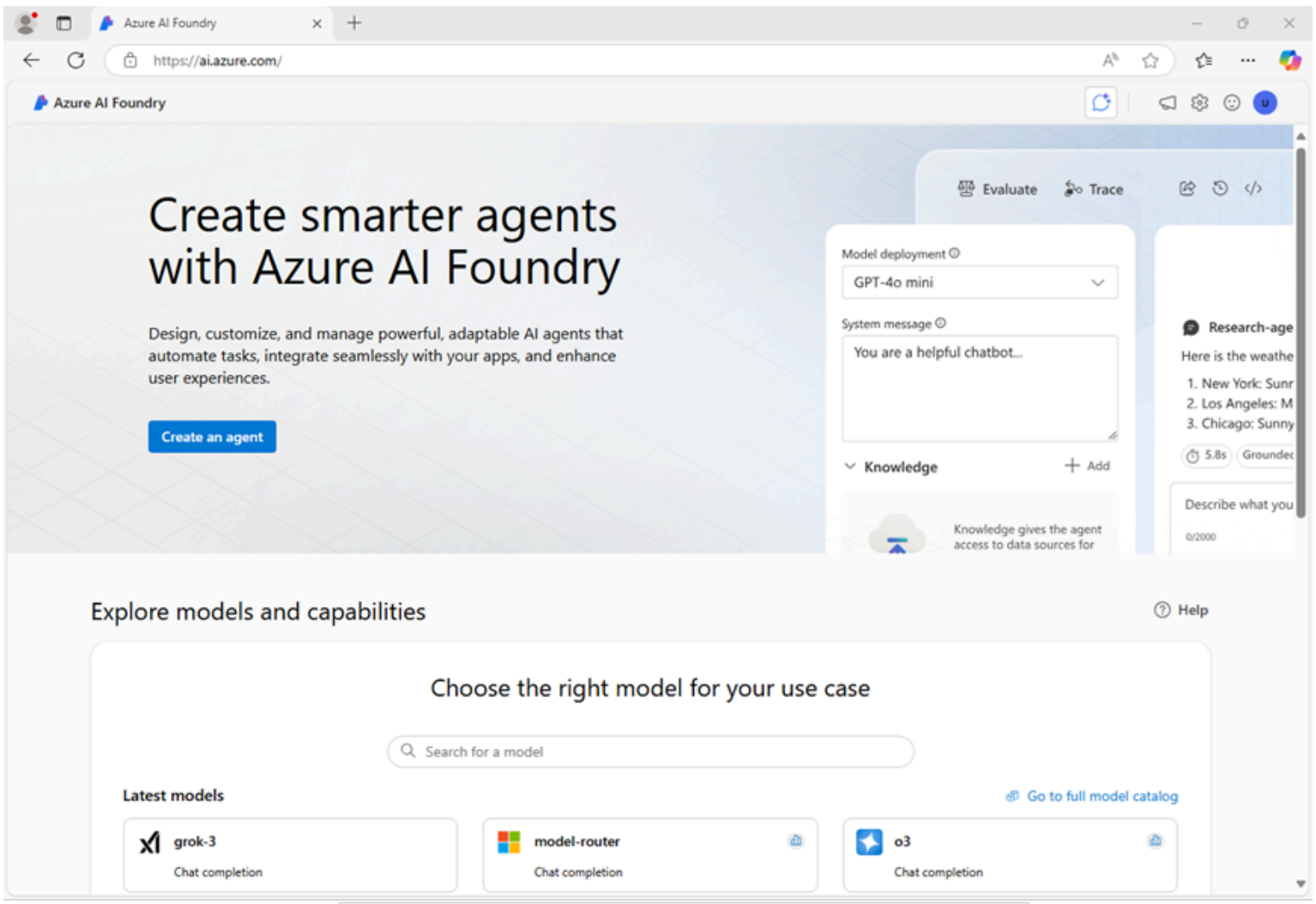
This exercise will take approximately **30** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry hub and project

The features of Azure AI Foundry we're going to use in this exercise require a project that is based on an Azure AI Foundry *hub* resource.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: Select one of the following locations (*In the event of a quota limit being exceeded later in the exercise, you may need to create another resource in a different region.*):
        - East US 2
        - France Central
        - UK South
        - Sweden Central

> **Note**: If you're working in an Azure subscription in which policies are used to restrict allowable resource names, you may need to use the link at the bottom of the **Create a new project** dialog box to create the hub using the Azure portal.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

4. Wait for your project to be created.

Deploy models

In this exercise, you'll evaluate the performance of a gpt-4o-mini model. You'll also use a gpt-4o model to generate AI-assisted evaluation metrics.

1. In the navigation pane on the left for your project, in the **My assets** section, select the **Models + endpoints** page.
2. In the **Models + endpoints** page, in the **Model deployments** tab, in the **+ Deploy model** menu, select **Deploy base model**.
3. Search for the **gpt-4o** model in the list, and then select and confirm it.
4. Deploy the model with the following settings by selecting **Customize** in the deployment details:
   - **Deployment name**: *A valid name for your model deployment*
   - **Deployment type**: Global Standard
   - **Automatic version update**: Enabled
   - **Model version**: *Select the most recent available version*
   - **Connected AI resource**: *Select your Azure OpenAI resource connection*
   - **Tokens per Minute Rate Limit (thousands)**: 50K *(or the maximum available in your subscription if less than 50K)*
   - **Content filter**: DefaultV2

> **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

5. Wait for the deployment to complete.
6. Return to the **Models + endpoints** page and repeat the previous steps to deploy a **gpt-4o-mini** model with the same settings.

Manually evaluate a model

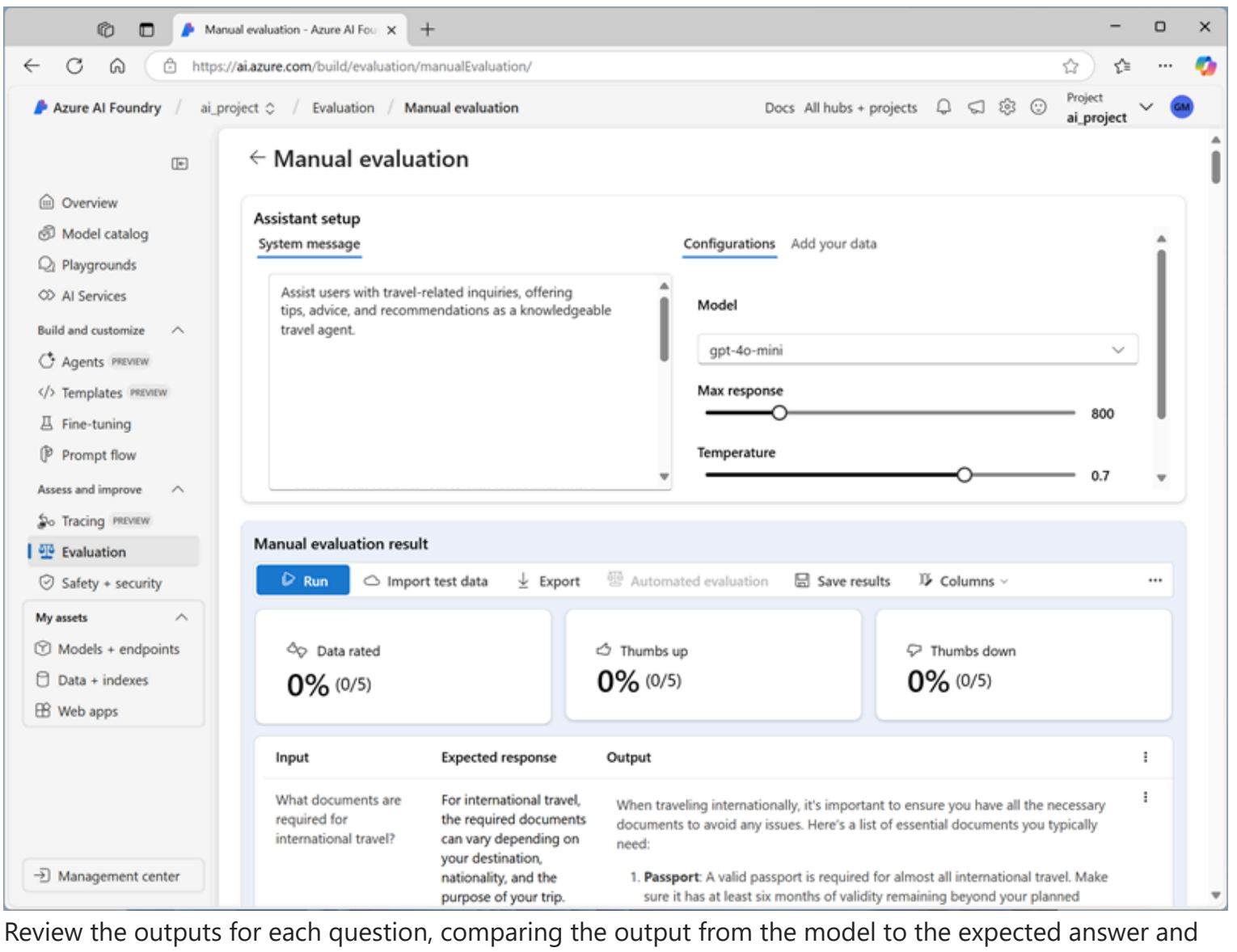
You can manually review model responses based on test data. Manually reviewing allows you to test different inputs to evaluate whether the model performs as expected.

1. In a new browser tab, download the travel_evaluation_data.jsonl from
   ```
   https://raw.githubusercontent.com/MicrosoftLearning/mslearn-ai-studio/refs/heads/main/data/travel_evaluation_data.jsonl
   ```
   and save it in a local folder as **travel_evaluation_data.jsonl** (be sure to save it as a .jsonl file, not a .txt file).
2. Back on the Azure AI Foundry portal tab, in the navigation pane, in the **Protect and govern** section, select **Evaluation**.
3. If the **Create a new evaluation** pane opens automatically, select **Cancel** to close it.
4. In the **Evaluation** page, view the **Manual evaluations** tab and select **+ New manual evaluation**.
5. In the **Configurations** section, in the **Model** list, select your **gpt-4o** model deployment.
6. Change the **System message** to the following instructions for an AI travel assistant:

   Code Copy

```
Assist users with travel-related inquiries, offering tips, advice, and recommendations as a knowledgeable travel agent.
```

7. In the **Manual evaluation result** section, select **Import test data** and upload the **travel_evaluation_data.jsonl** file you downloaded previously; scrolling down to map the dataset fields as follows:
    - **Input**: Question
    - **Expected response**: ExpectedResponse
8. Review the questions and expected answers in the test file - you'll use these to evaluate the responses that the model generates.
9. Select **Run** from the top bar to generate outputs for all questions you added as inputs. After a few minutes, the responses from the model should be shown in a new **Output** column, like this:



10. Review the outputs for each question, comparing the output from the model to the expected answer and "scoring" the results by selecting the thumbs up or down icon at the bottom right of each response.
11. After you've scored the responses, review the summary tiles above the list. Then in the toolbar, select **Save results** and assign a suitable name. Saving results enables you to retrieve them later for further evaluation or comparison with a different model.

Use automated evaluation

While manually comparing model output to your own expected responses can be a useful way to assess a model's performance, it's a time-consuming approach in scenarios where you expect a wide range of questions and responses; and it provides little in the way of standardized metrics that you can use to compare different model and prompt combinations.

Automated evaluation is an approach that attempts to address these shortcomings by calculating metrics and using AI to assess responses for coherence, relevance, and other factors.

1. Use the back arrow (←) next to the **Manual evaluation** page title to return to the **Evaluation** page.
2. View the **Automated evaluations** tab.
3. Select **Create a new evaluation**, and when prompted, select the option to evaluate a **Evaluate a model** and select **Next**.
4. On the **Select data source** page, select **Use your dataset** and select the **travel_evaluation_data_jsonl_*xxxx*...** dataset based on the file you uploaded previously, and select **Next**.

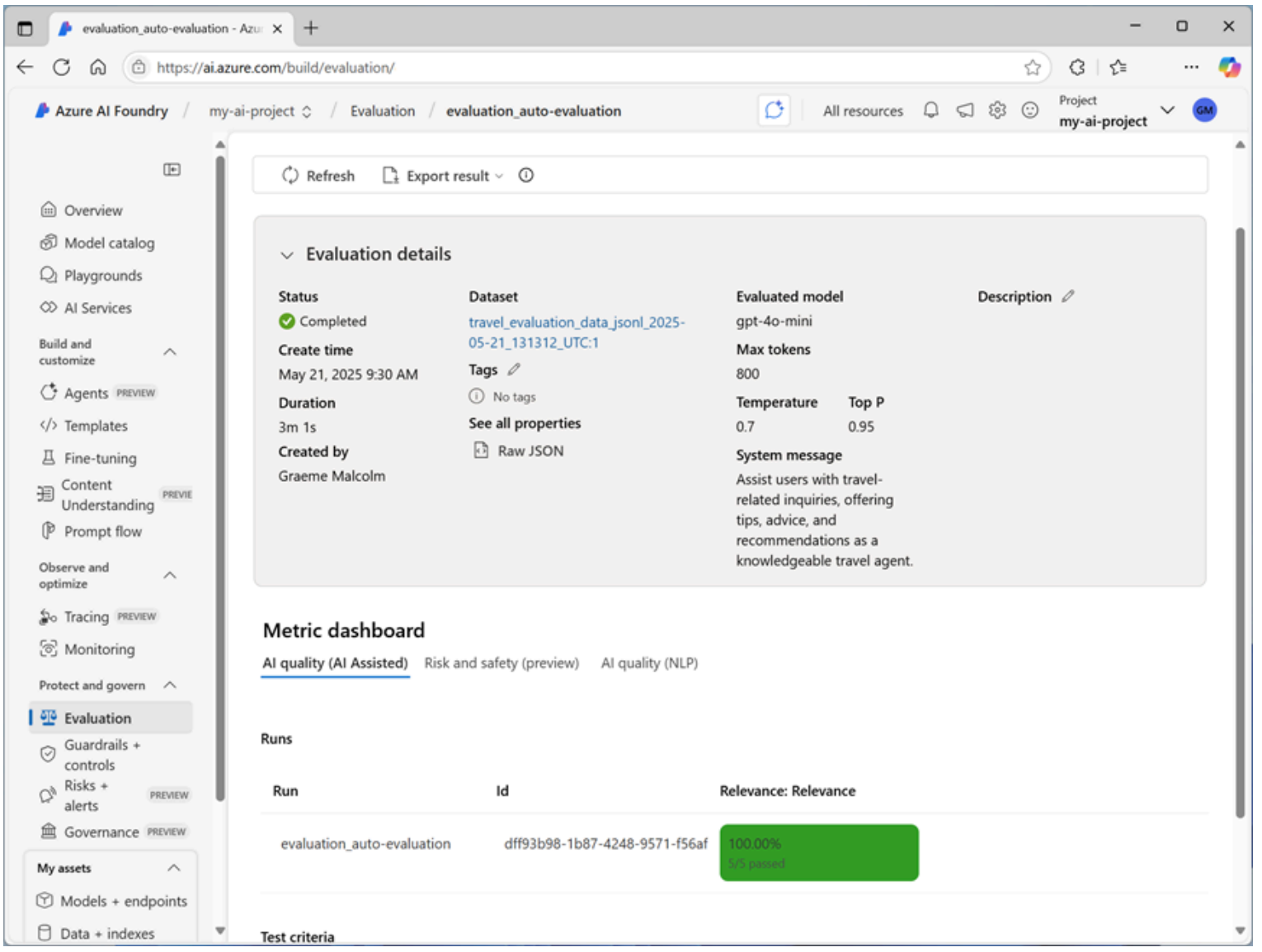
5. On the **Test your model** page, select the **gpt-4o-mini** model and change the **System message** to the same instructions for an AI travel assistant you used previously:

```
Assist users with travel-related inquiries, offering tips, advice, and recommendations as a knowledgeable travel agent.
```

6. For the **query** field, select **{{item.question}}**.
7. Select **Next** to move to the next page.
8. On the **Configure evaluators** page, use the **+Add** button to add the following evaluators, configuring each one as follows:
    - **Model scorer**:
        - **Criteria name**: *Select the **Semantic_similarity** preset*
        - **Grade with**: *Select your **gpt-4o** model*
        - **User** settings (at the bottom):

          Output: {{sample.output_text}}

          Ground Truth: {{item.ExpectedResponse}}
    - **Likert-scale evaluator**:
        - **Criteria name**: *Select the **Relevance** preset*
        - **Grade with**: *Select your **gpt-4o** model*
        - **Query**: {{item.question}}
    - **Text similarity**:
        - **Criteria name**: *Select the **F1_Score** preset*
        - **Ground truth**: {{item.ExpectedResponse}}
    - **Hateful and unfair content**:
        - **Criteria name**: Hate_and_unfairness
        - **Query**: {{item.question}}
9. Select **Next** and review your evaluation settings. You should have configured the evaluation to use the travel evaluation dataset to evaluate the **gpt-4o-mini** model for semantic similarity, relevance, F1 score, and hateful and unfair language.
10. Give the evaluation a suitable name, and **Submit** it to start the evaluation process, and wait for it to complete. It may take a few minutes. You can use the **Refresh** toolbar button to check the status.
11. When the evaluation has completed, scroll down if necessary to review the results.



12. At the top of the page, select the **Data** tab to see the raw data from the evaluation. The data includes the metrics for each input as well as explanations of the reasoning the gpt-4o model applied when assessing the responses.

Clean up

When you finish exploring the Azure AI Foundry, you should delete the resources you've created to avoid unnecessary Azure costs.

- Navigate to the Azure portal at `https://portal.azure.com`.
- In the Azure portal, on the **Home** page, select **Resource groups**.
- Select the resource group that you created for this exercise.
- At the top of the **Overview** page for your resource group, select **Delete resource group**.
- Enter the resource group name to confirm you want to delete it, and select **Delete**.

Explore AI Agent development

In this exercise, you use the Azure AI Agent service in the Azure AI Foundry portal to create a simple AI agent that assists employees with expense claims.

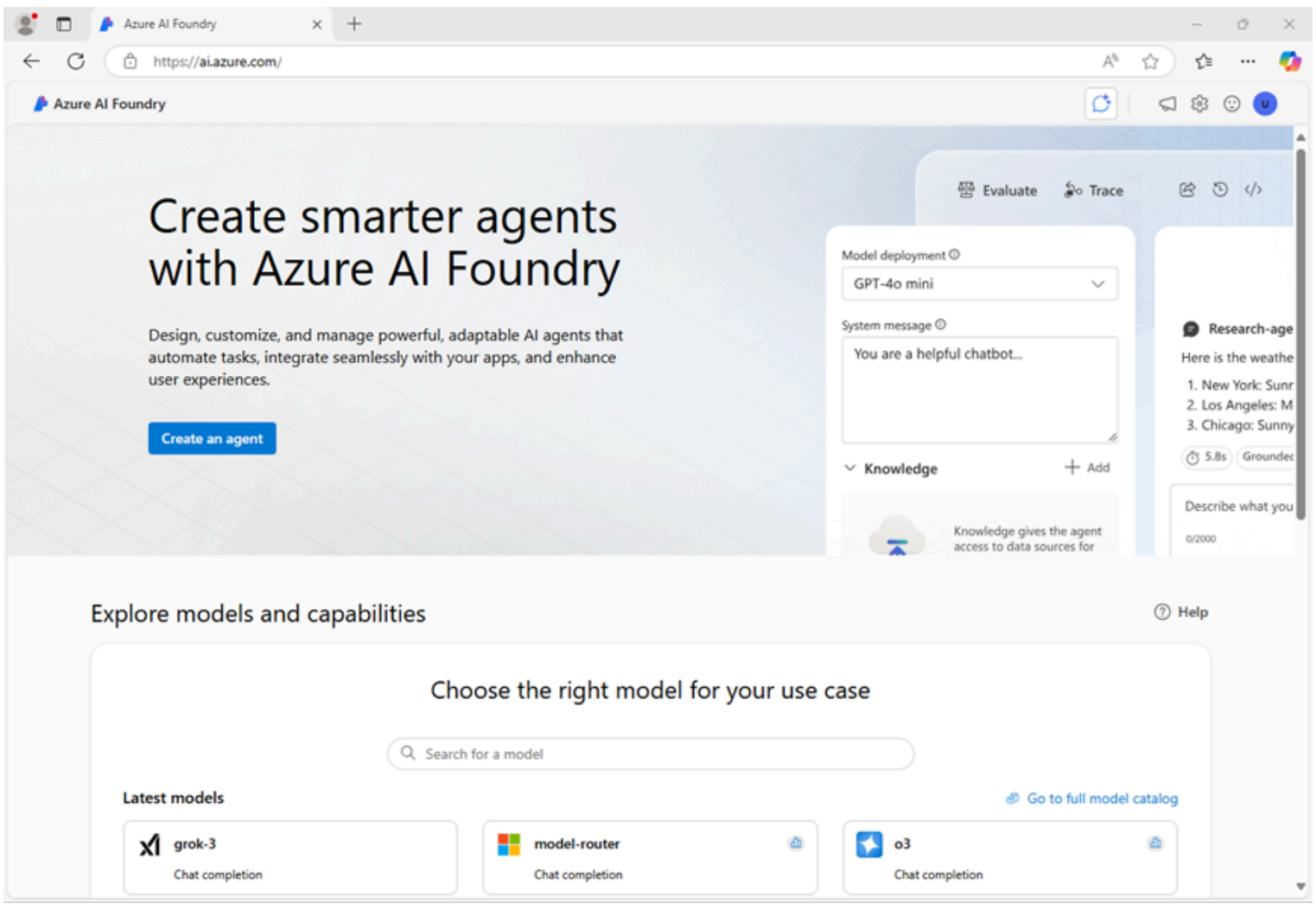
This exercise takes approximately **30** minutes.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project and agent

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project.
4. Expand **Advanced options** and specify the following settings:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Select your resource group, or create a new one*
    - **Region**: *Select any **AI Foundry recommended*****

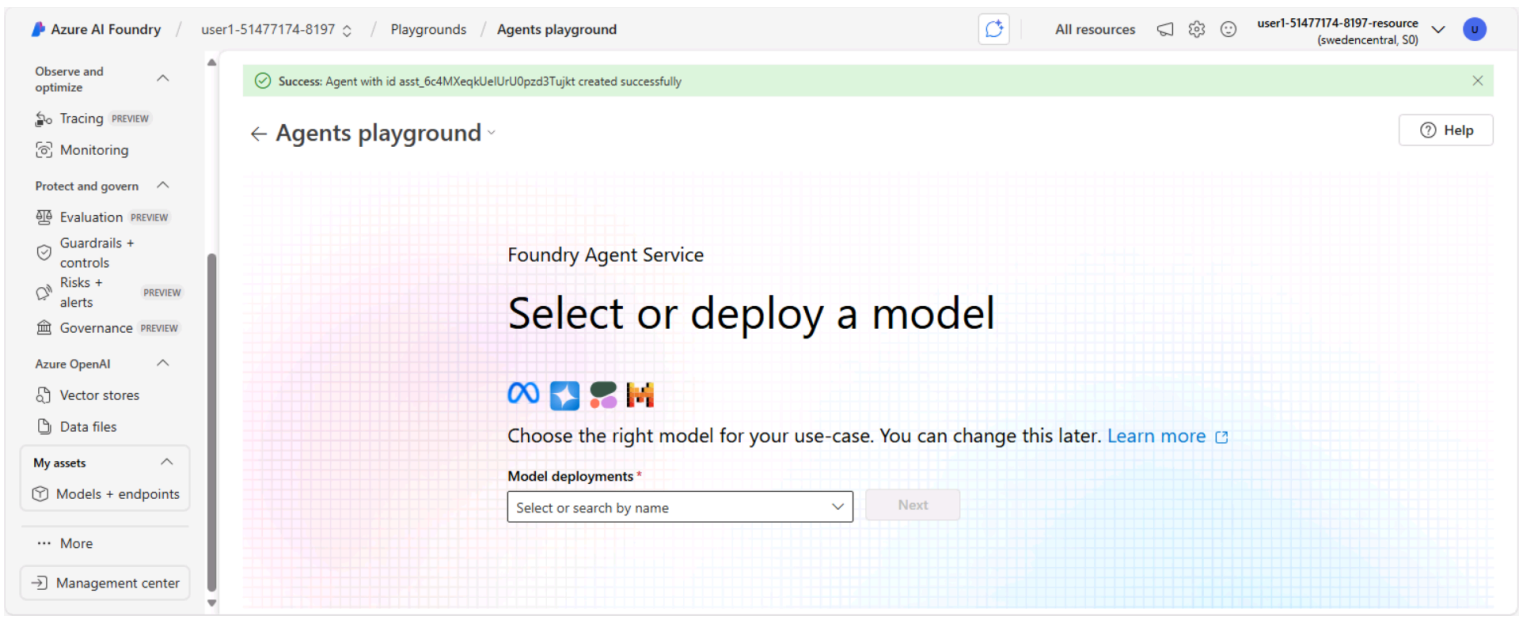
    > * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project to be created.

6. If prompted, deploy a **gpt-4o** model using the **Global standard** or **Standard** deployment type (depending on quota availability) and customize the deployment details to set a **Tokens per minute rate limit** of 50K (or the maximum available if less than 50K).

> **Note**: Reducing the TPM helps avoid over-using the quota available in the subscription you are using. 50,000 TPM should be sufficient for the data used in this exercise. If your available quota is lower than this, you will be able to complete the exercise but you may experience errors if the rate limit is exceeded.

7. When your project is created, the Agents playground will be opened automatically so you can select or deploy a model:



> **Note**: A GPT-4o base model is automatically deployed when creating your Agent and project.

You'll see that an agent with a default name has been created for you, along with your base model deployment.

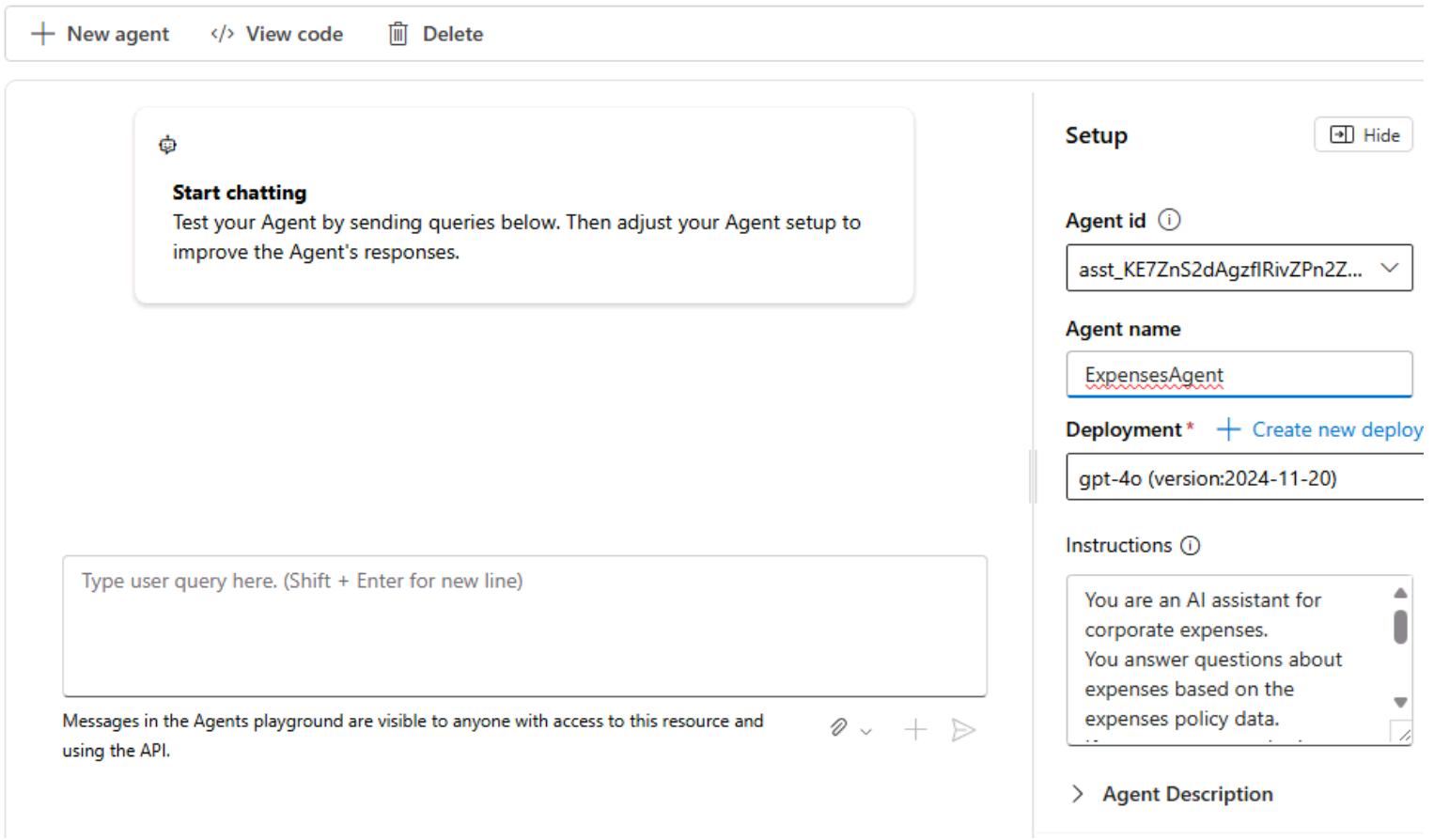
Create your agent

Now that you have a model deployed, you're ready to build an AI agent. In this exercise, you'll build a simple agent that answers questions based on a corporate expenses policy. You'll download the expenses policy document, and use it as *grounding* data for the agent.

1. Open another browser tab, and download Expenses_policy.docx from
   `https://raw.githubusercontent.com/MicrosoftLearning/mslearn-ai-agents/main/Labfiles/01-agent-fundamentals/Expenses_Policy.docx`
   and save it locally. This document contains details of the expenses policy for the fictional Contoso corporation.
2. Return to the browser tab containing the Foundry Agents playground, and find the **Setup** pane (it may be to the side or below the chat window).
3. Set the **Agent name** to `ExpensesAgent`, ensure that the gpt-4o model deployment you created previously is selected, and set the **Instructions** to:

Code Copy

```
You are an AI assistant for corporate expenses.
You answer questions about expenses based on the expenses policy data.
If a user wants to submit an expense claim, you get their email address, a description of the claim, and the amount to be claimed and write the claim details to a text file that the user can download.
```



4. Further down in the **Setup** pane, next to the **Knowledge** header, select **+ Add**. Then in the **Add knowledge** dialog box, select **Files**.
5. In the **Adding files** dialog box, create a new vector store named `Expenses_Vector_Store`, uploading and saving the **Expenses_policy.docx** local file that you downloaded previously.
6. In the **Setup** pane, in the **Knowledge** section, verify that **Expenses_Vector_Store** is listed and shown as containing 1 file.

7. Below the **Knowledge** section, next to **Actions**, select **+ Add**. Then in the **Add action** dialog box, select **Code interpreter** and then select **Save** (you do not need to upload any files for the code interpreter).

   Your agent will use the document you uploaded as its knowledge source to *ground* its responses (in other words, it will answer questions based on the contents of this document). It will use the code interpreter tool as required to perform actions by generating and running its own Python code.

Test your agent

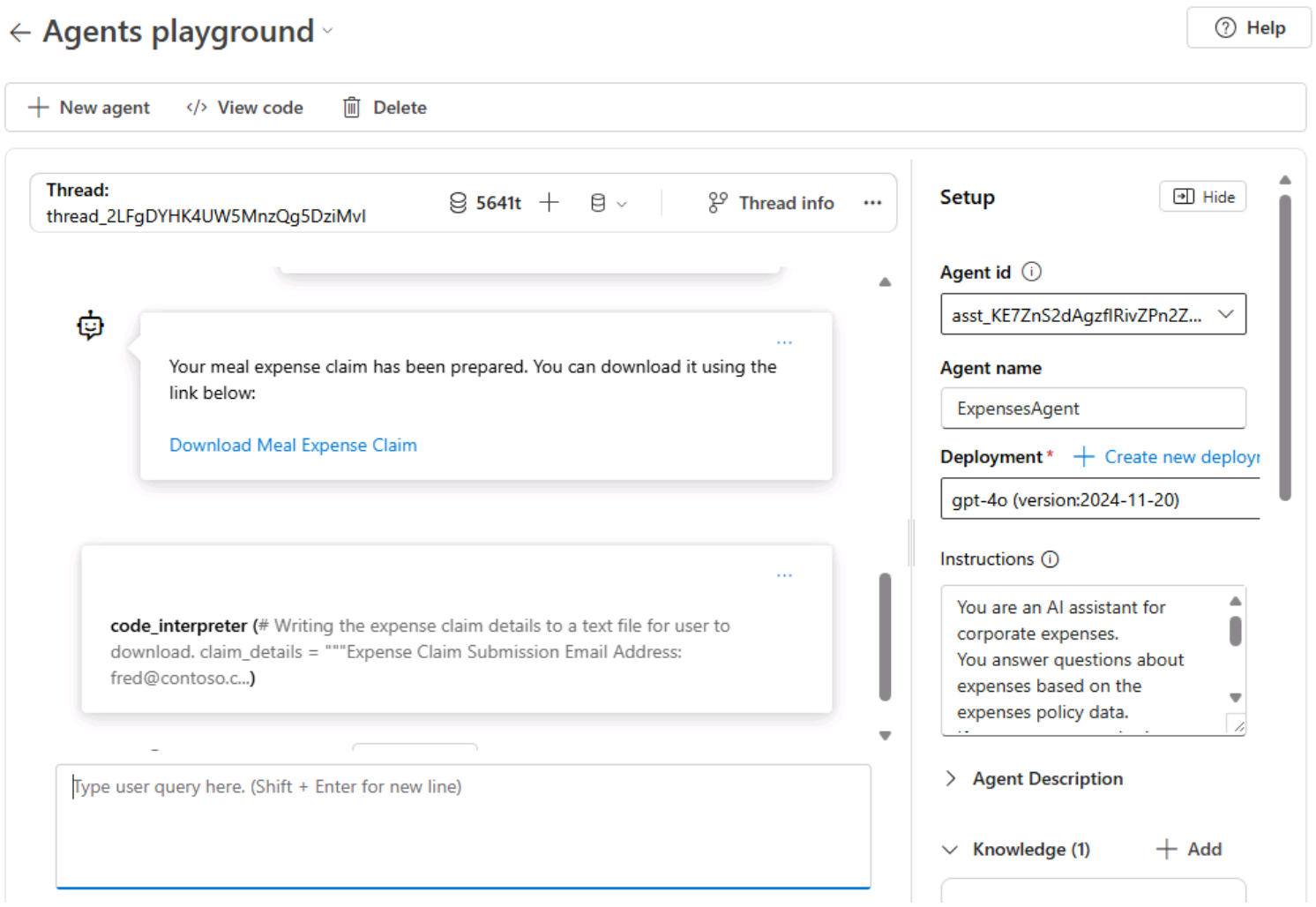
Now that you've created an agent, you can test it in the playground chat.

1. In the playground chat entry, enter the prompt: `What's the maximum I can claim for meals?` and review the agent's response - which should be based on information in the expenses policy document you added as knowledge to the agent setup.

   > **Note**: If the agent fails to respond because the rate limit is exceeded. Wait a few seconds and try again. If there is insufficient quota available in your subscription, the model may not be able to respond. If the problem persists, try to increase the quota for your model on the **Models + endpoints** page.

2. Try the following follow-up prompt: `I'd like to submit a claim for a meal.` and review the response. The agent should ask you for the required information to submit a claim.
3. Provide the agent with an email address; for example, `fred@contoso.com`. The agent should acknowledge the response and request the remaining information required for the expense claim (description and amount)
4. Submit a prompt that describes the claim and the amount; for example, `Breakfast cost me $20`.

5. The agent should use the code interpreter to prepare the expense claim text file, and provide a link so you can download it.



6. Download and open the text document to see the expense claim details.

Clean up

Now that you've finished the exercise, you should delete the cloud resources you've created to avoid unnecessary resource usage.

1. Open the Azure portal at `https://portal.azure.com` and view the contents of the resource group where you deployed the hub resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop an AI agent

In this exercise, you'll use Azure AI Agent Service to create a simple agent that analyzes data and creates charts. The agent can use the built-in *Code Interpreter* tool to dynamically generate any code required to analyze data.

> **Tip**: The code used in this exercise is based on the for Azure AI Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Azure AI Foundry SDK client libraries for details.

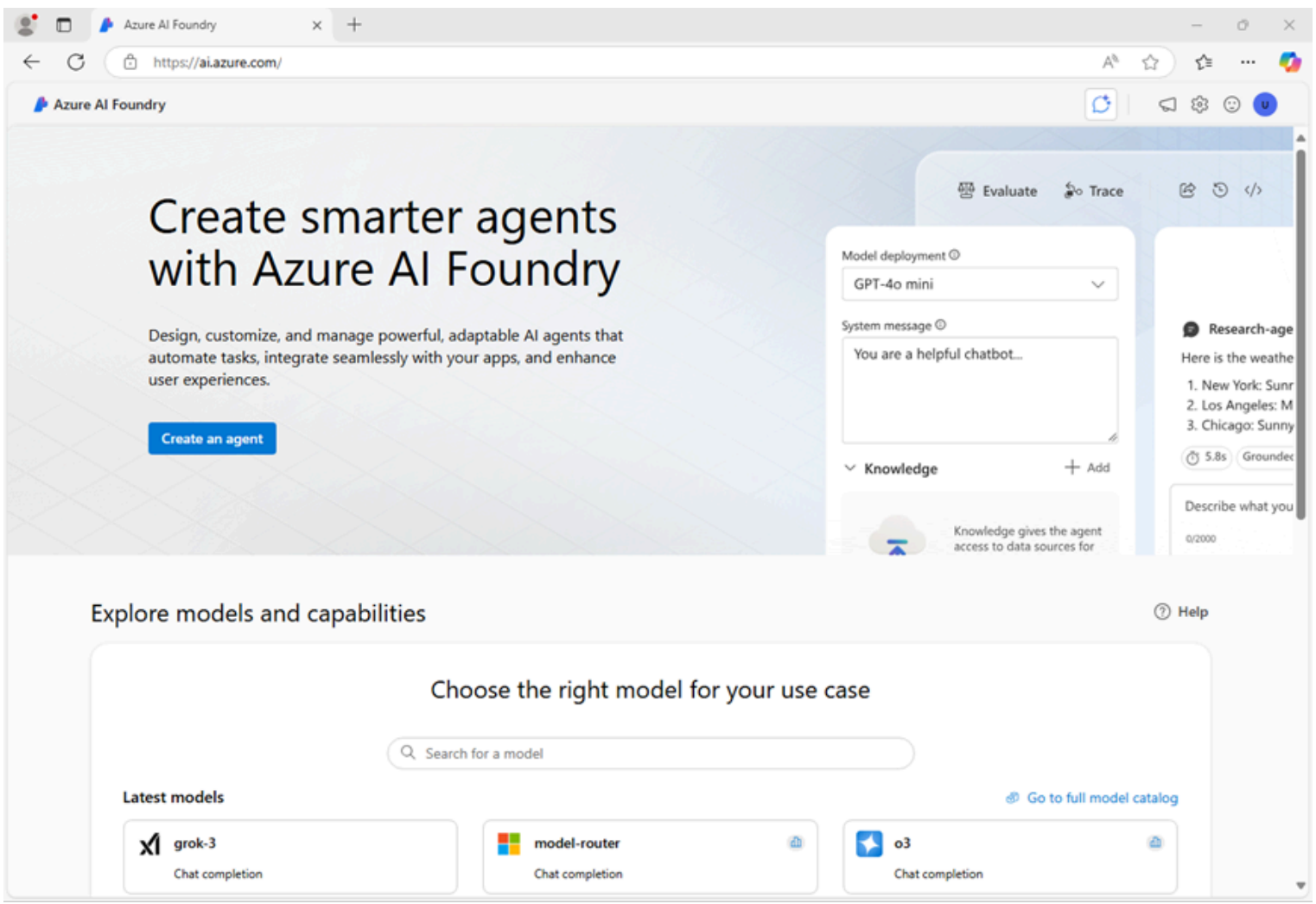
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any* ***AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

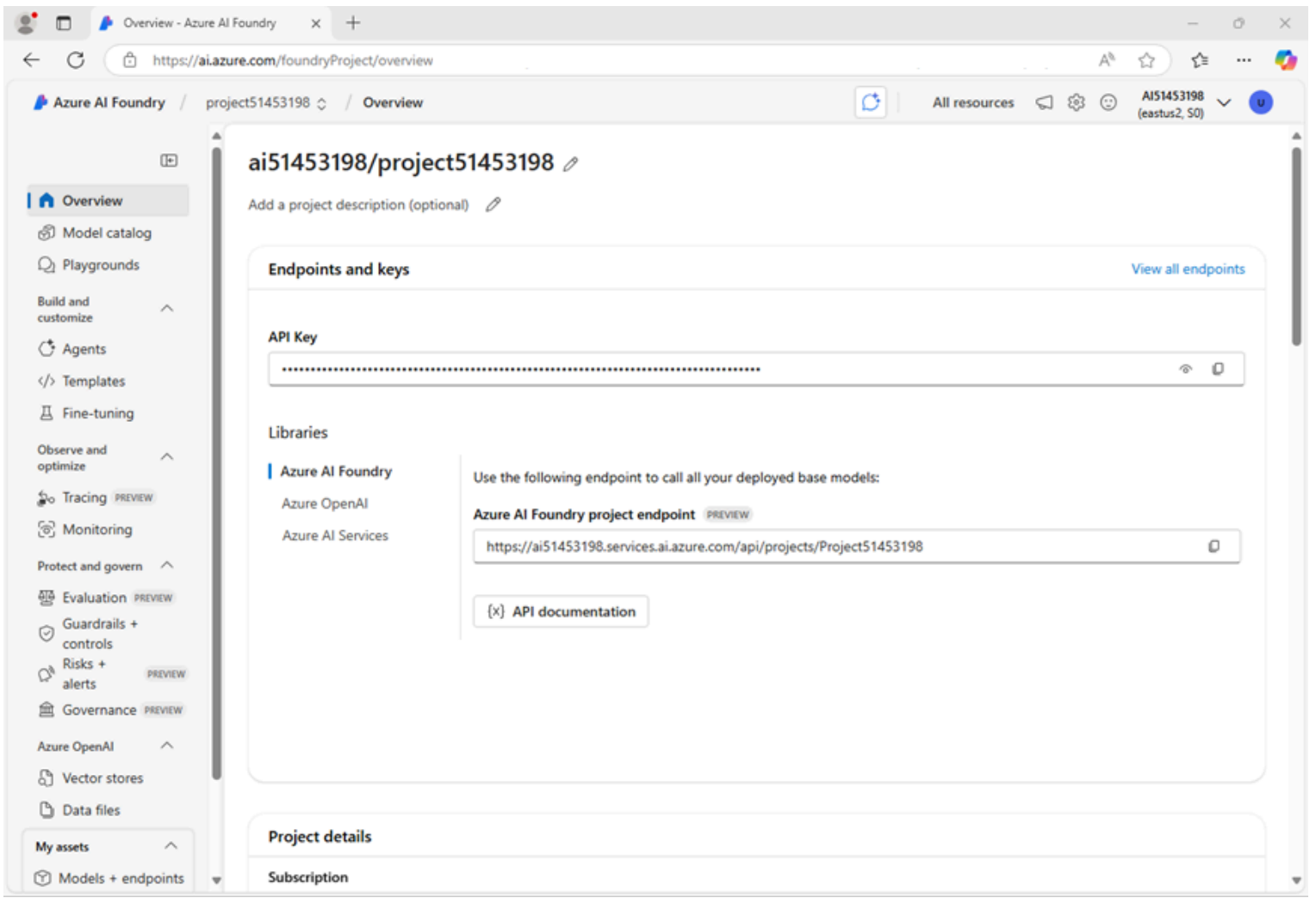
5. Select **Create** and wait for your project to be created.

6. If prompted, deploy a **gpt-4o** model using either the *Global Standard* or *Standard* deployment option (depending on your quota availability).

> **Note**: If quota is available, a GPT-4o base model may be deployed automatically when creating your Agent and project.

7. When your project is created, the Agents playground will be opened.

8. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



9. Copy the **Azure AI Foundry project endpoint** values to a notepad, as you'll use them to connect to your project in a client application.

Create an agent client app

Now you're ready to create a client app that uses an agent. Some code has been provided for you in a GitHub repository.

Clone the repo containing the application code

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

```
rm -r ai-agents -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
```

> **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. Enter the following command to change the working directory to the folder containing the code files and list them all.

```
cd ai-agents/Labfiles/02-build-ai-agent/Python
ls -a -l
```

The provided files include application code, configuration settings, and data.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-projects
```

2. Enter the following command to edit the configuration file that has been provided:

```
code .env
```

The file is opened in a code editor.

3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal) and ensure that the MODEL_DEPLOYMENT_NAME variable is set to your model deployment name (which should be *gpt-4o*).
4. After you've replaced the placeholder, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Write code for an agent app

> **Tip**: As you add code, be sure to maintain the correct indentation. Use the comment indentation levels as a guide.

1. Enter the following command to edit the code file that has been provided:

```
code agent.py
```

2. Review the existing code, which retrieves the application configuration settings and loads data from *data.txt* to be analyzed. The rest of the file includes comments where you'll add the necessary code to implement your data analysis agent.

3. Find the comment **Add references** and add the following code to import the classes you'll need to build an Azure AI agent that uses the built-in code interpreter tool:

Code Copy

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.agents import AgentsClient
from azure.ai.agents.models import FilePurpose, CodeInterpreterTool, ListSortOrder,
MessageRole
```

4. Find the comment **Connect to the Agent client** and add the following code to connect to the Azure AI project.

> **Tip**: Be careful to maintain the correct indentation level.

Code Copy

```
# Connect to the Agent client
agent_client = AgentsClient(
    endpoint=project_endpoint,
    credential=DefaultAzureCredential
        (exclude_environment_credential=True,
         exclude_managed_identity_credential=True)
)
with agent_client:
```

The code connects to the Azure AI Foundry project using the current Azure credentials. The final *with agent_client* statement starts a code block that defines the scope of the client, ensuring it's cleaned up when the code within the block is finished.

5. Find the comment **Upload the data file and create a CodeInterpreterTool**, within the *with agent_client* block, and add the following code to upload the data file to the project and create a CodeInterpreterTool that can access the data in it:

Code Copy

```
# Upload the data file and create a CodeInterpreterTool
file = agent_client.files.upload_and_poll(
     file_path=file_path, purpose=FilePurpose.AGENTS
)
print(f"Uploaded {file.filename}")

code_interpreter = CodeInterpreterTool(file_ids=[file.id])
```

6. Find the comment **Define an agent that uses the CodeInterpreterTool** and add the following code to define an AI agent that analyzes data and can use the code interpreter tool you defined previously:

Code Copy

```
# Define an agent that uses the CodeInterpreterTool
agent = agent_client.create_agent(
     model=model_deployment,
     name="data-agent",
     instructions="You are an AI agent that analyzes the data in the file that has been
uploaded. Use Python to calculate statistical metrics as necessary.",
     tools=code_interpreter.definitions,
     tool_resources=code_interpreter.resources,
)
print(f"Using agent: {agent.name}")
```

7. Find the comment **Create a thread for the conversation** and add the following code to start a thread on which the chat session with the agent will run:

Code Copy

```
# Create a thread for the conversation
thread = agent_client.threads.create()
```

8. Note that the next section of code sets up a loop for a user to enter a prompt, ending when the user enters “quit”.

9. Find the comment **Send a prompt to the agent** and add the following code to add a user message to the prompt (along with the data from the file that was loaded previously), and then run thread with the agent.

Code Copy

```
# Send a prompt to the agent
message = agent_client.messages.create(
     thread_id=thread.id,
     role="user",
     content=user_prompt,
 )

run = agent_client.runs.create_and_process(thread_id=thread.id, agent_id=agent.id)
```

10. Find the comment **Check the run status for failures** and add the following code to check for any errors.

Code Copy

```
# Check the run status for failures
if run.status == "failed":
     print(f"Run failed: {run.last_error}")
```

11. Find the comment **Show the latest response from the agent** and add the following code to retrieve the messages from the completed thread and display the last one that was sent by the agent.

Code Copy

```
# Show the latest response from the agent
last_msg = agent_client.messages.get_last_message_text_by_role(
    thread_id=thread.id,
    role=MessageRole.AGENT,
)
if last_msg:
    print(f"Last Message: {last_msg.text.value}")
```

12. Find the comment **Get the conversation history**, which is after the loop ends, and add the following code to print out the messages from the conversation thread; reversing the order to show them in chronological sequence

Code Copy

```
# Get the conversation history
print("\nConversation Log:\n")
messages = agent_client.messages.list(thread_id=thread.id, order=ListSortOrder.ASCENDING)
for message in messages:
    if message.text_messages:
        last_msg = message.text_messages[-1]
        print(f"{message.role}: {last_msg.text.value}\n")
```

13. Find the comment **Clean up** and add the following code to delete the agent and thread when no longer needed.

Code Copy

```
# Clean up
agent_client.delete_agent(agent.id)
```

14. Review the code, using the comments to understand how it:
    - Connects to the AI Foundry project.
    - Uploads the data file and creates a code interpreter tool that can access it.
    - Creates a new agent that uses the code interpreter tool and has explicit instructions to use Python as necessary for statistical analysis.
    - Runs a thread with a prompt message from the user along with the data to be analyzed.
    - Checks the status of the run in case there's a failure
    - Retrieves the messages from the completed thread and displays the last one sent by the agent.
    - Displays the conversation history
    - Deletes the agent and thread when they're no longer required.
15. Save the code file (*CTRL+S*) when you have finished. You can also close the code editor (*CTRL+Q*); though you may want to keep it open in case you need to make any edits to the code you added. In either case, keep the cloud shell command-line pane open.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. After you have signed in, enter the following command to run the application:

```
python agent.py
```

   The application runs using the credentials for your authenticated Azure session to connect to your project and create and run the agent.

4. When prompted, view the data that the app has loaded from the *data.txt* text file. Then enter a prompt such as:

```
What's the category with the highest cost?
```

> **Tip**: If the app fails because the rate limit is exceeded. Wait a few seconds and try again. If there is insufficient quota available in your subscription, the model may not be able to respond.

5. View the response. Then enter another prompt, this time requesting a visualization:

```
Create a text-based bar chart showing cost by category
```

6. View the response. Then enter another prompt, this time requesting a statistical metric:

```
What's the standard deviation of cost?
```

   View the response.
7. You can continue the conversation if you like. The thread is *stateful*, so it retains the conversation history - meaning that the agent has the full context for each response. Enter `quit` when you're done.
8. Review the conversation messages that were retrieved from the thread - which may include messages the agent generated to explain its steps when using the code interpreter tool.

Summary

In this exercise, you used the Azure AI Agent Service SDK to create a client application that uses an AI agent. The agent can use the built-in Code Interpreter tool to run dynamic Python code to perform statistical analyses.

Clean up

If you've finished exploring Azure AI Agent Service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop an AI agent with VS Code extension

In this exercise, you'll use the Azure AI Foundry VS Code extension to create an agent that can use Model Context Protocol (MCP) server tools to access external data sources and APIs. The agent will be able to retrieve up-to-date information and interact with various services through MCP tools.

This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Prerequisites

Before starting this exercise, ensure you have:

- Visual Studio Code installed
- An active Azure subscription

Install the Azure AI Foundry VS Code extension

Let's start by installing and setting up the VS Code extension.

1. Open Visual Studio Code.
2. Select **Extensions** from the left pane (or press **Ctrl+Shift+X**).
3. In the search bar, type **Azure AI Foundry** and press Enter.
4. Select the **Azure AI Foundry** extension from Microsoft and click **Install**.
5. After installation is complete, verify the extension appears in the primary navigation bar on the left side of Visual Studio Code.

Sign in to Azure and create a project

Now you'll connect to your Azure resources and create a new AI Foundry project.

1. In the VS Code sidebar, select the **Azure AI Foundry** extension icon.
2. In the Azure Resources view, select **Sign in to Azure...** and follow the authentication prompts.
3. After signing in, select your Azure subscription from the dropdown.
4. Create a new Azure AI Foundry project by selecting the **+** (plus) icon next to **Resources** in the Azure AI Foundry Extension view.
5. Choose whether to create a new resource group or use an existing one:

   To create a new resource group:

   - Select **Create new resource group** and press Enter
   - Enter a name for your resource group (e.g., "rg-ai-agents-lab") and press Enter
   - Select a location from the available options and press Enter

   To use an existing resource group:

   - Select the resource group you want to use from the list and press Enter

6. Enter a name for your Azure AI Foundry project (e.g., "ai-agents-project") in the textbox and press Enter.

7. Wait for the project deployment to complete. A popup will appear with the message “Project deployed successfully.”

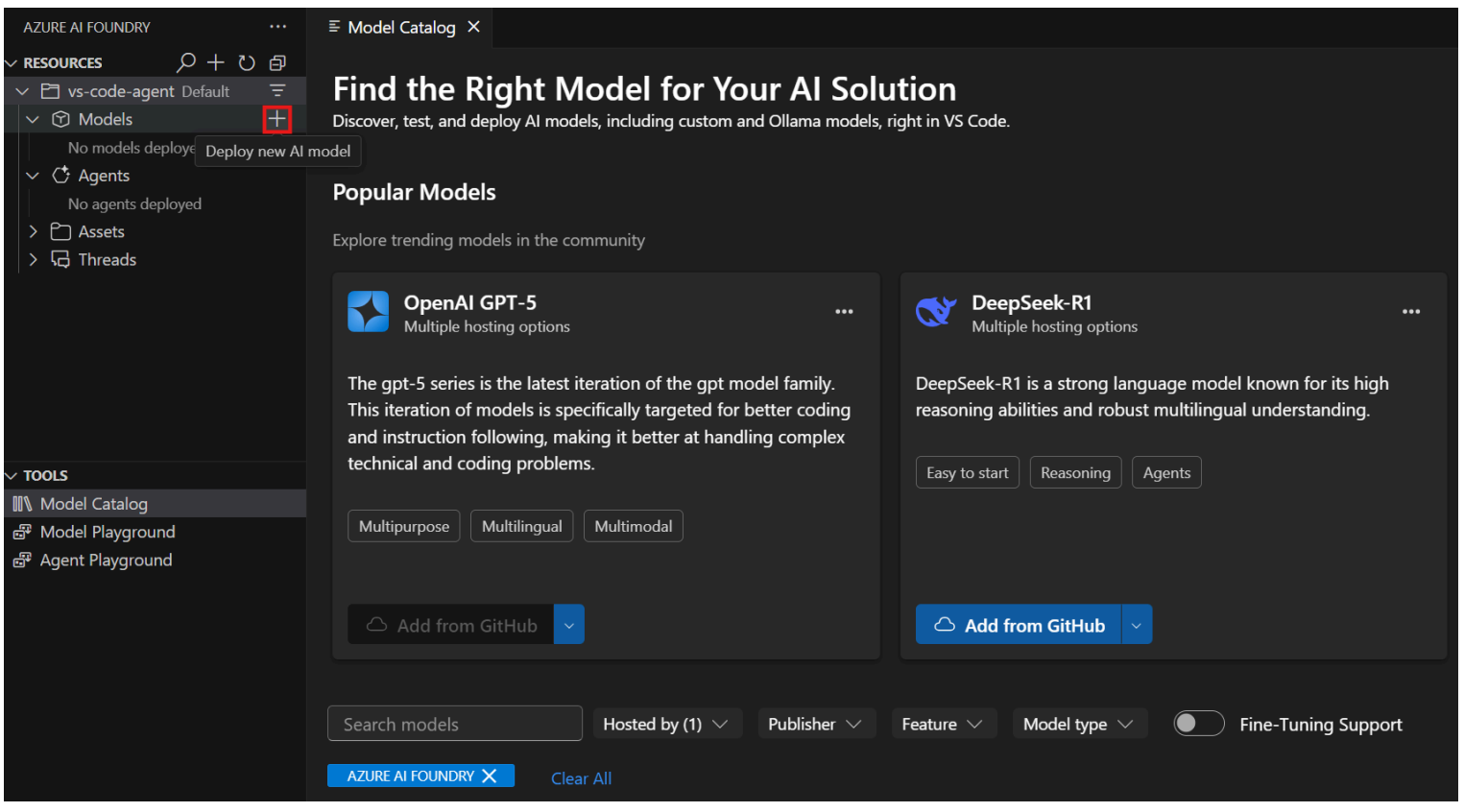
Deploy a model

You’ll need a deployed model to use with your agent.

1. When the “Project deployed successfully” popup appears, select the **Deploy a model** button. This opens the Model Catalog.

> **Tip**: You can also access the Model Catalog by selecting the **+** icon next to **Models** in the Resources section, or by pressing **F1** and running the command **Azure AI Foundry: Open Model Catalog**.

2. In the Model Catalog, locate the **gpt-4o** model (you can use the search bar to find it quickly).

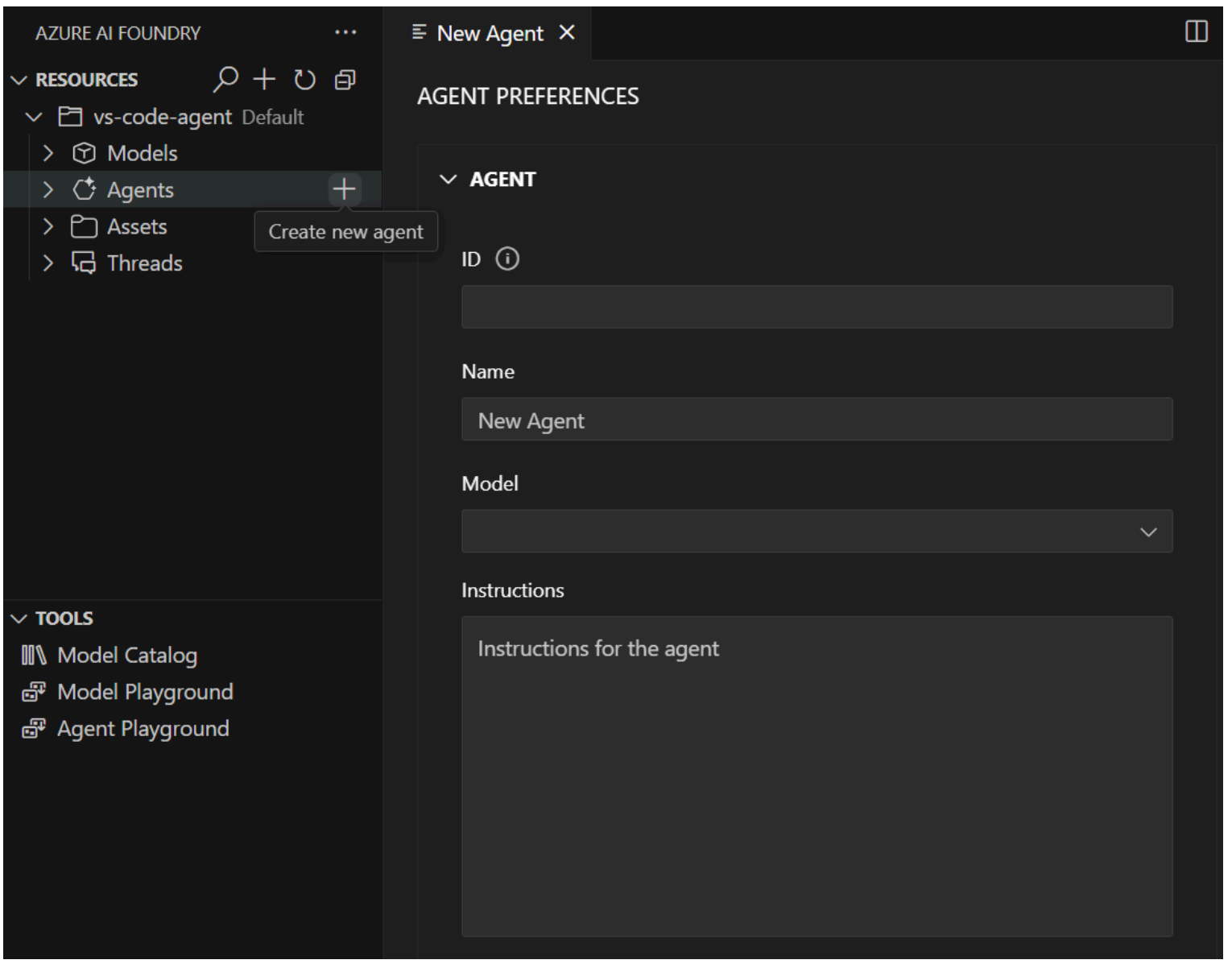


3. Select **Deploy in Azure** next to the gpt-4o model.
4. Configure the deployment settings:
    - **Deployment name**: Enter a name like “gpt-4o-deployment”
    - **Deployment type**: Select **Global Standard** (or **Standard** if Global Standard is not available)
    - **Model version**: Leave as default
    - **Tokens per minute**: Leave as default
5. Select **Deploy in Azure AI Foundry** in the bottom-left corner.
6. In the confirmation dialog, select **Deploy** to deploy the model.
7. Wait for the deployment to complete. Your deployed model will appear under the **Models** section in the Resources view.

Create an AI agent with the designer view

Now you’ll create an AI agent using the visual designer interface.

1. In the Azure AI Foundry extension view, find the **Resources** section.
2. Select the **+** (plus) icon next to the **Agents** subsection to create a new AI Agent.



3. Choose a location to save your agent files when prompted.

4. The agent designer view will open along with a `.yaml` configuration file.

Configure your agent in the designer

1. In the agent designer, configure the following fields:

    - **Name**: Enter a descriptive name for your agent (e.g., “data-research-agent”)
    - **Description**: Add a description explaining the agent’s purpose
    - **Model**: Select your GPT-4o deployment from the dropdown
    - **Instructions**: Enter system instructions such as:

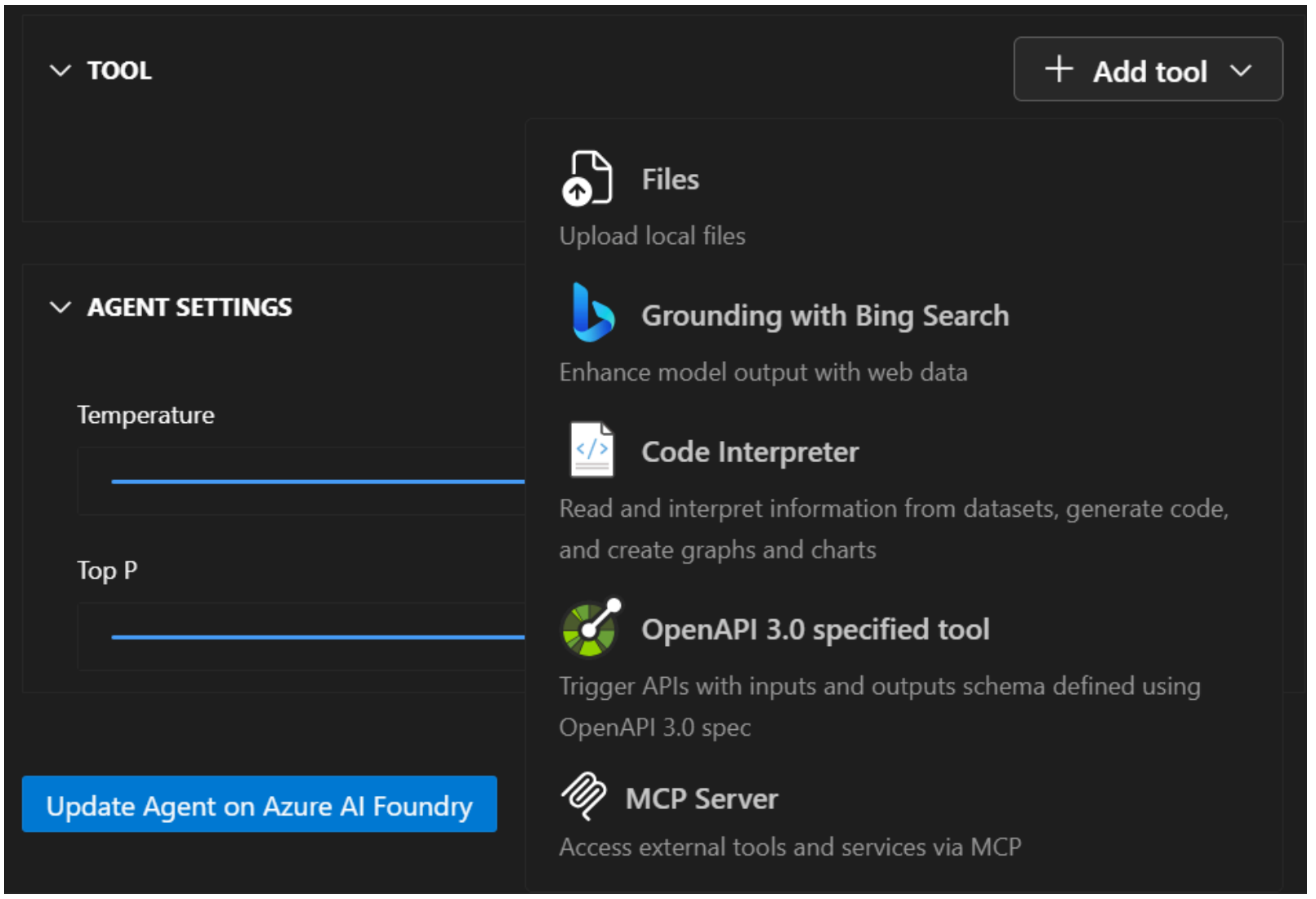
```
You are an AI agent that helps users research information from various sources. Use the available tools to access up-to-date information and provide comprehensive responses based on external data sources.
```

2. Save the configuration by selecting **File > Save** from the VS Code menu bar.

Add an MCP Server tool to your agent

You’ll now add a Model Context Protocol (MCP) server tool that allows your agent to access external APIs and data sources.

1. In the **TOOL** section of the designer, select the **Add tool** button in the top-right corner.



1. From the dropdown menu, choose **MCP Server**.
2. Configure the MCP Server tool with the following information:
   - **Server URL**: Enter the URL of an MCP server (e.g., `https://gitmcp.io/Azure/azure-rest-api-specs`)
   - **Server Label**: Enter a unique identifier (e.g., “github_docs_server”)
3. Leave the **Allowed tools** dropdown empty to allow all tools from the MCP server.
4. Select the **Create tool** button to add the tool to your agent.

Deploy your agent to Azure AI Foundry

1. In the designer view, select the **Create on Azure AI Foundry** button in the bottom-left corner.
2. Wait for the deployment to complete.
3. In the VS Code navbar, refresh the **Azure Resources** view. Your deployed agent should now appear under the **Agents** subsection.

Test your agent in the playground

1. Right-click on your deployed agent in the **Agents** subsection.
2. Select **Open Playground** from the context menu.
3. The Agents Playground will open in a new tab within VS Code.
4. Type a test prompt such as:

```
Can you help me find documentation about Azure Container Apps and provide an example of how to create one?
```

5. Send the message and observe the authentication and approval prompts for the MCP Server tool:
   - For this exercise, select **No Authentication** when prompted.
   - For the MCP Tools approval preference, you can select **Always approve**.
6. Review the agent’s response and note how it uses the MCP server tool to retrieve external information.
7. Check the **Agent Annotations** section to see the sources of information used by the agent.

Generate sample code for your agent

1. Right-click on your deployed agent and select **Open Code File**, or select the **Open Code File** button in the Agent Preferences page.
2. Choose your preferred SDK from the dropdown (Python, .NET, JavaScript, or Java).
3. Select your preferred programming language.
4. Choose your preferred authentication method.
5. Review the generated sample code that demonstrates how to interact with your agent programmatically.

You can use this code as a starting point for building applications that leverage your AI agent.

View conversation history and threads

1. In the **Azure Resources** view, expand the **Threads** subsection to see conversations created during your agent interactions.
2. Select a thread to view the **Thread Details** page, which shows:
   - Individual messages in the conversation
   - Run information and execution details
   - Agent responses and tool usage
3. Select **View run info** to see detailed JSON information about each run.

Summary

In this exercise, you used the Azure AI Foundry VS Code extension to create an AI agent with MCP server tools. The agent can access external data sources and APIs through the Model Context Protocol, enabling it to provide up-to-date information and interact with various services. You also learned how to test the agent in the playground and generate sample code for programmatic interaction.

Clean up

When you've finished exploring the Azure AI Foundry VS Code extension, you should clean up the resources to avoid incurring unnecessary Azure costs.

Delete your agents

1. In the Azure AI Foundry portal, select **Agents** from the navigation menu.
2. Select your agent and then select the **Delete** button.

Delete your models

1. In VS Code, refresh the **Azure Resources** view.
2. Expand the **Models** subsection.
3. Right-click on your deployed model and select **Delete**.

Delete other resources

1. Open the Azure portal.
2. Navigate to the resource group containing your AI Foundry resources.
3. Select **Delete resource group** and confirm the deletion.

Use a custom function in an AI agent

In this exercise you'll explore creating an agent that can use custom functions as a tool to complete tasks. You'll build a simple technical support agent that can collect details of a technical problem and generate a support ticket.

> **Tip**: The code used in this exercise is based on the for Azure AI Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Azure AI Foundry SDK client libraries for details.

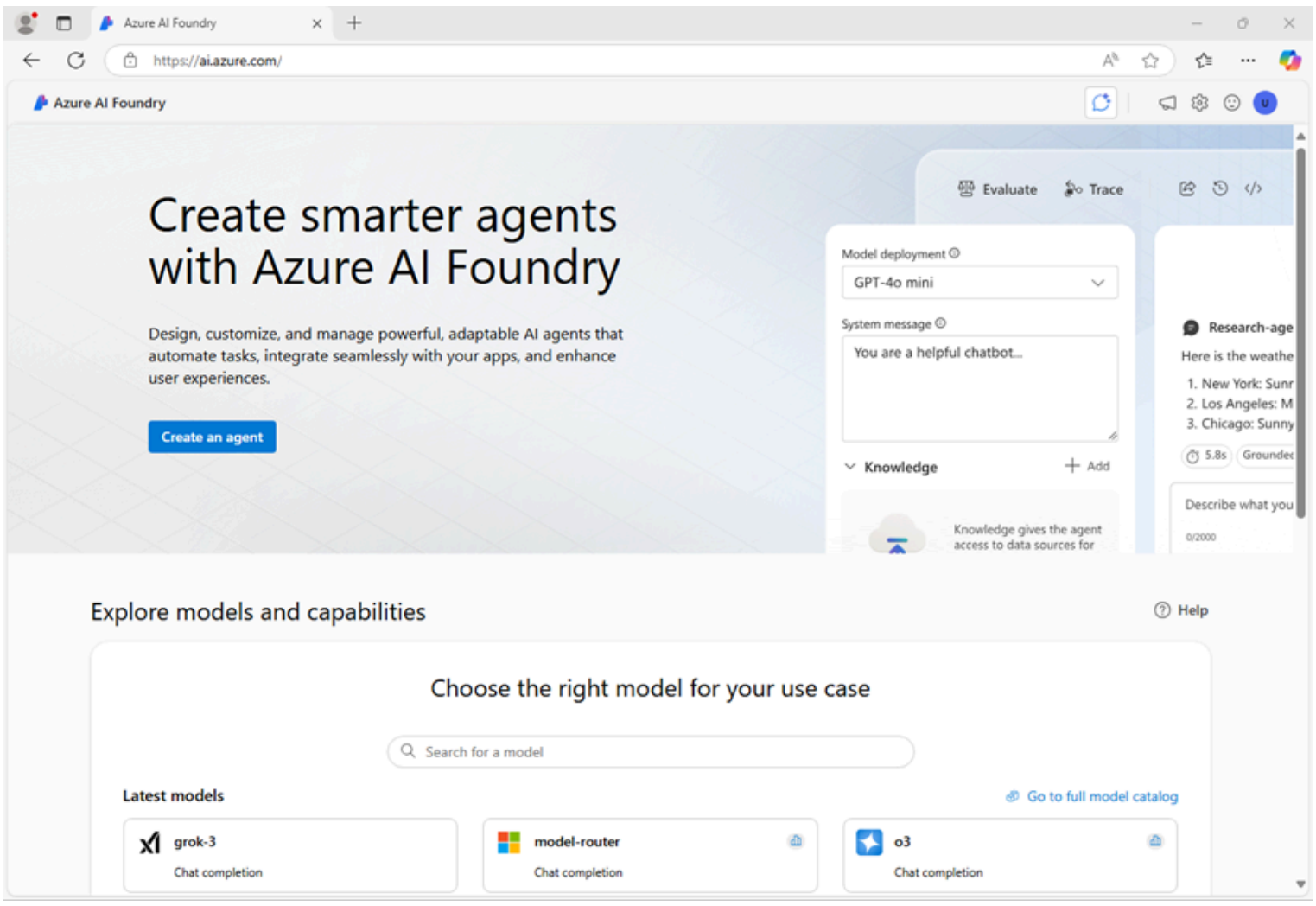
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



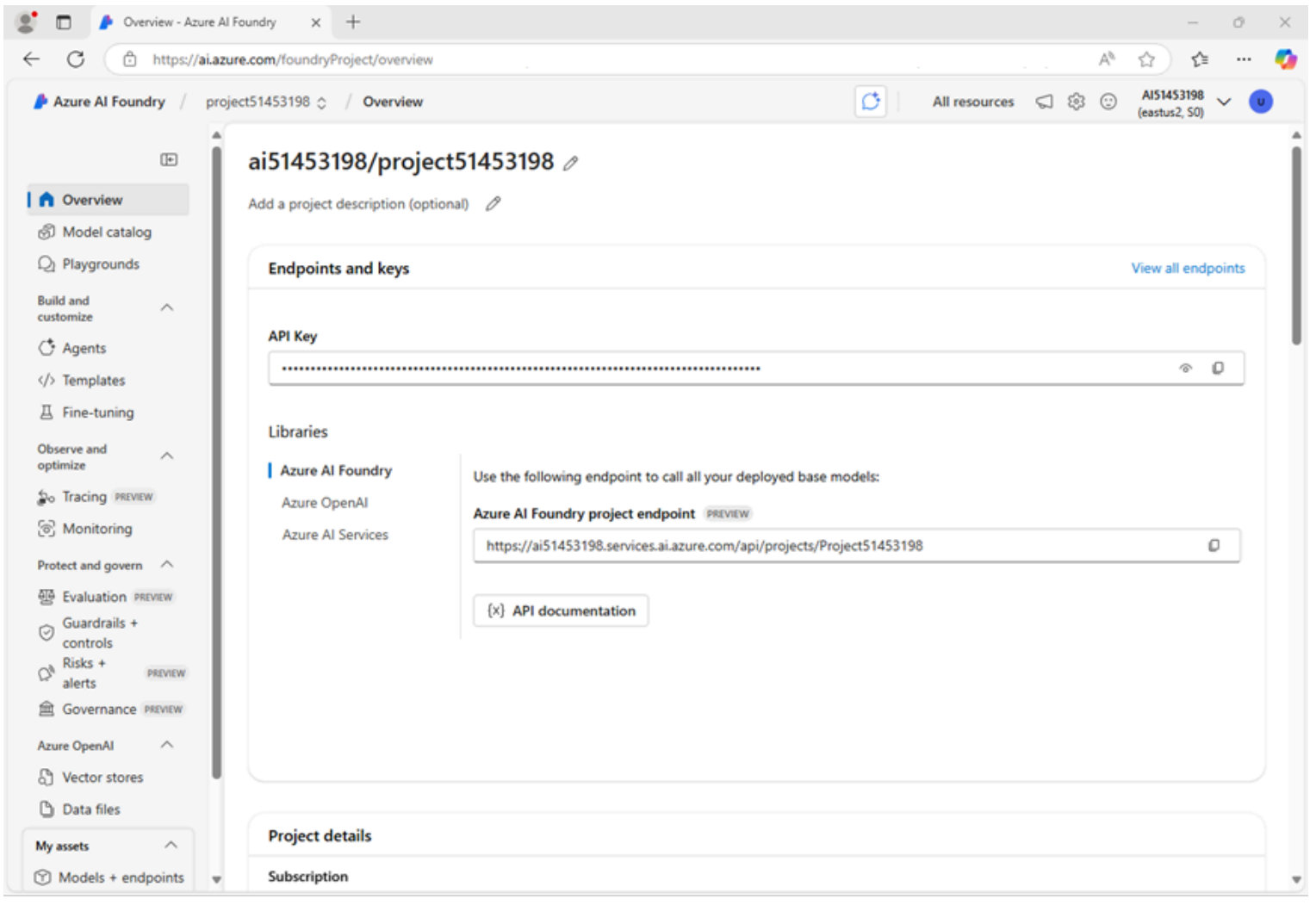
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project to be created.
6. If prompted, deploy a **gpt-4o** model using either the *Global Standard* or *Standard* deployment option (depending on your quota availability).

> **Note**: If quota is available, a GPT-4o base model may be deployed automatically when creating your Agent and project.

7. When your project is created, the Agents playground will be opened.
8. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



9. Copy the **Azure AI Foundry project endpoint** values to a notepad, as you'll use them to connect to your project in a client application.

Develop an agent that uses function tools

Now that you've created your project in AI Foundry, let's develop an app that implements an agent using custom function tools.

Clone the repo containing the application code

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

```
rm -r ai-agents -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
```

> **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. Enter the following command to change the working directory to the folder containing the code files and list them all.

```
cd ai-agents/Labfiles/03-ai-agent-functions/Python
ls -a -l
```

   The provided files include application code and a file for configuration settings.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-projects
```

> **Note:** You can ignore any warning or error messages displayed during the library installation.

2. Enter the following command to edit the configuration file that has been provided:

```
code .env
```

   The file is opened in a code editor.
3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal) and ensure that the MODEL_DEPLOYMENT_NAME variable is set to your model deployment name (which should be *gpt-4o*).
4. After you've replaced the placeholder, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Define a custom function

1. Enter the following command to edit the code file that has been provided for your function code:

```
code user_functions.py
```

2. Find the comment **Create a function to submit a support ticket** and add the following code, which generates a ticket number and saves a support ticket as a text file.

```
# Create a function to submit a support ticket
def submit_support_ticket(email_address: str, description: str) -> str:
     script_dir = Path(__file__).parent  # Get the directory of the script
     ticket_number = str(uuid.uuid4()).replace('-', '')[:6]
     file_name = f"ticket-{ticket_number}.txt"
     file_path = script_dir / file_name
     text = f"Support ticket: {ticket_number}\nSubmitted by:
{email_address}\nDescription:\n{description}"
     file_path.write_text(text)

     message_json = json.dumps({"message": f"Support ticket {ticket_number} submitted. The
ticket file is saved as {file_name}"})
     return message_json
```

3. Find the comment **Define a set of callable functions** and add the following code, which statically defines a set of callable functions in this code file (in this case, there's only one - but in a real solution you may have multiple functions that your agent can call):

```
# Define a set of callable functions
user_functions: Set[Callable[..., Any]] = {
     submit_support_ticket
 }
```

4. Save the file (*CTRL+S*).

Write code to implement an agent that can use your function

1. Enter the following command to begin editing the agent code.

```
code agent.py
```

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

2. Review the existing code, which retrieves the application configuration settings and sets up a loop in which the user can enter prompts for the agent. The rest of the file includes comments where you'll add the necessary code to implement your technical support agent.

3. Find the comment **Add references** and add the following code to import the classes you'll need to build an Azure AI agent that uses your function code as a tool:

Code Copy

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.agents import AgentsClient
from azure.ai.agents.models import FunctionTool, ToolSet, ListSortOrder, MessageRole
from user_functions import user_functions
```

4. Find the comment **Connect to the Agent client** and add the following code to connect to the Azure AI project using the current Azure credentials.

> **Tip**: Be careful to maintain the correct indentation level.

Code Copy

```
# Connect to the Agent client
agent_client = AgentsClient(
    endpoint=project_endpoint,
    credential=DefaultAzureCredential
        (exclude_environment_credential=True,
         exclude_managed_identity_credential=True)
)
```

5. Find the comment **Define an agent that can use the custom functions** section, and add the following code to add your function code to a toolset, and then create an agent that can use the toolset and a thread on which to run the chat session.

Code Copy

```
# Define an agent that can use the custom functions
with agent_client:

     functions = FunctionTool(user_functions)
     toolset = ToolSet()
     toolset.add(functions)
     agent_client.enable_auto_function_calls(toolset)

     agent = agent_client.create_agent(
         model=model_deployment,
         name="support-agent",
         instructions="""You are a technical support agent.
                         When a user has a technical issue, you get their email address and
a description of the issue.
                         Then you use those values to submit a support ticket using the
function available to you.
                         If a file is saved, tell the user the file name.
                        """,
         toolset=toolset
     )

     thread = agent_client.threads.create()
     print(f"You're chatting with: {agent.name} ({agent.id})")
```

6. Find the comment **Send a prompt to the agent** and add the following code to add the user's prompt as a message and run the thread.

Code Copy

```
# Send a prompt to the agent
message = agent_client.messages.create(
     thread_id=thread.id,
     role="user",
     content=user_prompt
)
run = agent_client.runs.create_and_process(thread_id=thread.id, agent_id=agent.id)
```

> **Note**: Using the **create_and_process** method to run the thread enables the agent to automatically find your functions and choose to use them based on their names and parameters. As an alternative, you could use the **create_run** method, in which case you would be responsible for writing code to poll for run status to determine when a function call is required, call the function, and return the results to the agent.

7. Find the comment **Check the run status for failures** and add the following code to show any errors that occur.

Code Copy

```
# Check the run status for failures
if run.status == "failed":
     print(f"Run failed: {run.last_error}")
```

8. Find the comment **Show the latest response from the agent** and add the following code to retrieve the messages from the completed thread and display the last one that was sent by the agent.

Code Copy

```
# Show the latest response from the agent
last_msg = agent_client.messages.get_last_message_text_by_role(
    thread_id=thread.id,
    role=MessageRole.AGENT,
)
if last_msg:
     print(f"Last Message: {last_msg.text.value}")
```

9. Find the comment **Get the conversation history** and add the following code to print out the messages from the conversation thread; ordering them in chronological sequence

Code Copy

```
# Get the conversation history
print("\nConversation Log:\n")
messages = agent_client.messages.list(thread_id=thread.id, order=ListSortOrder.ASCENDING)
for message in messages:
     if message.text_messages:
        last_msg = message.text_messages[-1]
        print(f"{message.role}: {last_msg.text.value}\n")
```

10. Find the comment **Clean up** and add the following code to delete the agent and thread when no longer needed.

Code Copy

```
# Clean up
agent_client.delete_agent(agent.id)
print("Deleted agent")
```

11. Review the code, using the comments to understand how it:
    - Adds your set of custom functions to a toolset
    - Creates an agent that uses the toolset.
    - Runs a thread with a prompt message from the user.
    - Checks the status of the run in case there's a failure
    - Retrieves the messages from the completed thread and displays the last one sent by the agent.
    - Displays the conversation history
    - Deletes the agent and thread when they're no longer required.
12. Save the code file (*CTRL+S*) when you have finished. You can also close the code editor (*CTRL+Q*); though you may want to keep it open in case you need to make any edits to the code you added. In either case, keep the cloud shell command-line pane open.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. After you have signed in, enter the following command to run the application:

Code Copy

```
python agent.py
```

The application runs using the credentials for your authenticated Azure session to connect to your project and create and run the agent.

4. When prompted, enter a prompt such as:

Code Copy

```
I have a technical problem
```

> **Tip**: If the app fails because the rate limit is exceeded. Wait a few seconds and try again. If there is insufficient quota available in your subscription, the model may not be able to respond.

Create an Azure AI Foundry project

Develop an agent that uses function tools

Clean up

5. View the response. The agent may ask for your email address and a description of the issue. You can use any email address (for example, `alex@contoso.com`) and any issue description (for example `my computer won't start`)

   When it has enough information, the agent should choose to use your function as required.
6. You can continue the conversation if you like. The thread is *stateful*, so it retains the conversation history - meaning that the agent has the full context for each response. Enter `quit` when you're done.
7. Review the conversation messages that were retrieved from the thread, and the tickets that were generated.

8. The tool should have saved support tickets in the app folder. You can use the `ls` command to check, and then use the `cat` command to view the file contents, like this:

   Code Copy

   ```
   cat ticket-<ticket_num>.txt
   ```

Clean up

Now that you've finished the exercise, you should delete the cloud resources you've created to avoid unnecessary resource usage.

1. Open the Azure portal at `https://portal.azure.com` and view the contents of the resource group where you deployed the hub resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop a multi-agent solution

In this exercise, you'll create a project that orchestrates multiple AI agents using Azure AI Foundry Agent Service. You'll design an AI solution that assists with ticket triage. The connected agents will assess the ticket's priority, suggest a team assignment, and determine the level of effort required to complete the ticket. Let's get started!

> **Tip**: The code used in this exercise is based on the for Azure AI Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Azure AI Foundry SDK client libraries for details.

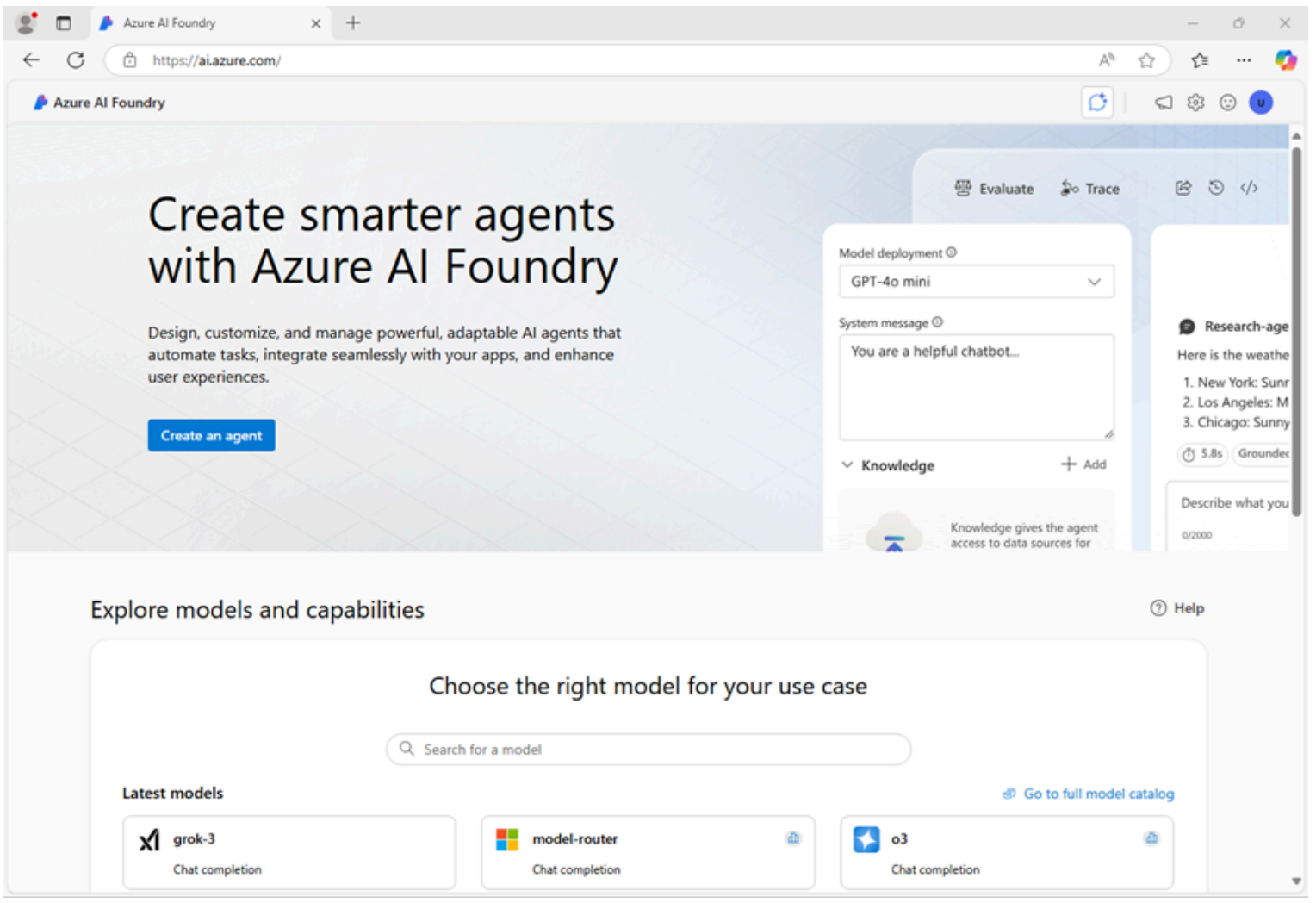
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



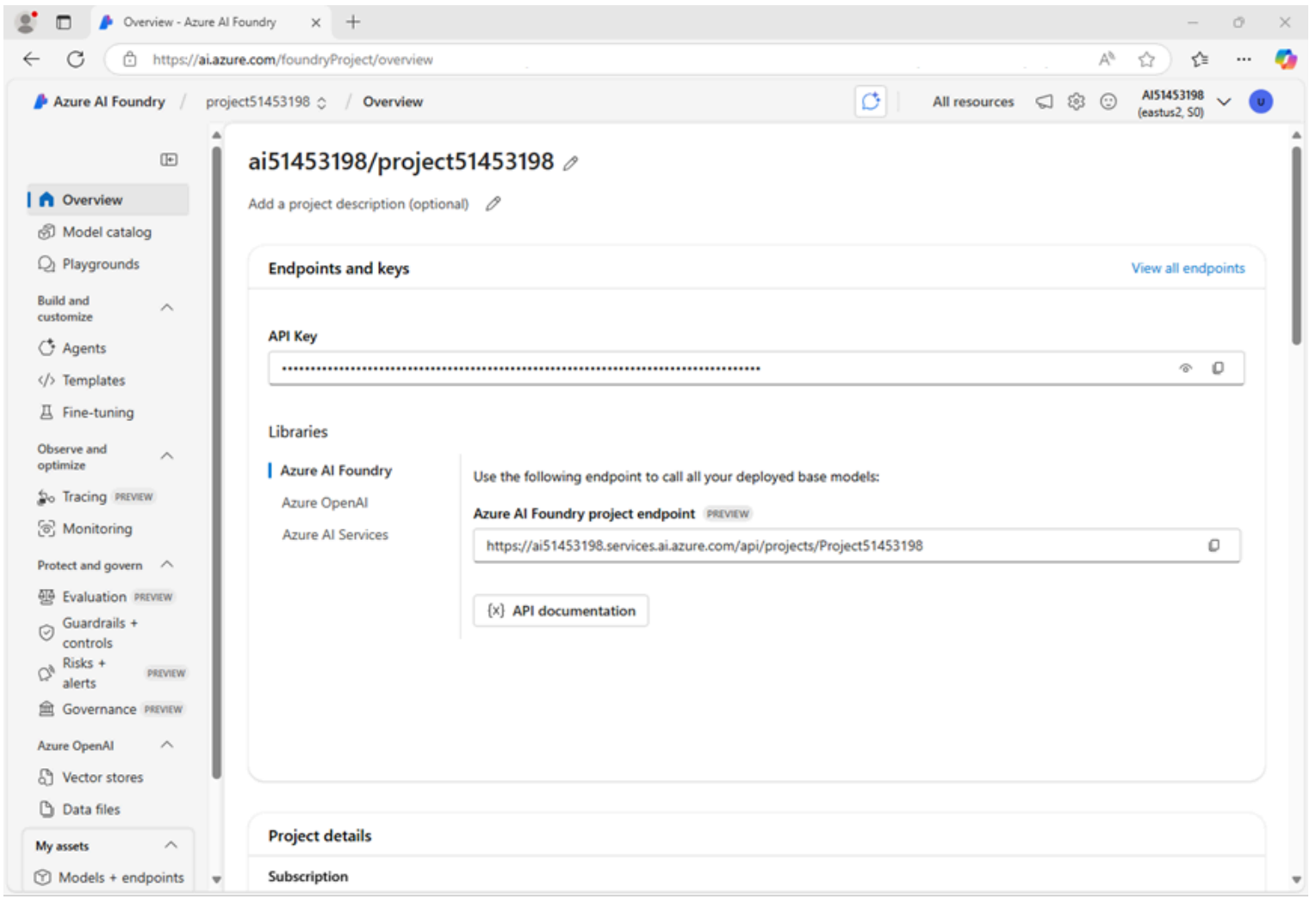
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project to be created.
6. If prompted, deploy a **gpt-4o** model using either the *Global Standard* or *Standard* deployment option (depending on your quota availability).

> **Note**: If quota is available, a GPT-4o base model may be deployed automatically when creating your Agent and project.

7. When your project is created, the Agents playground will be opened.
8. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



9. Copy the **Azure AI Foundry project endpoint** values to a notepad, as you'll use them to connect to your project in a client application.

Create an AI Agent client app

Now you're ready to create a client app that defines the agents and instructions. Some code is provided for you in a GitHub repository.

Prepare the environment

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r ai-agents -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
   ```

   > **Tip**: As you enter commands into the cloud shell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. When the repo has been cloned, enter the following command to change the working directory to the folder containing the code files and list them all.

   Code Copy

   ```
   cd ai-agents/Labfiles/03b-build-multi-agent-solution/Python
   ls -a -l
   ```

   The provided files include application code and a file for configuration settings.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-projects
   ```

2. Enter the following command to edit the configuration file that is provided:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal), and the **your_model_deployment** placeholder with the name you assigned to your gpt-4o model deployment (which by default is `gpt-4o`).

4. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Create AI agents

Now you're ready to create the agents for your multi-agent solution! Let's get started!

1. Enter the following command to edit the **agent_triage.py** file:

```
code agent_triage.py
```

2. Review the code in the file, noting that it contains strings for each agent name and instructions.

3. Find the comment **Add references** and add the following code to import the classes you'll need:

```
# Add references
from azure.ai.agents import AgentsClient
from azure.ai.agents.models import ConnectedAgentTool, MessageRole, ListSortOrder, ToolSet,
FunctionTool
from azure.identity import DefaultAzureCredential
```

4. Note that code to load the project endpoint and model name from your environment variables has been provided.

5. Find the comment **Connect to the agents client**, and add the following code to create an AgentsClient connected to your project:

```
# Connect to the agents client
agents_client = AgentsClient(
     endpoint=project_endpoint,
     credential=DefaultAzureCredential(
         exclude_environment_credential=True,
         exclude_managed_identity_credential=True
     ),
)
```

Now you'll add code that uses the AgentsClient to create multiple agents, each with a specific role to play in processing a support ticket.

> **Tip**: When adding subsequent code, be sure to maintain the right level of indentation under the `using agents_client:` statement.

6. Find the comment **Create an agent to prioritize support tickets**, and enter the following code (being careful to retain the right level of indentation):

```
# Create an agent to prioritize support tickets
priority_agent_name = "priority_agent"
priority_agent_instructions = """
Assess how urgent a ticket is based on its description.

Respond with one of the following levels:
- High: User-facing or blocking issues
- Medium: Time-sensitive but not breaking anything
- Low: Cosmetic or non-urgent tasks

Only output the urgency level and a very brief explanation.
"""

priority_agent = agents_client.create_agent(
     model=model_deployment,
     name=priority_agent_name,
     instructions=priority_agent_instructions
)
```

7. Find the comment **Create an agent to assign tickets to the appropriate team**, and enter the following code:

Code Copy

```
# Create an agent to assign tickets to the appropriate team
team_agent_name = "team_agent"
team_agent_instructions = """
Decide which team should own each ticket.

Choose from the following teams:
- Frontend
- Backend
- Infrastructure
- Marketing

Base your answer on the content of the ticket. Respond with the team name and a very brief
explanation.
"""

team_agent = agents_client.create_agent(
     model=model_deployment,
     name=team_agent_name,
     instructions=team_agent_instructions
)
```

8. Find the comment **Create an agent to estimate effort for a support ticket**, and enter the following code:

Code Copy

```
# Create an agent to estimate effort for a support ticket
effort_agent_name = "effort_agent"
effort_agent_instructions = """
Estimate how much work each ticket will require.

Use the following scale:
- Small: Can be completed in a day
- Medium: 2-3 days of work
- Large: Multi-day or cross-team effort

Base your estimate on the complexity implied by the ticket. Respond with the effort level
and a brief justification.
"""

effort_agent = agents_client.create_agent(
     model=model_deployment,
     name=effort_agent_name,
     instructions=effort_agent_instructions
)
```

So far, you've created three agents; each of which has a specific role in triaging a support ticket. Now let's create ConnectedAgentTool objects for each of these agents so they can be used by other agents.

9. Find the comment **Create connected agent tools for the support agents**, and enter the following code:

Code Copy

```
# Create connected agent tools for the support agents
priority_agent_tool = ConnectedAgentTool(
     id=priority_agent.id,
     name=priority_agent_name,
     description="Assess the priority of a ticket"
)

team_agent_tool = ConnectedAgentTool(
     id=team_agent.id,
     name=team_agent_name,
     description="Determines which team should take the ticket"
)

effort_agent_tool = ConnectedAgentTool(
     id=effort_agent.id,
     name=effort_agent_name,
     description="Determines the effort required to complete the ticket"
)
```

Now you're ready to create a primary agent that will coordinate the ticket triage process, using the connected agents as required.

10. Find the comment **Create an agent to triage support ticket processing by using connected agents**, and enter the following code:

Code Copy

```
# Create an agent to triage support ticket processing by using connected agents
triage_agent_name = "triage-agent"
triage_agent_instructions = """
Triage the given ticket. Use the connected tools to determine the ticket's priority,
which team it should be assigned to, and how much effort it may take.
"""

triage_agent = agents_client.create_agent(
     model=model_deployment,
     name=triage_agent_name,
     instructions=triage_agent_instructions,
     tools=[
         priority_agent_tool.definitions[0],
         team_agent_tool.definitions[0],
         effort_agent_tool.definitions[0]
     ]
)
```

Now that you have defined a primary agent, you can submit a prompt to it and have it use the other agents to triage a support issue.

11. Find the comment **Use the agents to triage a support issue**, and enter the following code:

Code Copy

```
# Use the agents to triage a support issue
print("Creating agent thread.")
thread = agents_client.threads.create()

# Create the ticket prompt
prompt = input("\nWhat's the support problem you need to resolve?: ")

# Send a prompt to the agent
message = agents_client.messages.create(
     thread_id=thread.id,
     role=MessageRole.USER,
     content=prompt,
)

# Run the thread usng the primary agent
print("\nProcessing agent thread. Please wait.")
run = agents_client.runs.create_and_process(thread_id=thread.id, agent_id=triage_agent.id)

if run.status == "failed":
     print(f"Run failed: {run.last_error}")

# Fetch and display messages
messages = agents_client.messages.list(thread_id=thread.id, order=ListSortOrder.ASCENDING)
for message in messages:
     if message.text_messages:
         last_msg = message.text_messages[-1]
         print(f"{message.role}:\n{last_msg.text.value}\n")
```

12. Find the comment **Clean up**, and enter the following code to delete the agents when they are no longer required:

Code Copy

```
# Clean up
print("Cleaning up agents:")
agents_client.delete_agent(triage_agent.id)
print("Deleted triage agent.")
agents_client.delete_agent(priority_agent.id)
print("Deleted priority agent.")
agents_client.delete_agent(team_agent.id)
print("Deleted team agent.")
agents_client.delete_agent(effort_agent.id)
print("Deleted effort agent.")
```

13. Use the **CTRL+S** command to save your changes to the code file. You can keep it open (in case you need to edit the code to fix any errors) or use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Sign into Azure and run the app

Now you're ready to run your code and watch your AI agents collaborate.

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

   Code Copy

   ```
   az login
   ```

   You must sign into Azure - even though the cloud shell session is already authenticated.

   > **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. After you have signed in, enter the following command to run the application:

   Code Copy

   ```
   python agent_triage.py
   ```

4. Enter a prompt, such as `Users can't reset their password from the mobile app.`

   After the agents process the prompt, you should see some output similar to the following:

   Code Copy

```
Creating agent thread.
Processing agent thread. Please wait.

MessageRole.USER:
Users can't reset their password from the mobile app.

MessageRole.AGENT:
### Ticket Assessment

- **Priority:** High — This issue blocks users from resetting their passwords, limiting access to their accounts.
- **Assigned Team:** Frontend Team — The problem lies in the mobile app's user interface or functionality.
- **Effort Required:** Medium — Resolving this problem involves identifying the root cause, potentially updating the mobile app functionality, reviewing API/backend integration, and testing to ensure compatibility across Android/iOS platforms.

Cleaning up agents:
Deleted triage agent.
Deleted priority agent.
Deleted team agent.
Deleted effort agent.
```

You can try modifying the prompt using a different ticket scenario to see how the agents collaborate. For example, "Investigate occasional 502 errors from the search endpoint."

Clean up

If you've finished exploring Azure AI Agent Service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Connect AI agents to tools using Model Context Protocol (MCP)

In this exercise, you'll build an agent that connects to a cloud-hosted MCP server. The agent will use AI-powered search to help developers find accurate, real-time answers from Microsoft's official documentation. This is useful for building assistants that support developers with up-to-date guidance on tools like Azure, .NET, and Microsoft 365. The agent will use the provided `microsoft_docs_search` tool to query the documentation and return relevant results.

> **Tip**: The code used in this exercise is based on the Azure AI Agent service MCP support sample repository. Refer to Azure OpenAI demos or visit Connect to Model Context Protocol servers for more details.

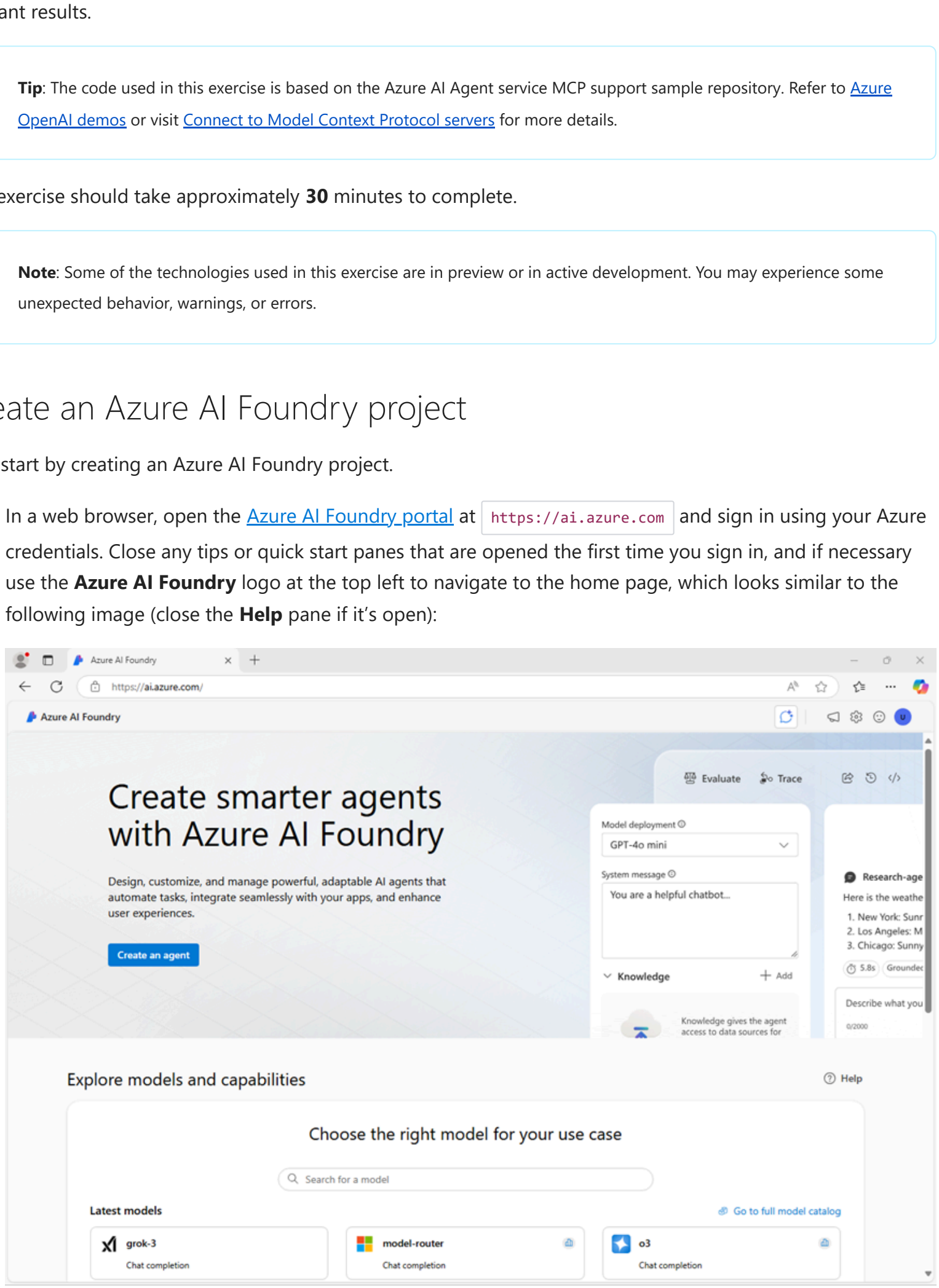
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any of the following supported locations: **

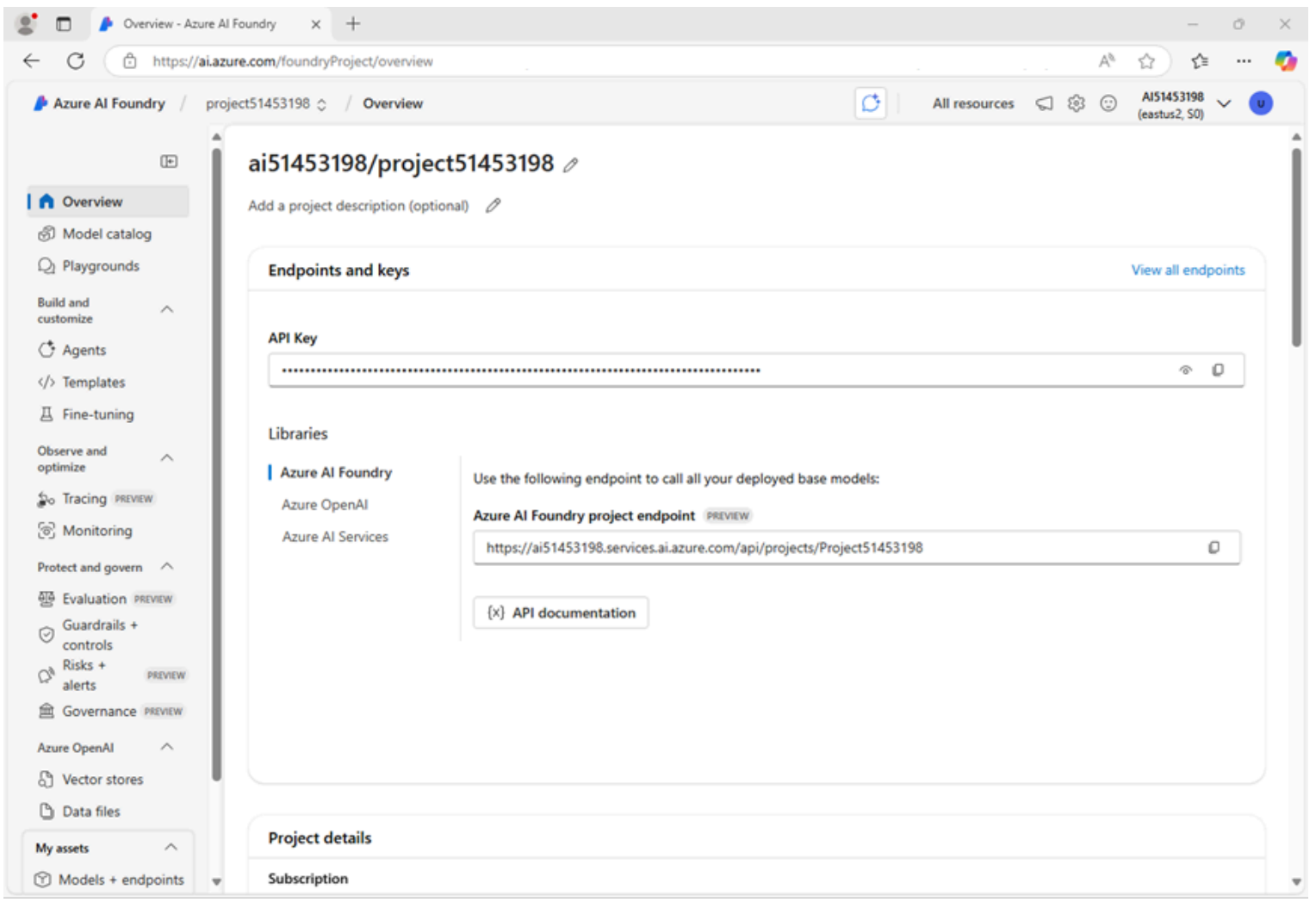
- West US 2
- West US
- Norway East
- Switzerland North
- UAE North
- South India

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project to be created.
6. If prompted, deploy a **gpt-4o** model using either the *Global Standard* or *Standard* deployment option (depending on your quota availability).

> **Note**: If quota is available, a GPT-4o base model may be deployed automatically when creating your Agent and project.

7. When your project is created, the Agents playground will be opened.
8. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



9. Copy the **Azure AI Foundry project endpoint** value. You'll use it to connect to your project in a client application.

Develop an agent that uses MCP function tools

Now that you've created your project in AI Foundry, let's develop an app that integrates an AI agent with an MCP server.

Clone the repo containing the application code

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r ai-agents -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
   ```

   > **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. Enter the following command to change the working directory to the folder containing the code files and list them all.

   Code Copy

   ```
   cd ai-agents/Labfiles/03c-use-agent-tools-with-mcp/Python
   ls -a -l
   ```

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt --pre azure-ai-projects mcp
   ```

   > **Note:** You can ignore any warning or error messages displayed during the library installation.

2. Enter the following command to edit the configuration file that has been provided:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal) and ensure that the MODEL_DEPLOYMENT_NAME variable is set to your model deployment name (which should be *gpt-4o*).

4. After you've replaced the placeholder, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Connect an Azure AI Agent to a remote MCP server

In this task, you'll connect to a remote MCP server, prepare the AI agent, and run a user prompt.

1. Enter the following command to edit the code file that has been provided:

```
code client.py
```

   The file is opened in the code editor.

2. Find the comment **Add references** and add the following code to import the classes:

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.agents import AgentsClient
from azure.ai.agents.models import McpTool, ToolSet, ListSortOrder
```

3. Find the comment **Connect to the agents client** and add the following code to connect to the Azure AI project using the current Azure credentials.

```
# Connect to the agents client
agents_client = AgentsClient(
     endpoint=project_endpoint,
     credential=DefaultAzureCredential(
         exclude_environment_credential=True,
         exclude_managed_identity_credential=True
     )
```

4. Under the comment **Initialize agent MCP tool**, add the following code:

```
# Initialize agent MCP tool
mcp_tool = McpTool(
     server_label=mcp_server_label,
     server_url=mcp_server_url,
)

mcp_tool.set_approval_mode("never")

toolset = ToolSet()
toolset.add(mcp_tool)
```

This code will connect to the Microsft Learn Docs remote MCP server. This is a cloud-hosted service that enables clients to access trusted and up-to-date information directly from Microsoft's official documentation.

5. Under the comment **Create a new agent** and add the following code:

Code Copy

```
# Create a new agent
agent = agents_client.create_agent(
     model=model_deployment,
     name="my-mcp-agent",
     instructions="""
     You have access to an MCP server called `microsoft.docs.mcp` - this tool allows you to
     search through Microsoft's latest official documentation. Use the available MCP tools
     to answer questions and perform tasks."""
)
```

In this code, you provide instructions for the agent and provide it with the MCO tool definitions.

6. Find the comment **Create thread for communication** and add the following code:

Code Copy

```
# Create thread for communication
thread = agents_client.threads.create()
print(f"Created thread, ID: {thread.id}")
```

7. Find the comment **Create a message on the thread** and add the following code:

Code Copy

```
# Create a message on the thread
prompt = input("\nHow can I help?: ")
message = agents_client.messages.create(
     thread_id=thread.id,
     role="user",
     content=prompt,
)
print(f"Created message, ID: {message.id}")
```

8. Find the comment **Set approval mode** and add the following code:

Code Copy

```
# Set approval mode
mcp_tool.set_approval_mode("never")
```

This allows the agent to automatically invoke the MCP tools without requiring user approval. If you want to require approval, you must supply a header value using `mcp_tool.update_headers`.

9. Find the comment **Create and process agent run in thread with MCP tools** and add the following code:

Code Copy

```
# Create and process agent run in thread with MCP tools
run = agents_client.runs.create_and_process(thread_id=thread.id, agent_id=agent.id,
toolset=toolset)
print(f"Created run, ID: {run.id}")
```

The AI Agent automatically invokes the connected MCP tools to process the prompt request. To illustrate this process, the code provided under the comment **Display run steps and tool calls** will output any invoked tools from the MCP server.

10. Save the code file (*CTRL+S*) when you have finished. You can also close the code editor (*CTRL+Q*); though you may want to keep it open in case you need to make any edits to the code you added. In either case, keep the cloud shell command-line pane open.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. After you have signed in, enter the following command to run the application:

Code Copy

```
python client.py
```

4. When prompted, enter a request for technical information such as:

Code Copy

```
Give me the Azure CLI commands to create an Azure Container App with a managed identity.
```

5. Wait for the agent to process your prompt, using the MCP server to find a suitable tool to retrieve the requested information. You should see some output similar to the following:

Code Copy

```
Created agent, ID: <<agent-id>>
MCP Server: mslearn at https://learn.microsoft.com/api/mcp
Created thread, ID: <<thread-id>>
Created message, ID: <<message-id>>
Created run, ID: <<run-id>>
Run completed with status: RunStatus.COMPLETED
Step <<step1-id>> status: completed

Step <<step2-id>> status: completed
MCP Tool calls:
    Tool Call ID: <<tool-call-id>>
    Type: mcp
    Type: microsoft_docs_search



Conversation:
--------------------------------------------------
ASSISTANT: You can use Azure CLI to create an Azure Container App with a managed identity
(either system-assigned or user-assigned). Below are the relevant commands and workflow:

 ---

 ### **1. Create a Resource Group**
 '''azurecli
 az group create --name myResourceGroup --location eastus
 '''




 By following these steps, you can deploy an Azure Container App with either system-assigned
or user-assigned managed identities to integrate seamlessly with other Azure services.
 --------------------------------------------------
 USER: Give me the Azure CLI commands to create an Azure Container App with a managed
identity.
 --------------------------------------------------
 Deleted agent
```

Notice that the agent was able to invoke the MCP tool `microsoft_docs_search` automatically to fulfill the request.

6. You can run the app again (using the command `python client.py`) to ask for different information, In each case, the agent will attempt to find technical documentation by using the MCP tool.

Clean up

Now that you've finished the exercise, you should delete the cloud resources you've created to avoid unnecessary resource usage.

1. Open the Azure portal at `https://portal.azure.com` and view the contents of the resource group where you deployed the hub resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop an Azure AI chat agent with the Microsoft Agent Framework SDK

In this exercise, you'll use Azure AI Agent Service and Microsoft Agent Framework to create an AI agent that processes expense claims.

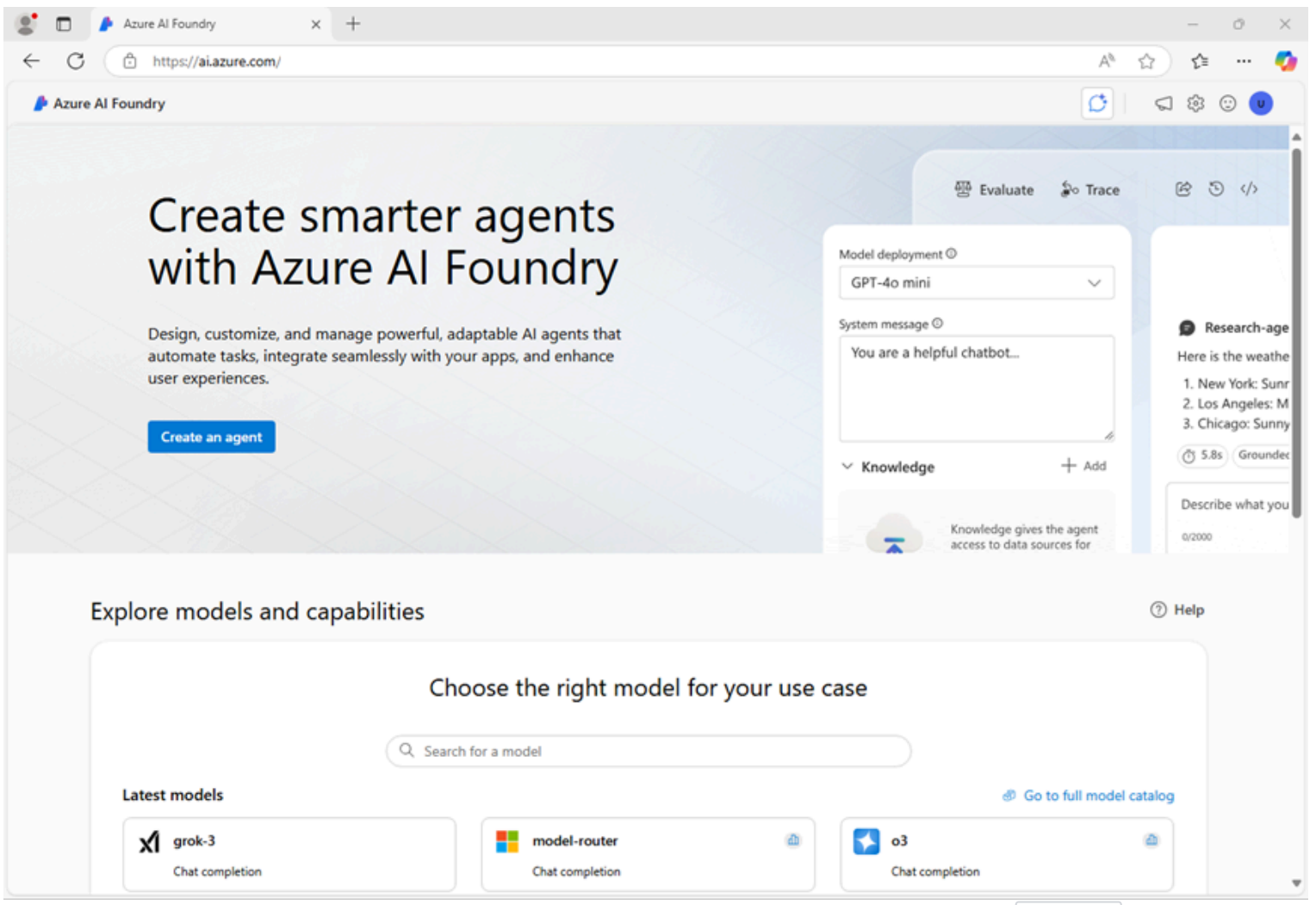
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Deploy a model in an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):

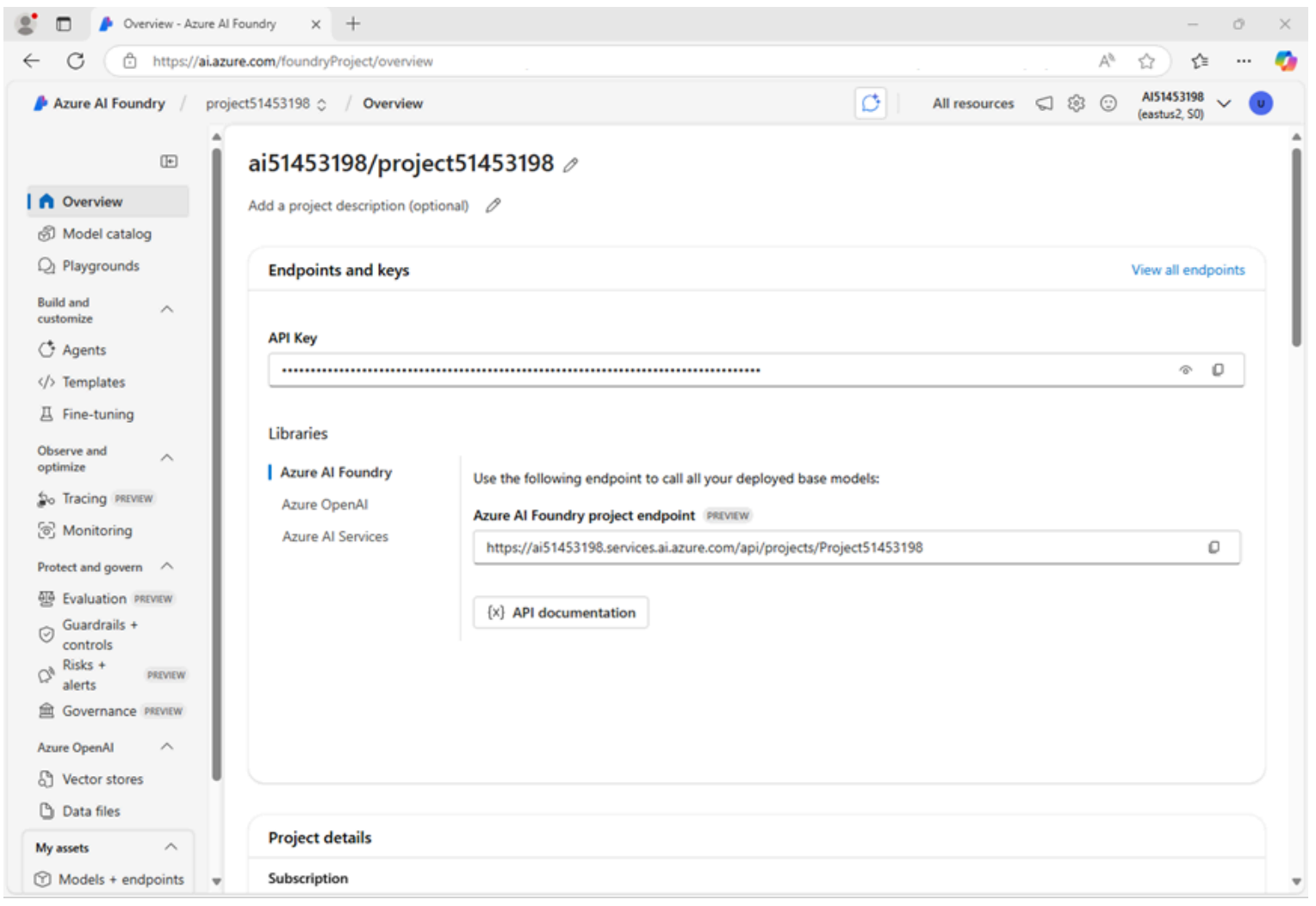


2. On the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any* ***AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

6. Select **Create** and wait for your project, including the gpt-4 model deployment you selected, to be created.
7. When your project is created, the chat playground will be opened automatically.
8. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**. You can confirm this by viewing the deployment in the **Models and endpoints** page (just open that page in the navigation pane on the left).

9. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



Create an agent client app

Now you're ready to create a client app that defines an agent and a custom function. Some code has been provided for you in a GitHub repository.

Prepare the environment

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

Code Copy

```
rm -r ai-agents -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
```

> **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. When the repo has been cloned, enter the following command to change the working directory to the folder containing the code files and list them all.

Code Copy

```
cd ai-agents/Labfiles/04-agent-framework/python
ls -a -l
```

The provided files include application code a file for configuration settings, and a file containing expenses data.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install azure-identity agent-framework
```

2. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

The file is opened in a code editor.

3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal), and the **your_model_deployment** placeholder with the name you assigned to your gpt-4o model deployment.
4. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Write code for an agent app

> **Tip**: As you add code, be sure to maintain the correct indentation. Use the existing comments as a guide, entering the new code at the same level of indentation.

1. Enter the following command to edit the agent code file that has been provided:

Code Copy

```
code agent-framework.py
```

2. Review the code in the file. It contains:
    - Some **import** statements to add references to commonly used namespaces
    - A *main* function that loads a file containing expenses data, asks the user for instructions, and and then calls...
    - A **process_expenses_data** function in which the code to create and use your agent must be added
3. At the top of the file, after the existing **import** statement, find the comment **Add references**, and add the following code to reference the namespaces in the libraries you'll need to implement your agent:

Code Copy

```
# Add references
from agent_framework import AgentThread, ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential
from pydantic import Field
from typing import Annotated
```

4. Near the bottom of the file, find the comment **Create a tool function for the email functionality**, and add the following code to define a function that your agent will use to send email (tools are a way to add custom functionality to agents)

Code Copy

```
# Create a tool function for the email functionality
def send_email(
 to: Annotated[str, Field(description="Who to send the email to")],
 subject: Annotated[str, Field(description="The subject of the email.")],
 body: Annotated[str, Field(description="The text body of the email.")]):
    print("\nTo:", to)
    print("Subject:", subject)
    print(body, "\n")
```

> **Note**: The function *simulates* sending an email by printing it to the console. In a real application, you'd use an SMTP service or similar to actually send the email!

5. Back up above the **send_email** code, in the **process_expenses_data** function, find the comment **Create a chat agent**, and add the following code to create a **ChatAgent** object with the tools and instructions. (Be sure to maintain the indentation level)

Code Copy

```
# Create a chat agent
async with (
    AzureCliCredential() as credential,
    ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        name="expenses_agent",
        instructions="""You are an AI assistant for expense claim submission.
                        When a user submits expenses data and requests an expense claim, use
the plug-in function to send an email to expenses@contoso.com with the subject 'Expense
Claim`and a body that contains itemized expenses with a total.
                        Then confirm to the user that you've done so.""",
        tools=send_email,
    ) as agent,
):
```

Note that the **AzureCliCredential** object will allow your code to authenticate to your Azure account. The **AzureAIAgentClient** object will automatically include the Azure AI Foundry project settings from the .env configuration.

6. Find the comment **Use the agent to process the expenses data**, and add the following code to create a thread for your agent to run on, and then invoke it with a chat message.

   (Be sure to maintain the indentation level):

   Code Copy

   ```
   # Use the agent to process the expenses data
   try:
       # Add the input prompt to a list of messages to be submitted
       prompt_messages = [f"{prompt}: {expenses_data}"]
       # Invoke the agent for the specified thread with the messages
       response = await agent.run(prompt_messages)
       # Display the response
       print(f"\n# Agent:\n{response}")
   except Exception as e:
       # Something went wrong
       print (e)
   ```

7. Review that the completed code for your agent, using the comments to help you understand what each block of code does, and then save your code changes (**CTRL+S**).
8. Keep the code editor open in case you need to correct any typo's in the code, but resize the panes so you can see more of the command line console.

Sign into Azure and run the app

1. In the cloud shell command-line pane beneath the code editor, enter the following command to sign into Azure.

   Code Copy

   ```
   az login
   ```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. After you have signed in, enter the following command to run the application:

```
python agent-framework.py
```

The application runs using the credentials for your authenticated Azure session to connect to your project and create and run the agent.

4. When asked what to do with the expenses data, enter the following prompt:

```
Submit an expense claim
```

5. When the application has finished, review the output. The agent should have composed an email for an expenses claim based on the data that was provided.

> **Tip**: If the app fails because the rate limit is exceeded. Wait a few seconds and try again. If there is insufficient quota available in your subscription, the model may not be able to respond.

Summary

In this exercise, you used the Microsoft Agent Framework SDK to create an agent with a custom tool.

Clean up

If you've finished exploring Azure AI Agent Service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop a multi-agent solution

In this exercise, you'll practice using the sequential orchestration pattern in the Microsoft Agent Framework SDK. You'll create a simple pipeline of three agents that work together to process customer feedback and suggest next steps. You'll create the following agents:

- The Summarizer agent will condense raw feedback into a short, neutral sentence.
- The Classifier agent will categorize the feedback as Positive, Negative, or a Feature request.
- Finally, the Recommended Action agent will recommend an appropriate follow-up step.

You'll learn how to use the Microsoft Agent Framework SDK to break down a problem, route it through the right agents, and produce actionable results. Let's get started!

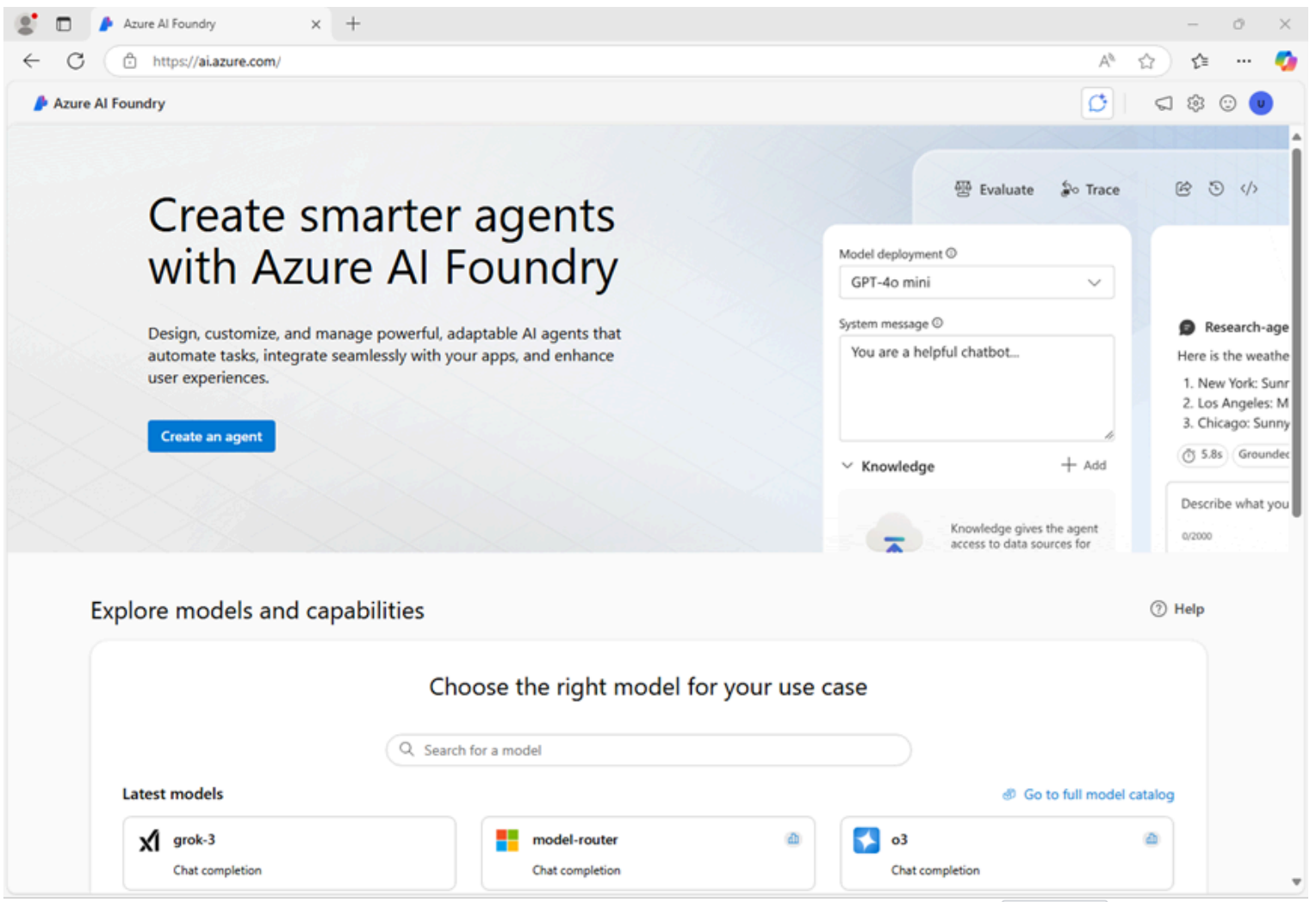
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Deploy a model in an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):

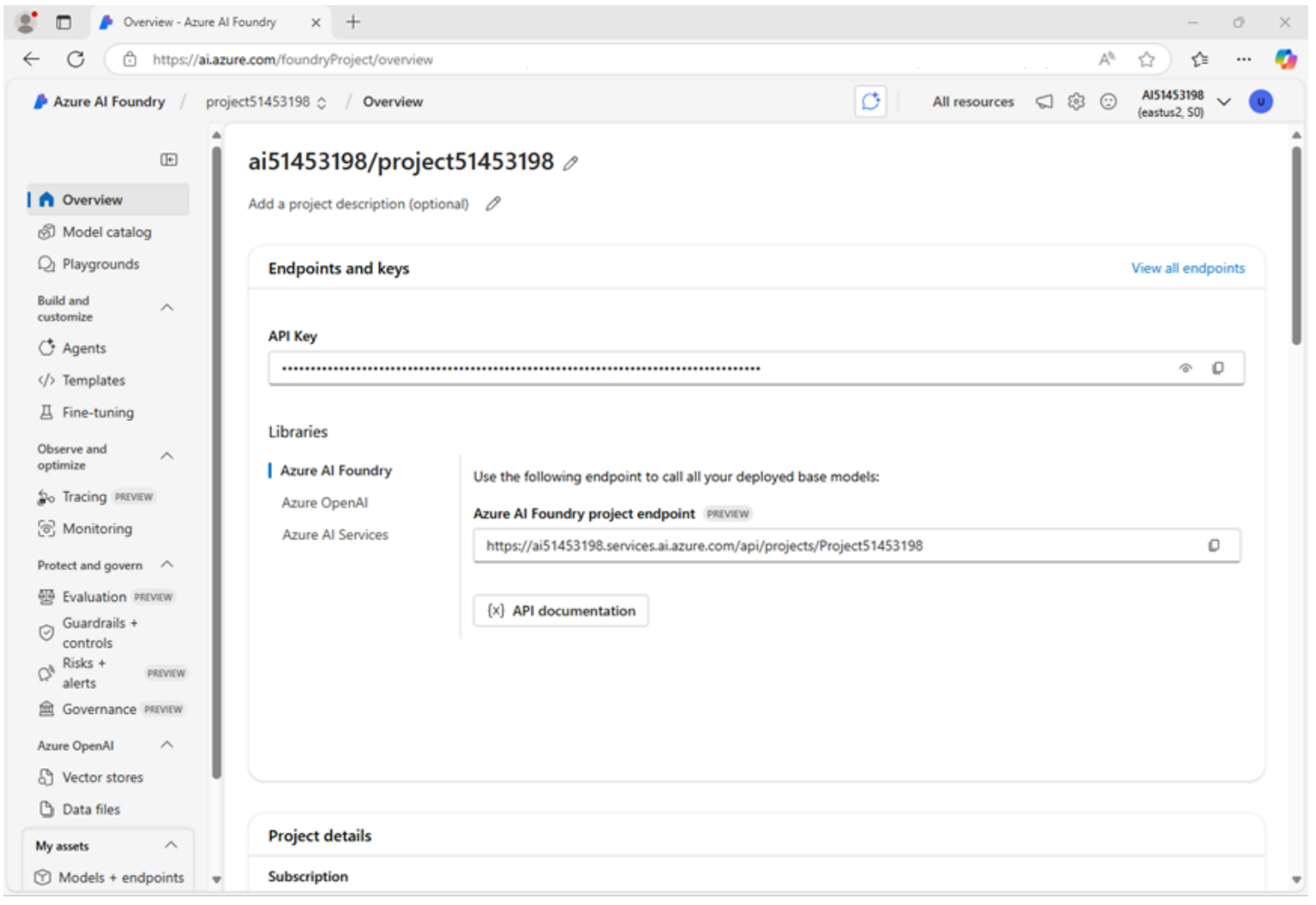


2. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*

- **Region**: *Select any **AI Foundry recommended****

> ! * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

6. Select **Create** and wait for your project, including the gpt-4 model deployment you selected, to be created.
7. When your project is created, the chat playground will be opened automatically.
8. In the navigation pane on the left, select **Models and endpoints** and select your **gpt-4o** deployment.
9. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**. You can confirm this by viewing the deployment in the **Models and endpoints** page (just open that page in the navigation pane on the left).
10. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



Create an AI Agent client app

Now you're ready to create a client app that defines an agent and a custom function. Some code is provided for you in a GitHub repository.

Prepare the environment

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r ai-agents -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
   ```

   > **Tip**: As you enter commands into the cloud shell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. When the repo has been cloned, enter the following command to change the working directory to the folder containing the code files and list them all.

   Code Copy

   ```
   cd ai-agents/Labfiles/05-agent-orchestration/Python
   ls -a -l
   ```

   The provided files include application code and a file for configuration settings.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install azure-identity agent-framework
   ```

2. Enter the following command to edit the configuration file that is provided:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

3. In the code file, replace the **your_openai_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal). Replace the **your_model_deployment** placeholder with the name you assigned to your gpt-4o model deployment.

4. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Create AI agents

Now you're ready to create the agents for your multi-agent solution! Let's get started!

1. Enter the following command to edit the **agents.py** file:

Code Copy

```
code agents.py
```

2. At the top of the file under the comment **Add references**, and add the following code to reference the namespaces in the libraries you'll need to implement your agent:

Code Copy

```
# Add references
import asyncio
from typing import cast
from agent_framework import ChatMessage, Role, SequentialBuilder, WorkflowOutputEvent
from agent_framework.azure import AzureAIAgentClient
from azure.identity import AzureCliCredential
```

3. In the **main** function, take a moment to review the agent instructions. These instructions define the behavior of each agent in the orchestration.

4. Add the following code under the comment **Create the chat client**:

Code Copy

```
# Create the chat client
credential = AzureCliCredential()
async with (
    AzureAIAgentClient(async_credential=credential) as chat_client,
):
```

Note that the **AzureCliCredential** object will allow your code to authenticate to your Azure account. The **AzureAIAgentClient** object will automatically include the Azure AI Foundry project settings from the .env configuration.

5. Add the following code under the comment **Create agents**:

(Be sure to maintain the indentation level)

Code Copy

```
# Create agents
summarizer = chat_client.create_agent(
    instructions=summarizer_instructions,
    name="summarizer",
)

classifier = chat_client.create_agent(
    instructions=classifier_instructions,
    name="classifier",
)

action = chat_client.create_agent(
    instructions=action_instructions,
    name="action",
)
```

Create a sequential orchestration

1. In the **main** function, find the comment **Initialize the current feedback** and add the following code:

   (Be sure to maintain the indentation level)

   Code Copy

   ```
   # Initialize the current feedback
   feedback="""
   I use the dashboard every day to monitor metrics, and it works well overall.
   But when I'm working late at night, the bright screen is really harsh on my eyes.
   If you added a dark mode option, it would make the experience much more comfortable.
   """
   ```

2. Under the comment **Build a sequential orchestration**, add the following code to define a sequential orchestration with the agents you defined:

   Code Copy

   ```
   # Build sequential orchestration
   workflow = SequentialBuilder().participants([summarizer, classifier, action]).build()
   ```

   The agents will process the feedback in the order they are added to the orchestration.

3. Add the following code under the comment **Run and collect outputs**:

   Code Copy

   ```
   # Run and collect outputs
   outputs: list[list[ChatMessage]] = []
   async for event in workflow.run_stream(f"Customer feedback: {feedback}"):
       if isinstance(event, WorkflowOutputEvent):
           outputs.append(cast(list[ChatMessage], event.data))
   ```

   This code runs the orchestration and collects the output from each of the participating agents.

4. Add the following code under the comment **Display outputs**:

   Code Copy

   ```
   # Display outputs
   if outputs:
       for i, msg in enumerate(outputs[-1], start=1):
           name = msg.author_name or ("assistant" if msg.role == Role.ASSISTANT else "user")
           print(f"{'-' * 60}\n{i:02d} [{name}]\n{msg.text}")
   ```

   This code formats and displays the messages from the workflow outputs you collected from the orchestration.

5. Use the **CTRL+S** command to save your changes to the code file. You can keep it open (in case you need to edit the code to fix any errors) or use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Sign into Azure and run the app

Now you're ready to run your code and watch your AI agents collaborate.

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. After you have signed in, enter the following command to run the application:

Code Copy

```
python agents.py
```

You should see some output similar to the following:

Code Copy

```
---------------------------------------------------------
01 [user]
Customer feedback:
    I use the dashboard every day to monitor metrics, and it works well overall.
    But when I'm working late at night, the bright screen is really harsh on my eyes.
    If you added a dark mode option, it would make the experience much more comfortable.

---------------------------------------------------------
02 [summarizer]
User requests a dark mode for better nighttime usability.
---------------------------------------------------------
03 [classifier]
Feature request
---------------------------------------------------------
04 [action]
  Log as enhancement request for product backlog.
```

4. Optionally, you can try running the code using different feedback inputs, such as:

Code Copy

```
 I use the dashboard every day to monitor metrics, and it works well overall. But when I'm working late at night, the bright screen is really harsh on my eyes. If you added a dark mode option, it would make the experience much more comfortable.
```

Code Copy

```
 I reached out to your customer support yesterday because I couldn't access my account. The representative responded almost immediately, was polite and professional, and fixed the issue within minutes. Honestly, it was one of the best support experiences I've ever had.
```

Summary

In this exercise, you practiced sequential orchestration with the Microsoft Agent Framework SDK, combining multiple agents into a single, streamlined workflow. Great work!

Clean up

If you've finished exploring Azure AI Agent Service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Connect to remote agents with A2A protocol

In this exercise, you'll use Azure AI Agent Service with the A2A protocol to create simple remote agents that interact with one another. These agents will assist technical writers with preparing their developer blog posts. A title agent will generate a headline, and an outline agent will use the title to develop a concise outline for the article. Let's get started

> **Tip**: The code used in this exercise is based on the for Azure AI Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Azure AI Foundry SDK client libraries for details.

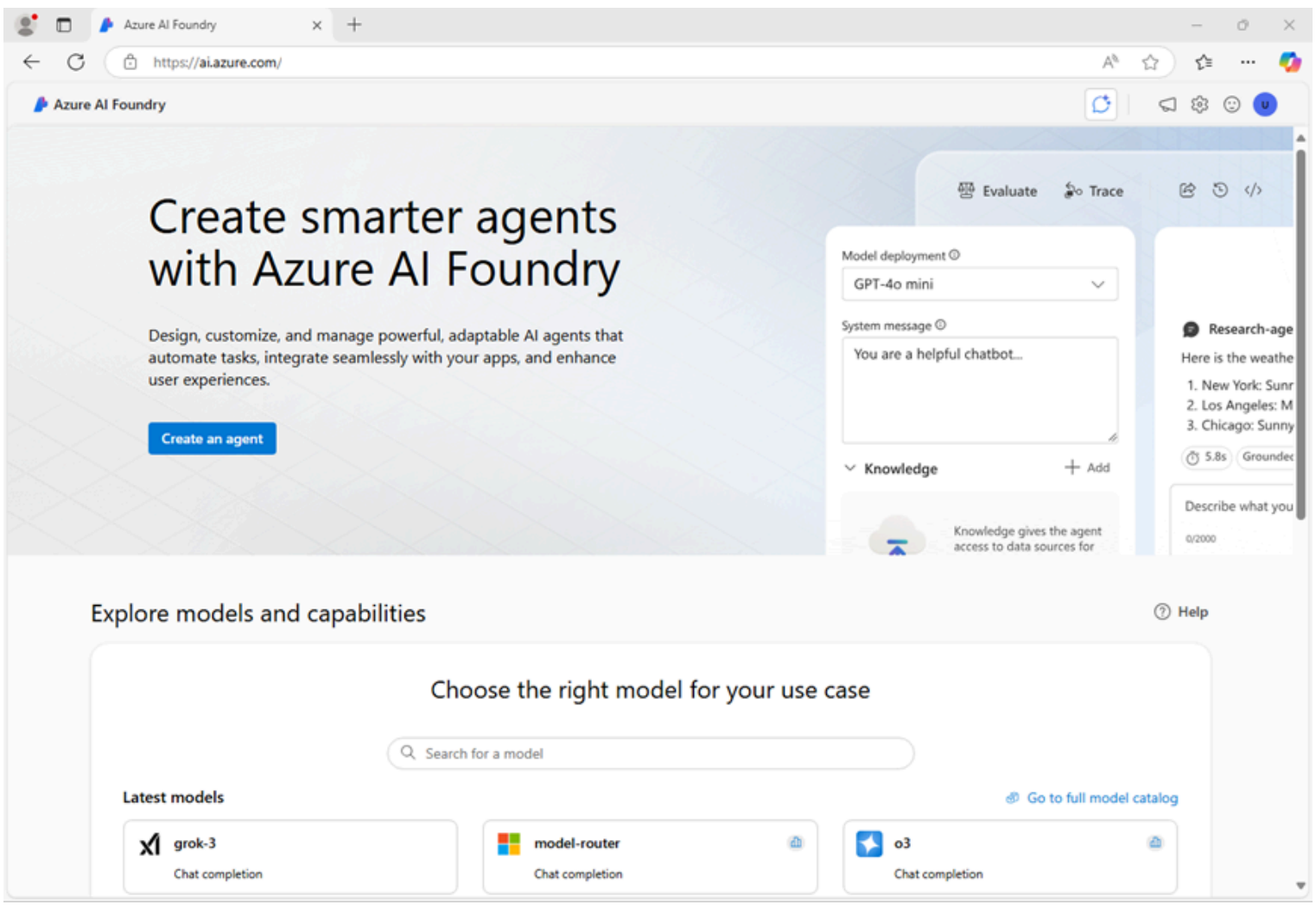
This exercise should take approximately **30** minutes to complete.

> **Note**: Some of the technologies used in this exercise are in preview or in active development. You may experience some unexpected behavior, warnings, or errors.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



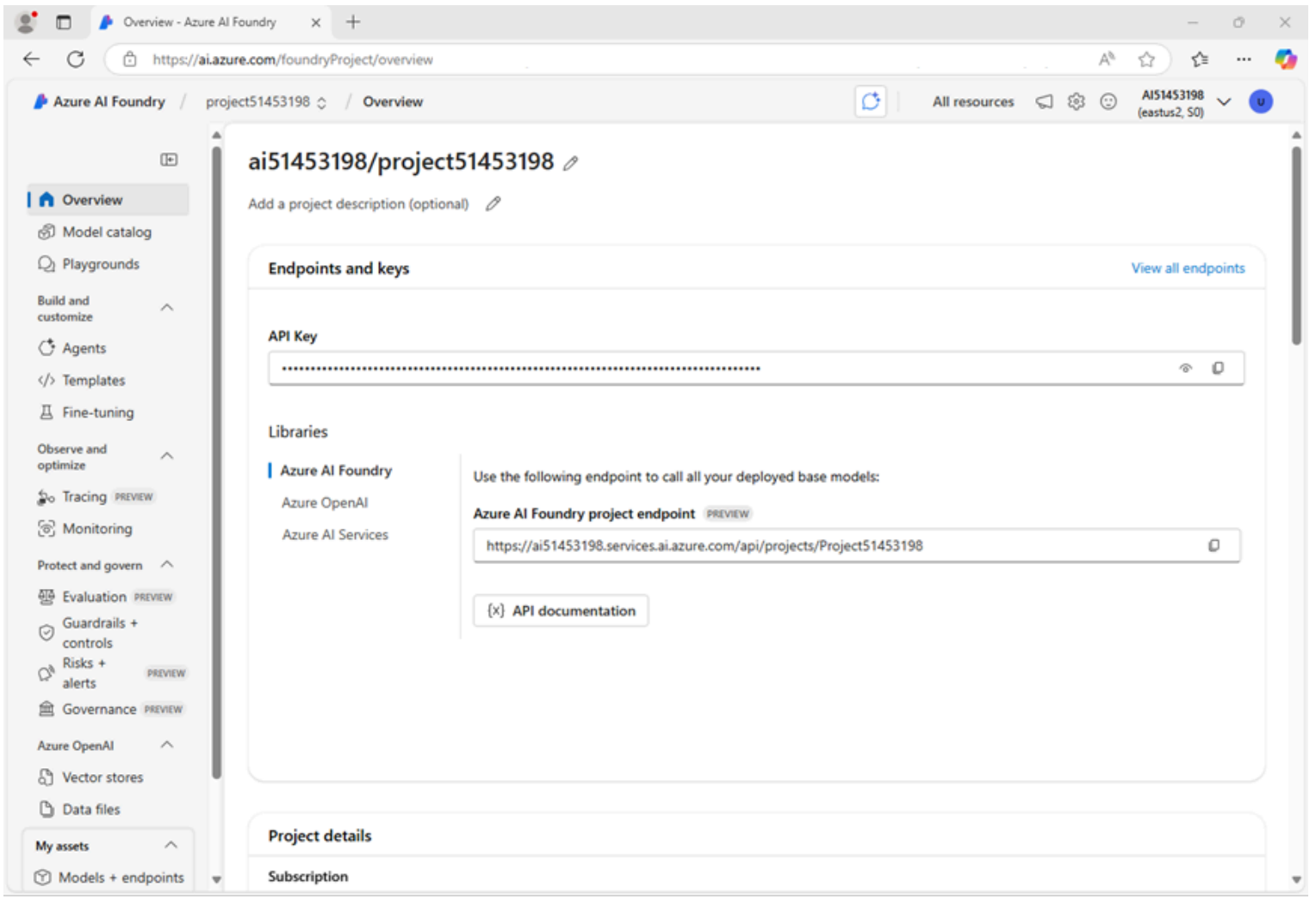
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project to be created.
6. If prompted, deploy a **gpt-4o** model using either the *Global Standard* or *Standard* deployment option (depending on your quota availability).

> **Note**: If quota is available, a GPT-4o base model may be deployed automatically when creating your Agent and project.

7. When your project is created, the Agents playground will be opened.
8. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:



9. Copy the **Azure AI Foundry project endpoint** values to a notepad, as you'll use them to connect to your project in a client application.

Create an A2A application

Now you're ready to create a client app that uses an agent. Some code has been provided for you in a GitHub repository.

Clone the repo containing the application code

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r ai-agents -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-agents ai-agents
   ```

   > **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer and the cursor on the current line may be obscured. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. Enter the following command to change the working directory to the folder containing the code files and list them all.

   Code Copy

   ```
   cd ai-agents/Labfiles/06-build-remote-agents-with-a2a/python
   ls -a -l
   ```

   The provided files include:

   Code Copy

   ```
   python
   ├── outline_agent/
   │   ├── agent.py
   │   ├── agent_executor.py
   │   └── server.py
   ├── routing_agent/
   │   ├── agent.py
   │   └── server.py
   ├── title_agent/
   │   ├── agent.py
   │   ├── agent_executor.py
   │   └── server.py
   ├── client.py
   └── run_all.py
   ```

   Each agent folder contains the Azure AI agent code and a server to host the agent. The **routing agen**t is responsible for discovering and communicating with the **title** and **outline** agents. The **client** allows users to submit prompts to the routing agent. `run_all.py` launches all the servers and runs the client.

Configure the application settings

1. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-projects a2a-sdk
```

2. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

The file is opened in a code editor.

3. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal) and ensure that the MODEL_DEPLOYMENT_NAME variable is set to your model deployment name (which should be *gpt-4o*).
4. After you've replaced the placeholder, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Create a discoverable agent

In this task, you create the title agent that helps writers create trendy headlines for their articles. You also define the agent's skills and card required by the A2A protocol to make the agent discoverable.

1. Navigate to the `title_agent` directory:

Code Copy

```
cd title_agent
```

> **Tip**: As you add code, be sure to maintain the correct indentation. Use the comment indentation levels as a guide.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code agent.py
```

2. Find the comment **Create the agents client** and add the following code to connect to the Azure AI project:

> **Tip**: Be careful to maintain the correct indentation level.

Code Copy

```
# Create the agents client
self.client = AgentsClient(
    endpoint=os.environ['PROJECT_ENDPOINT'],
    credential=DefaultAzureCredential(
        exclude_environment_credential=True,
        exclude_managed_identity_credential=True
    )
)
```

3. Find the comment **Create the title agent** and add the following code to create the agent:

```
# Create the title agent
self.agent = self.client.create_agent(
    model=os.environ['MODEL_DEPLOYMENT_NAME'],
    name='title-agent',
    instructions="""
    You are a helpful writing assistant.
    Given a topic the user wants to write about, suggest a single clear and catchy blog post title.
    """,
)
```

4. Find the comment **Create a thread for the chat session** and add the following code to create the chat thread:

```
# Create a thread for the chat session
thread = self.client.threads.create()
```

5. Locate the comment **Send user message** and add this code to submit the user's prompt:

```
# Send user message
self.client.messages.create(thread_id=thread.id, role=MessageRole.USER,
content=user_message)
```

6. Under the comment **Create and run the agent**, add the following code to initiate the agent's response generation:

```
# Create and run the agent
run = self.client.runs.create_and_process(thread_id=thread.id, agent_id=self.agent.id)
```

The code provided in the rest of the file will process and return the agent's response.

7. Save the code file (*CTRL+S*). Now you're ready to share the agent's skills and card with the A2A protocol.

8. Enter the following command to edit the title agent's `server.py` file

```
code server.py
```

9. Find the comment **Define agent skills** and add the following code to specify the agent's functionality:

```
# Define agent skills
skills = [
    AgentSkill(
        id='generate_blog_title',
        name='Generate Blog Title',
        description='Generates a blog title based on a topic',
        tags=['title'],
        examples=[
            'Can you give me a title for this article?',
        ],
    ),
]
```

10. Find the comment **Create agent card** and add this code to define the metadata that makes the agent discoverable:

Code Copy

```
# Create agent card
agent_card = AgentCard(
    name='AI Foundry Title Agent',
    description='An intelligent title generator agent powered by Azure AI Foundry. '
    'I can help you generate catchy titles for your articles.',
    url=f'http://{host}:{port}/',
    version='1.0.0',
    default_input_modes=['text'],
    default_output_modes=['text'],
    capabilities=AgentCapabilities(),
    skills=skills,
)
```

11. Locate the comment **Create agent executor** and add the following code to initialize the agent executor using the agent card:

Code Copy

```
# Create agent executor
agent_executor = create_foundry_agent_executor(agent_card)
```

The agent executor will act as a wrapper for the title agent you created.

12. Find the comment **Create request handler** and add the following to handle incoming requests using the executor:

Code Copy

```
# Create request handler
request_handler = DefaultRequestHandler(
    agent_executor=agent_executor, task_store=InMemoryTaskStore()
)
```

13. Under the comment **Create A2A application**, add this code to create the A2A-compatible application instance:

Code Copy

```
# Create A2A application
a2a_app = A2AStarletteApplication(
    agent_card=agent_card, http_handler=request_handler
)
```

This code creates an A2A server that will share the title agent's information and handle incoming requests for this agent using the title agent executor.

14. Save the code file (*CTRL+S*) when you have finished.

Enable messages between the agents

In this task, you use the A2A protocol to enable the routing agent to send messages to the other agents. You also allow the title agent to receive messages by implementing the agent executor class.

1. Navigate to the `routing_agent` directory:

Code Copy

```
cd ../routing_agent
```

2. Enter the following command to edit the code file that has been provided:

Code Copy

```
code agent.py
```

The routing agent acts as an orchestrator that handles user messages and determines which remote agent should process the request.

When a user message is received, the routing agent:

- Starts a conversation thread.
- Uses the `create_and_process` method to evaluate the best-matching agent for the user's message.
- The message is routed to the appropriate agent over HTTP using the `send_message` function.
- The remote agent processes the message and returns a response.

The routing agent finally captures the response and returns it to the user through the thread.

Notice that the `send_message` method is async and must be awaited for the agent run to complete successfully.

3. Add the following code under the comment **Retrieve the remote agent's A2A client using the agent name**:

Code Copy

```
# Retrieve the remote agent's A2A client using the agent name
client = self.remote_agent_connections[agent_name]
```

4. Locate the comment **Construct the payload to send to the remote agent** and add the following code:

Code Copy

```
# Construct the payload to send to the remote agent
payload: dict[str, Any] = {
    'message': {
        'role': 'user',
        'parts': [{'kind': 'text', 'text': task}],
        'messageId': message_id,
    },
}
```

5. Find the comment **Wrap the payload in a SendMessageRequest object** and add the following code:

Code Copy

```
# Wrap the payload in a SendMessageRequest object
message_request = SendMessageRequest(id=message_id,
params=MessageSendParams.model_validate(payload))
```

6. Add the following code under the comment **Send the message to the remote agent client and await the response**:

Code Copy

```
# Send the message to the remote agent client and await the response
send_response: SendMessageResponse = await
client.send_message(message_request=message_request)
```

7. Save the code file (*CTRL+S*) when you have finished. Now the routing agent is able to discover and send messages to the title agent. Let's create the agent executor code to handle those incoming messages from the routing agent.

8. Navigate to the `title_agent` directory:

Code Copy

```
cd ../title_agent
```

9. Enter the following command to edit the code file that has been provided:

Code Copy

```
code agent_executor.py
```

The `AgentExecutor` class implemenation must contain the methods `execute` and `cancel`. The cancel method has been provided for you. The `execute` method includes a `TaskUpdater` object that manages events and signals to the caller when the task is complete. Let's add the logic for task execution.

10. In the `execute` method, add the following code under the comment **Process the request**:

Code Copy

```
# Process the request
await self._process_request(context.message.parts, context.context_id, updater)
```

11. In the `_process_request` method, add the following code under the comment **Get the title agent**:

Code Copy

```
# Get the title agent
agent = await self._get_or_create_agent()
```

12. Add the following code under the comment **Update the task status**:

Code Copy

```
# Update the task status
await task_updater.update_status(
    TaskState.working,
    message=new_agent_text_message('Title Agent is processing your request...',
context_id=context_id),
)
```

13. Find the comment **Run the agent conversation** and add the following code:

Code Copy

```
# Run the agent conversation
responses = await agent.run_conversation(user_message)
```

14. Find the comment **Update the task with the responses** and add the following code:

Code Copy

```
# Update the task with the responses
for response in responses:
    await task_updater.update_status(
        TaskState.working,
        message=new_agent_text_message(response, context_id=context_id),
    )
```

15. Find the comment **Mark the task as complete** and add the following code:

Code Copy

```
# Mark the task as complete
final_message = responses[-1] if responses else 'Task completed.'
await task_updater.complete(
    message=new_agent_text_message(final_message, context_id=context_id)
)
```

Now your title agent has been wrapped with an agent executor that the A2A protocol will use to handle messages. Great work!

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. After you have signed in, enter the following command to run the application:

```
cd ..
python run_all.py
```

   The application runs using the credentials for your authenticated Azure session to connect to your project and create and run the agent. You should see some output from each server as it starts.

4. Wait until the prompt for input appears, then enter a prompt such as:

```
Create a title and outline for an article about React programming.
```

   After a few moments, you should see a response from the agent with the results.
5. Enter `quit` to exit the program and stop the servers.

Summary

In this exercise, you used the Azure AI Agent Service SDK and the A2A Python SDK to create a remote multi-agent solution. You created a discoverable A2A-compatible agent and set up a routing agent to access the agent's skills. You also implemented an agent executor to process incoming A2A messages and manage tasks. Great work!

Clean up

If you've finished exploring Azure AI Agent Service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Analyze Text

Azure AI Language supports analysis of text, including language detection, sentiment analysis, key phrase extraction, and entity recognition.

For example, suppose a travel agency wants to process hotel reviews that have been submitted to the company's web site. By using the Azure AI Language, they can determine the language each review is written in, the sentiment (positive, neutral, or negative) of the reviews, key phrases that might indicate the main topics discussed in the review, and named entities, such as places, landmarks, or people mentioned in the reviews. In this exercise, you'll use the Azure AI Language Python SDK for text analytics to implement a simple hotel review application based on this example.

While this exercise is based on Python, you can develop text analytics applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **30** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource in your Azure subscription.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select **Continue to create your resource**.
5. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Responsible AI Notice**: Agree.
6. Select **Review + create**, then select **Create** to provision the resource.
7. Wait for deployment to complete, and then go to the deployed resource.
8. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Clone the repository for this course

You'll develop your code using Cloud Shell from the Azure Portal. The code files for your app have been provided in a GitHub repo.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/01-analyze-text/Python/text-analysis
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **text-analysis** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**text-analysis.py**). The text your application will analyze is in the **reviews** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-textanalytics==5.3.0
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal)
5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to connect to your Azure AI Language resource

1. Enter the following command to edit the application code file:

```
code text-analysis.py
```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a client for the Text Analysis API:

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

```
python text-analysis.py
```

6. Observe the output as the code should run without error, displaying the contents of each review text file in the **reviews** folder. The application successfully creates a client for the Text Analytics API but doesn't make use of it. We'll fix that in the next section.

Add code to detect language

Now that you have created a client for the API, let's use it to detect the language in which each review is written.

1. In the code editor, find the comment **Get language**. Then add the code necessary to detect the language in each review document:

```
# Get language
detectedLanguage = ai_client.detect_language(documents=[text])[0]
print('\nLanguage: {}'.format(detectedLanguage.primary_language.name))
```

> **Note**: *In this example, each review is analyzed individually, resulting in a separate call to the service for each file. An alternative approach is to create a collection of documents and pass them to the service in a single call. In both approaches, the response from the service consists of a collection of documents; which is why in the Python code above, the index of the first (and only) document in the response ([0]) is specified.*

2. Save your changes. Then re-run the program.
3. Observe the output, noting that this time the language for each review is identified.

Add code to evaluate sentiment

Sentiment analysis is a commonly used technique to classify text as *positive* or *negative* (or possible *neutral* or *mixed*). It's commonly used to analyze social media posts, product reviews, and other items where the sentiment of the text may provide useful insights.

1. In the code editor, find the comment **Get sentiment**. Then add the code necessary to detect the sentiment of each review document:

Code Copy

```
# Get sentiment
sentimentAnalysis = ai_client.analyze_sentiment(documents=[text])[0]
print("\nSentiment: {}".format(sentimentAnalysis.sentiment))
```

2. Save your changes. Then close the code editor and re-run the program.
3. Observe the output, noting that the sentiment of the reviews is detected.

Add code to identify key phrases

It can be useful to identify key phrases in a body of text to help determine the main topics that it discusses.

1. In the code editor, find the comment **Get key phrases**. Then add the code necessary to detect the key phrases in each review document:

Code Copy

```
# Get key phrases
phrases = ai_client.extract_key_phrases(documents=[text])[0].key_phrases
if len(phrases) > 0:
     print("\nKey Phrases:")
     for phrase in phrases:
          print('\t{}'.format(phrase))
```

2. Save your changes and re-run the program.
3. Observe the output, noting that each document contains key phrases that give some insights into what the review is about.

Add code to extract entities

Often, documents or other bodies of text mention people, places, time periods, or other entities. The text Analytics API can detect multiple categories (and subcategories) of entity in your text.

1. In the code editor, find the comment **Get entities**. Then, add the code necessary to identify entities that are mentioned in each review:

Code Copy

```
# Get entities
entities = ai_client.recognize_entities(documents=[text])[0].entities
if len(entities) > 0:
     print("\nEntities")
     for entity in entities:
         print('\t{} ({})'.format(entity.text, entity.category))
```

2. Save your changes and re-run the program.
3. Observe the output, noting the entities that have been detected in the text.

Add code to extract linked entities

In addition to categorized entities, the Text Analytics API can detect entities for which there are known links to data sources, such as Wikipedia.

1. In the code editor, find the comment **Get linked entities**. Then, add the code necessary to identify linked entities that are mentioned in each review:

Code Copy

```
# Get linked entities
entities = ai_client.recognize_linked_entities(documents=[text])[0].entities
if len(entities) > 0:
     print("\nLinks")
     for linked_entity in entities:
         print('\t{} ({})'.format(linked_entity.name, linked_entity.url))
```

2. Save your changes and re-run the program.
3. Observe the output, noting the linked entities that are identified.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

For more information about using **Azure AI Language**, see the documentation.

Create a Question Answering Solution

One of the most common conversational scenarios is providing support through a knowledge base of frequently asked questions (FAQs). Many organizations publish FAQs as documents or web pages, which works well for a small set of question and answer pairs, but large documents can be difficult and time-consuming to search.

Azure AI Language includes a *question answering* capability that enables you to create a knowledge base of question and answer pairs that can be queried using natural language input, and is most commonly used as a resource that a bot can use to look up answers to questions submitted by users. In this exercise, you'll use the Azure AI Language Python SDK for text analytics to implement a simple question answering application.

While this exercise is based on Python, you can develop question answering applications using multiple language-specific SDKs; including:

- Azure AI Language Service Question Answering client library for Python
- Azure AI Language Service Question Answering client library for .NET

This exercise takes approximately **20** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, to create and host a knowledge base for question answering, you need to enable the **Question Answering** feature.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select the **Custom question answering** block. Then select **Continue to create your resource**. You will need to enter the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Choose or create a resource group*.
    - **Region**: *Choose any available location*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Azure Search region**: *Choose a location in the same global region as your Language resource*
    - **Azure Search pricing tier**: Free (F) (*If this tier is not available, select Basic (B)*)
    - **Responsible AI Notice**: *Agree*
5. Select **Create + review**, then select **Create**.

> **NOTE** Custom Question Answering uses Azure Search to index and query the knowledge base of questions and answers.

6. Wait for deployment to complete, and then go to the deployed resource.
7. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Create a question answering project

To create a knowledge base for question answering in your Azure AI Language resource, you can use the Language Studio portal to create a question answering project. In this case, you'll create a knowledge base containing questions and answers about Microsoft Learn.

1. In a new browser tab, go to the Language Studio portal at https://language.cognitive.azure.com/ and sign in using the Microsoft account associated with your Azure subscription.
2. If you're prompted to choose a Language resource, select the following settings:
    - **Azure Directory**: The Azure directory containing your subscription.
    - **Azure subscription**: Your Azure subscription.
    - **Resource type**: Language
    - **Resource name**: The Azure AI Language resource you created previously.

    If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

    a. On the bar at the top if the page, select the **Settings (⚙)** button.
    b. On the **Settings** page, view the **Resources** tab.
    c. Select the language resource you just created, and click **Switch resource**.
    d. At the top of the page, click **Language Studio** to return to the Language Studio home page.
3. At the top of the portal, in the **Create new** menu, select **Custom question answering**.
4. In the *__Create a project__ wizard, on the **Choose language setting** page, select the option to **Select the language for all projects**, and select **English** as the language. Then select **Next**.
5. On the **Enter basic information** page, enter the following details:
    - **Name** `LearnFAQ`
    - **Description**: `FAQ for Microsoft Learn`
    - **Default answer when no answer is returned**: `Sorry, I don't understand the question`
6. Select **Next**.
7. On the **Review and finish** page, select **Create project**.

Add sources to the knowledge base

You can create a knowledge base from scratch, but it's common to start by importing questions and answers from an existing FAQ page or document. In this case, you'll import data from an existing FAQ web page for Microsoft Learn, and you'll also import some pre-defined "chit chat" questions and answers to support common conversational exchanges.

1. On the **Manage sources** page for your question answering project, in the **+ Add source** list, select **URLs**. Then in the **Add URLs** dialog box, select **+ Add url** and set the following name and URL before you select **Add all** to add it to the knowledge base:
    - **Name**: `Learn FAQ Page`
    - **URL**: `https://learn.microsoft.com/en-us/training/support/faq?pivots=general`
2. On the **Manage sources** page for your question answering project, in the **+ Add source** list, select **Chitchat**. The in the **Add chit chat** dialog box, select **Friendly** and select **Add chit chat**.

Edit the knowledge base

Your knowledge base has been populated with question and answer pairs from the Microsoft Learn FAQ, supplemented with a set of conversational *chit-chat* question and answer pairs. You can extend the knowledge base by adding additional question and answer pairs.

1. In your **LearnFAQ** project in Language Studio, select the **Edit knowledge base** page to see the existing question and answer pairs (if some tips are displayed, read them and choose **Got it** to dismiss them, or select **Skip all**)

2. In the knowledge base, on the **Question answer pairs** tab, select **+**, and create a new question answer pair with the following settings:

   - **Source**: `https://learn.microsoft.com/en-us/training/support/faq?pivots=general`
   - **Question**: `What are the different types of modules on Microsoft Learn?`
   - **Answer**:

     ```
     Microsoft Learn offers various types of training modules, including role-based learning paths, product-specific modules, and hands-on labs. Each module contains units with lessons and knowledge checks to help you learn at your own pace.
     ```

3. Select **Done**.

4. In the page for the **What are the different types of modules on Microsoft Learn?** question that is created, expand **Alternate questions**. Then add the alternate question `How are training modules organized?`.

   In some cases, it makes sense to enable the user to follow up on an answer by creating a *multi-turn* conversation that enables the user to iteratively refine the question to get to the answer they need.

5. Under the answer you entered for the module types question, expand **Follow-up prompts** and add the following follow-up prompt:

   - **Text displayed in the prompt to the user**: `Learn more about training`.
   - Select the **Create link to new pair** tab, and enter this text:

     ```
     You can explore modules and learning paths on the [Microsoft Learn training page](https://learn.microsoft.com/training/).
     ```

   - Select **Show in contextual flow only**. This option ensures that the answer is only ever returned in the context of a follow-up question from the original module types question.

6. Select **Add prompt**.

Train and test the knowledge base

Now that you have a knowledge base, you can test it in Language Studio.

1. Save the changes to your knowledge base by selecting the **Save** button under the **Question answer pairs** tab on the left.
2. After the changes have been saved, select the **Test** button to open the test pane.
3. In the test pane, at the top, deselect **Include short answer response** (if not already unselected). Then at the bottom enter the message `Hello`. A suitable response should be returned.
4. In the test pane, at the bottom enter the message `What is Microsoft Learn?`. An appropriate response from the FAQ should be returned.
5. Enter the message `Thanks!` An appropriate chit-chat response should be returned.
6. Enter the message `What are the different types of modules on Microsoft Learn?`. The answer you created should be returned along with a follow-up prompt link.
7. Select the **Learn more about training** follow-up link. The follow-up answer with a link to the training page should be returned.
8. When you're done testing the knowledge base, close the test pane.

Deploy the knowledge base

The knowledge base provides a back-end service that client applications can use to answer questions. Now you are ready to publish your knowledge base and access its REST interface from a client.

1. In the **LearnFAQ** project in Language Studio, select the **Deploy knowledge base** page from the navigation menu on the left.

2. At the top of the page, select **Deploy**. Then select **Deploy** to confirm you want to deploy the knowledge base.
3. When deployment is complete, select **Get prediction URL** to view the REST endpoint for your knowledge base and note that the sample request includes parameters for:
    - **projectName**: The name of your project (which should be *LearnFAQ*)
    - **deploymentName**: The name of your deployment (which should be *production*)
4. Close the prediction URL dialog box.

Prepare to develop an app in Cloud Shell

You'll develop your question answering app using Cloud Shell in the Azure portal. The code files for your app have been provided in a GitHub repo.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

    Code Copy

    ```
    rm -r mslearn-ai-language -f
    git clone https://github.com/microsoftlearning/mslearn-ai-language
    ```

    > **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

    Code Copy

    ```
    cd mslearn-ai-language/Labfiles/02-qna/Python/qna-app
    ```

Configure your application

1. In the command line pane, run the following command to view the code files in the **qna-app** folder:

    Code Copy

    ```
    ls -a -l
    ```

    The files include a configuration file (**.env**) and a code file (**qna-app.py**).

2. Create a Python virtual environment and install the Azure AI Language Question Answering SDK package and other required packages by running the following command:

    Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-language-questionanswering
```

3. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

4. In the code file, update the configuration values it contains to reflect the **endpoint** and an authentication **key** for the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal). The project name and deployment name for your deployed knowledge base should also be in this file.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to user your knowledge base

1. Enter the following command to edit the application code file:

Code Copy

```
code qna-app.py
```

2. Review the existing code. You will add code to work with your knowledge base.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. In the code file, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Question Answering SDK:

Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.language.questionanswering import QuestionAnsweringClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a question answering client:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = QuestionAnsweringClient(endpoint=ai_endpoint, credential=credential)
```

5. In the code file, find the comment **Submit a question and display the answer**, and add the following code to repeatedly read questions from the command line, submit them to the service, and display details of the answers:

Code Copy

```
# Submit a question and display the answer
user_question = ''
while True:
    user_question = input('\nQuestion:\n')
    if user_question.lower() == "quit":
        break
    response = ai_client.get_answers(question=user_question,
                                     project_name=ai_project_name,
                                     deployment_name=ai_deployment_name)
    for candidate in response.answers:
        print(candidate.answer)
        print("Confidence: {}".format(candidate.confidence))
        print("Source: {}".format(candidate.source))
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python qna-app.py
```

7. When prompted, enter a question to be submitted to your question answering project; for example `What is a learning path?`.
8. Review the answer that is returned.
9. Ask more questions. When you're done, enter `quit`.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

To learn more about question answering in Azure AI Language, see the Azure AI Language documentation.

Create a language understanding model with the Language service

The Azure AI Language service enables you to define a *conversational language understanding* model that applications can use to interpret natural language *utterances* from users (text or spoken input), predict the users *intent* (what they want to achieve), and identify any *entities* to which the intent should be applied.

For example, a conversational language model for a clock application might be expected to process input such as:

What is the time in London?

This kind of input is an example of an *utterance* (something a user might say or type), for which the desired *intent* is to get the time in a specific location (an *entity*); in this case, London.

> **NOTE** The task of a conversational language model is to predict the user's intent and identify any entities to which the intent applies. It is not the job of a conversational language model to actually perform the actions required to satisfy the intent. For example, a clock application can use a conversational language model to discern that the user wants to know the time in London; but the client application itself must then implement the logic to determine the correct time and present it to the user.

In this exercise, you'll use the Azure AI Language service to create a conversational language understand model, and use the Python SDK to implement a client app that uses it.

While this exercise is based on Python, you can develop conversational understanding applications using multiple language-specific SDKs; including:

- Azure AI Conversations client library for Python
- Azure AI Conversations client library for .NET
- Azure AI Conversations client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource in your Azure subscription.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - China East 2
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North

    - UK South
    - West Europe
    - West US 2
    - West US 3
  - **Name**: *Enter a unique name*.
  - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
  - **Responsible AI Notice**: Agree.
5. Select **Review + create**, then select **Create** to provision the resource.
6. Wait for deployment to complete, and then go to the deployed resource.
7. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Create a conversational language understanding project

Now that you have created an authoring resource, you can use it to create a conversational language understanding project.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.
2. If prompted to choose a Language resource, select the following settings:

   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.
   - **Language resource**: The Azure AI Language resource you created previously.

   If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

   a. On the bar at the top of the page, select the **Settings (⚙)** button.
   b. On the **Settings** page, view the **Resources** tab.
   c. Select the language resource you just created, and click **Switch resource**.
   d. At the top of the page, click **Language Studio** to return to the Language Studio home page

3. At the top of the portal, in the **Create new** menu, select **Conversational language understanding**.
4. In the **Create a project** dialog box, on the **Enter basic information** page, enter the following details and then select **Next**:

   - **Name**: `Clock`
   - **Utterances primary language**: English
   - **Enable multiple languages in project?**: *Unselected*
   - **Description**: `Natural language clock`

5. On the **Review and finish** page, select **Create**.

Create intents

The first thing we'll do in the new project is to define some intents. The model will ultimately predict which of these intents a user is requesting when submitting a natural language utterance.

> **Tip**: When working on your project, if some tips are displayed, read them and select **Got it** to dismiss them, or select **Skip all**.

1. On the **Schema definition** page, on the **Intents** tab, select **+ Add** to add a new intent named `GetTime`.
2. Verify that the **GetTime** intent is listed (along with the default **None** intent). Then add the following additional intents:

   - `GetDay`
   - `GetDate`

Label each intent with sample utterances

To help the model predict which intent a user is requesting, you must label each intent with some sample utterances.

1. In the pane on the left, select the **Data Labeling** page.

> **Tip**: You can expand the pane with the **>>** icon to see the page names, and hide it again with the **<<** icon.

1. Select the new **GetTime** intent and enter the utterance `what is the time?`. This adds the utterance as sample input for the intent.
2. Add the following additional utterances for the **GetTime** intent:
    - `what's the time?`
    - `what time is it?`
    - `tell me the time`

> **NOTE** To add a new utterance, write the utterance in the textbox next to the intent and then press ENTER.

3. Select the **GetDay** intent and add the following utterances as example input for that intent:
    - `what day is it?`
    - `what's the day?`
    - `what is the day today?`
    - `what day of the week is it?`
4. Select the **GetDate** intent and add the following utterances for it:
    - `what date is it?`
    - `what's the date?`
    - `what is the date today?`
    - `what's today's date?`
5. After you've added utterances for each of your intents, select **Save changes**.

Train and test the model

Now that you've added some intents, let's train the language model and see if it can correctly predict them from user input.

1. In the pane on the left, select **Training jobs**. Then select **+ Start a training job**.
2. On the **Start a training job** dialog, select the option to train a new model, name it `Clock`. Select **Standard training** mode and the default **Data splitting** options.
3. To begin the process of training your model, select **Train**.
4. When training is complete (which may take several minutes) the job **Status** will change to **Training succeeded**.
5. Select the **Model performance** page, and then select the **Clock** model. Review the overall and per-intent evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may be included in the results).

> **NOTE** To learn more about the evaluation metrics, refer to the documentation

6. Go to the **Deploying a model** page, then select **Add deployment**.

7. On the **Add deployment** dialog, select **Create a new deployment name**, and then enter `production`.
8. Select the **Clock** model in the **Model** field then select **Deploy**. The deployment may take some time.
9. When the model has been deployed, select the **Testing deployments** page, then select the **production** deployment in the **Deployment name** field.
10. Enter the following text in the empty textbox, and then select **Run the test**:

    ```
    what's the time now?
    ```

    Review the result that is returned, noting that it includes the predicted intent (which should be **GetTime**) and a confidence score that indicates the probability the model calculated for the predicted intent. The JSON tab shows the comparative confidence for each potential intent (the one with the highest confidence score is the predicted intent)

11. Clear the text box, and then run another test with the following text:

    ```
    tell me the time
    ```

    Again, review the predicted intent and confidence score.

12. Try the following text:

    ```
    what's the day today?
    ```

    Hopefully the model predicts the **GetDay** intent.

Add entities

So far you've defined some simple utterances that map to intents. Most real applications include more complex utterances from which specific data entities must be extracted to get more context for the intent.

Add a learned entity

The most common kind of entity is a *learned* entity, in which the model learns to identify entity values based on examples.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Location` and ensure that the **Learned** tab is selected. Then select **Add entity**.
3. After the **Location** entity has been created, return to the **Data labeling** page.
4. Select the **GetTime** intent and enter the following new example utterance:

    ```
    what time is it in London?
    ```

5. When the utterance has been added, select the word **London**, and in the drop-down list that appears, select **Location** to indicate that "London" is an example of a location.
6. Add another example utterance for the **GetTime** intent:

    ```
    Tell me the time in Paris?
    ```

7. When the utterance has been added, select the word **Paris**, and map it to the **Location** entity.
8. Add another example utterance for the **GetTime** intent:

    ```
    what's the time in New York?
    ```

9. When the utterance has been added, select the words **New York**, and map them to the **Location** entity.
10. Select **Save changes** to save the new utterances.

Add a *list* entity

In some cases, valid values for an entity can be restricted to a list of specific terms and synonyms; which can help the app identify instances of the entity in utterances.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Weekday` and select the **List** entity tab. Then select **Add entity**.
3. On the page for the **Weekday** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **List** section, select **+ Add new list**. Then enter the following value and synonym and select **Save**:

List key	synonyms
`Sunday`	`Sun`

> **NOTE** To enter the fields of the new list, insert the value `Sunday` in the text field, then click on the field where 'Type in value and press enter...' is displayed, enter the synonyms, and press ENTER.

4. Repeat the previous step to add the following list components:

Value	synonyms
`Monday`	`Mon`
`Tuesday`	`Tue, Tues`
`Wednesday`	`Wed, Weds`
`Thursday`	`Thur, Thurs`
`Friday`	`Fri`
`Saturday`	`Sat`

5. After adding and saving the list values, return to the **Data labeling** page.
6. Select the **GetDate** intent and enter the following new example utterance:

```
what date was it on Saturday?
```

7. When the utterance has been added, select the word ***Saturday***, and in the drop-down list that appears, select **Weekday**.
8. Add another example utterance for the **GetDate** intent:

```
what date will it be on Friday?
```

9. When the utterance has been added, map **Friday** to the **Weekday** entity.
10. Add another example utterance for the **GetDate** intent:

```
what will the date be on Thurs?
```

11. When the utterance has been added, map **Thurs** to the **Weekday** entity.
12. select **Save changes** to save the new utterances.

Add a *prebuilt* entity

The Azure AI Language service provides a set of *prebuilt* entities that are commonly used in conversational applications.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Date` and select the **Prebuilt** entity tab. Then select **Add entity**.
3. On the page for the **Date** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **Prebuilt** section, select **+ Add new prebuilt**.
4. In the **Select prebuilt** list, select **DateTime** and then select **Save**.
5. After adding the prebuilt entity, return to the **Data labeling** page
6. Select the **GetDay** intent and enter the following new example utterance:

   ```
   what day was 01/01/1901?
   ```

7. When the utterance has been added, select ***01/01/1901***, and in the drop-down list that appears, select **Date**.
8. Add another example utterance for the **GetDay** intent:

   ```
   what day will it be on Dec 31st 2099?
   ```

9. When the utterance has been added, map **Dec 31st 2099** to the **Date** entity.
10. Select **Save changes** to save the new utterances.

Retrain the model

Now that you've modified the schema, you need to retrain and retest the model.

1. On the **Training jobs** page, select **Start a training job**.
2. On the **Start a training job** dialog, select **overwrite an existing model** and specify the **Clock** model. Select **Train** to train the model. If prompted, confirm you want to overwrite the existing model.
3. When training is complete the job **Status** will update to **Training succeeded**.
4. Select the **Model performance** page and then select the **Clock** model. Review the evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may be included in the results).
5. On the **Deploying a model** page, select **Add deployment**.
6. On the **Add deployment** dialog, select **Override an existing deployment name**, and then select **production**.
7. Select the **Clock** model in the **Model** field and then select **Deploy** to deploy it. This may take some time.
8. When the model is deployed, on the **Testing deployments** page, select the **production** deployment under the **Deployment name** field, and then test it with the following text:

   ```
   what's the time in Edinburgh?
   ```

9. Review the result that is returned, which should hopefully predict the **GetTime** intent and a **Location** entity with the text value "Edinburgh".
10. Try testing the following utterances:

    ```
    what time is it in Tokyo?
    ```

    ```
    what date is it on Friday?
    ```

    ```
    what's the date on Weds?
    ```

    ```
    what day was 01/01/2020?
    ```

```
what day will Mar 7th 2030 be?
```

Use the model from a client app

In a real project, you'd iteratively refine intents and entities, retrain, and retest until you are satisfied with the predictive performance. Then, when you've tested it and are satisfied with its predictive performance, you can use it in a client app by calling its REST interface or a runtime-specific SDK.

Prepare to develop an app in Cloud Shell

You'll develop your language understanding app using Cloud Shell in the Azure portal. The code files for your app have been provided in a GitHub repo.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-language/Labfiles/03-language/Python/clock-client
   ```

Configure your application

1. In the command line pane, run the following command to view the code files in the **clock-client** folder:

   Code Copy

   ```
   ls -a -l
   ```

   The files include a configuration file (**.env**) and a code file (**clock-client.py**).

2. Create a Python virtual environment and install the Azure AI Language Conversations SDK package and other required packages by running the following command:

   Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-language-conversations==1.1.0
```

3. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to the application

1. Enter the following command to edit the application code file:

Code Copy

```
code clock-client.py
```

2. Review the existing code. You will add code to work with the AI Language Conversations SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the AI Language Conversations SDK:

Code Copy

```
# Import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.language.conversations import ConversationAnalysisClient
```

4. In the **main** function, note that code to load the prediction endpoint and key from the configuration file has already been provided. Then find the comment **Create a client for the Language service model** and add the following code to create a conversation analysis client for your AI Language service:

Code Copy

```
# Create a client for the Language service model
client = ConversationAnalysisClient(
    ls_prediction_endpoint, AzureKeyCredential(ls_prediction_key))
```

5. Note that the code in the **main** function prompts for user input until the user enters "quit". Within this loop, find the comment **Call the Language service model to get intent and entities** and add the following code:

Code Copy

```
# Call the Language service model to get intent and entities
cls_project = 'Clock'
deployment_slot = 'production'

with client:
    query = userText
    result = client.analyze_conversation(
        task={
            "kind": "Conversation",
            "analysisInput": {
                "conversationItem": {
                    "participantId": "1",
                    "id": "1",
                    "modality": "text",
                    "language": "en",
                    "text": query
                },
                "isLoggingEnabled": False
            },
            "parameters": {
                "projectName": cls_project,
                "deploymentName": deployment_slot,
                "verbose": True
            }
        }
    )

top_intent = result["result"]["prediction"]["topIntent"]
entities = result["result"]["prediction"]["entities"]

print("view top intent:")
print("\ttop intent: {}".format(result["result"]["prediction"]["topIntent"]))
print("\tcategory: {}".format(result["result"]["prediction"]["intents"][0]["category"]))
print("\tconfidence score: {}\n".format(result["result"]["prediction"]["intents"][0]
["confidenceScore"]))

print("view entities:")
for entity in entities:
    print("\tcategory: {}".format(entity["category"]))
    print("\ttext: {}".format(entity["text"]))
    print("\tconfidence score: {}".format(entity["confidenceScore"]))

print("query: {}".format(result["result"]["query"]))
```

The call to the conversational understanding model returns a prediction/result, which includes the top (most likely) intent as well as any entities that were detected in the input utterance. Your client application must now use that prediction to determine and perform the appropriate action.

6. Find the comment **Apply the appropriate action**, and add the following code, which checks for intents supported by the application (**GetTime**, **GetDate**, and **GetDay**) and determines if any relevant entities have been detected, before calling an existing function to produce an appropriate response.

Code Copy

```
# Apply the appropriate action
if top_intent == 'GetTime':
    location = 'local'
    # Check for entities
    if len(entities) > 0:
        # Check for a location entity
        for entity in entities:
            if 'Location' == entity["category"]:
                # ML entities are strings, get the first one
                location = entity["text"]
    # Get the time for the specified location
    print(GetTime(location))

elif top_intent == 'GetDay':
    date_string = date.today().strftime("%m/%d/%Y")
    # Check for entities
    if len(entities) > 0:
        # Check for a Date entity
        for entity in entities:
            if 'Date' == entity["category"]:
                # Regex entities are strings, get the first one
                date_string = entity["text"]
    # Get the day for the specified date
    print(GetDay(date_string))

elif top_intent == 'GetDate':
    day = 'today'
    # Check for entities
    if len(entities) > 0:
        # Check for a Weekday entity
        for entity in entities:
            if 'Weekday' == entity["category"]:
                # List entities are lists
                day = entity["text"]
    # Get the date for the specified day
    print(GetDate(day))

else:
    # Some other intent (for example, "None") was predicted
    print('Try asking me for the time, the day, or the date.')
```

7. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python clock-client.py
```

8. When prompted, enter utterances to test the application. For example, try:

   Hello

   What time is it?

   What's the time in London?

   What's the date?

   What date is Sunday?

What day is it?

What day is 01/01/2025?

> **Note**: The logic in the application is deliberately simple, and has a number of limitations. For example, when getting the time, only a restricted set of cities is supported and daylight savings time is ignored. The goal is to see an example of a typical pattern for using Language Service in which your application must:
>
> a. Connect to a prediction endpoint.
> b. Submit an utterance to get a prediction.
> c. Implement logic to respond appropriately to the predicted intent and entities.

9. When you have finished testing, enter *quit*.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

To learn more about conversational language understanding in Azure AI Language, see the Azure AI Language documentation.

Custom text classification

Azure AI Language provides several NLP capabilities, including the key phrase identification, text summarization, and sentiment analysis. The Language service also provides custom features like custom question answering and custom text classification.

To test the custom text classification of the Azure AI Language service, you'll configure the model using Language Studio then use a Python application to test it.

While this exercise is based on Python, you can develop text classification applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, use custom text classification, you need to enable the **Custom text classification & extraction** feature.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select the box that includes **Custom text classification**. Then select **Continue to create your resource**.
5. Create a resource with the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Select or create a resource group*.
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North
        - UK South
        - West Europe
        - West US 2
        - West US 3
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Storage account**: New storage account
        - **Storage account name**: *Enter a unique name*.
        - **Storage account type**: Standard LRS
    - **Responsible AI notice**: Selected.
6. Select **Review + create,** then select **Create** to provision the resource.

7. Wait for deployment to complete, and then go to the resource group.

8. Find the storage account you created, select it, and verify the *Account kind* is **StorageV2**. If it's v1, upgrade your storage account kind on that resource page.

Configure role-based access for your user

> **NOTE**: If you skip this step, you'll get a 403 error when trying to connect to your custom project. It's important that your current user has this role to access storage account blob data, even if you're the owner of the storage account.

1. Go to your storage account page in the Azure portal.
2. Select **Access Control (IAM)** in the left navigation menu.
3. Select **Add** to Add Role Assignments, and choose the **Storage Blob Data Owner** role on the storage account.
4. Within **Assign access to**, select **User, group, or service principal**.
5. Select **Select members**.
6. Select your User. You can search for user names in the **Select** field.

Upload sample articles

Once you've created the Azure AI Language service and storage account, you'll need to upload example articles to train your model later.

1. In a new browser tab, download sample articles from `https://aka.ms/classification-articles` and extract the files to a folder of your choice.

2. In the Azure portal, navigate to the storage account you created, and select it.

3. In your storage account select **Configuration**, located below **Settings**. In the Configuration screen enable the option to **Allow Blob anonymous access** then select **Save**.

4. Select **Containers** in the left menu, located below **Data storage**. On the screen that appears, select **+ Container**. Give the container the name `articles`, and set **Anonymous access level** to **Container (anonymous read access for containers and blobs)**.

   > **NOTE**: When you configure a storage account for a real solution, be careful to assign the appropriate access level. To learn more about each access level, see the Azure Storage documentation.

5. After you've created the container, select it then select the **Upload** button. Select **Browse for files** to browse for the sample articles you downloaded. Then select **Upload**.

Create a custom text classification project

After configuration is complete, create a custom text classification project. This project provides a working place to build, train, and deploy your model.

> **NOTE**: This lab utilizes **Language Studio**, but you can also create, build, train, and deploy your model through the REST API.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.

2. If prompted to choose a Language resource, select the following settings:

   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.

- **Language resource**: The Azure AI Language resource you created previously.

If you are <u>not</u> prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

a. On the bar at the top if the page, select the **Settings (⚙)** button.
b. On the **Settings** page, view the **Resources** tab.
c. Select the language resource you just created, and click **Switch resource**.
d. At the top of the page, click **Language Studio** to return to the Language Studio home page

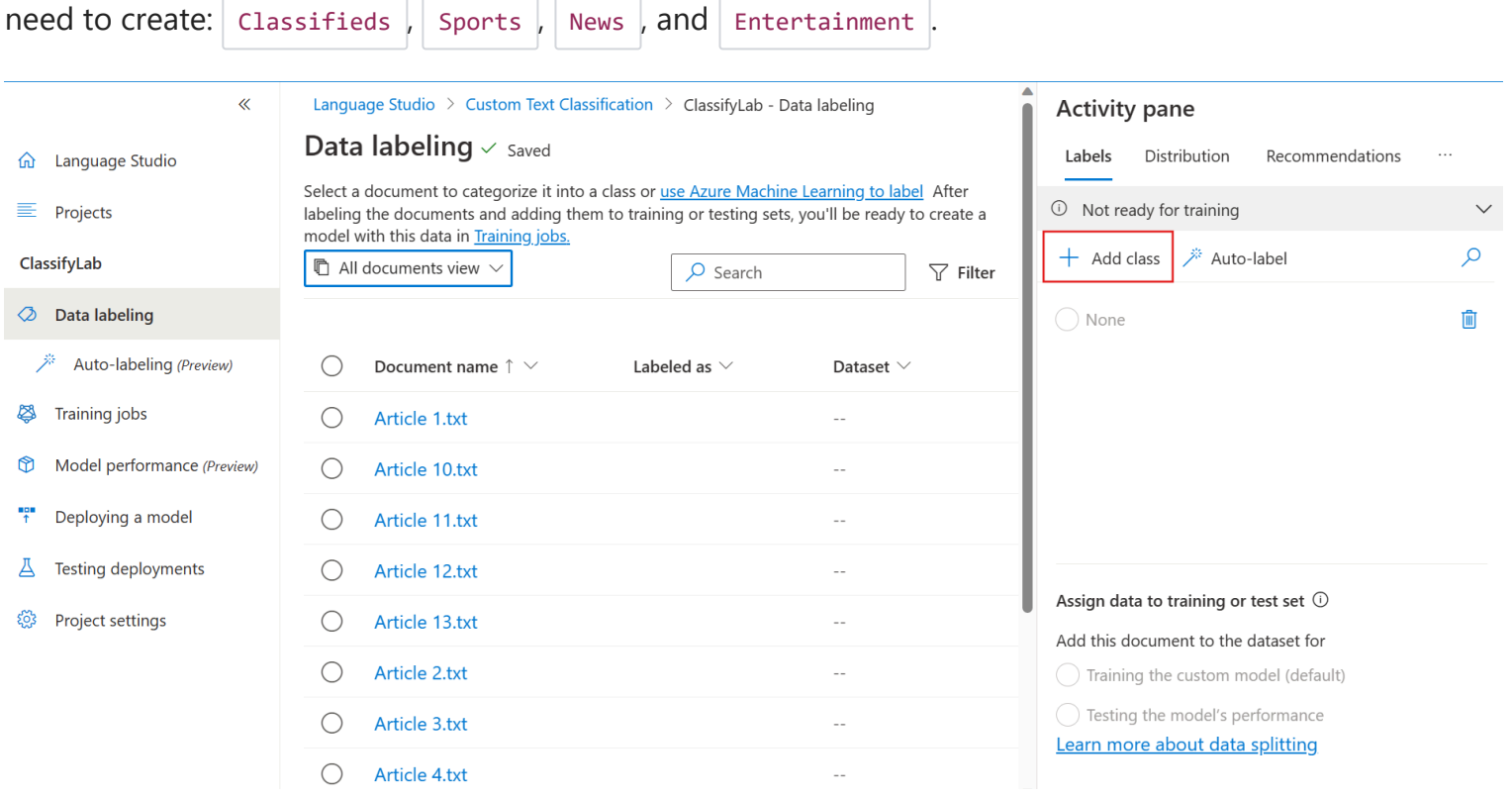
3. At the top of the portal, in the **Create new** menu, select **Custom text classification**.
4. The **Connect storage** page appears. All values will already have been filled. So select **Next**.
5. On the **Select project type** page, select **Single label classification**. Then select **Next**.
6. On the **Enter basic information** pane, set the following:
   - **Name**: `ClassifyLab`
   - **Text primary language**: English (US)
   - **Description**: `Custom text lab`
7. Select **Next**.
8. On the **Choose container** page, set the **Blob store container** dropdown to your *articles* container.
9. Select the **No, I need to label my files as part of this project** option. Then select **Next**.
10. Select **Create project**.

> **Tip**: If you get an error about not being authorized to perform this operation, you'll need to add a role assignment. To fix this, we add the role "Storage Blob Data Contributor" on the storage account for the user running the lab. More details can be found on the documentation page

Label your data

Now that your project is created, you need to label, or tag, your data to train your model how to classify text.

1. On the left, select **Data labeling**, if not already selected. You'll see a list of the files you uploaded to your storage account.
2. On the right side, in the **Activity** pane, select **+ Add class**. The articles in this lab fall into four classes you'll need to create: `Classifieds`, `Sports`, `News`, and `Entertainment`.



3. After you've created your four classes, select **Article 1** to start. Here you can read the article, define which class this file is, and which dataset (training or testing) to assign it to.

4. Assign each article the appropriate class and dataset (training or testing) using the **Activity** pane on the right. You can select a label from the list of labels on the right, and set each article to **training** or **testing** using the options at the bottom of the Activity pane. You select **Next document** to move to the next document. For the purposes of this lab, we'll define which are to be used for training the model and testing the model:

Article	Class	Dataset
Article 1	Sports	Training
Article 10	News	Training
Article 11	Entertainment	Testing
Article 12	News	Testing
Article 13	Sports	Testing
Article 2	Sports	Training
Article 3	Classifieds	Training
Article 4	Classifieds	Training
Article 5	Entertainment	Training
Article 6	Entertainment	Training
Article 7	News	Training
Article 8	News	Training
Article 9	Entertainment	Training

> **NOTE** Files in Language Studio are listed alphabetically, which is why the above list is not in sequential order. Make sure you visit both pages of documents when labeling your articles.

5. Select **Save labels** to save your labels.

Train your model

After you've labeled your data, you need to train your model.

1. Select **Training jobs** on the left side menu.
2. Select **Start a training job**.
3. Train a new model named `ClassifyArticles`.
4. Select **Use a manual split of training and testing data**.

> **TIP** In your own classification projects, the Azure AI Language service will automatically split the testing set by percentage which is useful with a large dataset. With smaller datasets, it's important to train with the right class distribution.

5. Select **Train**

> **IMPORTANT** Training your model can sometimes take several minutes. You'll get a notification when it's complete.

Evaluate your model

In real world applications of text classification, it's important to evaluate and improve your model to verify it's performing as you expect.

1. Select **Model performance**, and select your **ClassifyArticles** model. There you can see the scoring of your model, performance metrics, and when it was trained. If the scoring of your model isn't 100%, it means that one of the documents used for testing didn't evaluate to what it was labeled. These failures can help you understand where to improve.
2. Select **Test set details** tab. If there are any errors, this tab allows you to see the articles you indicated for testing and what the model predicted them as and whether that conflicts with their test label. The tab defaults to show incorrect predictions only. You can toggle the **Show mismatches only** option to see all the articles you indicated for testing and what they each of them predicted as.

Deploy your model

When you're satisfied with the training of your model, it's time to deploy it, which allows you to start classifying text through the API.

1. On the left panel, select **Deploying model**.
2. Select **Add deployment**, then enter `articles` in the **Create a new deployment name** field, and select **ClassifyArticles** in the **Model** field.
3. Select **Deploy** to deploy your model.
4. Once your model is deployed, leave that page open. You'll need your project and deployment name in the next step.

Prepare to develop an app in Cloud Shell

To test the custom text classification capabilities of the Azure AI Language service, you'll develop a simple console application in the Azure Cloud Shell.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

```
cd mslearn-ai-language/Labfiles/04-text-classification/Python/classify-text
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **classify-text** folder:

   Code Copy

   ```
   ls -a -l
   ```

   The files include a configuration file (**.env**) and a code file (**classify-text.py**). The text your application will analyze is in the **articles** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-textanalytics==5.3.0
   ```

3. Enter the following command to edit the application configuration file:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).The file should already contain the project and deployment names for your text classification model.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to classify documents

1. Enter the following command to edit the application code file:

   Code Copy

   ```
   code classify-text.py
   ```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

   > **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

   Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key and the project and deployment names from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a text analysis client:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Note that the existing code reads all of the files in the **articles** folder and creates a list containing their contents. Then find the comment **Get Classifications** and add the following code:

Code Copy

```
# Get Classifications
operation = ai_client.begin_single_label_classify(
    batchedDocuments,
    project_name=project_name,
    deployment_name=deployment_name
)

document_results = operation.result()

for doc, classification_result in zip(files, document_results):
    if classification_result.kind == "CustomDocumentClassification":
        classification = classification_result.classifications[0]
        print("{} was classified as '{}' with confidence score {}.".format(
            doc, classification.category, classification.confidence_score)
        )
    elif classification_result.is_error is True:
        print("{} has an error with code '{}' and message '{}'".format(
            doc, classification_result.error.code, classification_result.error.message)
        )
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python classify-text.py
```

7. Observe the output. The application should list a classification and confidence score for each text file.

Clean up

When you don't need your project any more, you can delete if from your **Projects** page in Language Studio. You can also remove the Azure AI Language service and associated storage account in the Azure portal.

Extract custom entities

In addition to other natural language processing capabilities, Azure AI Language Service enables you to define custom entities, and extract instances of them from text.

To test the custom entity extraction, we'll create a model and train it through Azure AI Language Studio, then use a Python application to test it.

While this exercise is based on Python, you can develop text classification applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, use custom text classification, you need to enable the **Custom text classification & extraction** feature.

1. In a browser, open the Azure portal at `https://portal.azure.com`, and sign in with your Microsoft account.
2. Select the **Create a resource** button, search for *Language*, and create a **Language Service** resource. When on the page for *Select additional features*, select the custom feature containing **Custom named entity recognition extraction**. Create the resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Select or create a resource group*
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North
        - UK South
        - West Europe
        - West US 2
        - West US 3
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Storage account**: New storage account:
        - **Storage account name**: *Enter a unique name*.
        - **Storage account type**: Standard LRS
    - **Responsible AI notice**: Selected.
3. Select **Review + create**, then select **Create** to provision the resource.
4. Wait for deployment to complete, and then go to the deployed resource.
5. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Configure role-based access for your user

> **NOTE**: If you skip this step, you'll have a 403 error when trying to connect to your custom project. It's important that your current user has this role to access storage account blob data, even if you're the owner of the storage account.

1. Go to your storage account page in the Azure portal.
2. Select **Access Control (IAM)** in the left navigation menu.
3. Select **Add** to Add Role Assignments, and choose the **Storage Blob Data Contributor** role on the storage account.
4. Within **Assign access to**, select **User, group, or service principal**.
5. Select **Select members**.
6. Select your User. You can search for user names in the **Select** field.

Upload sample ads

After you've created the Azure AI Language Service and storage account, you'll need to upload example ads to train your model later.

1. In a new browser tab, download sample classified ads from `https://aka.ms/entity-extraction-ads` and extract the files to a folder of your choice.
2. In the Azure portal, navigate to the storage account you created, and select it.
3. In your storage account select **Configuration**, located below **Settings**, and screen enable the option to **Allow Blob anonymous access** then select **Save**.
4. Select **Containers** from the left menu, located below **Data storage**. On the screen that appears, select **+ Container**. Give the container the name `classifieds`, and set **Anonymous access level** to **Container (anonymous read access for containers and blobs)**.

   > **NOTE**: When you configure a storage account for a real solution, be careful to assign the appropriate access level. To learn more about each access level, see the Azure Storage documentation.

5. After creating the container, select it and click the **Upload** button and upload the sample ads you downloaded.

Create a custom named entity recognition project

Now you're ready to create a custom named entity recognition project. This project provides a working place to build, train, and deploy your model.

> **NOTE**: You can also create, build, train, and deploy your model through the REST API.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.
2. If prompted to choose a Language resource, select the following settings:
   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.
   - **Language resource**: The Azure AI Language resource you created previously.

   If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

   a. On the bar at the top of the page, select the **Settings (⚙)** button.
   b. On the **Settings** page, view the **Resources** tab.

c. Select the language resource you just created, and click **Switch resource**.
d. At the top of the page, click **Language Studio** to return to the Language Studio home page.

3. At the top of the portal, in the **Create new** menu, select **Custom named entity recognition**.

4. Create a new project with the following settings:

- **Connect storage**: *This value is likely already filled. Change it to your storage account if it isn't already*
- **Basic information**:
- **Name**: `CustomEntityLab`
    - **Text primary language**: English (US)
    - **Does your dataset include documents that are not in the same language?** : *No*
    - **Description**: `Custom entities in classified ads`
- **Container**:
    - **Blob store container**: classifieds
    - **Are your files labeled with classes?**: No, I need to label my files as part of this project

> **Tip**: If you get an error about not being authorized to perform this operation, you'll need to add a role assignment. To fix this, we add the role "Storage Blob Data Contributor" on the storage account for the user running the lab. More details can be found on the documentation page

Label your data

Now that your project is created, you need to label your data to train your model how to identity entities.

1. If the **Data labeling** page is not already open, in the pane on the left, select **Data labeling**. You'll see a list of the files you uploaded to your storage account.
2. On the right side, in the **Activity** pane, select **Add entity** and add a new entity named `ItemForSale`.
3. Repeat the previous step to create the following entities:
    - `Price`
    - `Location`
4. After you've created your three entities, select **Ad 1.txt** so you can read it.
5. In *Ad 1.txt*:
    a. Highlight the text *face cord of firewood* and select the **ItemForSale** entity.
    b. Highlight the text *Denver, CO* and select the **Location** entity.
    c. Highlight the text *$90* and select the **Price** entity.
6. In the **Activity** pane, note that this document will be added to the dataset for training the model.
7. Use the **Next document** button to move to the next document, and continue assigning text to appropriate entities for the entire set of documents, adding them all to the training dataset.
8. When you have labeled the last document (*Ad 9.txt*), save the labels.

Train your model

After you've labeled your data, you need to train your model.

1. Select **Training jobs** in the pane on the left.
2. Select **Start a training job**
3. Train a new model named `ExtractAds`
4. Choose **Automatically split the testing set from training data**

> **TIP**: In your own extraction projects, use the testing split that best suits your data. For more consistent data and larger datasets, the Azure AI Language Service will automatically split the testing set by percentage. With smaller datasets, it's important to train with the right variety of possible input documents.

5. Click **Train**

> **IMPORTANT**: Training your model can sometimes take several minutes. You'll get a notification when it's complete.

Evaluate your model

In real world applications, it's important to evaluate and improve your model to verify it's performing as you expect. Two pages on the left show you the details of your trained model, and any testing that failed.

Select **Model performance** on the left side menu, and select your `ExtractAds` model. There you can see the scoring of your model, performance metrics, and when it was trained. You'll be able to see if any testing documents failed, and these failures help you understand where to improve.

Deploy your model

When you're satisfied with the training of your model, it's time to deploy it, which allows you to start extracting entities through the API.

1. In the left pane, select **Deploying a model**.
2. Select **Add deployment**, then enter the name `AdEntities` and select the **ExtractAds** model.
3. Click **Deploy** to deploy your model.

Prepare to develop an app in Cloud Shell

To test the custom entity extraction capabilities of the Azure AI Language service, you'll develop a simple console application in the Azure Cloud Shell.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

   Code Copy

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/05-custom-entity-recognition/Python/custom-entities
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **custom-entities** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**custom-entities.py**). The text your application will analyze is in the **ads** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-textanalytics==5.3.0
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).The file should already contain the project and deployment names for your custom entity extraction model.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to extract entities

1. Enter the following command to edit the application code file:

```
code custom-entities.py
```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key and the project and deployment names from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a text analytics client:

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Note that the existing code reads all of the files in the **ads** folder and creates a list containing their contents. Then find the comment **Extract entities** and add the following code:

```
# Extract entities
operation = ai_client.begin_recognize_custom_entities(
    batchedDocuments,
    project_name=project_name,
    deployment_name=deployment_name
)

document_results = operation.result()

for doc, custom_entities_result in zip(files, document_results):
    print(doc)
    if custom_entities_result.kind == "CustomEntityRecognition":
        for entity in custom_entities_result.entities:
            print(
                "\tEntity '{}' has category '{}' with confidence score of '{}'".format(
                    entity.text, entity.category, entity.confidence_score
                )
            )
    elif custom_entities_result.is_error is True:
        print("\tError with code '{}' and message '{}'".format(
            custom_entities_result.error.code, custom_entities_result.error.message
            )
        )
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

```
python custom-entities.py
```

7. Observe the output. The application should list details of the entities found in each text file.

Clean up

When you don't need your project anymore, you can delete if from your **Projects** page in Language Studio. You can also remove the Azure AI Language service and associated storage account in the Azure portal.

Translate Text

Azure AI Translator is a service that enables you to translate text between languages. In this exercise, you'll use it to create a simple app that translates input in any supported language to the target language of your choice.

While this exercise is based on Python, you can develop text translation applications using multiple language-specific SDKs; including:

- Azure AI Translation client library for Python
- Azure AI Translation client library for .NET
- Azure AI Translation client library for JavaScript

This exercise takes approximately **30** minutes.

Provision an *Azure AI Translator* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Translator** resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the search field at the top, search for **Translators** then select **Translators** in the results.
3. Create a resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Choose or create a resource group*
    - **Region**: *Choose any available region*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Prepare to develop an app in Cloud Shell

To test the text translation capabilities of Azure AI Translator, you'll develop a simple console application in the Azure Cloud Shell.

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/06-translator-sdk/Python/translate-text
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **translate-text** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**translate.py**).

2. Create a Python virtual environment and install the Azure AI Translation SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-translation-text==1.0.1
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **region** and a **key** from the Azure AI Translator resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

> **NOTE**: Be sure to add the *region* for your resource, not the endpoint!

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to translate text

1. Enter the following command to edit the application code file:

```
code translate.py
```

2. Review the existing code. You will add code to work with the Azure AI Translation SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Translation SDK:

Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.translation.text import *
from azure.ai.translation.text.models import InputTextItem
```

4. In the **main** function, note that the existing code reads the configuration settings.

5. Find the comment **Create client using endpoint and key** and add the following code:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(translatorKey)
client = TextTranslationClient(credential=credential, region=translatorRegion)
```

6. Find the comment **Choose target language** and add the following code, which uses the Text Translator service to return list of supported languages for translation, and prompts the user to select a language code for the target language:

Code Copy

```
# Choose target language
languagesResponse = client.get_supported_languages(scope="translation")
print("{} languages supported.".format(len(languagesResponse.translation)))
print("(See https://learn.microsoft.com/azure/ai-services/translator/language-support#translation)")
print("Enter a target language code for translation (for example, 'en'):")
targetLanguage = "xx"
supportedLanguage = False
while supportedLanguage == False:
    targetLanguage = input()
    if  targetLanguage in languagesResponse.translation.keys():
        supportedLanguage = True
    else:
        print("{} is not a supported language.".format(targetLanguage))
```

7. Find the comment **Translate text** and add the following code, which repeatedly prompts the user for text to be translated, uses the Azure AI Translator service to translate it to the target language (detecting the source language automatically), and displays the results until the user enters *quit*:

Code Copy

```
# Translate text
inputText = ""
while inputText.lower() != "quit":
    inputText = input("Enter text to translate ('quit' to exit):")
    if inputText != "quit":
        input_text_elements = [InputTextItem(text=inputText)]
        translationResponse = client.translate(body=input_text_elements, to_language=
[targetLanguage])
        translation = translationResponse[0] if translationResponse else None
        if translation:
            sourceLanguage = translation.detected_language
            for translated_text in translation.translations:
                print(f"'{inputText}' was translated from {sourceLanguage.language} to
{translated_text.to} as '{translated_text.text}'.")
```

8. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python translate.py
```

9. When prompted, enter a valid target language from the list displayed.
10. Enter a phrase to be translated (for example `This is a test` or `C'est un test`) and view the results, which should detect the source language and translate the text to the target language.
11. When you're done, enter `quit`. You can run the application again and choose a different target language.

Clean up resources

If you're finished exploring the Azure AI Translator service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Translator resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

For more information about using **Azure AI Translator**, see the Azure AI Translator documentation.

Recognize and synthesize speech

Azure AI Speech is a service that provides speech-related functionality, including:

- A *speech-to-text* API that enables you to implement speech recognition (converting audible spoken words into text).
- A *text-to-speech* API that enables you to implement speech synthesis (converting text into audible speech).

In this exercise, you'll use both of these APIs to implement a speaking clock application.

While this exercise is based on Python, you can develop speech applications using multiple language-specific SDKs; including:

- Azure AI Speech SDK for Python
- Azure AI Speech SDK for .NET
- Azure AI Speech SDK for JavaScript

This exercise takes approximately **30** minutes.

> **NOTE** This exercise is designed to be completed in the Azure cloud shell, where direct access to your computer's sound hardware is not supported. The lab will therefore use audio files for speech input and output streams. The code to achieve the same results using a mic and speaker is provided for your reference.

Create an Azure AI Speech resource

Let's start by creating an Azure AI Speech resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search field, search for **Speech service**. Select it from the list, then select **Create**.
3. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Prepare and configure the speaking clock app

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

Code Copy

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the speaking clock application code files:

Code Copy

```
cd mslearn-ai-language/Labfiles/07-speech/Python/speaking-clock
```

5. In the command line pane, run the following command to view the code files in the **speaking-clock** folder:

Code Copy

```
ls -a -l
```

The files include a configuration file (**.env**) and a code file (**speaking-clock.py**). The audio files your application will use are in the **audio** subfolder.

6. Create a Python virtual environment and install the Azure AI Speech SDK package and other required packages by running the following command:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-speech==1.42.0
```

7. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

8. Update the configuration values to include the **region** and a **key** from the Azure AI Speech resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to use the Azure AI Speech SDK

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code speaking-clock.py
```

2. At the top of the code file, under the existing namespace references, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Azure AI Speech SDK:

Code Copy

```
# Import namespaces
from azure.core.credentials import AzureKeyCredential
import azure.cognitiveservices.speech as speech_sdk
```

3. In the **main** function, under the comment **Get config settings**, note that the code loads the key and region you defined in the configuration file.

4. Find the comment **Configure speech service**, and add the following code to use the AI Services key and your region to configure your connection to the Azure AI Services Speech endpoint:

Code Copy

```
# Configure speech service
speech_config = speech_sdk.SpeechConfig(speech_key, speech_region)
print('Ready to use speech service in:', speech_config.region)
```

5. Save your changes (*CTRL+S*), but leave the code editor open.

Run the app

So far, the app doesn't do anything other than connect to your Azure AI Speech service, but it's useful to run it and check that it works before adding speech functionality.

1. In the command line, enter the following command to run the speaking clock app:

Code Copy

```
python speaking-clock.py
```

The code should display the region of the speech service resource the application will use. A successful run indicates that the app has connected to your Azure AI Speech resource.

Add code to recognize speech

Now that you have a **SpeechConfig** for the speech service in your project's Azure AI Services resource, you can use the **Speech-to-text** API to recognize speech and transcribe it to text.

In this procedure, the speech input is captured from an audio file, which you can play here:

1. In the code file, note that the code uses the **TranscribeCommand** function to accept spoken input. Then in the **TranscribeCommand** function, find the comment **Configure speech recognition** and add the appropriate code below to create a **SpeechRecognizer** client that can be used to recognize and transcribe speech from an audio file:

Code Copy

```
# Configure speech recognition
current_dir = os.getcwd()
audioFile = current_dir + '/time.wav'
audio_config = speech_sdk.AudioConfig(filename=audioFile)
speech_recognizer = speech_sdk.SpeechRecognizer(speech_config, audio_config)
```

2. In the **TranscribeCommand** function, under the comment **Process speech input**, add the following code to listen for spoken input, being careful not to replace the code at the end of the function that returns the command:

Code Copy

```
# Process speech input
print("Listening...")
speech = speech_recognizer.recognize_once_async().get()
if speech.reason == speech_sdk.ResultReason.RecognizedSpeech:
    command = speech.text
    print(command)
else:
    print(speech.reason)
    if speech.reason == speech_sdk.ResultReason.Canceled:
        cancellation = speech.cancellation_details
        print(cancellation.reason)
        print(cancellation.error_details)
```

3. Save your changes (*CTRL+S*), and then in the command line below the code editor, re-run the program:
4. Review the output, which should successfully "hear" the speech in the audio file and return an appropriate response (note that your Azure cloud shell may be running on a server that is in a different time-zone to yours!)

> **Tip**: If the SpeechRecognizer encounters an error, it produces a result of "Cancelled". The code in the application will then display the error message. The most likely cause is an incorrect region value in the configuration file.

Synthesize speech

Your speaking clock application accepts spoken input, but it doesn't actually speak! Let's fix that by adding code to synthesize speech.

Once again, due to the hardware limitations of the cloud shell we'll direct the synthesized speech output to a file.

1. In the code file, note that the code uses the **TellTime** function to tell the user the current time.
2. In the **TellTime** function, under the comment **Configure speech synthesis**, add the following code to create a **SpeechSynthesizer** client that can be used to generate spoken output:

Code Copy

```
# Configure speech synthesis
output_file = "output.wav"
speech_config.speech_synthesis_voice_name = "en-GB-RyanNeural"
audio_config = speech_sdk.audio.AudioConfig(filename=output_file)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config,)
```

3. In the **TellTime** function, under the comment **Synthesize spoken output**, add the following code to generate spoken output, being careful not to replace the code at the end of the function that prints the response:

```
# Synthesize spoken output
speak = speech_synthesizer.speak_text_async(response_text).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
else:
    print("Spoken output saved in " + output_file)
```

4. Save your changes (*CTRL+S*) and re-run the program, which should indicate that the spoken output was saved in a file.

5. If you have a media player capable of playing .wav audio files, download the file that was generated by entering the following command:

```
download ./output.wav
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file.

The file should sound similar to this:

Use Speech Synthesis Markup Language

Speech Synthesis Markup Language (SSML) enables you to customize the way your speech is synthesized using an XML-based format.

1. In the **TellTime** function, replace all of the current code under the comment **Synthesize spoken output** with the following code (leave the code under the comment **Print the response**):

```
# Synthesize spoken output
responseSsml = " \
   <speak version='1.0' xmlns='http://www.w3.org/2001/10/synthesis' xml:lang='en-US'> \
       <voice name='en-GB-LibbyNeural'> \
           {} \
           <break strength='weak'/> \
           Time to end this lab! \
       </voice> \
   </speak>".format(response_text)
speak = speech_synthesizer.speak_ssml_async(responseSsml).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
   print(speak.reason)
else:
   print("Spoken output saved in " + output_file)
```

2. Save your changes and re-run the program, which should once again indicate that the spoken output was saved in a file.
3. Download and play the generated file, which should sound similar to this:

Clean up

If you've finished exploring Azure AI Speech, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Speech resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

What if you have a mic and speaker?

In this exercise, the Azure Cloud Shell environment we used doesn't support audio hardware, so you used audio files for the speech input and output. Let's see how the code can be modified to use audio hardware if you have it available.

Using speech recognition with a microphone

If you have a mic, you can use the following code to capture spoken input for speech recognition:

Code Copy

```
# Configure speech recognition
audio_config = speech_sdk.AudioConfig(use_default_microphone=True)
speech_recognizer = speech_sdk.SpeechRecognizer(speech_config, audio_config)
print('Speak now...')

# Process speech input
speech = speech_recognizer.recognize_once_async().get()
if speech.reason == speech_sdk.ResultReason.RecognizedSpeech:
    command = speech.text
    print(command)
else:
    print(speech.reason)
    if speech.reason == speech_sdk.ResultReason.Canceled:
        cancellation = speech.cancellation_details
        print(cancellation.reason)
        print(cancellation.error_details)
```

> **Note**: The system default microphone is the default audio input, so you could also just omit the AudioConfig altogether!

Using speech synthesis with a speaker

If you have a speaker, you can use the following code to synthesize speech.

Code Copy

```
response_text = 'The time is {}:{:02d}'.format(now.hour,now.minute)

# Configure speech synthesis
speech_config.speech_synthesis_voice_name = "en-GB-RyanNeural"
audio_config = speech_sdk.audio.AudioOutputConfig(use_default_speaker=True)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config)

# Synthesize spoken output
speak = speech_synthesizer.speak_text_async(response_text).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
```

> **Note**: The system default speaker is the default audio output, so you could also just omit the AudioConfig altogether!

More information

For more information about using the **Speech-to-text** and **Text-to-speech** APIs, see the Speech-to-text documentation and Text-to-speech documentation.

Translate Speech

Azure AI Speech includes a speech translation API that you can use to translate spoken language. For example, suppose you want to develop a translator application that people can use when traveling in places where they don't speak the local language. They would be able to say phrases such as "Where is the station?" or "I need to find a pharmacy" in their own language, and have it translate them to the local language. In this exercise, you'll use the Azure AI Speech SDK for Python to create a simple application based on this example.

While this exercise is based on Python, you can develop speech translation applications using multiple language-specific SDKs; including:

- Azure AI Speech SDK for Python
- Azure AI Speech SDK for .NET
- Azure AI Speech SDK for JavaScript

This exercise takes approximately **30** minutes.

> **NOTE** This exercise is designed to be completed in the Azure cloud shell, where direct access to your computer's sound hardware is not supported. The lab will therefore use audio files for speech input and output streams. The code to achieve the same results using a mic and speaker is provided for your reference.

Create an Azure AI Speech resource

Let's start by creating an Azure AI Speech resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search field, search for **Speech service**. Select it from the list, then select **Create**.
3. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Prepare to develop an app in Cloud Shell

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

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

    Code Copy

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the code files:

Code Copy

```
cd mslearn-ai-language/Labfiles/08-speech-translation/Python/translator
```

5. In the command line pane, run the following command to view the code files in the **translator** folder:

Code Copy

```
ls -a -l
```

The files include a configuration file (**.env**) and a code file (**translator.py**).

6. Create a Python virtual environment and install the Azure AI Speech SDK package and other required packages by running the following command:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-speech==1.42.0
```

7. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

The file is opened in a code editor.

8. Update the configuration values to include the **region** and a **key** from the Azure AI Speech resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to use the Azure AI Speech SDK

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code translator.py
```

2. At the top of the code file, under the existing namespace references, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Azure AI Speech SDK:

```
Code
Copy

# Import namespaces
from azure.core.credentials import AzureKeyCredential
import azure.cognitiveservices.speech as speech_sdk
```

3. In the **main** function, under the comment **Get config settings**, note that the code loads the key and region you defined in the configuration file.

4. Find the following code under the comment **Configure translation**, and add the following code to configure your connection to the Azure AI Services Speech endpoint:

```
Code
Copy

# Configure translation
translation_config = speech_sdk.translation.SpeechTranslationConfig(speech_key,
speech_region)
translation_config.speech_recognition_language = 'en-US'
translation_config.add_target_language('fr')
translation_config.add_target_language('es')
translation_config.add_target_language('hi')
print('Ready to translate from',translation_config.speech_recognition_language)
```

5. You will use the **SpeechTranslationConfig** to translate speech into text, but you will also use a **SpeechConfig** to synthesize translations into speech. Add the following code under the comment **Configure speech**:

```
Code
Copy

# Configure speech
speech_config = speech_sdk.SpeechConfig(speech_key, speech_region)
print('Ready to use speech service in:', speech_config.region)
```

6. Save your changes (*CTRL+S*), but leave the code editor open.

Run the app

So far, the app doesn't do anything other than connect to your Azure AI Speech resource, but it's useful to run it and check that it works before adding speech functionality.

1. In the command line, enter the following command to run the translator app:

```
Code
Copy

python translator.py
```

The code should display the region of the speech service resource the application will use, a message that it is ready to translate from en-US and prompt you for a target language. A successful run indicates that the app has connected to your Azure AI Speech service. Press ENTER to end the program.

Implement speech translation

Now that you have a **SpeechTranslationConfig** for the Azure AI Speech service, you can use the Azure AI Speech translation API to recognize and translate speech.

1. In the code file, note that the code uses the **Translate** function to translate spoken input. Then in the **Translate** function, under the comment **Translate speech**, add the following code to create a **TranslationRecognizer** client that can be used to recognize and translate speech from a file.

```
# Translate speech
current_dir = os.getcwd()
audioFile = current_dir + '/station.wav'
audio_config_in = speech_sdk.AudioConfig(filename=audioFile)
translator = speech_sdk.translation.TranslationRecognizer(translation_config, audio_config =
audio_config_in)
print("Getting speech from file...")
result = translator.recognize_once_async().get()
print('Translating "{}"'.format(result.text))
translation = result.translations[targetLanguage]
print(translation)
```

2. Save your changes (*CTRL+S*), and re-run the program:

```
python translator.py
```

3. When prompted, enter a valid language code (*fr*, *es*, or *hi*). The program should transcribe your input file and translate it to the language you specified (French, Spanish, or Hindi). Repeat this process, trying each language supported by the application.

> **NOTE**: The translation to Hindi may not always be displayed correctly in the Console window due to character encoding issues.

4. When you're finished, press ENTER to end the program.

> **NOTE**: The code in your application translates the input to all three languages in a single call. Only the translation for the specific language is displayed, but you could retrieve any of the translations by specifying the target language code in the **translations** collection of the result.

Synthesize the translation to speech

So far, your application translates spoken input to text; which might be sufficient if you need to ask someone for help while traveling. However, it would be better to have the translation spoken aloud in a suitable voice.

> **Note**: Due to the hardware limitations of the cloud shell, we'll direct the synthesized speech output to a file.

1. In the **Translate** function, find the comment **Synthesize translation**, and add the following code to use a **SpeechSynthesizer** client to synthesize the translation as speech and save it as a .wav file:

```
# Synthesize translation
output_file = "output.wav"
voices = {
        "fr": "fr-FR-HenriNeural",
        "es": "es-ES-ElviraNeural",
        "hi": "hi-IN-MadhurNeural"
}
speech_config.speech_synthesis_voice_name = voices.get(targetLanguage)
audio_config_out = speech_sdk.audio.AudioConfig(filename=output_file)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config_out)
speak = speech_synthesizer.speak_text_async(translation).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
else:
    print("Spoken output saved in " + output_file)
```

2. Save your changes (*CTRL+S*), and re-run the program:

Code Copy

```
python translator.py
```

3. Review the output from the application, which should indicate that the spoken output translation was saved in a file. When you're finished, press **ENTER** to end the program.

4. If you have a media player capable of playing .wav audio files, download the file that was generated by entering the following command:

Code Copy

```
download ./output.wav
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file.

> **NOTE** *In this example, you've used a **SpeechTranslationConfig** to translate speech to text, and then used a **SpeechConfig** to synthesize the translation as speech. You can in fact use the **SpeechTranslationConfig** to synthesize the translation directly, but this only works when translating to a single language, and results in an audio stream that is typically saved as a file.*

Clean up resources

If you're finished exploring the Azure AI Speech service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Speech resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

What if you have a mic and speaker?

In this exercise, the Azure Cloud Shell environment we used doesn't support audio hardware, so you used audio files for the speech input and output. Let's see how the code can be modified to use audio hardware if you have it available.

Using speech translation with a microphone

1. If you have a mic, you can use the following code to capture spoken input for speech translation:

Code Copy

```
# Translate speech
audio_config_in = speech_sdk.AudioConfig(use_default_microphone=True)
translator = speech_sdk.translation.TranslationRecognizer(translation_config, audio_config =
audio_config_in)
print("Speak now...")
result = translator.recognize_once_async().get()
print('Translating "{}"'.format(result.text))
translation = result.translations[targetLanguage]
print(translation)
```

> **Note**: The system default microphone is the default audio input, so you could also just omit the AudioConfig altogether!

Using speech synthesis with a speaker

1. If you have a speaker, you can use the following code to synthesize speech.

Code Copy

```
# Synthesize translation
voices = {
        "fr": "fr-FR-HenriNeural",
        "es": "es-ES-ElviraNeural",
        "hi": "hi-IN-MadhurNeural"
}
speech_config.speech_synthesis_voice_name = voices.get(targetLanguage)
audio_config_out = speech_sdk.audio.AudioOutputConfig(use_default_speaker=True)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config_out)
speak = speech_synthesizer.speak_text_async(translation).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
```

> **Note**: The system default speaker is the default audio output, so you could also just omit the AudioConfig altogether!

More information

For more information about using the Azure AI Speech translation API, see the Speech translation documentation.

Develop an audio-enabled chat app

In this exercise, you use the *Phi-4-multimodal-instruct* generative AI model to generate responses to prompts that include audio files. You'll develop an app that provides AI assistance for a produce supplier company by using Azure AI Foundry and the Python OpenAI SDK to summarize voice messages left by customers.

While this exercise is based on Python, you can develop similar applications using multiple language-specific SDKs; including:

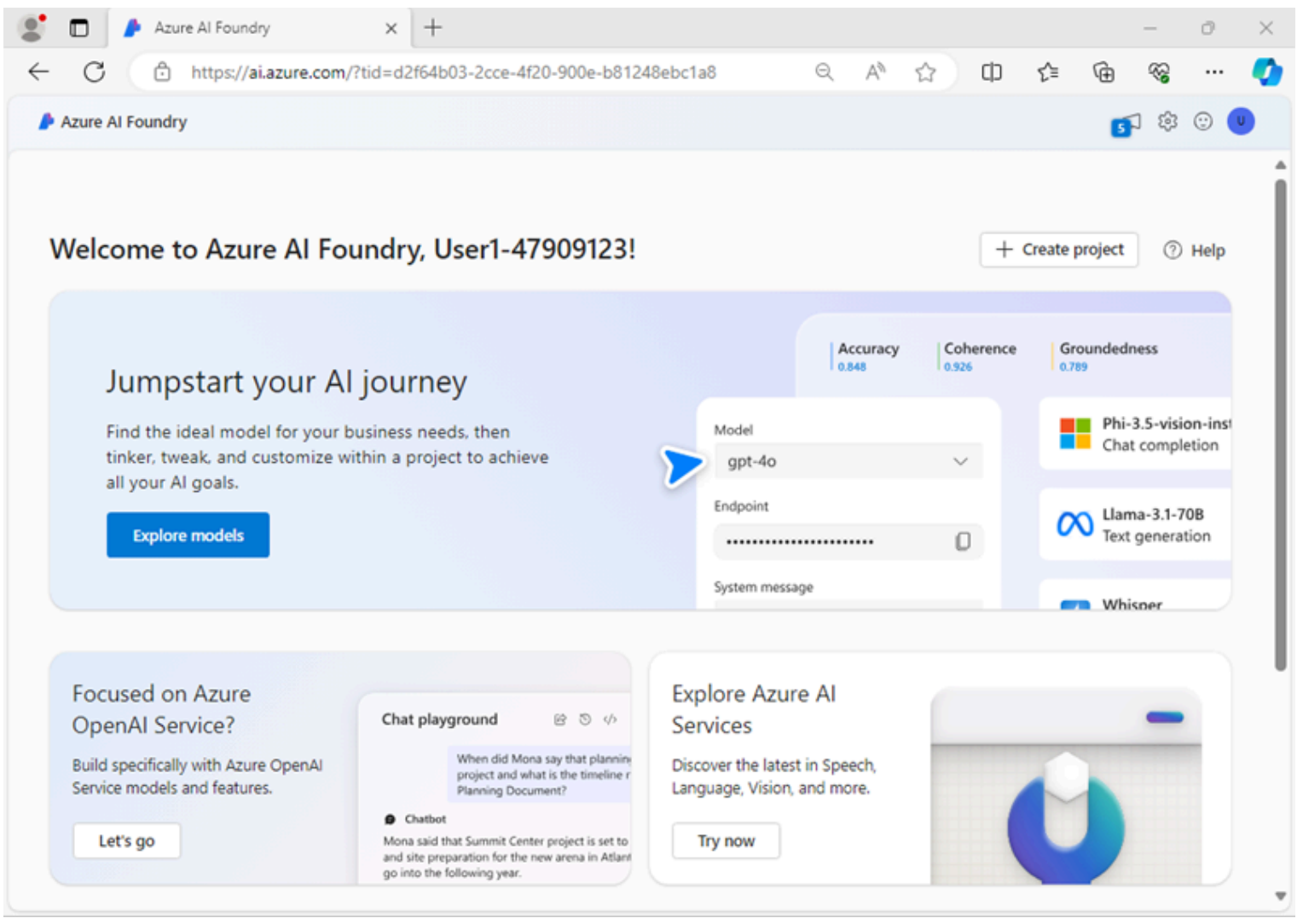
- Azure AI Projects for Python
- OpenAI library for Python
- Azure AI Projects for Microsoft .NET
- Azure OpenAI client library for Microsoft .NET
- Azure AI Projects for JavaScript
- Azure OpenAI library for TypeScript

This exercise takes approximately **30** minutes.

Create an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image:



2. In the home page, in the **Explore models and capabilities** section, search for the `Phi-4-multimodal-instruct` model; which we'll use in our project.
3. In the search results, select the **Phi-4-multimodal-instruct** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Select **Customize** and specify the following settings for your hub:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*

- **Subscription**: *Your Azure subscription*
- **Resource group**: *Create or select a resource group*
- **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region. You can check the latest regional availability for specific models in the Azure AI Foundry documentation

6. Select **Create** and wait for your project to be created.

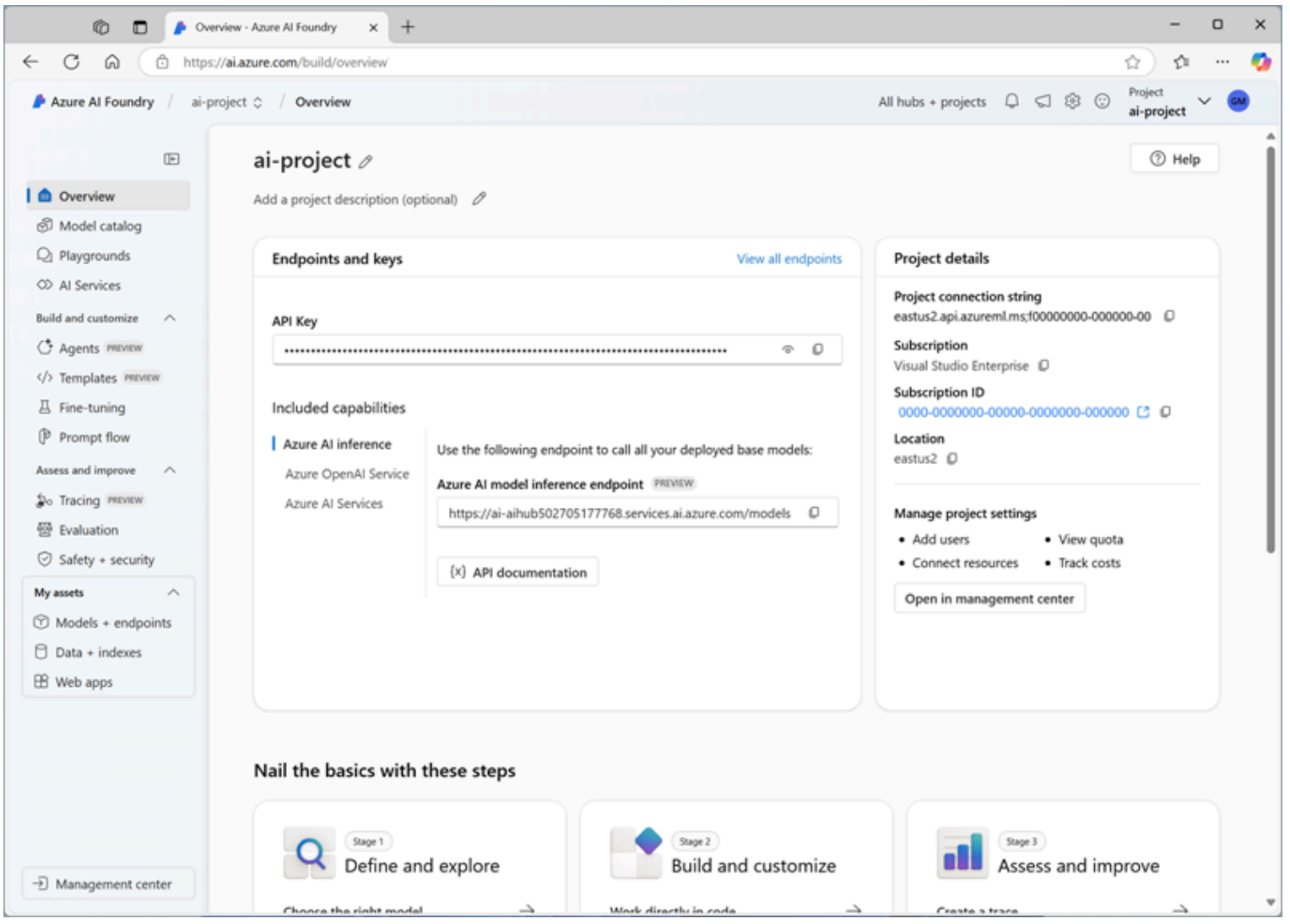
   It may take a few moments for the operation to complete.

7. Select **Agree and Proceed** to agree to the model terms, then select **Deploy** to complete the Phi model deployment.

8. When your project is created, the model details will be opened automatically. Note the name of your model deployment; which should be **Phi-4-multimodal-instruct**

9. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:

> **Note**: If an *Insufficient permissions** error is displayed, use the **Fix me** button to resolve it.



Create a client application

Now that you deployed a model, you can use the Azure AI Foundry and Azure AI Model Inference SDKs to develop an application that chats with it.

> **Tip**: You can choose to develop your solution using Python or Microsoft C#. Follow the instructions in the appropriate section for your chosen language.

Prepare the application configuration

1. In the Azure AI Foundry portal, view the **Overview** page for your project.

2. In the **Project details** area, note the **Azure AI Foundry project endpoint**. You'll use this endpoint to connect to your project in a client application.

3. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

4. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

5. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

6. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-audio -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

7. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-language/Labfiles/09-audio-chat/Python
   ```

8. In the cloud shell command line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-identity azure-ai-projects openai
   ```

9. Enter the following command to edit the configuration file that has been provided:

   Code Copy

   ```
   code .env
   ```

   The file should open in a code editor.

10. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal), and the **your_model_deployment** placeholder with the name you assigned to your Phi-4-multimodal-instruct model deployment.

11. After you replace the placeholders, in the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Write code to connect to your project and get a chat client for your model

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file:

```
code audio-chat.py
```

2. In the code file, note the existing statements that have been added at the top of the file to import the necessary SDK namespaces. Then, Find the comment **Add references**, add the following code to reference the namespaces in the libraries you installed previously:

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient
```

3. In the **main** function, under the comment **Get configuration settings**, note that the code loads the project connection string and model deployment name values you defined in the configuration file.

4. Find the comment **Initialize the project client** and add the following code to connect to your Azure AI Foundry project:

> **Tip**: Be careful to maintain the correct indentation level for your code.

```
# Initialize the project client
project_client = AIProjectClient(
   credential=DefaultAzureCredential(
       exclude_environment_credential=True,
       exclude_managed_identity_credential=True
   ),
   endpoint=project_endpoint,
)
```

5. Find the comment **Get a chat client**, add the following code to create a client object for chatting with your model:

```
# Get a chat client
openai_client = project_client.get_openai_client(api_version="2024-10-21")
```

Write code to submit an audio-based prompt

Before submitting the prompt, we need to encode the audio file for the request. Then we can attach the audio data to the user's message with a prompt for the LLM. Note that the code includes a loop to allow the user to input a prompt until they enter "quit".

1. Under the comment **Encode the audio file**, enter the following code to prepare the following audio file:

Code Copy

```
# Encode the audio file
file_path = "https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/Labfiles/09-audio-chat/data/avocados.mp3"
response = requests.get(file_path)
response.raise_for_status()
audio_data = base64.b64encode(response.content).decode('utf-8')
```

2. Under the comment **Get a response to audio input**, add the following code to submit a prompt:

Code Copy

```
# Get a response to audio input
response = openai_client.chat.completions.create(
   model=model_deployment,
   messages=[
       {"role": "system", "content": system_message},
       { "role": "user",
           "content": [
           {
               "type": "text",
               "text": prompt
           },
           {
               "type": "input_audio",
               "input_audio": {
                   "data": audio_data,
                   "format": "mp3"
               }
           }
       ] }
   ]
)
print(response.choices[0].message.content)
```

3. Use the **CTRL+S** command to save your changes to the code file. You can also close the code editor (**CTRL+Q**) if you like.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *--tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. In the cloud shell command-line pane, enter the following command to run the app:

```
python audio-chat.py
```

4. When prompted, enter the prompt

```
Can you summarize this customer's voice message?
```

5. Review the response.

Use a different audio file

1. In the code editor for your app code, find the code you added previously under the comment **Encode the audio file**. Then modify the file path url as follows to use a different audio file for the request (leaving the existing code after the file path):

```
# Encode the audio file
file_path = "https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/Labfiles/09-audio-chat/data/fresas.mp3"
```

The new file sounds like this:

2. Use the **CTRL+S** command to save your changes to the code file. You can also close the code editor (**CTRL+Q**) if you like.
3. In the cloud shell command line pane beneath the code editor, enter the following command to run the app:

```
python audio-chat.py
```

4. When prompted, enter the following prompt:

```
Can you summarize this customer's voice message? Is it time-sensitive?
```

5. Review the response. Then enter `quit` to exit the program.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

6. You can continue to run the app, choosing different prompt types and trying different prompts. When you're finished, enter `quit` to exit the program.

   If you have time, you can modify the code to use a different system prompt and your own internet-accessible audio files.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

Summary

In this exercise, you used Azure AI Foundry and the Azure AI Inference SDK to create a client application uses a multimodal model to generate responses to audio.

Clean up

If you've finished exploring Azure AI Foundry, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Develop an Azure AI Voice Live voice agent

In this exercise, you complete a Flask-based Python web app based that enables real-time voice interactions with an agent. You add the code to initialize the session, and handle session events. You use a deployment script that: deploys the AI model; creates an image of the app in Azure Container Registry (ACR) using ACR tasks; and then creates an Azure App Service instance that pulls the the image. To test the app you will need an audio device with microphone and speaker capabilities.

While this exercise is based on Python, you can develop similar applications other language-specific SDKs; including:

- Azure VoiceLive client library for .NET

Tasks performed in this exercise:

- Download the base files for the app
- Add code to complete the web app
- Review the overall code base
- Update and run the deployment script
- View and test the application

This exercise takes approximately **30** minutes to complete.

Launch the Azure Cloud Shell and download the files

In this section of the exercise you download the a zipped file containing the base files for the app.

1. In your browser navigate to the Azure portal https://portal.azure.com; signing in with your Azure credentials if prompted.

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

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

4. Run the following command in the **Bash** shell to download and unzip the exercise files. The second command will also change to the directory for the exercise files.

Code Copy

```
wget https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/downloads/python/voice-live-web.zip
```

Code Copy

```
unzip voice-live-web.zip && cd voice-live-web
```

Add code to complete the web app

Now that the exercise files are downloaded, the the next step is to add code to complete the application. The following steps are performed in the cloud shell.

> **Tip:** Resize the cloud shell to display more information, and code, by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

Run the following command to change into the *src* directory before you continue with the exercise.

Code Copy

```
cd src
```

Add code to implement the voice live assistant

In this section you add code to implement the voice live assistant. The **__init__** method initializes the voice assistant by storing the Azure VoiceLive connection parameters (endpoint, credentials, model, voice, and system instructions) and setting up runtime state variables to manage the connection lifecycle and handle user interruptions during conversations. The **start** method imports the necessary Azure VoiceLive SDK components that will be used to establish the WebSocket connection and configure the real-time voice session.

1. Run the following command to open the *flask_app.py* file for editing.

   Code Copy

   ```
   code flask_app.py
   ```

2. Search for the **# BEGIN VOICE LIVE ASSISTANT IMPLEMENTATION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

   Code Copy

```
def __init__(
    self,
    endpoint: str,
    credential,
    model: str,
    voice: str,
    instructions: str,
    state_callback=None,
):
    # Store Azure Voice Live connection and configuration parameters
    self.endpoint = endpoint
    self.credential = credential
    self.model = model
    self.voice = voice
    self.instructions = instructions

    # Initialize runtime state - connection established in start()
    self.connection = None
    self._response_cancelled = False  # Used to handle user interruptions
    self._stopping = False  # Signals graceful shutdown
    self.state_callback = state_callback or (lambda *_: None)

async def start(self):
    # Import Voice Live SDK components needed for establishing connection and configuring
session
    from azure.ai.voicelive.aio import connect  # type: ignore
    from azure.ai.voicelive.models import (
        RequestSession,
        ServerVad,
        AzureStandardVoice,
        Modality,
        InputAudioFormat,
        OutputAudioFormat,
    )  # type: ignore
```

3. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to implement the voice live assistant

In this section you add code to configure the voice live session. This specifies the modalities (audio-only is not supported by the API), the system instructions that define the assistant's behavior, the Azure TTS voice for responses, the audio format for both input and output streams, and Server-side Voice Activity Detection (VAD) which specifies how the model detects when users start and stop speaking.

1. Search for the **# BEGIN CONFIGURE VOICE LIVE SESSION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

Code Copy

```
# Configure VoiceLive session with audio/text modalities and voice activity detection
session_config = RequestSession(
    modalities=[Modality.TEXT, Modality.AUDIO],
    instructions=self.instructions,
    voice=voice_cfg,
    input_audio_format=InputAudioFormat.PCM16,
    output_audio_format=OutputAudioFormat.PCM16,
    turn_detection=ServerVad(threshold=0.5, prefix_padding_ms=300, silence_duration_ms=500),
)
await conn.session.update(session=session_config)
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to handle session events

In this section you add code to add event handlers for the voice live session. The event handlers respond to key VoiceLive session events during the conversation lifecycle: **_handle_session_updated** signals when the session is ready for user input, **_handle_speech_started** detects when the user begins speaking and implements interruption logic by stopping any ongoing assistant audio playback and canceling in-progress responses to allow natural conversation flow, and **_handle_speech_stopped** handles when the user has finished speaking and the assistant begins processing the input.

1. Search for the **# BEGIN HANDLE SESSION EVENTS - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment, be sure to check the indentation.

Code Copy

```
async def _handle_event(self, event, conn, verbose=False):
    """Handle Voice Live events with clear separation by event type."""
    # Import event types for processing different Voice Live server events
    from azure.ai.voicelive.models import ServerEventType

    event_type = event.type
    if verbose:
        _broadcast({"type": "log", "level": "debug", "event_type": str(event_type)})

    # Route Voice Live server events to appropriate handlers
    if event_type == ServerEventType.SESSION_UPDATED:
        await self._handle_session_updated()
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STARTED:
        await self._handle_speech_started(conn)
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STOPPED:
        await self._handle_speech_stopped()
    elif event_type == ServerEventType.RESPONSE_AUDIO_DELTA:
        await self._handle_audio_delta(event)
    elif event_type == ServerEventType.RESPONSE_AUDIO_DONE:
        await self._handle_audio_done()
    elif event_type == ServerEventType.RESPONSE_DONE:
        # Reset cancellation flag but don't change state - _handle_audio_done already did
        self._response_cancelled = False
    elif event_type == ServerEventType.ERROR:
        await self._handle_error(event)

async def _handle_session_updated(self):
    """Session is ready for conversation."""
    self.state_callback("ready", "Session ready. You can start speaking now.")

async def _handle_speech_started(self, conn):
    """User started speaking - handle interruption if needed."""
    self.state_callback("listening", "Listening… speak now")

    try:
        # Stop any ongoing audio playback on the client side
        _broadcast({"type": "control", "action": "stop_playback"})

        # If assistant is currently speaking or processing, cancel the response to allow
interruption
        current_state = assistant_state.get("state")
        if current_state in {"assistant_speaking", "processing"}:
            self._response_cancelled = True
            await conn.response.cancel()
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"Interrupted assistant during {current_state}"})
        else:
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"User speaking during {current_state} - no cancellation
needed"})
    except Exception as e:
        _broadcast({"type": "log", "level": "debug",
                    "msg": f"Exception in speech handler: {e}"})

async def _handle_speech_stopped(self):
    """User stopped speaking - processing input."""
    self.state_callback("processing", "Processing your input…")
```

```
async def _handle_audio_delta(self, event):
    """Stream assistant audio to clients."""
    if self._response_cancelled:
        return  # Skip cancelled responses

    # Update state when assistant starts speaking
    if assistant_state.get("state") != "assistant_speaking":
        self.state_callback("assistant_speaking", "Assistant speaking…")

    # Extract and broadcast Voice Live audio delta as base64 to WebSocket clients
    audio_data = getattr(event, "delta", None)
    if audio_data:
        audio_b64 = base64.b64encode(audio_data).decode("utf-8")
        _broadcast({"type": "audio", "audio": audio_b64})

async def _handle_audio_done(self):
    """Assistant finished speaking."""
    self._response_cancelled = False
    self.state_callback("ready", "Assistant finished. You can speak again.")

async def _handle_error(self, event):
    """Handle Voice Live errors."""
    error = getattr(event, "error", None)
    message = getattr(error, "message", "Unknown error") if error else "Unknown error"
    self.state_callback("error", f"Error: {message}")

def request_stop(self):
    self._stopping = True
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Review the code in the app

So far, you've added code to the app to implement the agent and handle agent events. Take a few minutes to review the full code and comments to get a better understanding of how the app is handling client state and operations.

1. When you're finished enter **ctrl+q** to exit out of the editor.

Update and run the deployment script

In this section you make a small change to the **azdeploy.sh** deployment script and then run the deployment.

Update the deployment script

There are only two values you should change at the top of the **azdeploy.sh** deployment script.

- The **rg** value specifies the resource group to contain the deployment. You can accept the default value, or enter your own value if you need to deploy to a specific resource group.
- The **location** value sets the region for the deployment. The *gpt-4o* model used in the exercise can be deployed to other regions, but there can be limits in any particular region. If the deployment fails in your chosen region, try **eastus2** or **swedencentral**.

Code Copy

```
rg="rg-voicelive" # Replace with your resource group
location="eastus2" # Or a location near you
```

1. Run the following commands in the Cloud Shell to begin editing the deployment script.

```
cd ~/voice-live-web
```

```
code azdeploy.sh
```

2. Update the values for **rg** and **location** to meet your needs and then enter **ctrl+s** to save your changes and **ctrl+q** to exit the editor.

Run the deployment script

The deployment script deploys the AI model and creates the necessary resources in Azure to run a containerized app in App Service.

1. Run the following command in the Cloud Shell to begin deploying the Azure resources and the application.

```
bash azdeploy.sh
```

2. Select **option 1** for the initial deployment.

   The deployment should complete in 5-10 minutes. During the deployment you might be prompted for the following information/actions:

   - If you are prompted to authenticate to Azure follow the directions presented to you.
   - If you are prompted to select a subscription use the arrow keys to highlight your subscription and press **Enter**.
   - You will likely see some warnings during deployment and these can be ignored.
   - If the deployment fails during the AI model deployment change the region in the deployment script and try again.
   - Regions in Azure can get busy at times and interrupt the timing of the deployments. If the deployment fails after the model deployment re-run the deployment script.

View and test the app

When the deployment completes a "Deployment complete!" message will be in the shell along with a link to the web app. You can select that link, or navigate to the App Service resource and launch the app from there. It can take a few minutes for the application to load.

1. Select the **Start session** button to connect to the model.
2. You will be prompted to give the application access to you audio devices.
3. Begin talking to the model when the app prompts you to start speaking.

Troubleshooting:

- If the app reports missing environment variables, restart the application in App Service.
- If you see excessive *audio chunk* messages in the log shown in the application select **Stop session** and then start the session again.
- If the app fails to function at all, double-check you added all of the code and for proper indentation. If you need to make any changes re-run the deployment and select **option 2** to only update the image.

Clean up resources

Run the following command in the Cloud Shell to remove all of the resources deployed for this exercise. You will be prompted to confirm the resource deletion.

Code Copy

```
azd down --purge
```

Analyze images

Azure AI Vision is an artificial intelligence capability that enables software systems to interpret visual input by analyzing images. In Microsoft Azure, the **Vision** Azure AI service provides pre-built models for common computer vision tasks, including analysis of images to suggest captions and tags, detection of common objects and people. You can also use the Azure AI Vision service to remove the background or create a foreground matting of images.

> **Note**: This exercise is based on pre-release SDK software, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

While this exercise is based on the Azure Vision Python SDK, you can develop vision applications using multiple language-specific SDKs; including:

- Azure AI Vision Analysis for JavaScript
- Azure AI Vision Analysis for Microsoft .NET
- Azure AI Vision Analysis for Java

This exercise takes approximately **30** minutes.

Provision an Azure AI Vision resource

If you don't already have one in your subscription, you'll need to provision an Azure AI Vision resource.

> **Note**: In this exercise, you'll use a standalone **Computer Vision** resource. You can also use Azure AI Vision services in an *Azure AI Services* multi-service resource, either directly or in an *Azure AI Foundry* project.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Close any welcome messages or tips that are displayed.
2. Select **Create a resource**.
3. In the search bar, search for `Computer Vision`, select **Computer Vision**, and create the resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Choose from **East US**, **West US**, **France Central**, **Korea Central**, **North Europe**, **Southeast Asia**, **West Europe**, or **East Asia****
    - **Name**: *A valid name for your Computer Vision resource*
    - **Pricing tier**: Free F0

    *Azure AI Vision 4.0 full feature sets are currently only available in these regions.
4. Select the required checkboxes and create the resource.
5. Wait for deployment to complete, and then view the deployment details.
6. When the resource has been deployed, go to it and under the **Resource management** node in the navigation pane, view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

Develop an image analysis app with the Azure AI Vision SDK

In this exercise, you'll complete a partially implemented client application that uses the Azure AI Vision SDK to analyze images.

Prepare the application configuration

1. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. Resize the cloud shell pane so you can still see the **Keys and Endpoint** page for your Computer Vision resource.

   > **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-vision -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
   ```

   > **Tip**: As you paste commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, use the following command to navigate to and view the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-vision/Labfiles/analyze-images/python/image-analysis
   ls -a -l
   ```

   The folder contains application configuration and code files for your app. It also contains a **/images** subfolder, which contains some image files for your app to analyze.

6. Install the Azure AI Vision SDK package and other required packages by running the following commands:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-vision-imageanalysis==1.0.0
   ```

7. Enter the following command to edit the configuration file for your app:

   Code Copy

```
code .env
```

The file is opened in a code editor.

8. In the code file, update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your Computer Vision resource (copied from its **Keys and Endpoint** page in the Azure portal).
9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to suggest a caption

1. In the cloud shell command line, enter the following command to open the code file for the client application:

Code Copy

```
code image-analysis.py
```

> **Tip**: You might want to maximize the cloud shell pane and move the split-bar between the command line cosole and the code editor so you can see the code more easily.

2. In the code file, find the comment **Import namespaces**, and add the following code to import the namespaces you will need to use the Azure AI Vision SDK:

Code Copy

```
# import namespaces
from azure.ai.vision.imageanalysis import ImageAnalysisClient
from azure.ai.vision.imageanalysis.models import VisualFeatures
from azure.core.credentials import AzureKeyCredential
```

3. In the **Main** function, note that the code to load the configuration settings and determine the image file to be analyzed has been provided. Then find the comment **Authenticate Azure AI Vision client** and add the following code to create and authenticate a Azure AI Vision client object (be sure to maintain the correct indentation levels):

Code Copy

```
# Authenticate Azure AI Vision client
cv_client = ImageAnalysisClient(
     endpoint=ai_endpoint,
     credential=AzureKeyCredential(ai_key))
```

4. In the **Main** function, under the code you just added, find the comment **Analyze image** and add the following code:

Code Copy

```
# Analyze image
with open(image_file, "rb") as f:
     image_data = f.read()
print(f'\nAnalyzing {image_file}\n')

result = cv_client.analyze(
     image_data=image_data,
     visual_features=[
         VisualFeatures.CAPTION,
         VisualFeatures.DENSE_CAPTIONS,
         VisualFeatures.TAGS,
         VisualFeatures.OBJECTS,
         VisualFeatures.PEOPLE],
)
```

5. Find the comment **Get image captions**, add the following code to display image captions and dense captions:

Code Copy

```
# Get image captions
if result.caption is not None:
     print("\nCaption:")
     print(" Caption: '{}' (confidence: {:.2f}%)".format(result.caption.text,
result.caption.confidence * 100))

if result.dense_captions is not None:
     print("\nDense Captions:")
     for caption in result.dense_captions.list:
         print(" Caption: '{}' (confidence: {:.2f}%)".format(caption.text,
caption.confidence * 100))
```

6. Save your changes (*CTRL+S*) and resize the panes so you can clearly see the command line console while keeping the code editor open. Then enter the following command to run the program with the argument **images/street.jpg**:

Code Copy

```
python image-analysis.py images/street.jpg
```

7. Observe the output, which should include a suggested caption for the **street.jpg** image, which looks like this:

8. Run the program again, this time with the argument **images/building.jpg** to see the caption that gets generated for the **building.jpg** image, which looks like this:

9. Repeat the previous step to generate a caption for the **images/person.jpg** file, which looks like this:

Add code to generate suggested tags

It can sometimes be useful to identify relevant *tags* that provide clues about the contents of an image.

1. In the code editor, in the **AnalyzeImage** function, find the comment **Get image tags** and add the following code:

Code Copy

```
# Get image tags
if result.tags is not None:
     print("\nTags:")
     for tag in result.tags.list:
         print(" Tag: '{}' (confidence: {:.2f}%)".format(tag.name, tag.confidence * 100))
```

2. Save your changes (*CTRL+S*) and run the program with the argument **images/street.jpg**, observing that in addition to the image caption, a list of suggested tags is displayed.
3. Rerun the program for the **images/building.jpg** and **images/person.jpg** files.

Add code to detect and locate objects

1. In the code editor, in the **AnalyzeImage** function, find the comment **Get objects in the image** and add the following code to list the objects detected in the image, and call the provided function to annotate an image with the detected objects:

Code Copy

```
# Get objects in the image
if result.objects is not None:
     print("\nObjects in image:")
     for detected_object in result.objects.list:
         # Print object tag and confidence
         print(" {} (confidence: {:.2f}%)".format(detected_object.tags[0].name,
detected_object.tags[0].confidence * 100))
     # Annotate objects in the image
     show_objects(image_file, result.objects.list)
```

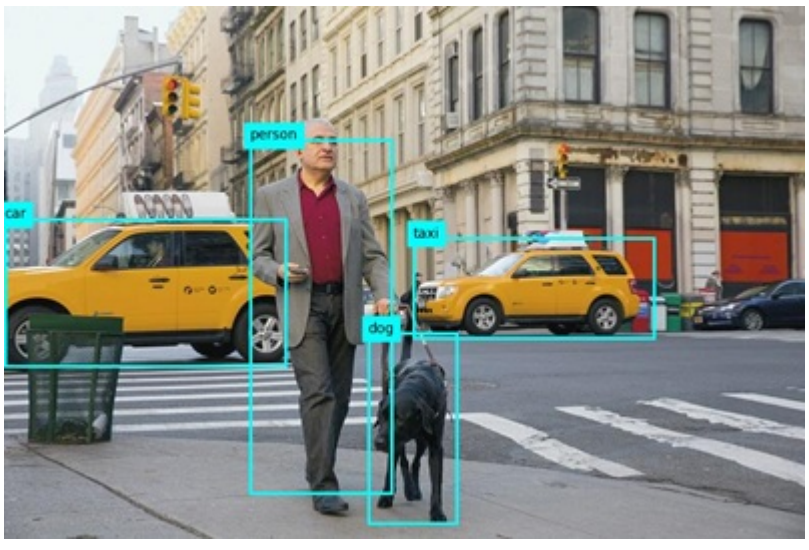
2. Save your changes (*CTRL+S*) and run the program with the argument **images/street.jpg**, observing that in addition to the image caption and suggested tags; a file named **objects.jpg** is generated.

3. Use the (Azure cloud shell-specific) **download** command to download the **objects.jpg** file:

Code Copy

```
download objects.jpg
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file. The image should look similar to this:



4. Rerun the program for the **images/building.jpg** and **images/person.jpg** files, downloading the generated objects.jpg file after each run.

Add code to detect and locate people

1. In the code editor, in the **AnalyzeImage** function, find the comment **Get people in the image** and add the following code to list any detected people with a confidence level of 20% or more, and call a provided function to annotate them in an image:

Code Copy

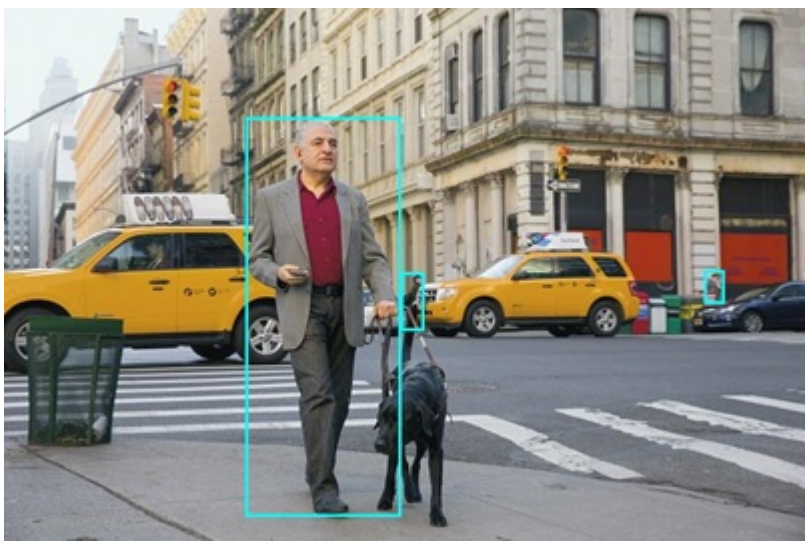
```
# Get people in the image
if result.people is not None:
     print("\nPeople in image:")

     for detected_person in result.people.list:
         if detected_person.confidence > 0.2:
             # Print location and confidence of each person detected
             print(" {} (confidence: {:.2f}%)".format(detected_person.bounding_box,
detected_person.confidence * 100))
     # Annotate people in the image
     show_people(image_file, result.people.list)
```

2. Save your changes (*CTRL+S*) and run the program with the argument **images/street.jpg**, observing that in addition to the image caption, suggested tags, and objects.jpg file; a list of person locations and file named **people.jpg** is generated.

3. Use the (Azure cloud shell-specific) **download** command to download the **objects.jpg** file:

```
download people.jpg
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file. The image should look similar to this:



4. Rerun the program for the **images/building.jpg** and **images/person.jpg** files, downloading the generated people.jpg file after each run.

> **Tip:** If you see bounding boxes returned from the model that don't make sense, check the JSON confidence score and try increasing the confidence score filtering in your app.

Clean up resources

If you've finished exploring Azure AI Vision, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs:

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.

2. In the top search bar, search for *Computer Vision*, and select the Computer Vision resource you created in this lab.

3. On the resource page, select **Delete** and follow the instructions to delete the resource.

Read text in images

Optical character recognition (OCR) is a subset of computer vision that deals with reading text in images and documents. The **Azure AI Vision** Image Analysis service provides an API for reading text, which you'll explore in this exercise.

> **Note**: This exercise is based on pre-release SDK software, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

While this exercise is based on the Azure Vision Analysis Python SDK, you can develop vision applications using multiple language-specific SDKs; including:

- Azure AI Vision Analysis for JavaScript
- Azure AI Vision Analysis for Microsoft .NET
- Azure AI Vision Analysis for Java

This exercise takes approximately **30** minutes.

Provision an Azure AI Vision resource

If you don't already have one in your subscription, you'll need to provision an Azure AI Vision resource.

> **Note**: In this exercise, you'll use a standalone **Computer Vision** resource. You can also use Azure AI Vision services in an *Azure AI Services* multi-service resource, either directly or in an *Azure AI Foundry* project.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Close any welcome messages or tips that are displayed.
2. Select **Create a resource**.
3. In the search bar, search for `Computer Vision`, select **Computer Vision**, and create the resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Choose from **East US**, **West US**, **France Central**, **Korea Central**, **North Europe**, **Southeast Asia**, **West Europe**, or **East Asia****
    - **Name**: *A valid name for your Computer Vision resource*
    - **Pricing tier**: Free F0

    *Azure AI Vision 4.0 full feature sets are currently only available in these regions.
4. Select the required checkboxes and create the resource.
5. Wait for deployment to complete, and then view the deployment details.
6. When the resource has been deployed, go to it and under the **Resource management** node in the navigation pane, view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

Develop a text extraction app with the Azure AI Vision SDK

In this exercise, you'll complete a partially implemented client application that uses the Azure AI Vision SDK to extract text from images.

Prepare the application configuration

1. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. Resize the cloud shell pane so you can still see the **Keys and Endpoint** page for your Computer Vision resource.

   > **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-vision -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
   ```

   > **Tip**: As you paste commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, use the following command to navigate to the application code files:

   Code Copy

   ```
   cd mslearn-ai-vision/Labfiles/ocr/python/read-text
   ls -a -l
   ```

   The folder contains application configuration and code files for your app. It also contains an **/images** subfolder, which contains some image files for your app to analyze.

6. Install the Azure AI Vision SDK package and other required packages by running the following commands:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-vision-imageanalysis==1.0.0
   ```

7. Enter the following command to edit the configuration file for your app:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

8. In the code file, update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your Computer Vision resource (copied from its **Keys and Endpoint** page in the Azure portal).

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to read text from an image

1. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code read-text.py
```

> **Tip**: You might want to maximize the cloud shell pane and move the split-bar between the command line cosole and the code editor so you can see the code more easily.

2. In the code file, find the comment **Import namespaces**, and add the following code to import the namespaces you will need to use the Azure AI Vision SDK:

```
# import namespaces
from azure.ai.vision.imageanalysis import ImageAnalysisClient
from azure.ai.vision.imageanalysis.models import VisualFeatures
from azure.core.credentials import AzureKeyCredential
```

3. In the **Main** function, the code to load the configuration settings and determine the file to be analyzed has been provided. Then find the comment **Authenticate Azure AI Vision client** and add the following language-specific code to create and authenticate an Azure AI Vision Image Analysis client object:

```
# Authenticate Azure AI Vision client
cv_client = ImageAnalysisClient(
     endpoint=ai_endpoint,
     credential=AzureKeyCredential(ai_key))
```

4. In the **Main** function, under the code you just added, find the comment **Read text in image** and add the following code to use the Image Analysis client to read the text in the image:

```
# Read text in image
with open(image_file, "rb") as f:
     image_data = f.read()
print (f"\nReading text in {image_file}")

result = cv_client.analyze(
     image_data=image_data,
     visual_features=[VisualFeatures.READ])
```

5. Find the comment **Print the text** and add the following code (including the final comment) to print the lines of text that were found and call a function to annotate them in the image (using the **bounding_polygon** returned for each line of text):

```
# Print the text
if result.read is not None:
     print("\nText:")

     for line in result.read.blocks[0].lines:
         print(f" {line.text}")
     # Annotate the text in the image
     annotate_lines(image_file, result.read)

     # Find individual words in each line
```

6. Save your changes (*CTRL+S*) but keep the code editor open in case you need to fix any typo's.

7. Resize the panes so you can see more of the console, then enter the following command to run the program:

Code Copy

```
python read-text.py images/Lincoln.jpg
```

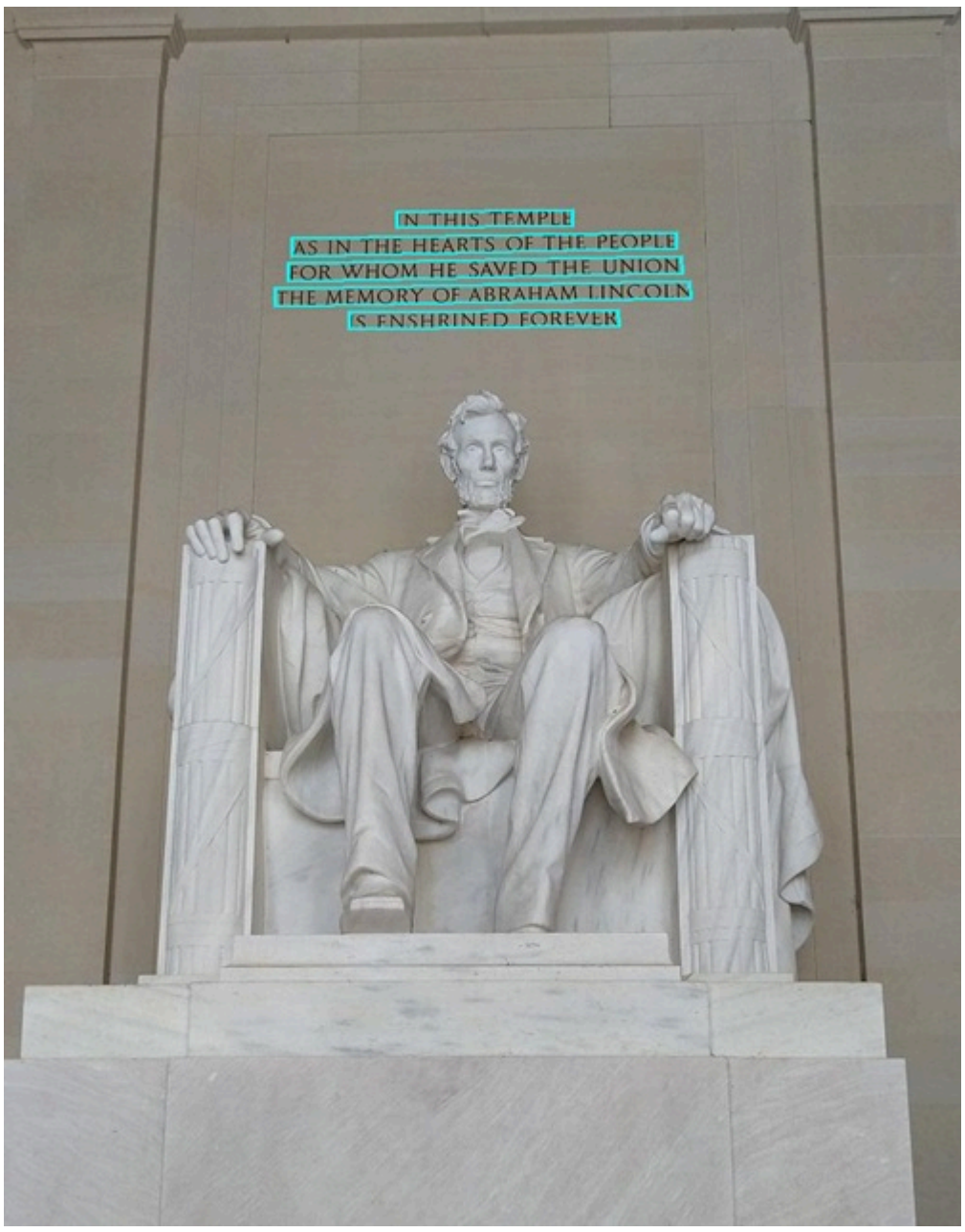
8. The program reads the text in the specified image file (*images/Lincoln.jpg*), which looks like this:

9. In the **read-text** folder, a **lines.jpg** image has been created. Use the (Azure cloud shell-specific) **download** command to download it:

Code Copy

```
download lines.jpg
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file. The image should look simlar to this:



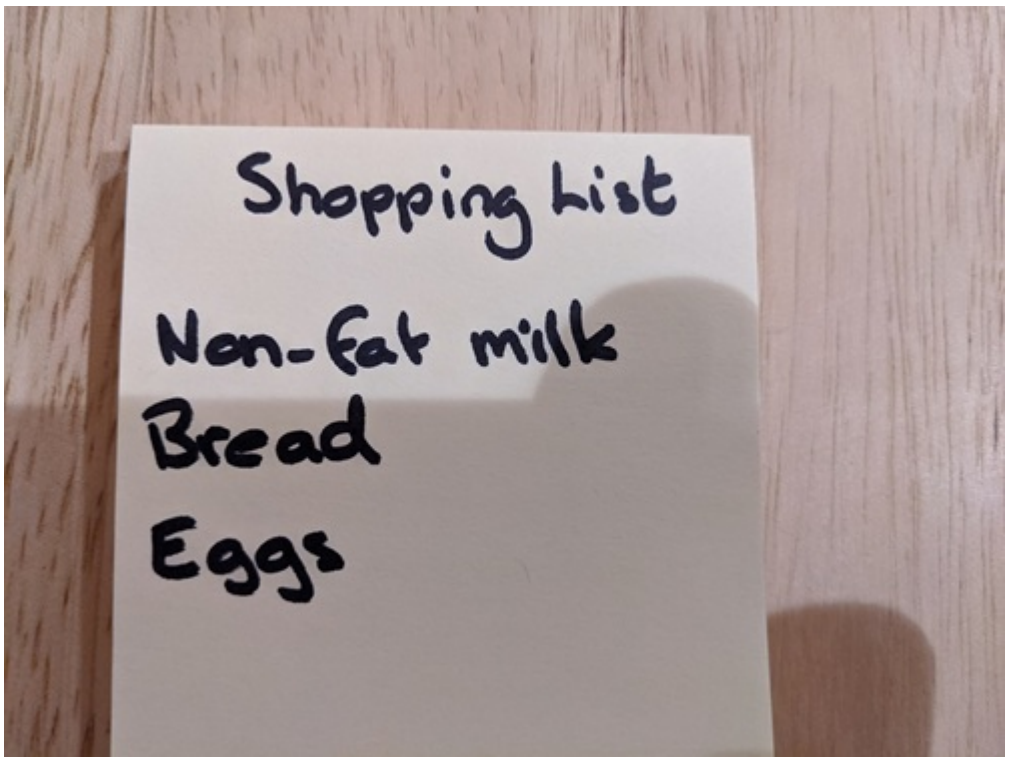
10. Run the program again, this time specifying the parameter *images/Business-card.jpg* to extract text from the following image:


```
python read-text.py images/Business-card.jpg
```

11. Download and view the resulting **lines.jpg** file:

```
download lines.jpg
```

12. Run the program one more time, this time specifying the parameter *images/Note.jpg* to extract text from this image:



Code Copy

```
python read-text.py images/Note.jpg
```

13. Download and view the resulting **lines.jpg** file:

Code Copy

```
download lines.jpg
```

Add code to return the position of individual words

1. Resize the panes so you can see more of the code file. Then find the comment **Find individual words in each line** and add the following code (being careful to maintain the correct indentation level):

Code Copy

```
# Find individual words in each line
print ("\nIndividual words:")
for line in result.read.blocks[0].lines:
     for word in line.words:
          print(f"  {word.text} (Confidence: {word.confidence:.2f}%)")
# Annotate the words in the image
annotate_words(image_file, result.read)
```

2. Save your changes (*CTRL+S*). Then, in the command line pane, rerun the program to extract text from *images/Lincoln.jpg*.
3. Observe the output, which should include each individual word in the image, and the confidence associated with their prediction.
4. In the **read-text** folder, a **words.jpg** image has been created. Use the (Azure cloud shell-specific) **download** command to download and view it:

Code Copy

```
download words.jpg
```

5. Rerun the program for *images/Business-card.jpg* and *images/Note.jpg*; viewing the **words.jpg** file generated for each image.

Clean up resources

If you've finished exploring Azure AI Vision, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs:

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search bar, search for *Computer Vision*, and select the Computer Vision resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

Detect and analyze faces

The ability to detect and analyze human faces is a core AI capability. In this exercise, you'll explore the **Face** service to work with faces.

> **Note**: This exercise is based on pre-release SDK software, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

While this exercise is based on the Azure Vision Face Python SDK, you can develop vision applications using multiple language-specific SDKs; including:

- Azure AI Vision Face for JavaScript
- Azure AI Vision Face for Microsoft .NET
- Azure AI Vision Face for Java

This exercise takes approximately **30** minutes.

> **Note**: Capabilities of Azure AI services that return personally identifiable information are restricted to customers who have been granted limited access. This exercise does not include facial recognition tasks, and can be completed without requesting any additional access to restricted features.

Provision an Azure AI Face API resource

If you don't already have one in your subscription, you'll need to provision an Azure AI Face API resource.

> **Note**: In this exercise, you'll use a standalone **Face** resource. You can also use Azure AI Face services in an *Azure AI Services* multi-service resource, either directly or in an *Azure AI Foundry* project.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Close any welcome messages or tips that are displayed.
2. Select **Create a resource**.
3. In the search bar, search for `Face`, select **Face**, and create the resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Choose any available region*
    - **Name**: *A valid name for your Face resource*
    - **Pricing tier**: Free F0
4. Create the resource and wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and under the **Resource management** node in the navigation pane, view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

Develop a facial analysis app with the Face SDK

In this exercise, you'll complete a partially implemented client application that uses the Azure Face SDK to detect and analyze human faces in images.

Prepare the application configuration

1. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. Resize the cloud shell pane so you can still see the **Keys and Endpoint** page for your Face resource.

   > **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-vision -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
   ```

   > **Tip**: As you paste commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, use the following command to navigate to the application code files:

   Code Copy

   ```
   cd mslearn-ai-vision/Labfiles/face/python/face-api
   ls -a -l
   ```

   The folder contains application configuration and code files for your app. It also contains an **/images** subfolder, which contains some image files for your app to analyze.

6. Install the Azure AI Vision SDK package and other required packages by running the following commands:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-vision-face==1.0.0b2
   ```

7. Enter the following command to edit the configuration file for your app:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

8. In the code file, update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your Face resource (copied from its **Keys and Endpoint** page in the Azure portal).
9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to create a Face API client

1. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code analyze-faces.py
```

> **Tip**: You might want to maximize the cloud shell pane and move the split-bar between the command line cosole and the code editor so you can see the code more easily.

2. In the code file, find the comment **Import namespaces**, and add the following code to import the namespaces you will need to use the Azure AI Vision SDK:

```
# Import namespaces
from azure.ai.vision.face import FaceClient
from azure.ai.vision.face.models import FaceDetectionModel, FaceRecognitionModel,
FaceAttributeTypeDetection01
from azure.core.credentials import AzureKeyCredential
```

3. In the **Main** function, note that the code to load the configuration settings and determine the image to be analyzed has been provided. Then find the comment **Authenticate Face client** and add the following code to create and authenticate a **FaceClient** object:

```
# Authenticate Face client
face_client = FaceClient(
     endpoint=cog_endpoint,
     credential=AzureKeyCredential(cog_key))
```

Add code to detect and analyze faces

1. In the code file for your application, in the **Main** function, find the comment **Specify facial features to be retrieved** and add the following code:

```
# Specify facial features to be retrieved
features = [FaceAttributeTypeDetection01.HEAD_POSE,
            FaceAttributeTypeDetection01.OCCLUSION,
            FaceAttributeTypeDetection01.ACCESSORIES]
```

2. In the **Main** function, under the code you just added, find the comment **Get faces** and add the following code to print the facial feature information and call a function that annotates the image with the bounding box for each detected face (based on the **face_rectangle** property of each face):

```
# Get faces
with open(image_file, mode="rb") as image_data:
     detected_faces = face_client.detect(
         image_content=image_data.read(),
         detection_model=FaceDetectionModel.DETECTION01,
         recognition_model=FaceRecognitionModel.RECOGNITION01,
         return_face_id=False,
         return_face_attributes=features,
     )

face_count = 0
if len(detected_faces) > 0:
     print(len(detected_faces), 'faces detected.')
     for face in detected_faces:

         # Get face properties
         face_count += 1
         print('\nFace number {}'.format(face_count))
         print(' - Head Pose (Yaw): {}'.format(face.face_attributes.head_pose.yaw))
         print(' - Head Pose (Pitch): {}'.format(face.face_attributes.head_pose.pitch))
         print(' - Head Pose (Roll): {}'.format(face.face_attributes.head_pose.roll))
         print(' - Forehead occluded?:
{}'.format(face.face_attributes.occlusion["foreheadOccluded"]))
         print(' - Eye occluded?: {}'.format(face.face_attributes.occlusion["eyeOccluded"]))
         print(' - Mouth occluded?:
{}'.format(face.face_attributes.occlusion["mouthOccluded"]))
         print(' - Accessories:')
         for accessory in face.face_attributes.accessories:
             print('    - {}'.format(accessory.type))
         # Annotate faces in the image
         annotate_faces(image_file, detected_faces)
```

3. Examine the code you added to the **Main** function. It analyzes an image file and detects any faces it contains, including attributes for head pose, occlusion, and the presence of accessories such as glasses. Additionally, a function is called to annotate the original image with a bounding box for each detected face.

4. Save your changes (*CTRL+S*) but keep the code editor open in case you need to fix any typo's.

5. Resize the panes so you can see more of the console, then enter the following command to run the program with the argument *images/face1.jpg*:

Code Copy

```
python analyze-faces.py images/face1.jpg
```

The app runs and analyzes the following image:

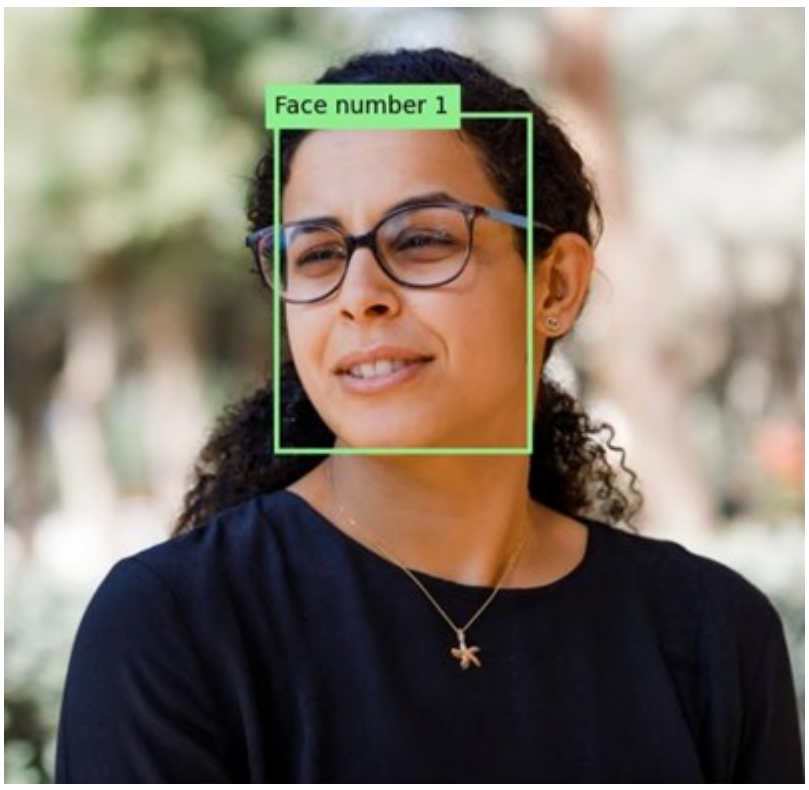
6. Observe the output, which should include the ID and attributes of each face detected.

7. Note that an image file named **detected_faces.jpg** is also generated. Use the (Azure cloud shell-specific) **download** command to download it:

Code Copy

```
download detected_faces.jpg
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file. The image should look simlar to this:



8. Run the program again, this time specifying the parameter *images/face2.jpg* to extract text from the following image:

Code Copy

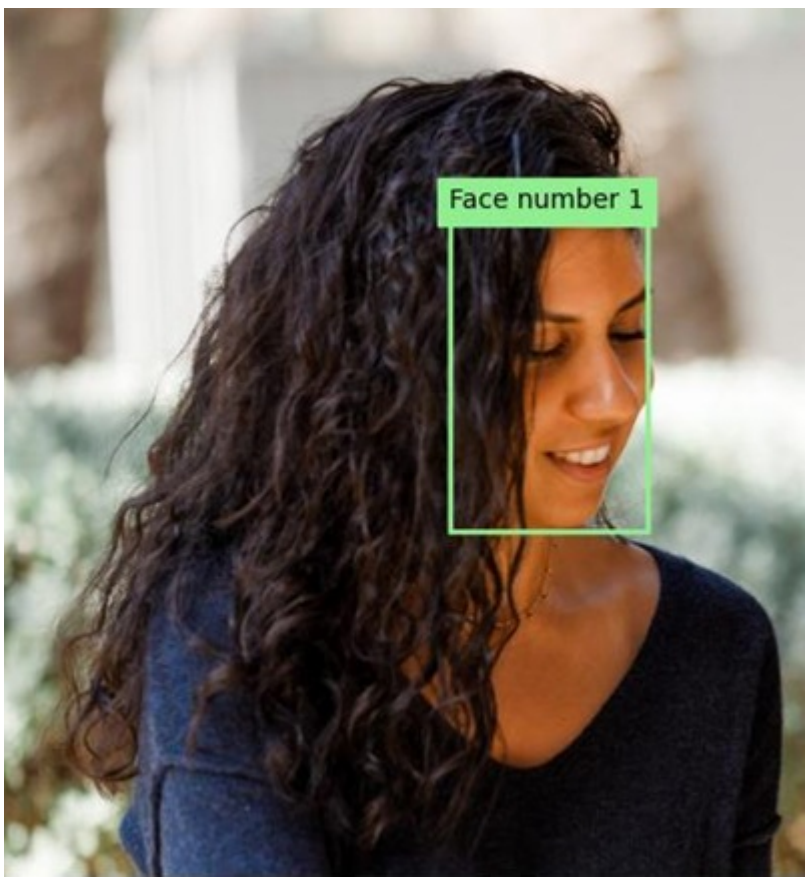
```
python analyze-faces.py images/face2.jpg
```

9. Download and view the resulting **detected_faces.jpg** file:

Code Copy

```
download detected_faces.jpg
```

The resulting image should look like this:



10. Run the program one more time, this time specifying the parameter *images/faces.jpg* to extract text from this image:

Code Copy

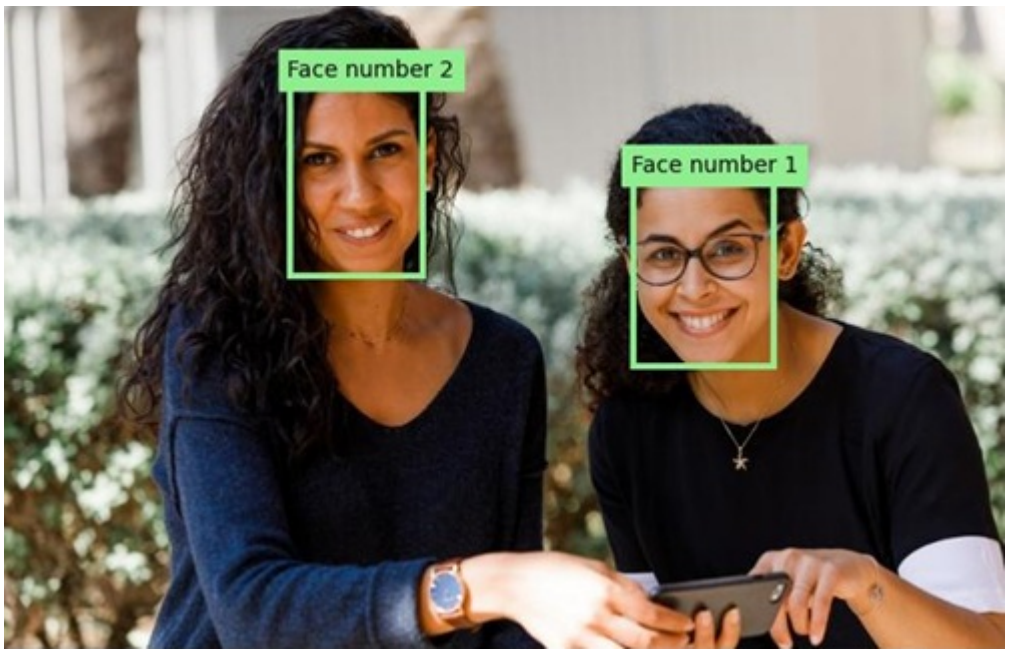
```
python analyze-faces.py images/faces.jpg
```

11. Download and view the resulting **detected_faces.jpg** file:

Code Copy

```
download detected_faces.jpg
```

The resulting image should look like this:



Clean up resources

If you've finished exploring Azure AI Vision, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs:

1. Open the Azure portal at `https://portal.azure.com`, and in the top search bar, search for the resources you created in this lab.
2. On the resource page, select **Delete** and follow the instructions to delete the resource. Alternatively, you can delete the entire resource group to clean up all resources at the same time.

Classify images

The **Azure AI Custom Vision** service enables you to create computer vision models that are trained on your own images. You can use it to train *image classification* and *object detection* models; which you can then publish and consume from applications.

In this exercise, you will use the Custom Vision service to train an image classification model that can identify three classes of fruit (apple, banana, and orange).

While this exercise is based on the Azure Custom Vision Python SDK, you can develop vision applications using multiple language-specific SDKs; including:

- Azure Custom Vision for JavaScript (training)
- Azure Custom Vision for JavaScript (prediction)
- Azure Custom Vision for Microsoft .NET (training)
- Azure Custom Vision for Microsoft .NET (prediction)
- Azure Custom Vision for Java (training)
- Azure Custom Vision for Java (prediction)

This exercise takes approximately **45** minutes.

Create Custom Vision resources

Before you can train a model, you will need Azure resources for *training* and *prediction*. You can create **Custom Vision** resources for each of these tasks, or you can create a single resource and use it for both. In this exercise, you'll create **Custom Vision** resources for training and prediction.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Close any welcome messages or tips that are displayed.
2. Select **Create a resource**.
3. In the search bar, search for `Custom Vision`, select **Custom Vision**, and create the resource with the following settings:
    - **Create options**: Both
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Choose any available region*
    - **Name**: *A valid name for your Custom Vision resource*
    - **Training pricing tier**: F0
    - **Prediction pricing tier**: F0
4. Create the resource and wait for deployment to complete, and then view the deployment details. Note that two Custom Vision resources are provisioned; one for training, and another for prediction.

    > **Note**: Each resource has its own *endpoint* and *keys*, which are used to manage access from your code. To train an image classification model, your code must use the *training* resource (with its endpoint and key); and to use the trained model to predict image classes, your code must use the *prediction* resource (with its endpoint and key).

5. When the resources have been deployed, go to the resource group to view them. You should see two custom vision resources, one with the suffix ***-Prediction***.

Create a Custom Vision project in the Custom Vision portal

To train an image classification model, you need to create a Custom Vision project based on your training resource. To do this, you'll use the Custom Vision portal.

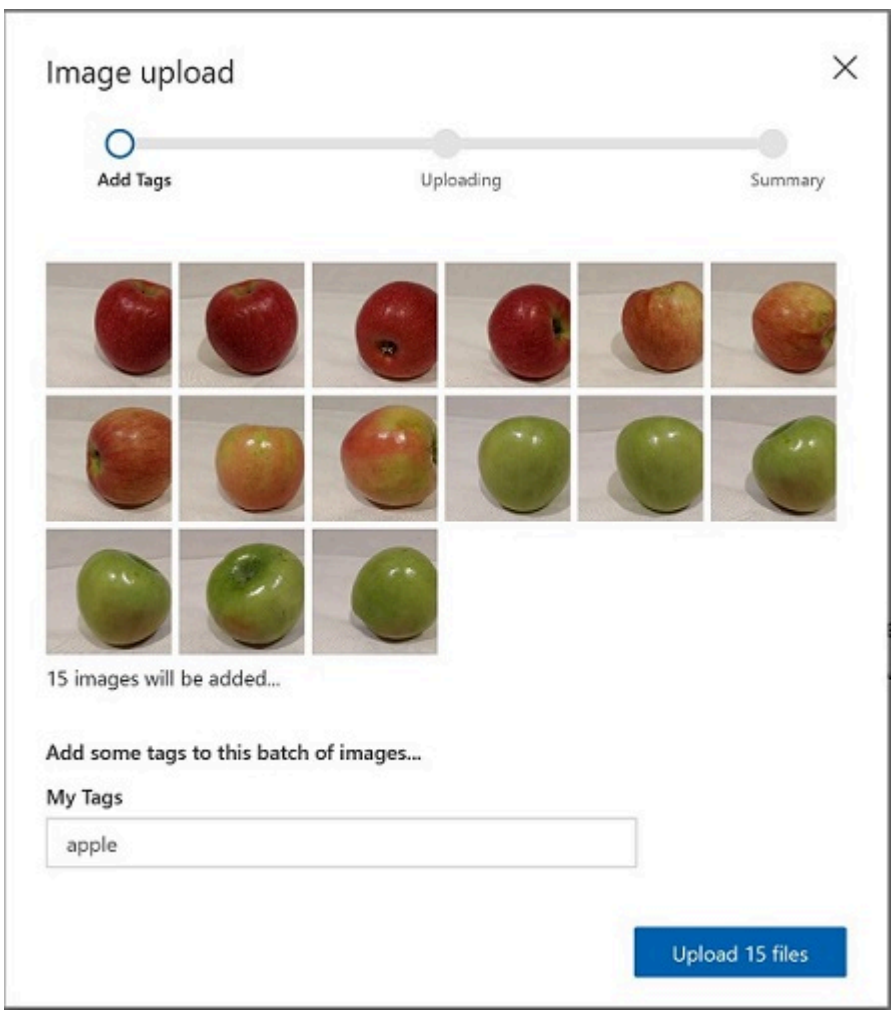
1. Open a new browser tab (keeping the Azure portal tab open - you'll return to it later).

2. In the new browser tab, open the Custom Vision portal at `https://customvision.ai`. If prompted, sign in using your Azure credentials and agree to the terms of service.
3. In the Custom Vision portal, create a new project with the following settings:
    - **Name**: `Classify Fruit`
    - **Description**: `Image classification for fruit`
    - **Resource**: *Your Custom Vision resource*
    - **Project Types**: Classification
    - **Classification Types**: Multiclass (single tag per image)
    - **Domains**: Food

Upload and tag images

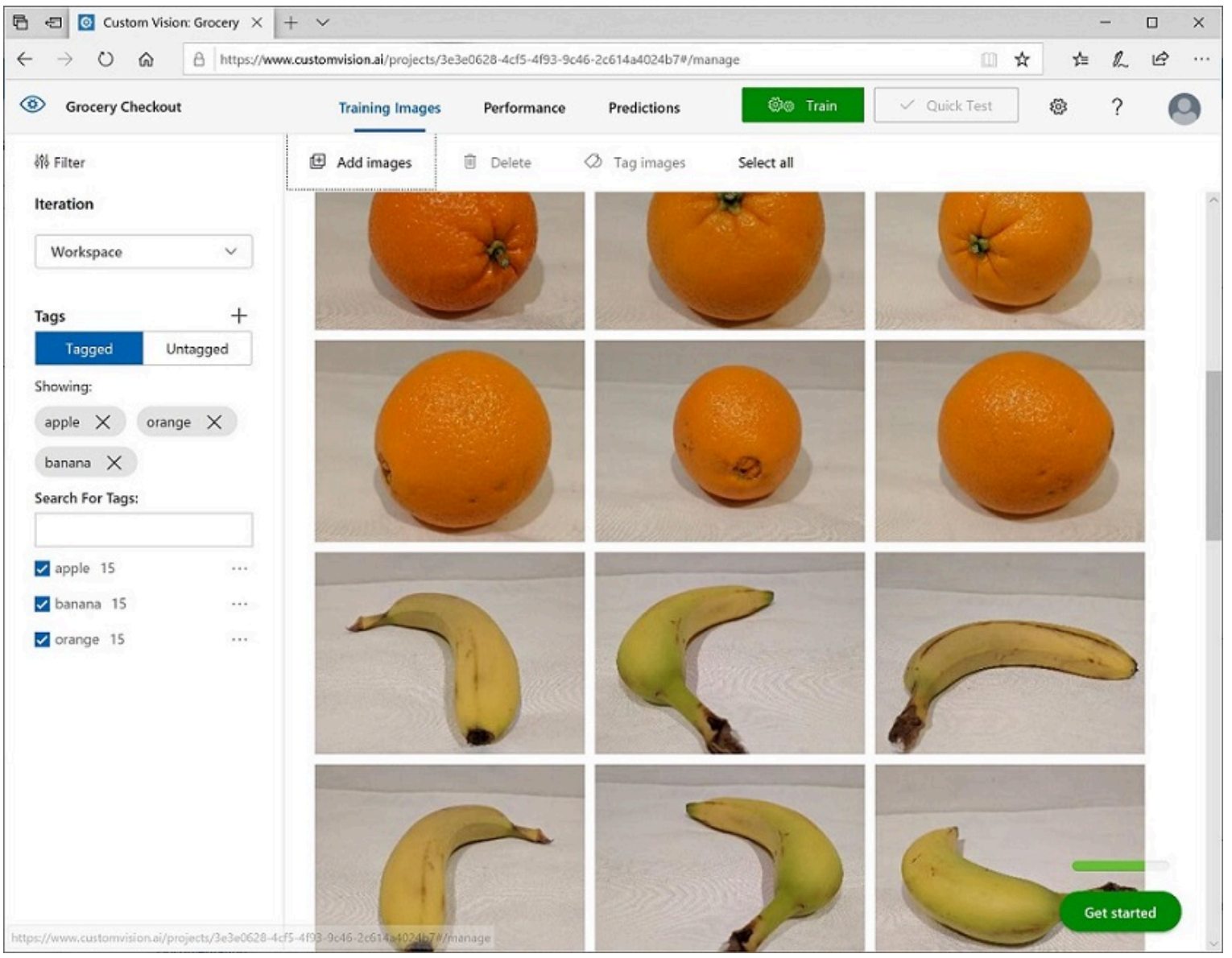
1. In a new browser tab, download the training images from `https://github.com/MicrosoftLearning/mslearn-ai-vision/raw/main/Labfiles/image-classification/training-images.zip` and extract the zip folder to view its contents. This folder contains subfolders of apple, banana, and orange images.

2. In the Custom Vision portal, in your image classification project, click **Add images**, and select all of the files in the **training-images/apple** folder you downloaded and extracted previously. Then upload the image files, specifying the tag `apple`, like this:



3. Use the **Add Images** (**[+]**) toolbar icon to repeat the previous step to upload the images in the **banana** folder with the tag `banana`, and the images in the **orange** folder with the tag `orange`.

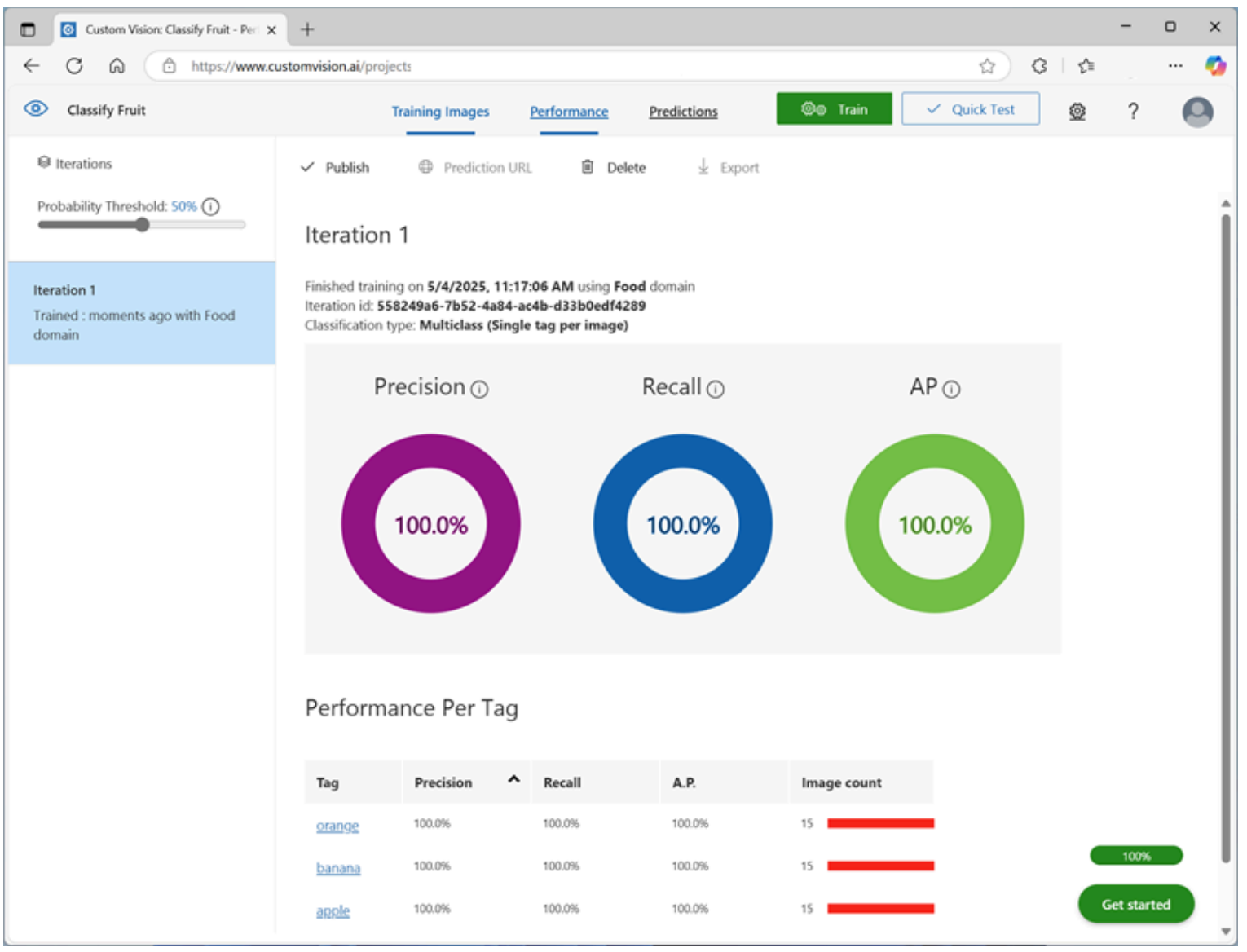
4. Explore the images you have uploaded in the Custom Vision project - there should be 15 images of each class, like this:



Train a model

1. In the Custom Vision project, above the images, click **Train** (⚙) to train a classification model using the tagged images. Select the **Quick Training** option, and then wait for the training iteration to complete (this may take a minute or so).

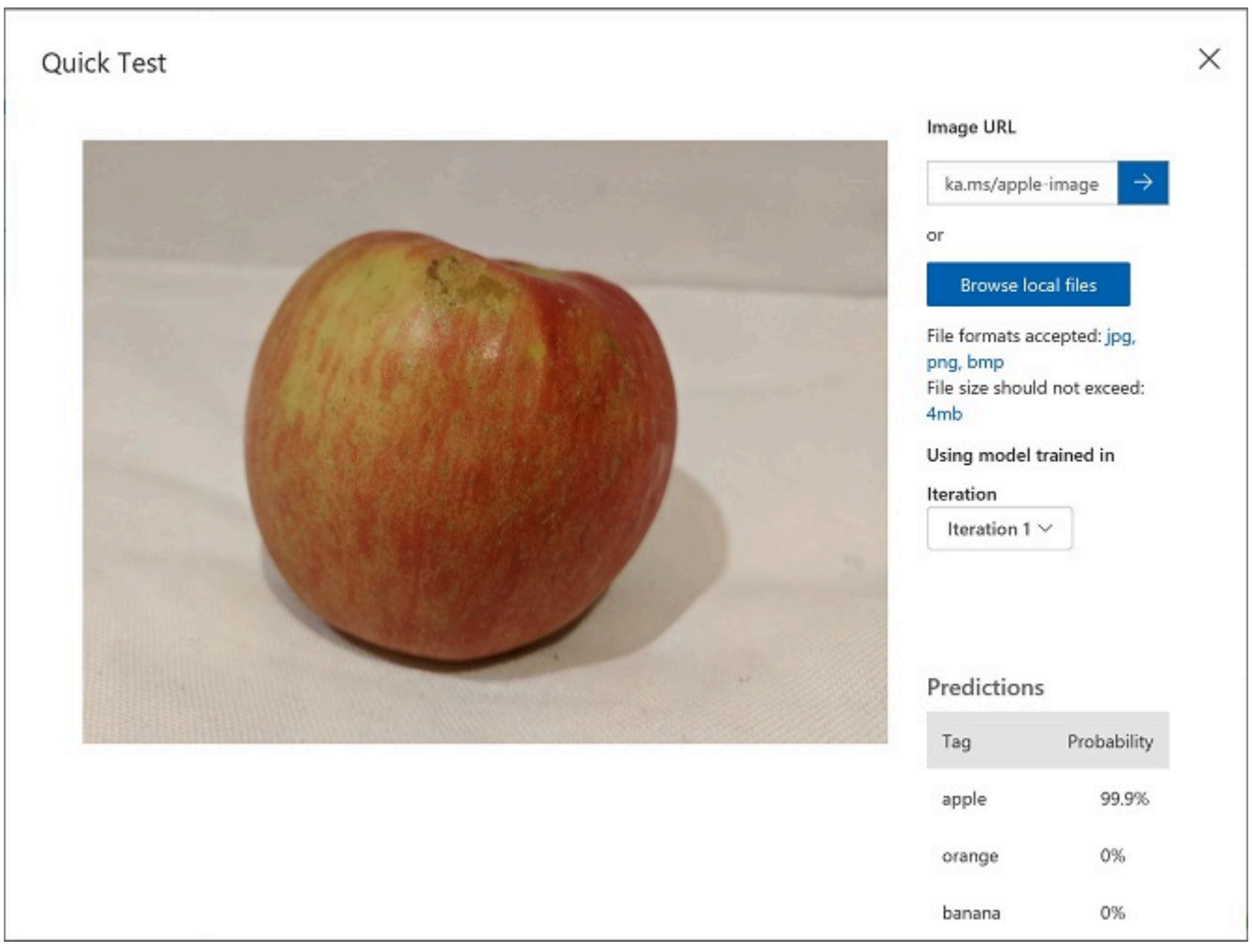
2. When the model iteration has been trained, review the *Precision*, *Recall*, and *AP* performance metrics - these measure the prediction accuracy of the classification model, and should all be high.



> **Note**: The performance metrics are based on a probability threshold of 50% for each prediction (in other words, if the model calculates a 50% or higher probability that an image is of a particular class, then that class is predicted). You can adjust this at the top-left of the page.

Test the model

1. Above the performance metrics, click **Quick Test**.
2. In the **Image URL** box, type `https://aka.ms/test-apple` and click the *quick test image* (➔) button.
3. View the predictions returned by your model - the probability score for *apple* should be the highest, like this:



4. Try testing the following images:
   - `https://aka.ms/test-banana`
   - `https://aka.ms/test-orange`
5. Close the **Quick Test** window.

View the project settings

The project you have created has been assigned a unique identifier, which you will need to specify in any code that interacts with it.

1. Click the *settings* (⚙) icon at the top right of the **Performance** page to view the project settings.
2. Under **General** (on the left), note the **Project Id** that uniquely identifies this project.
3. On the right, under **Resources** note that the key and endpoint are shown. These are the details for the *training* resource (you can also obtain this information by viewing the resource in the Azure portal).

Use the *training* API

The Custom Vision portal provides a convenient user interface that you can use to upload and tag images, and train models. However, in some scenarios you may want to automate model training by using the Custom Vision training API.

Prepare the application configuration

1. Return to the browser tab containing the Azure portal (keeping the Custom Vision portal tab open - you'll return to it later).

2. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. Resize the cloud shell pane so you can see more of it.

   > **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

5. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-vision -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

6. After the repo has been cloned, use the following command to navigate to the application code files:

   Code Copy

   ```
   cd mslearn-ai-vision/Labfiles/image-classification/python/train-classifier
   ls -a -l
   ```

   The folder contains application configuration and code files for your app. It also contains an **/more-training-images** subfolder, which contains some image files you'll use to perform additional training of your model.

7. Install the Azure AI Custom Vision SDK package for training and any other required packages by running the following commands:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-cognitiveservices-vision-customvision
   ```

8. Enter the following command to edit the configuration file for your app:

   Code Copy

   ```
   code .env
   ```

The file is opened in a code editor.

9. In the code file, update the configuration values it contains to reflect the **Endpoint** and an authentication **Key** for your Custom Vision *training* resource, and the **Project ID** for the custom vision project you created previously.
10. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Write code to perform model training

1. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code train-classifier.py
```

2. Note the following details in the code file:
   - The namespaces for the Azure AI Custom Vision SDK are imported.
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated.
   - **CustomVisionTrainingClient**, which is then used with the project ID to create a **Project** reference to your project.
   - The **Upload_Images** function retrieves the tags that are defined in the Custom Vision project and then uploads image files from correspondingly named folders to the project, assigning the appropriate tag ID.
   - The **Train_Model** function creates a new training iteration for the project and waits for training to complete.
3. Close the code editor (*CTRL+Q*) and enter the following command to run the program:

```
python train-classifier.py
```

4. Wait for the program to end. Then return to the browser tab containing the Custom Vision portal, and view the **Training Images** page for your project (refreshing the browser if necessary).
5. Verify that some new tagged images have been added to the project. Then view the **Performance** page and verify that a new iteration has been created.

Use the image classifier in a client application

Now you're ready to publish your trained model and use it in a client application.

Publish the image classification model

1. In the Custom Vision portal, on the **Performance** page, click ✓ **Publish** to publish the trained model with the following settings:
   - **Model name**: `fruit-classifier`
   - **Prediction Resource**: *The **prediction** resource you created previously which ends with "-Prediction" (not the training resource)*.
2. At the top left of the **Project Settings** page, click the *Projects Gallery* (👁) icon to return to the Custom Vision portal home page, where your project is now listed.
3. On the Custom Vision portal home page, at the top right, click the *settings* (⚙) icon to view the settings for your Custom Vision service. Then, under **Resources**, find your *prediction* resource which ends with "-Prediction" (not the training resource) to determine its **Key** and **Endpoint** values (you can also obtain this information by viewing the resource in the Azure portal).

Use the image classifier from a client application

1. Return to the browser tab containing the Azure portal and the cloud shell pane.
2. In cloud shell, run the following commands to switch to the folder for your client application and view the files it contains:

```
cd ../test-classifier
ls -a -l
```

   The folder contains application configuration and code files for your app. It also contains a **/test-images** subfolder, which contains some image files you'll use to test your model.

3. Install the Azure AI Custom Vision SDK package for prediction and any other required packages by running the following commands:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-vision-customvision
```

4. Enter the following command to edit the configuration file for your app:

```
code .env
```

   The file is opened in a code editor.
5. Update the configuration values to reflect the **Endpoint** and **Key** for your Custom Vision *prediction* resource, the **Project ID** for the classification project, and the name of your published model (which should be *fruit-classifier*). Save your changes (*CTRL+S*) and close the code editor (*CTRL+Q*).
6. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code test-classifier.py
```

7. Review the code, noting the following details:
   - The namespaces for the Azure AI Custom Vision SDK are imported.
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionPredictionClient**.
   - The prediction client object is used to predict a class for each image in the **test-images** folder, specifying the project ID and model name for each request. Each prediction includes a probability for each possible class, and only predicted tags with a probability greater than 50% are displayed.
8. Close the code editor and enter the following command to run the program:

```
python test-classifier.py
```

   The program submits each of the following images to the model for classification:

IMG_TEST_1.jpg

IMG_TEST_2.jpg

IMG_TEST_3.jpg

9. View the label (tag) and probability scores for each prediction.

Clean up resources

If you've finished exploring Azure AI Custom Vision, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs:

1. Open the Azure portal at `https://portal.azure.com`, and in the top search bar, search for the resources you created in this lab.

2. On the resource page, select **Delete** and follow the instructions to delete the resource. Alternatively, you can delete the entire resource group to clean up all resources at the same time.

Detect objects in images

The **Azure AI Custom Vision** service enables you to create computer vision models that are trained on your own images. You can use it to train *image classification* and *object detection* models; which you can then publish and consume from applications.

In this exercise, you will use the Custom Vision service to train an *object detection* model that can detect and locate three classes of fruit (apple, banana, and orange) in an image.

While this exercise is based on the Azure Custom Vision Python SDK, you can develop vision applications using multiple language-specific SDKs; including:

- Azure Custom Vision for JavaScript (training)
- Azure Custom Vision for JavaScript (prediction)
- Azure Custom Vision for Microsoft .NET (training)
- Azure Custom Vision for Microsoft .NET (prediction)
- Azure Custom Vision for Java (training)
- Azure Custom Vision for Java (prediction)

This exercise takes approximately **45** minutes.

Create Custom Vision resources

Before you can train a model, you will need Azure resources for *training* and *prediction*. You can create **Custom Vision** resources for each of these tasks, or you can create a single resource and use it for both. In this exercise, you'll create **Custom Vision** resources for training and prediction.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Close any welcome messages or tips that are displayed.
2. Select **Create a resource**.
3. In the search bar, search for `Custom Vision`, select **Custom Vision**, and create the resource with the following settings:
    - **Create options**: Both
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Choose any available region*
    - **Name**: *A valid name for your Custom Vision resource*
    - **Training pricing tier**: F0
    - **Prediction pricing tier**: F0
4. Create the resource and wait for deployment to complete, and then view the deployment details. Note that two Custom Vision resources are provisioned; one for training, and another for prediction.

> **Note**: Each resource has its own *endpoint* and *keys*, which are used to manage access from your code. To train an image classification model, your code must use the *training* resource (with its endpoint and key); and to use the trained model to predict image classes, your code must use the *prediction* resource (with its endpoint and key).

5. When the resources have been deployed, go to the resource group to view them. You should see two custom vision resources, one with the suffix ***-Prediction***.

Create a Custom Vision project in the Custom Vision portal

To train an object detection model, you need to create a Custom Vision project based on your training resource. To do this, you'll use the Custom Vision portal.

1. Open a new browser tab (keeping the Azure portal tab open - you'll return to it later).

2. In the new browser tab, open the Custom Vision portal at `https://customvision.ai`. If prompted, sign in using your Azure credentials and agree to the terms of service.
3. Create a new project with the following settings:
    - **Name**: `Detect Fruit`
    - **Description**: `Object detection for fruit.`
    - **Resource**: *Your Custom Vision resource*
    - **Project Types**: Object Detection
    - **Domains**: General
4. Wait for the project to be created and opened in the browser.

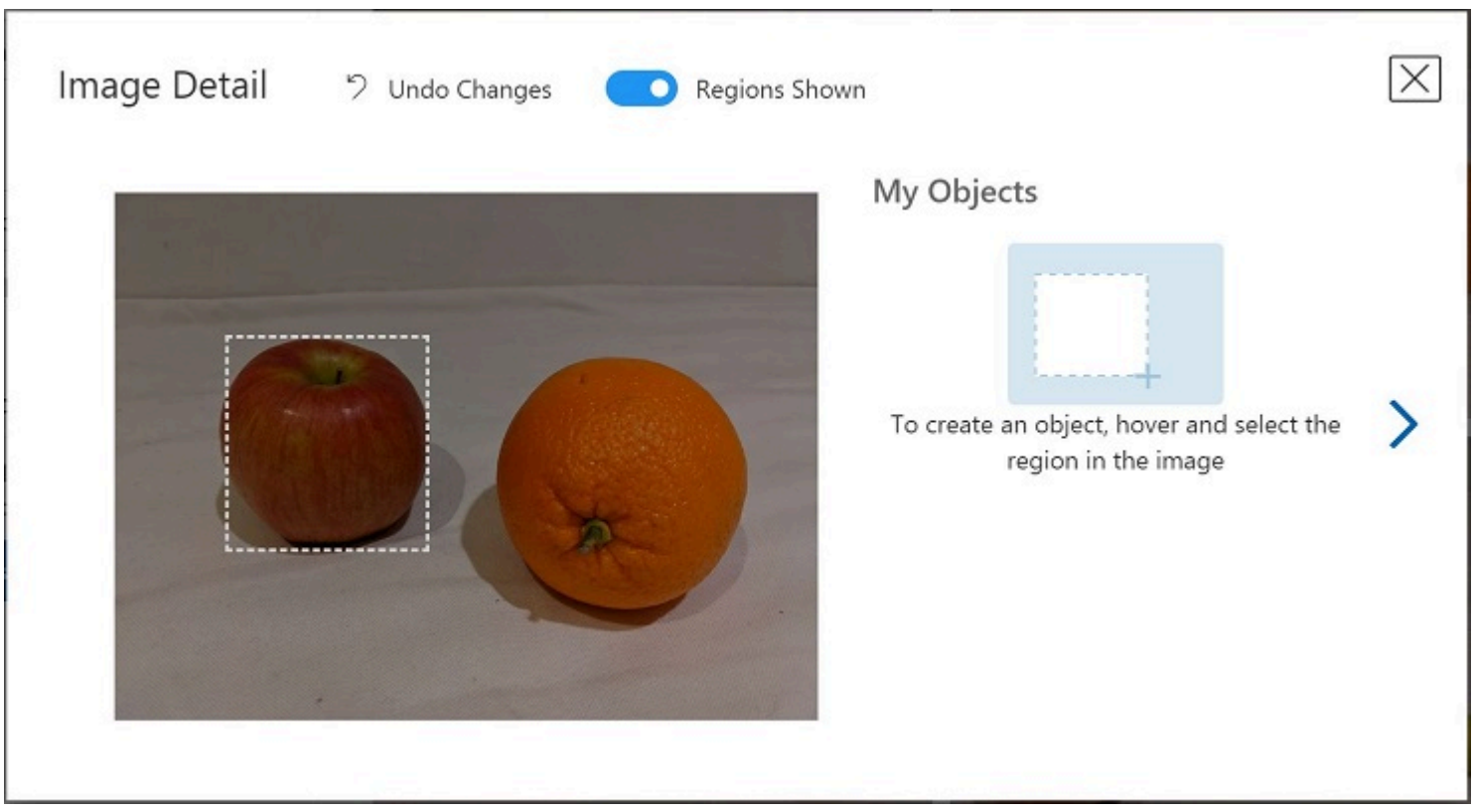
Upload and tag images

Now that you have an object detection project, you can upload and tag images to train a model.

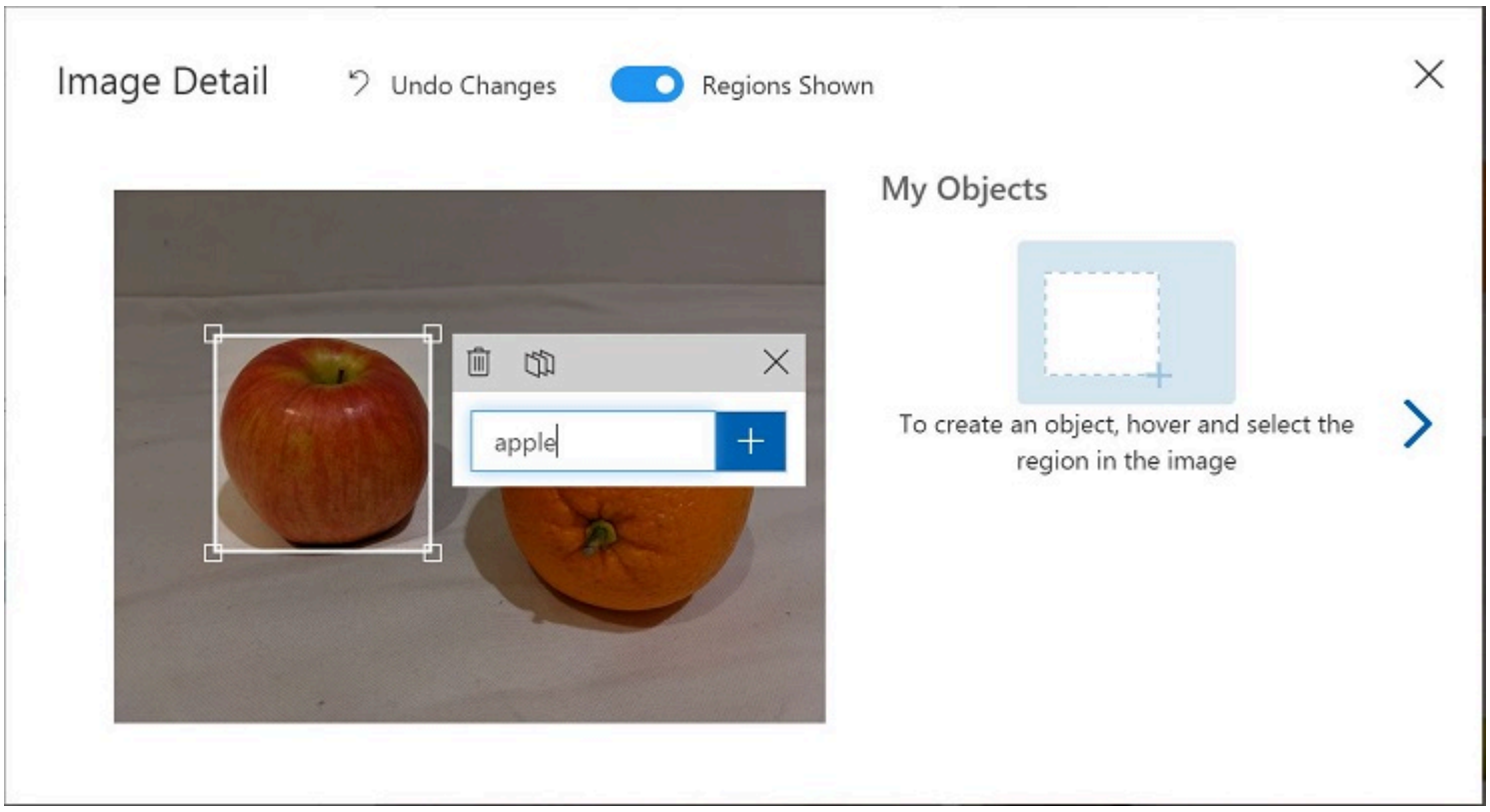
Upload and tag images in the Custom Vision portal

The Custom Vision portal includes visual tools that you can use to upload images and tag regions within them that contain multiple types of object.

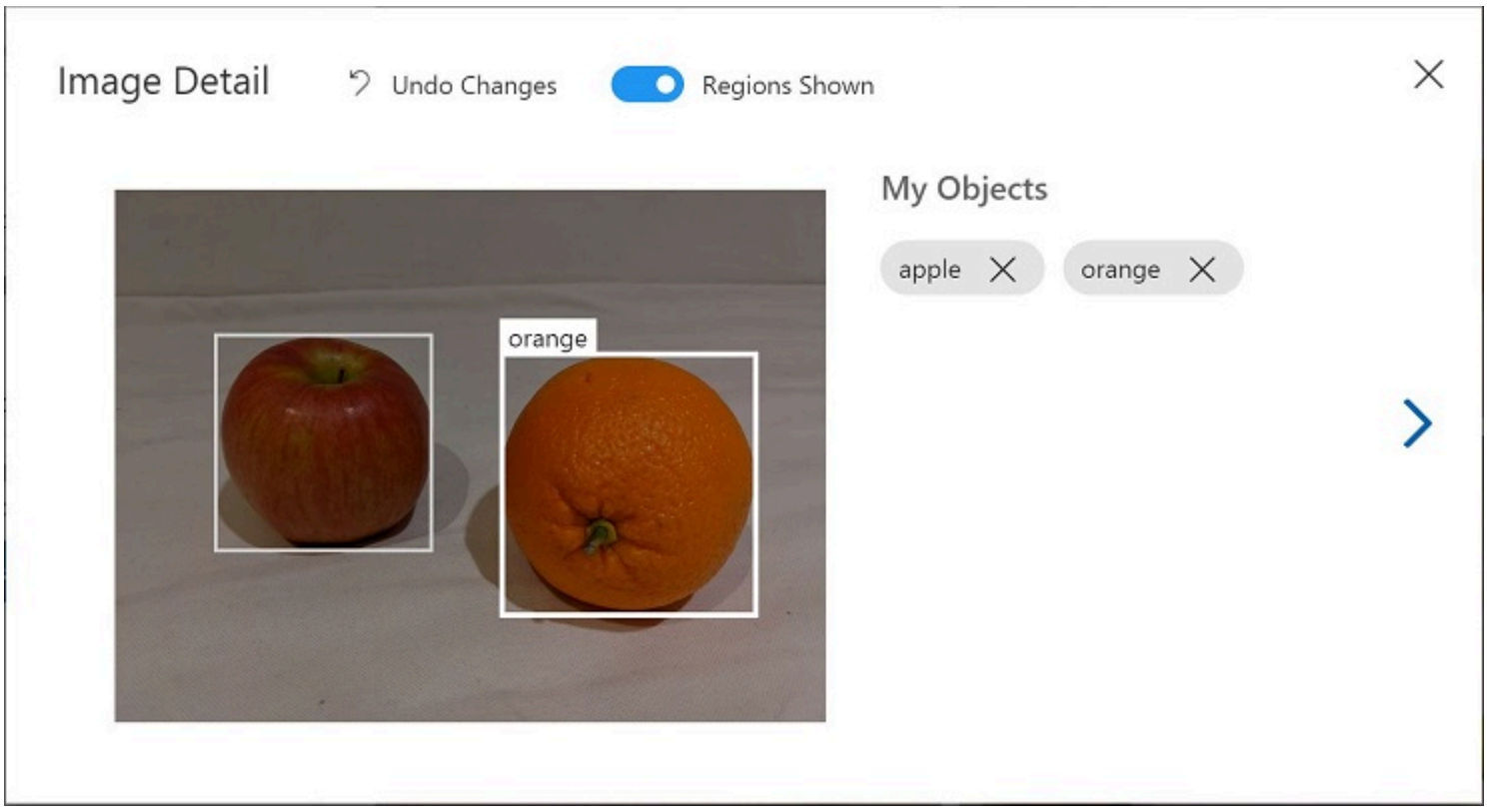
1. In a new browser tab, download the training images from `https://github.com/MicrosoftLearning/mslearn-ai-vision/raw/main/Labfiles/object-detection/training-images.zip` and extract the zip folder to view its contents. This folder contains images of fruit.
2. In the Custom Vision portal, in your object detection project, select **Add images** and upload all of the images in the extracted folder.
3. After the images have been uploaded, select the first one to open it.
4. Hold the mouse over any object in the image until an automatically detected region is displayed like the image below. Then select the object, and if necessary resize the region to surround it.



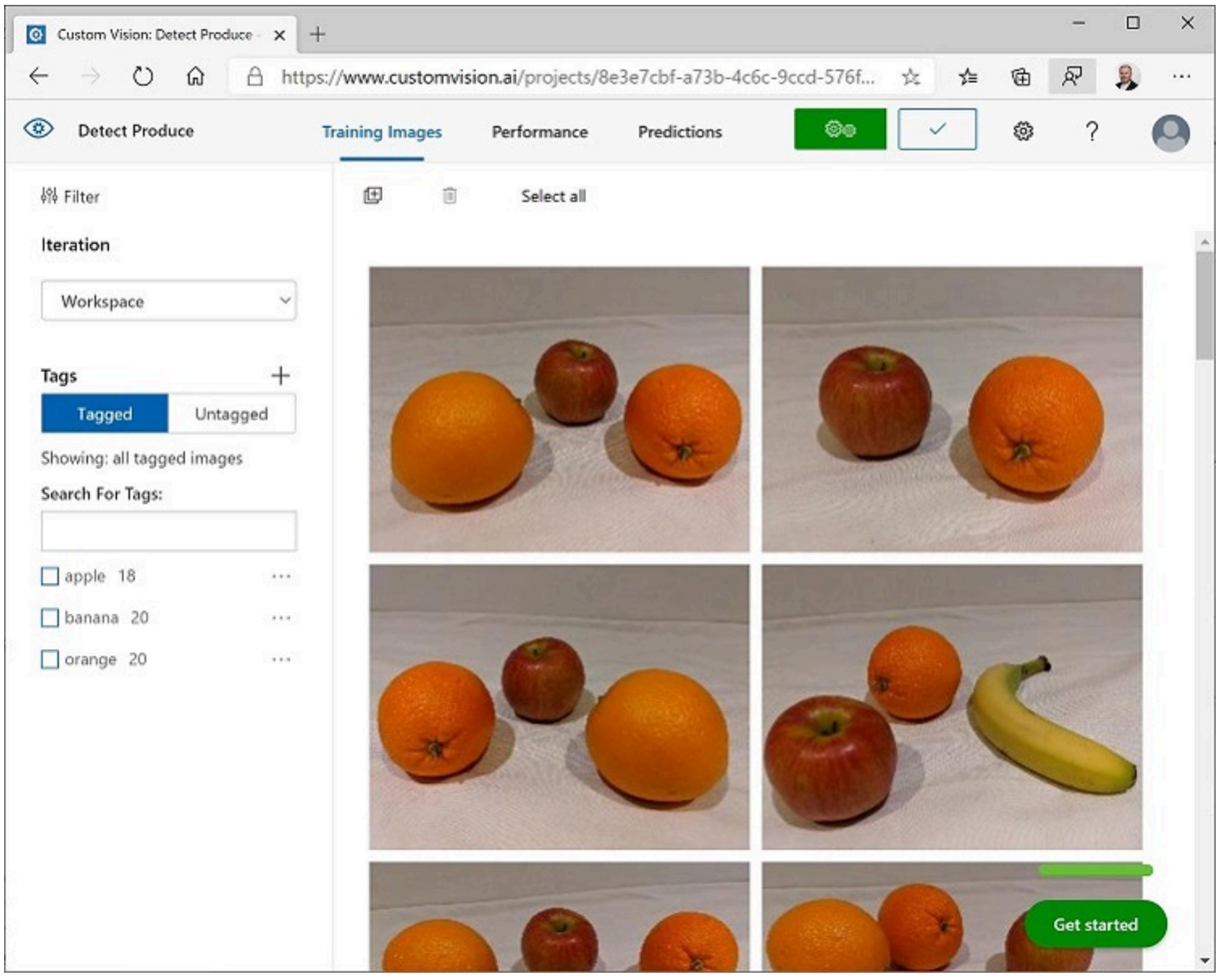
    Alternatively, you can simply drag around the object to create a region.
5. When the region surrounds the object, add a new tag with the appropriate object type (*apple*, *banana*, or *orange*) as shown here:



6. Select and tag each other object in the image, resizing the regions and adding new tags as required.



7. Use the **>** link on the right to go to the next image, and tag its objects. Then just keep working through the entire image collection, tagging each apple, banana, and orange.
8. When you have finished tagging the last image, close the **Image Detail** editor. On the **Training Images** page, under **Tags**, select **Tagged** to see all of your tagged images:



Use the Custom Vision SDK to upload images

You can use the UI in the Custom Vision portal to tag your images, but many AI development teams use other tools that generate files containing information about tags and object regions in images. In scenarios like this, you can use the Custom Vision training API to upload tagged images to the project.

1. Click the *settings* (⚙) icon at the top right of the **Training Images** page in the Custom Vision portal to view the project settings.
2. Under **General** (on the left), note the **Project Id** that uniquely identifies this project.
3. On the right, under **Resources** note that the **Key** and **Endpoint** are shown. These are the details for the *training* resource (you can also obtain this information by viewing the resource in the Azure portal).
4. Return to the browser tab containing the Azure portal (keeping the Custom Vision portal tab open - you'll return to it later).

5. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

6. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

7. Resize the cloud shell pane so you can see more of it.

   > **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

8. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

Code Copy

```
rm -r mslearn-ai-vision -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
```

> **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

9. After the repo has been cloned, use the following command to navigate to the application code files:

Code Copy

```
cd mslearn-ai-vision/Labfiles/object-detection/python/train-detector
ls -a -l
```

The folder contains application configuration and code files for your app. It also contains a **tagged-images.json** file which contains bounding box coordinates for objects in multiple images, and an **/images** subfolder, which contains the images.

10. Install the Azure AI Custom Vision SDK package for training and any other required packages by running the following commands:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-vision-customvision
```

11. Enter the following command to edit the configuration file for your app:

Code Copy

```
code .env
```

The file is opened in a code editor.

12. In the code file, update the configuration values it contains to reflect the **Endpoint** and an authentication **Key** for your Custom Vision *training* resource, and the **Project ID** for the custom vision project you created previously.
13. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

14. In the cloud shell command line, enter the following command to open the **tagged-images.json** file to see the tagging information for the image files in the **/images** subfolder:

Code Copy

```
code tagged-images.json
```

JSON defines a list of images, each containing one or more tagged regions. Each tagged region includes a tag name, and the top and left coordinates and width and height dimensions of the bounding box containing the tagged object.

> **Note**: The coordinates and dimensions in this file indicate relative points on the image. For example, a *height* value of 0.7 indicates a box that is 70% of the height of the image. Some tagging tools generate other formats of file in which the coordinate and dimension values represent pixels, inches, or other units of measurements.

15. Close the JSON file without saving any changes (*CTRL_Q*).
16. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code add-tagged-images.py
```

17. Note the following details in the code file:
    - The namespaces for the Azure AI Custom Vision SDK are imported.
    - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionTrainingClient**, which is then used with the project ID to create a **Project** reference to your project.
    - The **Upload_Images** function extracts the tagged region information from the JSON file and uses it to create a batch of images with regions, which it then uploads to the project.
18. Close the code editor (*CTRL+Q*) and enter the following command to run the program:

```
python add-tagged-images.py
```

19. Wait for the program to end.
20. Switch back to the browser tab containing the Custom Vision portal (keeping the Azure portal cloud shell tab open), and view the **Training Images** page for your project (refreshing the browser if necessary).
21. Verify that some new tagged images have been added to the project.

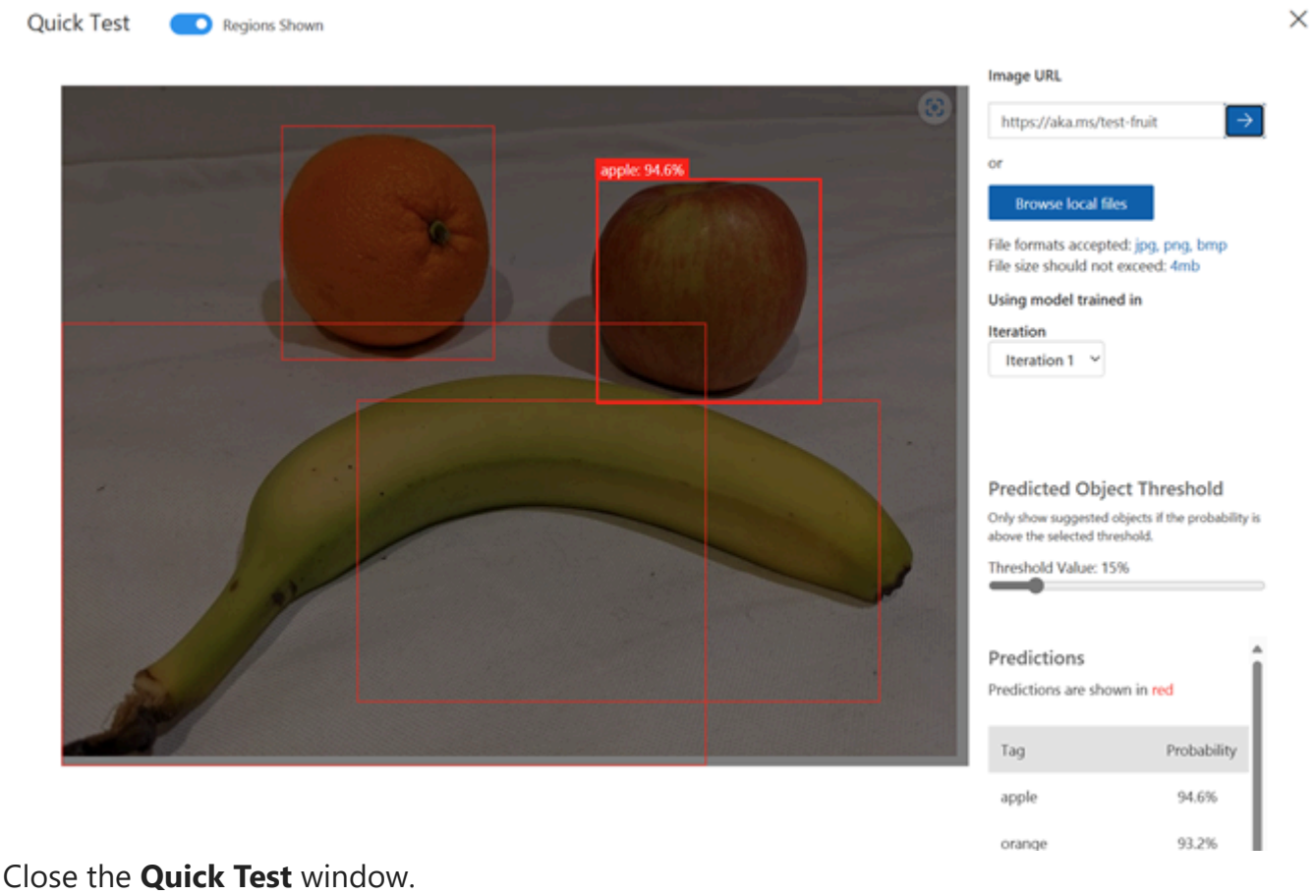
Train and test a model

Now that you've tagged the images in your project, you're ready to train a model.

1. In the Custom Vision project, click **Train** (⚙) to train an object detection model using the tagged images. Select the **Quick Training** option.
2. Wait for training to complete (it might take ten minutes or so).

> **Tip**: The Azure cloud shell has a 20-minute inactivity timeout, after which the session is abandoned. While you wait for training to finish, occassionally return to the cloud shell and enter a colland like `ls` to keep the session active.

3. In the Custom Vision portal, when training has finished, review the *Precision*, *Recall*, and *mAP* performance metrics - these measure the prediction accuracy of the object detection model, and should all be high.
4. At the top right of the page, click **Quick Test**, and then in the **Image URL** box, type `https://aka.ms/test-fruit` and click the *quick test image* (➔) button.
5. View the prediction that is generated.



6. Close the **Quick Test** window.

Use the object detector in a client application

Now you're ready to publish your trained model and use it in a client application.

Publish the object detection model

1. In the Custom Vision portal, on the **Performance** page, click ✓ **Publish** to publish the trained model with the following settings:
    - **Model name**: `fruit-detector`
    - **Prediction Resource**: *The **prediction** resource you created previously which ends with "-Prediction" (not the training resource)*.
2. At the top left of the **Project Settings** page, click the *Projects Gallery* (👁) icon to return to the Custom Vision portal home page, where your project is now listed.
3. On the Custom Vision portal home page, at the top right, click the *settings* (⚙) icon to view the settings for your Custom Vision service. Then, under **Resources**, find your *prediction* resource which ends with "-Prediction" (not the training resource) to determine its **Key** and **Endpoint** values (you can also obtain this information by viewing the resource in the Azure portal).

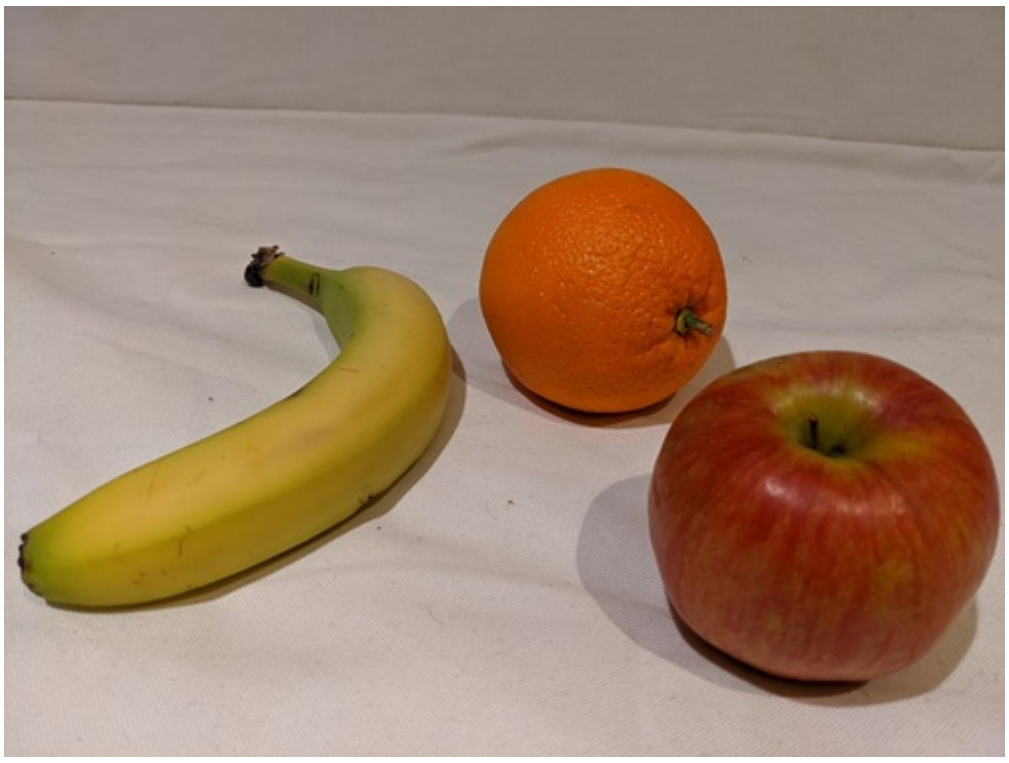
Use the image classifier from a client application

Now that you've published the image classification model, you can use it from a client application. Once again, you can choose to use **C#** or **Python**.

1. Return to the browser tab containing the Azure portal and the cloud shell pane.
2. In cloud shell, run the following commands to switch to the folder for you client application and view the files it contains:

```
cd ../test-detector
ls -a -l
```

The folder contains application configuration and code files for your app. It also contains the following **produce.jpg** image file, which you'll use to test your model.



3. Install the Azure AI Custom Vision SDK package for prediction and any other required packages by running the following commands:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-vision-customvision
```

4. Enter the following command to edit the configuration file for your app:

```
code .env
```

The file is opened in a code editor.

5. Update the configuration values to reflect the **Endpoint** and **Key** for your Custom Vision *prediction* resource, the **Project ID** for the object detection project, and the name of your published model (which should be *fruit-detector*). Save your changes (*CTRL*+*S*) and close the code editor (*CTRL*+*Q*).

6. In the cloud shell command line, enter the following command to open the code file for the client application:

```
code test-detector.py
```

7. Review the code, noting the following details:

   - The namespaces for the Azure AI Custom Vision SDK are imported.
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionPredictionClient**.
   - The prediction client object is used to get object detection predictions for the **produce.jpg** image, specifying the project ID and model name in the request. The predicted tagged regions are then drawn on the image, and the result is saved as **output.jpg**.

8. Close the code editor and enter the following command to run the program:

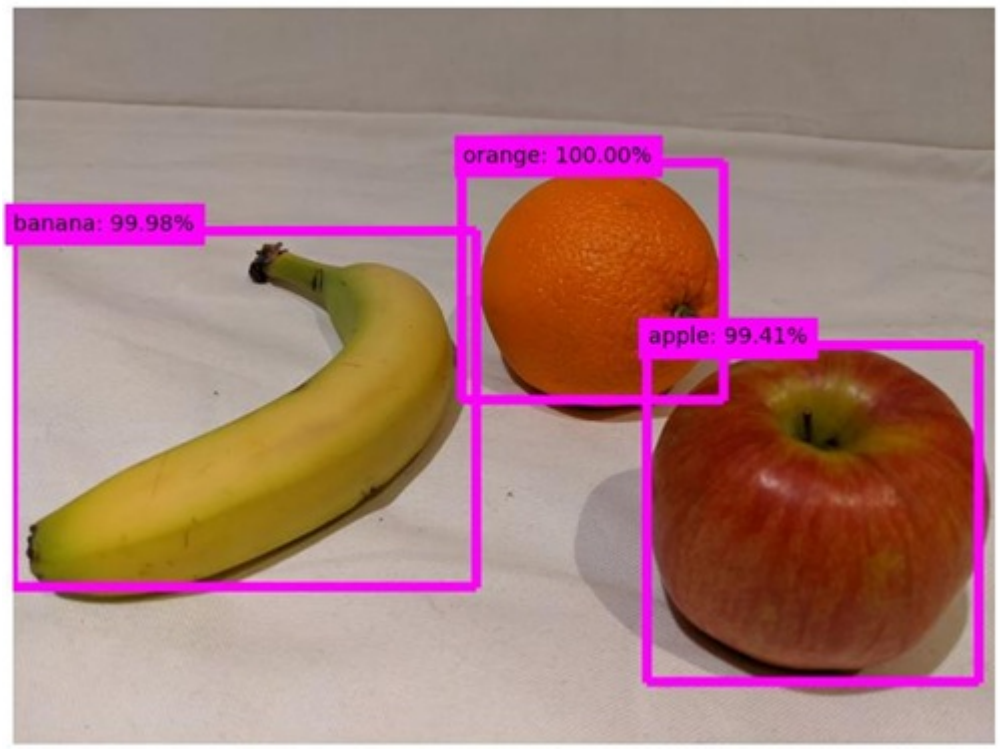
```
python test-detector.py
```

9. Review the program output, which lists each object detected in the image.

10. Note that an image file named **output.jpg** is generated. Use the (Azure cloud shell-specific) **download** command to download it:

```
download output.jpg
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file. The image should look simlar to this:



Clean up resources

If you're not using the Azure resources created in this lab for other training modules, you can delete them to avoid incurring further charges.

1. Open the Azure portal at `https://portal.azure.com`, and in the top search bar, search for the resources you created in this lab.
2. On the resource page, select **Delete** and follow the instructions to delete the resource. Alternatively, you can delete the entire resource group to clean up all resources at the same time.

More information

For more information about object detection with the Custom Vision service, see the Custom Vision documentation.

Analyze video

A large proportion of the data created and consumed today is in the format of video. **Azure AI Video Indexer** is an AI-powered service that you can use to index videos and extract insights from them.

> **Note**: From June 21st 2022, capabilities of Azure AI services that return personally identifiable information are restricted to customers who have been granted limited access. Without getting limited access approval, recognizing people and celebrities with Video Indexer for this lab is not available. For more details about the changes Microsoft has made, and why - see Responsible AI investments and safeguards for facial recognition.

Upload a video to Video Indexer

First, you'll need to sign into the Video Indexer portal and upload a video.

1. In your browser, open the Video Indexer portal at `https://www.videoindexer.ai`.
2. If you have an existing Video Indexer account, sign in. Otherwise, sign up for a free account and sign in using your Microsoft account (or any other valid account type). If you have difficulty signing in, try opening a private browser session.

   > Note: If this is your first time signing in you might see a pop-up form asking you to verify how you're going to use the service.

3. In a new tab, download the Responsible AI video by visiting `https://aka.ms/responsible-ai-video`. Save the file.
4. In Video Indexer, select the **Upload** option. Then select the option to **Browse for files**, select the downloaded video, and click **Add**. Change the text in the **File name** field to **Responsible AI**. Select **Review + upload**, review the summary overview, select the checkbox to verify compliance with Microsoft's policies for facial recognition, and upload the file.
5. After the file has uploaded, wait a few minutes while Video Indexer automatically indexes it.

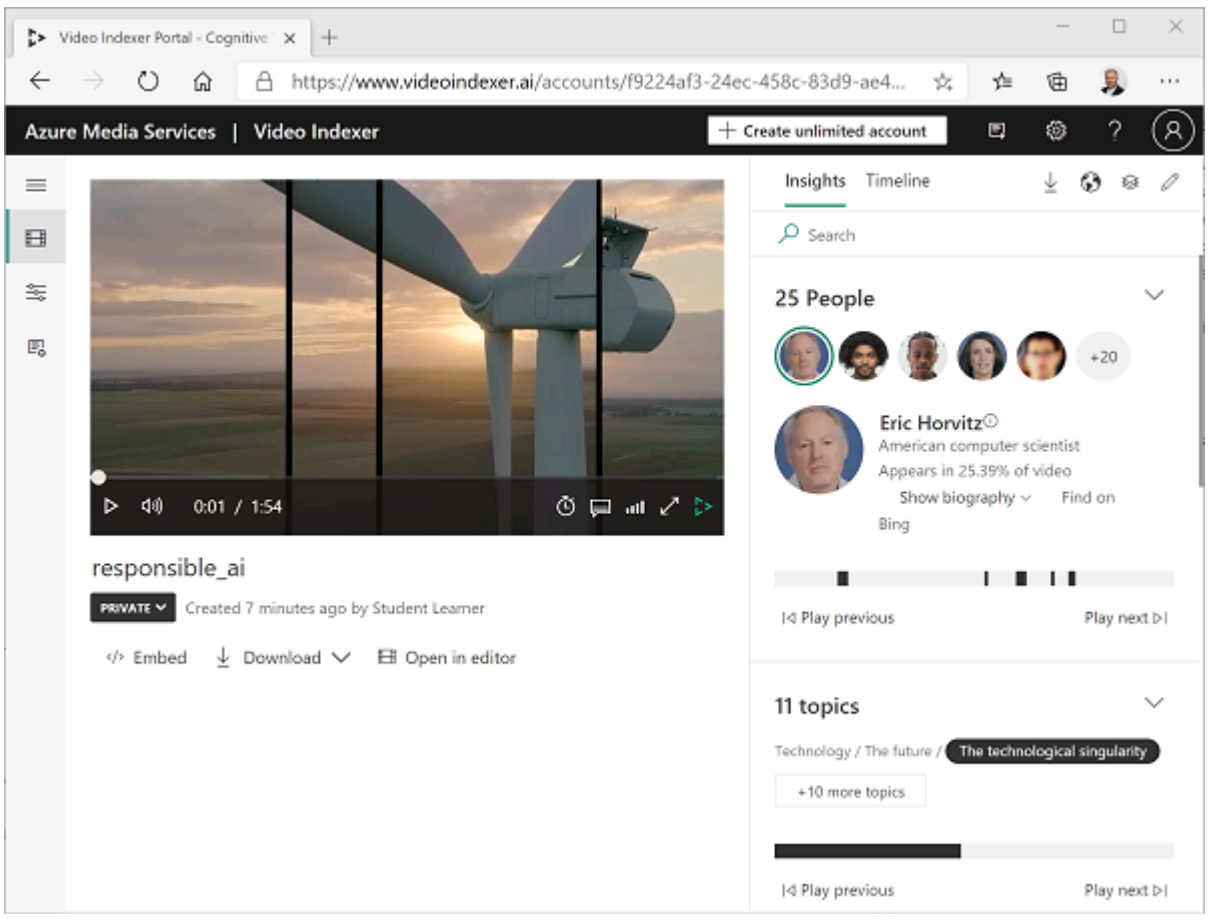
> **Note**: In this exercise, we're using this video to explore Video Indexer functionality; but you should take the time to watch it in full when you've finished the exercise as it contains useful information and guidance for developing AI-enabled applications responsibly!

Review video insights

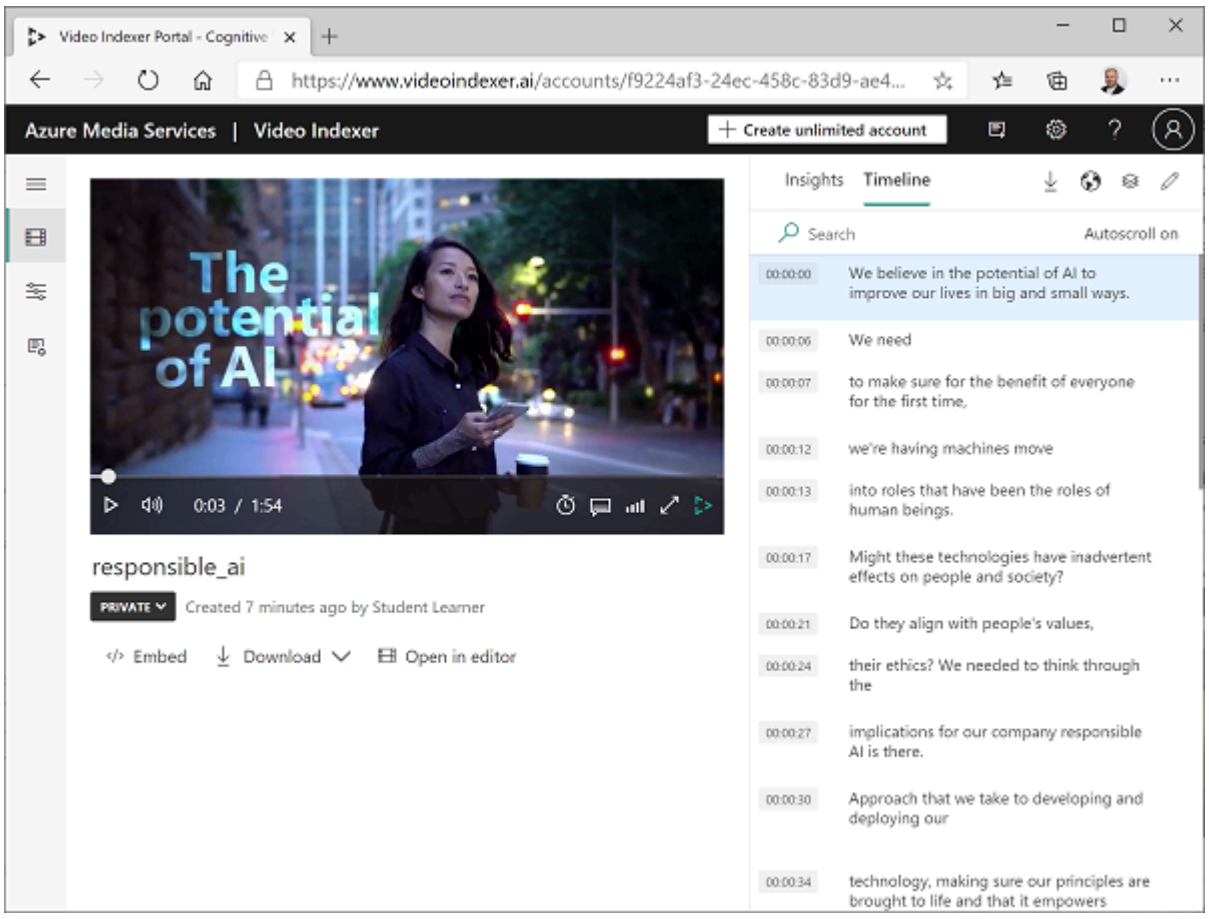
The indexing process extracts insights from the video, which you can view in the portal.

1. In the Video Indexer portal, when the video is indexed, select it to view it. You'll see the video player alongside a pane that shows insights extracted from the video.

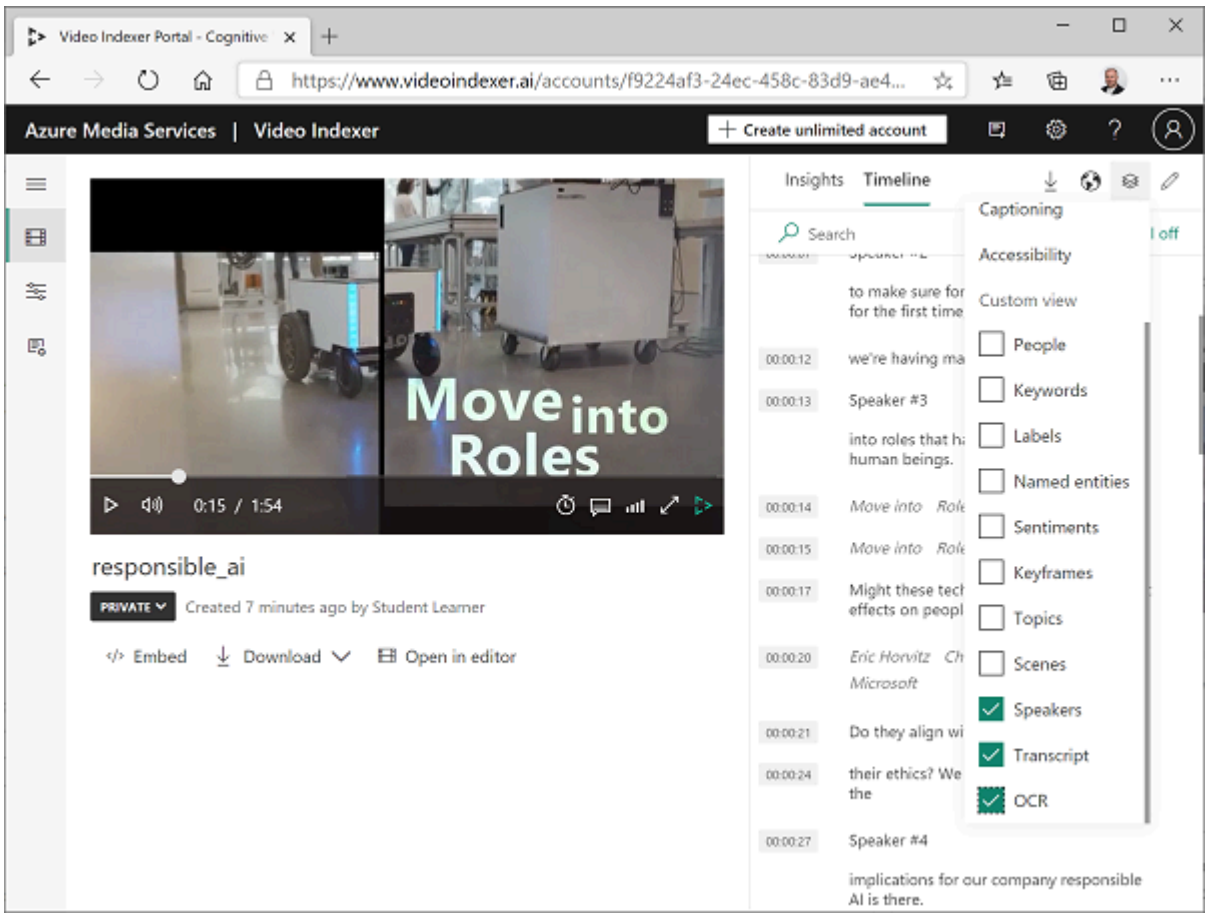
   > **Note**: Due to the limited access policy to protect individuals identities, you may not see names when you index the video.



2. As the video plays, select the **Timeline** tab to view a transcript of the video audio.



3. At the top right of the portal, select the **View** symbol (which looks similar to), and in the list of insights, in addition to **Transcript**, select **OCR** and **Speakers**.



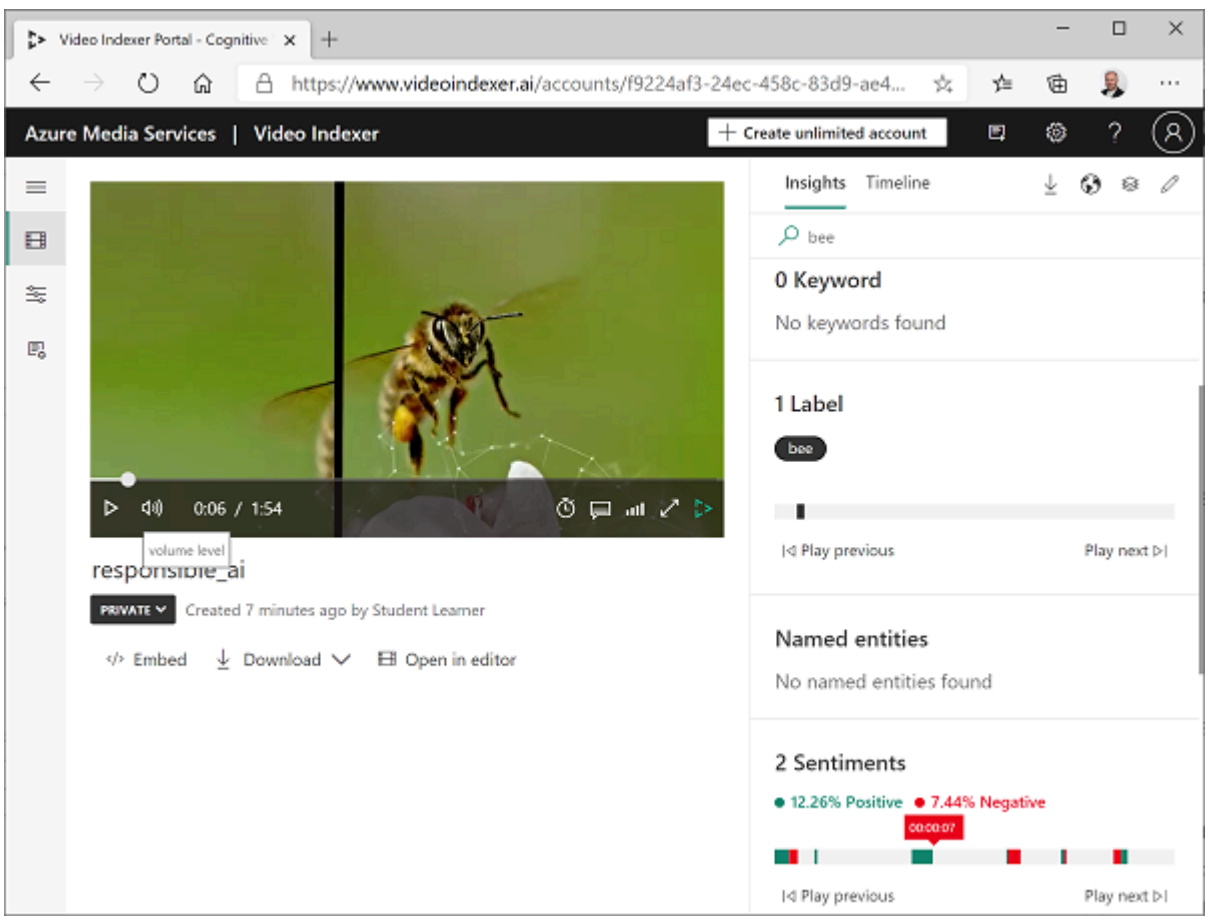
4. Observe that the **Timeline** pane now includes:
    - Transcript of audio narration.
    - Text visible in the video.
    - Indications of speakers who appear in the video. Some well-known people are automatically recognized by name, others are indicated by number (for example *Speaker #1*).
5. Switch back to the **Insights** pane and view the insights show there. They include:
    - Individual people who appear in the video.
    - Topics discussed in the video.
    - Labels for objects that appear in the video.
    - Named entities, such as people and brands that appear in the video.
    - Key scenes.
6. With the **Insights** pane visible, select the **View** symbol again, and in the list of insights, add **Keywords** and **Sentiments** to the pane.

    The insights found can help you determine the main themes in the video. For example, the **topics** for this video show that it is clearly about technology, social responsibility, and ethics.

Search for insights

You can use Video Indexer to search the video for insights.

1. In the **Insights** pane, in the **Search** box, enter *Bee*. You may need to scroll down in the Insights pane to see results for all types of insight.
2. Observe that one matching *label* is found, with its location in the video indicated beneath.
3. Select the beginning of the section where the presence of a bee is indicated, and view the video at that point (you may need to pause the video and select carefully - the bee only appears briefly!)



4. Clear the **Search** box to show all insights for the video.

Use the Video Indexer REST API

Video Indexer provides a REST API that you can use to upload and manage videos in your account.

1. In a new browser tab, open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials. Keep the existing tab with the Video Indexer portal open.
2. In the Azure portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

    The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. Resize the cloud shell pane so you can see more of it.

> **Tip**" You can resize the pane by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

5. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

```
Code                                                                    Copy

rm -r mslearn-ai-vision -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
```

> **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

6. After the repo has been cloned, navigate to the folder containing the application code file for this exercise:

```
Code                                                                    Copy

cd mslearn-ai-vision/Labfiles/video-indexer
```

Get your API details

To use the Video Indexer API, you need some information to authenticate requests:

1. In the Video Indexer portal, expand the left pane and select the **Account settings** page.
2. Note the **Account ID** on this page - you will need it later.
3. Open a new browser tab and go to the Video Indexer developer portal at `https://api-portal.videoindexer.ai, signing with your Azure credentials.
4. On the **Profile** page, view the **Subscriptions** associated with your profile.
5. On the page with your subscription(s), observe that you have been assigned two keys (primary and secondary) for each subscription. Then select **Show** for any of the keys to see it. You will need this key shortly.

Use the REST API

Now that you have the account ID and an API key, you can use the REST API to work with videos in your account. In this procedure, you'll use a PowerShell script to make REST calls; but the same principles apply with HTTP utilities such as cURL or Postman, or any programming language capable of sending and receiving JSON over HTTP.

All interactions with the Video Indexer REST API follow the same pattern:

- An initial request to the **AccessToken** method with the API key in the header is used to obtain an access token.
- Subsequent requests use the access token to authenticate when calling REST methods to work with videos.

1. In the cloud shell, use the following command to open the PowerShell script:

Code Copy

```
code get-videos.ps1
```

2. In the PowerShell script, replace the **YOUR_ACCOUNT_ID** and **YOUR_API_KEY** placeholders with the account ID and API key values you identified previously.
3. Observe that the *location* for a free account is "trial". If you have created an unrestricted Video Indexer account (with an associated Azure resource), you can change this to the location where your Azure resource is provisioned (for example "eastus").
4. Review the code in the script, noting that invokes two REST methods: one to get an access token, and another to list the videos in your account.

5. Save your changes (press *CTRL*+*S*), close the code editor (press *CTRL*+*Q*) and then run the following command to execute the script:

Code Copy

```
./get-videos.ps1
```

6. View the JSON response from the REST service, which should contain details of the **Responsible AI** video you indexed previously.

Use Video Indexer widgets

The Video Indexer portal is a useful interface to manage video indexing projects. However, there may be occasions when you want to make the video and its insights available to people who don't have access to your Video Indexer account. Video Indexer provides widgets that you can embed in a web page for this purpose.

1. Use the `ls` command,to view the contents of the **video-indexer** folder. Note that it contains a **analyze-video.html** file. This is a basic HTML page to which you will add the Video Indexer **Player** and **Insights** widgets.

2. Enter the following command to edit the file:

Code Copy

```
code analyze-video.html
```

The file is opened in a code editor.

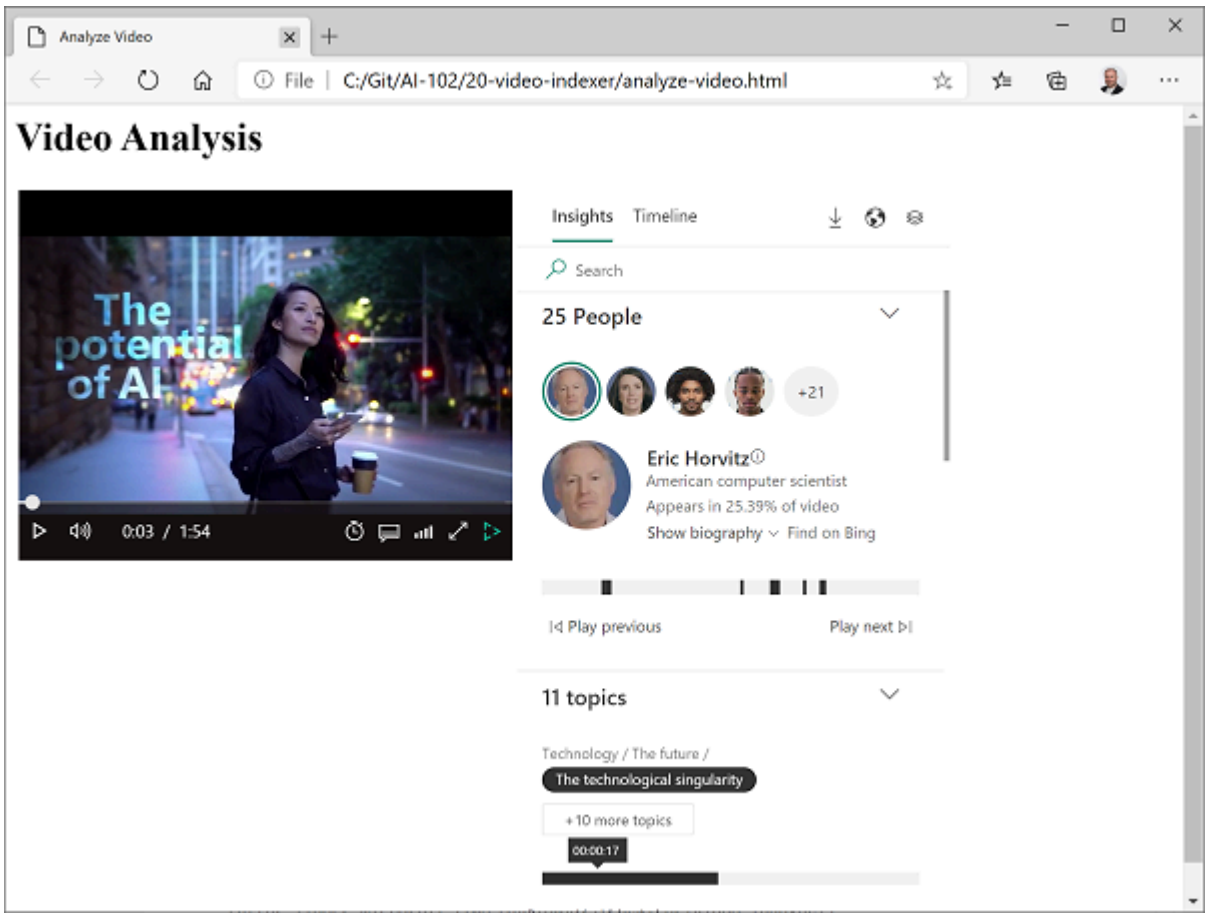
3. Note the reference to the **vb.widgets.mediator.js** script in the header - this script enables multiple Video Indexer widgets on the page to interact with one another.
4. In the Video Indexer portal, return to the **Media files** page and open your **Responsible AI** video.
5. Under the video player, select **</> Embed** to view the HTML iframe code to embed the widgets.
6. In the **Share and Embed** dialog box, select the **Player** widget, set the video size to 560 x 315, and then copy the embed code to the clipboard.
7. In the Azure portal cloud shell, in the code editor for the **analyze-video.html** file, paste the copied code under the comment **<– Player widget goes here – >**.
8. Back in the Video Indexer portal, in the **Share and Embed** dialog box, select the **Insights** widget and then copy the embed code to the clipboard. Then close the **Share and Embed** dialog box, switch back to Azure portal, and paste the copied code under the comment **<– Insights widget goes here – >**.
9. After editing the file, within the code editor, save your changes (*CTRL*+*S*) and then close the code editor (*CTRL*+*Q*) while keeping the cloud shell command line open.

10. In the cloud shell toolbar, enter the following (Cloud shell-specific) command to download the HTML file you edited:

Code Copy

```
download analyze-video.html
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file; which should look like this:



11. Experiment with the widgets, using the **Insights** widget to search for insights and jump to them in the video.

Develop a vision-enabled chat app

In this exercise, you use the *Phi-4-multimodal-instruct* generative AI model to generate responses to prompts that include images. You'll develop an app that provides AI assistance with fresh produce in a grocery store by using Azure AI Foundry and the Azure AI Model Inference service.

> **Note**: This exercise is based on pre-release SDK software, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

While this exercise is based on the Azure AI Foundry Python SDK, you can develop AI chat applications using multiple language-specific SDKs; including:

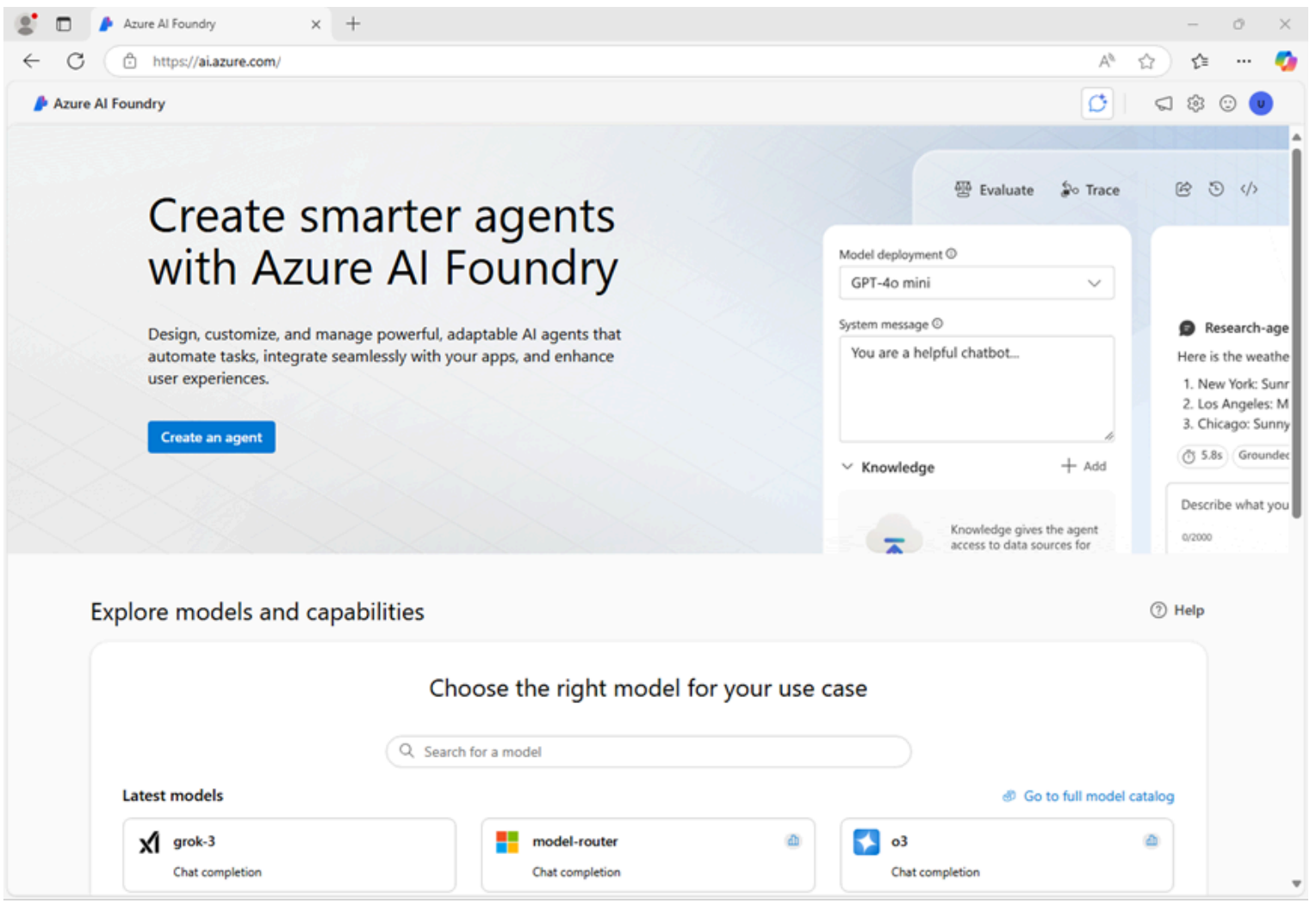
- Azure AI Projects for Python
- Azure AI Projects for Microsoft .NET
- Azure AI Projects for JavaScript

This exercise takes approximately **30** minutes.

Open Azure AI Foundry portal

Let's start by signing into Azure AI Foundry portal.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. Review the information on the home page.

Choose a model to start a project

An Azure AI *project* provides a collaborative workspace for AI development. Let's start by choosing a model that we want to work with and creating a project to use it in.

> **Note**: AI Foundry projects can be based on an *Azure AI Foundry* resource, which provides access to AI models (including Azure OpenAI), Azure AI services, and other resources for developing AI agents and chat solutions. Alternatively, projects can be based on *AI hub* resources; which include connections to Azure resources for secure storage, compute, and specialized tools. Azure AI Foundry based projects are great for developers who want to manage resources for AI agent or chat app development. AI hub based projects are more suitable for enterprise development teams working on complex AI solutions.

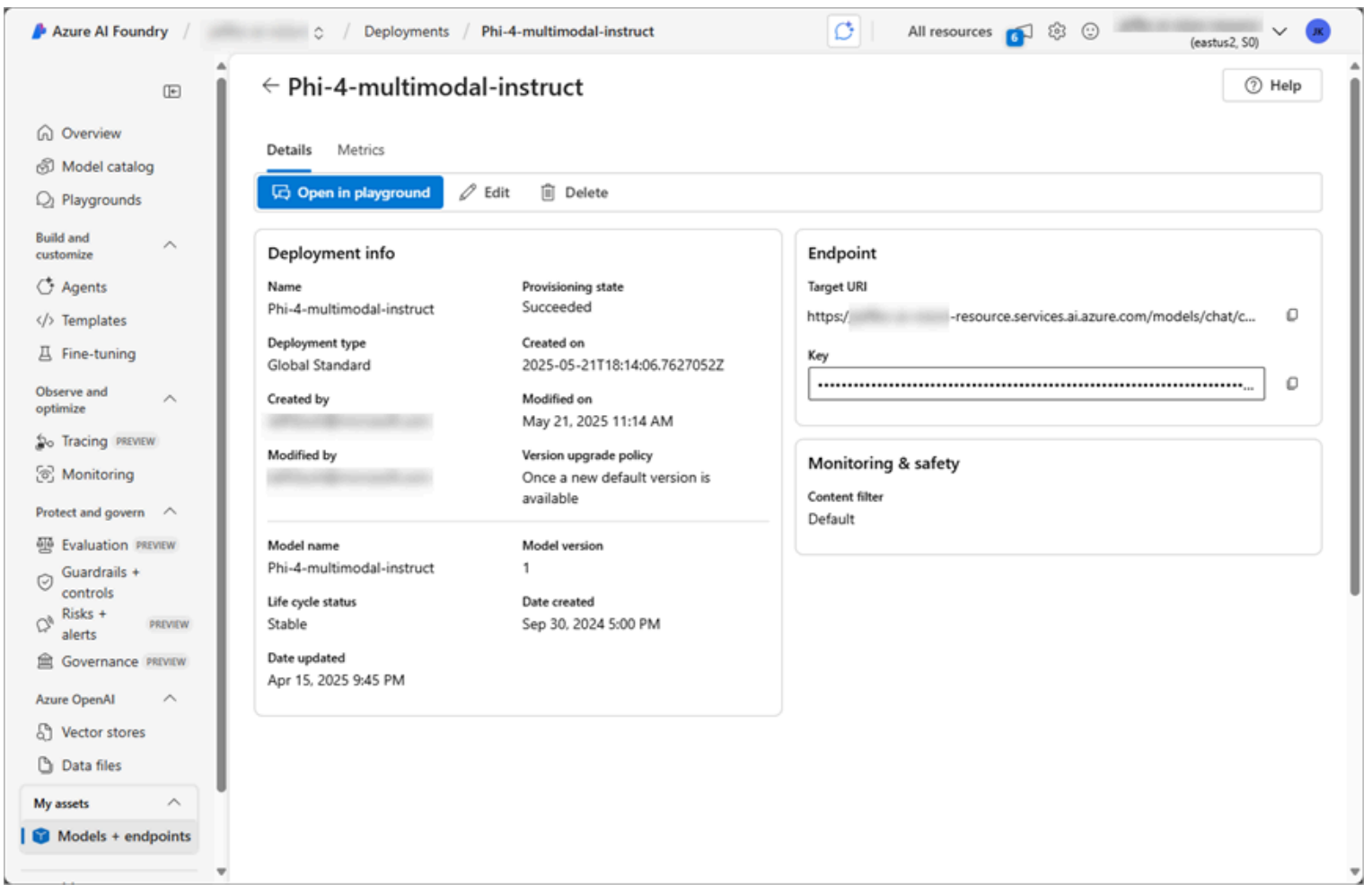
1. In the home page, in the **Explore models and capabilities** section, search for the `Phi-4-multimodal-instruct` model; which we'll use in our project.
2. In the search results, select the **Phi-4-multimodal-instruct** model to see its details, and then at the top of the page for the model, select **Use this model**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Select **Customize** and specify the following settings for your hub:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project, including the Phi-4-multimodal-instruct model deployment you selected, to be created.

> Note: Depending on your model selection you might receive additional prompts during the project creation process. Agree to the terms and finalize the deployment.

6. When your project is created, your model will be displayed in the **Models + endpoints** page:



Test the model in the playground

Now you can test your multimodal model deployment with an image-based prompt in the chat playground.

1. Select **Open in playground** on the model deployment page.

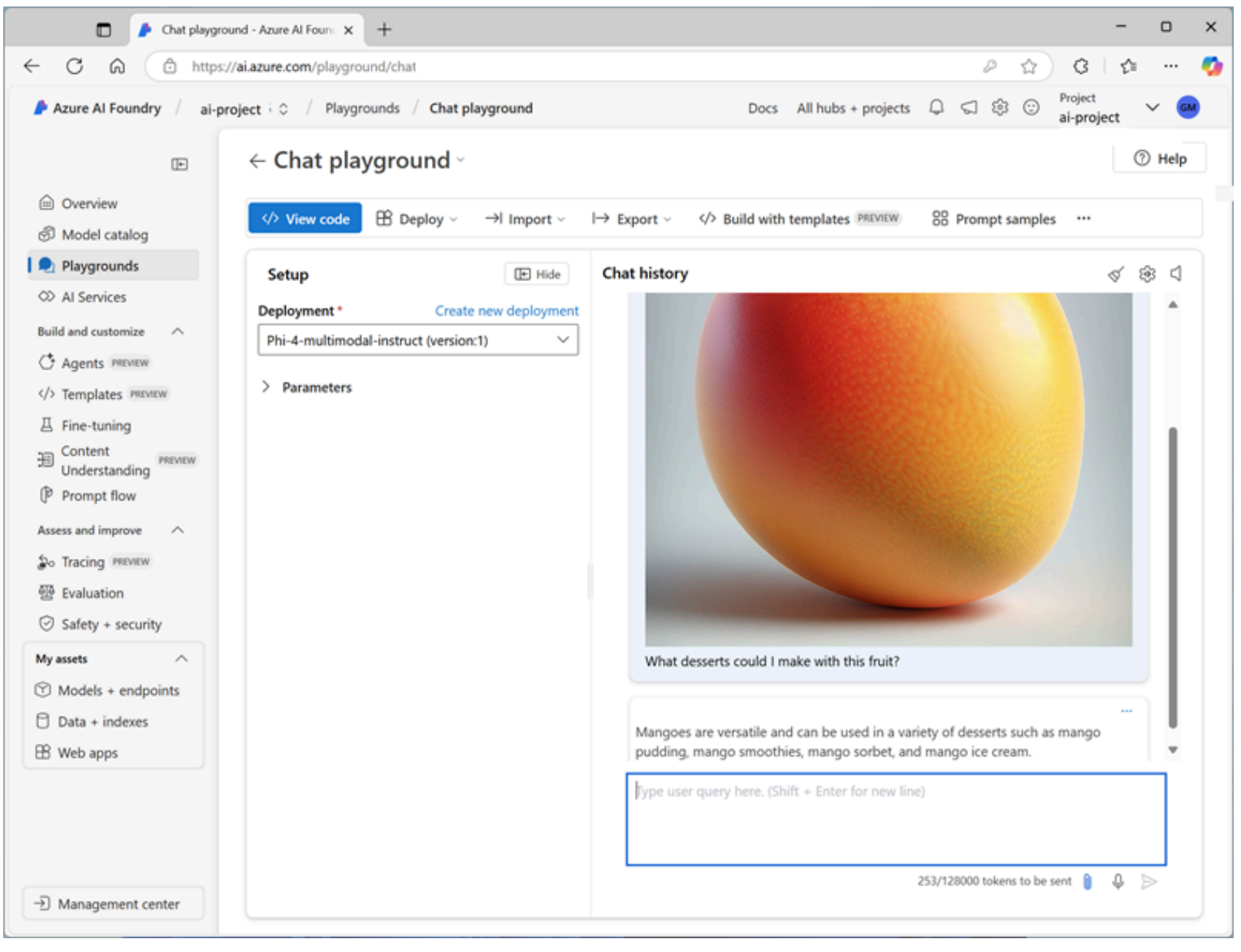
2. In a new browser tab, download mango.jpeg from

   ```
   https://github.com/MicrosoftLearning/mslearn-ai-vision/raw/refs/heads/main/Labfiles/gen-ai-vision/mango.jpeg
   ```

   and save it to a folder on your local file system.

3. On the chat playground page, in the **Setup** pane, ensure that your **Phi-4-multimodal-instruct** model model deployment is selected.

4. In the main chat session panel, under the chat input box, use the attach button (📎) to upload the *mango.jpeg* image file, and then add the text `What desserts could I make with this fruit?` and submit the prompt.



5. Review the response, which should hopefully provide relevant guidance for desserts you can make using a mango.

Create a client application

Now that you've deployed the model, you can use the deployment in a client application.

Prepare the application configuration

1. In the Azure AI Foundry portal, view the **Overview** page for your project.

2. In the **Endpoints and keys** area, ensure the **Azure AI Foundry** library is selected, and note the **Azure AI Foundry project endpoint**. You'll use this connection string to connect to your project in a client application.

3. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

4. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

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

5. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

6. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-vision -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

7. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-vision/Labfiles/gen-ai-vision/python
   ```

8. In the cloud shell command line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-identity azure-ai-projects openai
   ```

9. Enter the following command to edit the configuration file that has been provided:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

10. In the code file, replace the **your_project_endpoint** placeholder with the Foundry project endpoint (copied from the project **Overview** page in the Azure AI Foundry portal), and the **your_model_deployment** placeholder with the name you assigned to your Phi-4-multimodal-instruct model deployment.

11. After you've replaced the placeholders, in the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Write code to connect to your project and get a chat client for your model

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code chat-app.py
```

2. In the code file, note the existing statements that have been added at the top of the file to import the necessary SDK namespaces. Then, Find the comment **Add references**, add the following code to reference the namespaces in the libraries you installed previously:

Code Copy

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient
from openai import AzureOpenAI
```

3. In the **main** function, under the comment **Get configuration settings**, note that the code loads the project connection string and model deployment name values you defined in the configuration file.
4. In the **main** function, under the comment **Get configuration settings**, note that the code loads the project connection string and model deployment name values you defined in the configuration file.
5. Find the comment **Initialize the project client**, and add the following code to connect to your Azure AI Foundry project:

> **Tip**: Be careful to maintain the correct indentation level for your code.

Code Copy

```
# Initialize the project client
project_client = AIProjectClient(
         credential=DefaultAzureCredential(
             exclude_environment_credential=True,
             exclude_managed_identity_credential=True
         ),
         endpoint=project_endpoint,
     )
```

6. Find the comment **Get a chat client**, and add the following code to create a client object for chatting with a model:

Code Copy

```
# Get a chat client
openai_client = project_client.get_openai_client(api_version="2024-10-21")
```

Write code to submit a URL-based image prompt

1. Note that the code includes a loop to allow a user to input a prompt until they enter “quit”. Then in the loop section, find the comment **Get a response to image input**, add the following code to submit a prompt that includes the following image:

Code Copy

```
# Get a response to image input
image_url = "https://github.com/MicrosoftLearning/mslearn-ai-vision/raw/refs/heads/main/Labfiles/gen-ai-vision/orange.jpeg"
image_format = "jpeg"
request = Request(image_url, headers={"User-Agent": "Mozilla/5.0"})
image_data = base64.b64encode(urlopen(request).read()).decode("utf-8")
data_url = f"{image_data}"

response = openai_client.chat.completions.create(
     model=model_deployment,
     messages=[
         {"role": "system", "content": system_message},
         { "role": "user", "content": [
             { "type": "text", "text": prompt},
             { "type": "image_url", "image_url": {"url": data_url}}
         ] }
     ]
)
print(response.choices[0].message.content)
```

2. Use the **CTRL+S** command to save your changes to the code file - don't close it yet though.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. After you have signed in, enter the following command to run the application:

```
python chat-app.py
```

4. When prompted, enter the following prompt:

```
Suggest some recipes that include this fruit
```

5. Review the response. Then enter `quit` to exit the program.

Modify the code to upload a local image file

1. In the code editor for your app code, in the loop section, find the code you added previously under the comment **Get a response to image input**. Then modify the code as follows, to upload this local image file:

Code Copy

```
# Get a response to image input
script_dir = Path(__file__).parent  # Get the directory of the script
image_path = script_dir / 'mystery-fruit.jpeg'
mime_type = "image/jpeg"

# Read and encode the image file
with open(image_path, "rb") as image_file:
     base64_encoded_data = base64.b64encode(image_file.read()).decode('utf-8')

# Include the image file data in the prompt
data_url = f"data:{mime_type};base64,{base64_encoded_data}"
response = openai_client.chat.completions.create(
         model=model_deployment,
         messages=[
             {"role": "system", "content": system_message},
             { "role": "user", "content": [
                 { "type": "text", "text": prompt},
                 { "type": "image_url", "image_url": {"url": data_url}}
             ] }
         ]
)
print(response.choices[0].message.content)
```

2. Use the **CTRL+S** command to save your changes to the code file. You can also close the code editor (**CTRL+Q**) if you like.
3. In the cloud shell command line pane beneath the code editor, enter the following command to run the app:

Code Copy

```
python chat-app.py
```

4. When prompted, enter the following prompt:

Code Copy

```
What is this fruit? What recipes could I use it in?
```

5. Review the response. Then enter `quit` to exit the program.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

Clean up

If you've finished exploring Azure AI Foundry portal, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Open the Azure portal and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Generate images with AI

In this exercise, you use the OpenAI DALL-E generative AI model to generate images. You also use the OpenAI Python SDK to create a simple app to generate images based on your prompts.

> **Note**: This exercise is based on pre-release SDK software, which may be subject to change. Where necessary, we've used specific versions of packages; which may not reflect the latest available versions. You may experience some unexpected behavior, warnings, or errors.

While this exercise is based on the OpenAI Python SDK, you can develop AI chat applications using multiple language-specific SDKs; including:

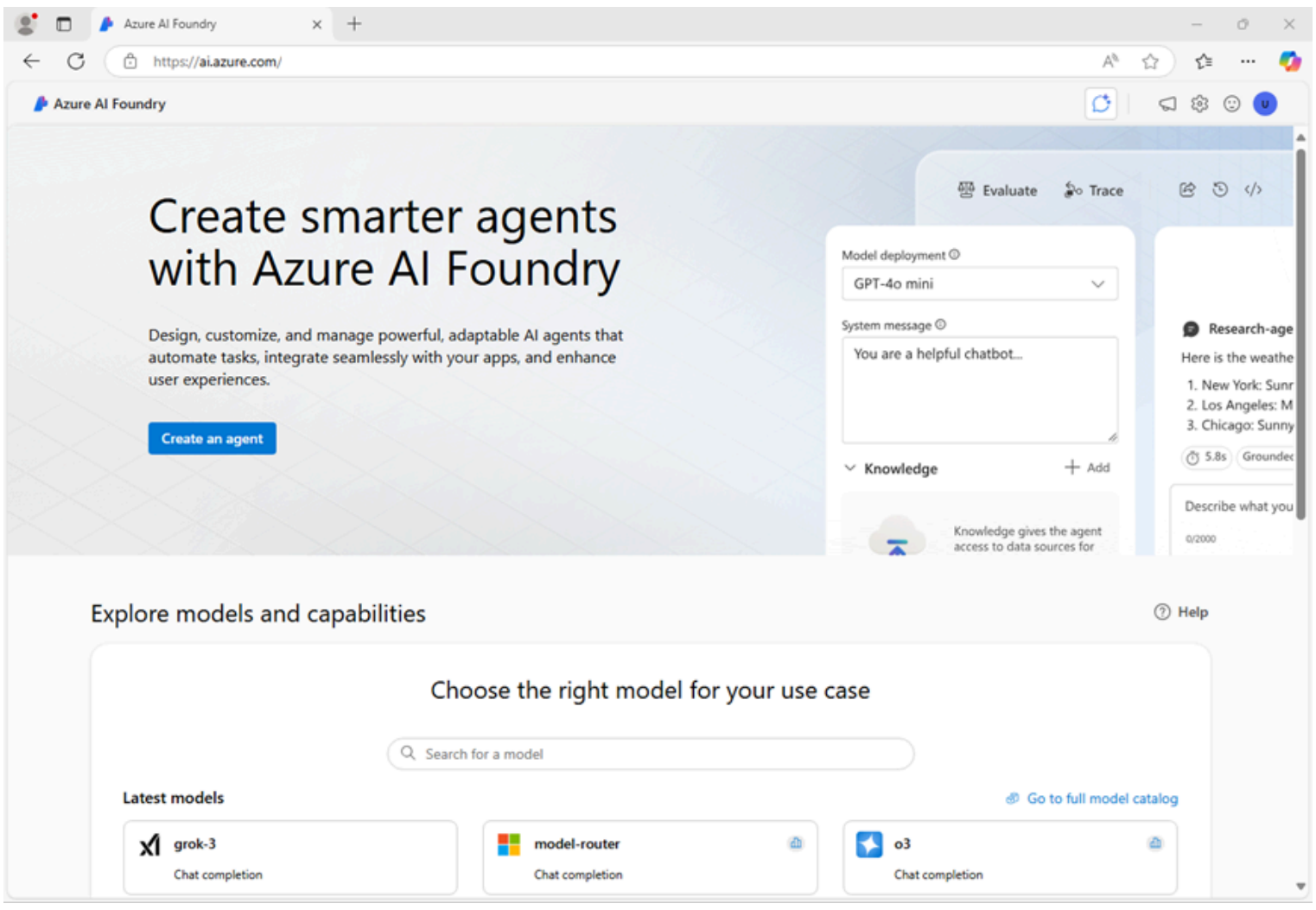
- OpenAI Projects for Microsoft .NET
- OpenAI Projects for JavaScript

This exercise takes approximately **30** minutes.

Open Azure AI Foundry portal

Let's start by signing into Azure AI Foundry portal.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. Review the information on the home page.

Choose a model to start a project

An Azure AI *project* provides a collaborative workspace for AI development. Let's start by choosing a model that we want to work with and creating a project to use it in.

> **Note**: AI Foundry projects can be based on an *Azure AI Foundry* resource, which provides access to AI models (including Azure OpenAI), Azure AI services, and other resources for developing AI agents and chat solutions. Alternatively, projects can be based on *AI hub* resources; which include connections to Azure resources for secure storage, compute, and specialized tools. Azure AI Foundry based projects are great for developers who want to manage resources for AI agent or chat app development. AI hub based projects are more suitable for enterprise development teams working on complex AI solutions.

1. In the home page, in the **Explore models and capabilities** section, search for the `dall-e-3` model; which we'll use in our project.
2. In the search results, select the **dall-e-3** model to see its details, and then at the top of the page for the model, select **Use this model**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Select **Customize** and specify the following settings for your hub:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Select any **AI Foundry recommended****

    > * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

5. Select **Create** and wait for your project, including the dall-e-3 model deployment you selected, to be created.

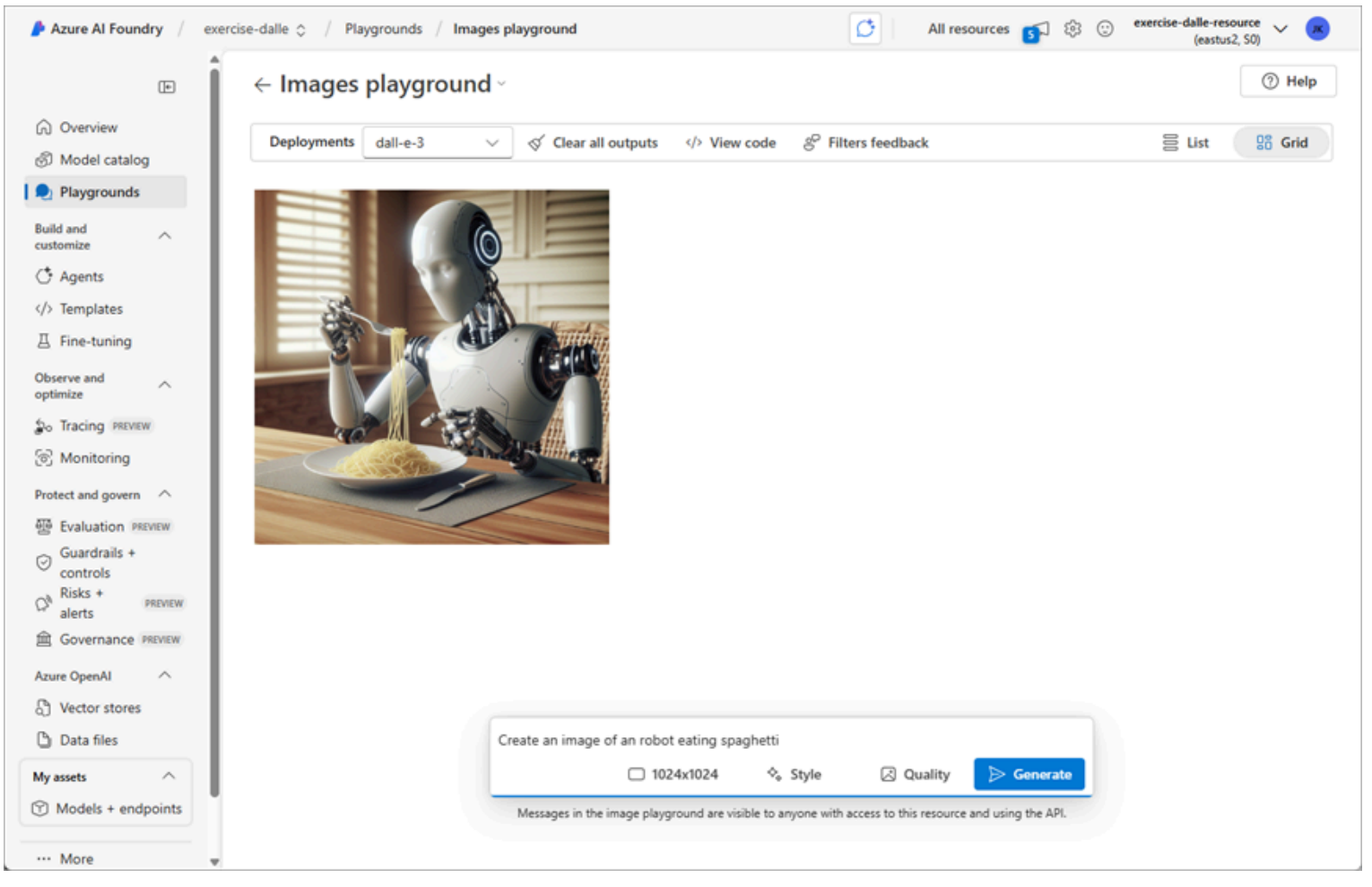
    > Note: Depending on your model selection you might receive additional prompts during the project creation process. Agree to the terms and finalize the deployment.

6. When your project is created, your model will be displayed in the **Models + endpoints** page.

Test the model in the playground

Before creating a client application, let's test the DALL-E model in the playground.

1. Select **Playgrounds**, and then **Images playground**.
2. Ensure your DALL-E model deployment is selected. Then, in the box near the bottom of the page, enter a prompt such as `Create an image of an robot eating spaghetti` and select **Generate**.
3. Review the resulting image in the playground:



4. Enter a follow-up prompt, such as `Show the robot in a restaurant` and review the resulting image.
5. Continue testing with new prompts to refine the image until you are happy with it.
6. Select the **</> View Code** button and ensure you are on the **Entra ID authentication** tab. Then record the following information for use later in the exercise. Note the values are examples, be sure to record the information from your deployment.
    - OpenAI Endpoint: *https://dall-e-aus-resource.cognitiveservices.azure.com/*
    - OpenAI API version: *2024-04-01-preview*
    - Deployment name (model name): *dall-e-3*

Create a client application

The model seems to work in the playground. Now you can use the OpenAI SDK to use it in a client application.

Prepare the application configuration

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.
2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

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

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

Code Copy

```
rm -r mslearn-ai-vision -f
git clone https://github.com/MicrosoftLearning/mslearn-ai-vision
```

> **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, navigate to the folder containing the application code files:

Code Copy

```
cd mslearn-ai-vision/Labfiles/dalle-client/python
```

6. In the cloud shell command line pane, enter the following command to install the libraries you'll use:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-identity openai requests
```

7. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

   The file is opened in a code editor.

8. Replace the **your_endpoint**, **your_model_deployment**, and **your_api_version** placeholders with the values you recorded from the from the **Images playground**.

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Write code to connect to your project and chat with your model

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code dalle-client.py
```

2. In the code file, note the existing statements that have been added at the top of the file to import the necessary SDK namespaces. Then, under the comment **Add references**, add the following code to reference the namespaces in the libraries you installed previously:

Code Copy

```
# Add references
from dotenv import load_dotenv
from azure.identity import DefaultAzureCredential, get_bearer_token_provider
from openai import AzureOpenAI
import requests
```

3. In the **main** function, under the comment **Get configuration settings**, note that the code loads the endpoint, API version, and model deployment name values you defined in the configuration file.

4. Under the comment **Initialize the client**, add the following code to connect to your model using the Azure credentials you are currently signed in with:

Code Copy

```
# Initialize the client
token_provider = get_bearer_token_provider(
    DefaultAzureCredential(exclude_environment_credential=True,
        exclude_managed_identity_credential=True),
    "https://cognitiveservices.azure.com/.default"
)

client = AzureOpenAI(
    api_version=api_version,
    azure_endpoint=endpoint,
    azure_ad_token_provider=token_provider
)
```


5. Note that the code includes a loop to allow a user to input a prompt until they enter “quit”. Then in the loop section, under the comment **Generate an image**, add the following code to submit the prompt and retrieve the URL for the generated image from your model:

Python

Code Copy

```
# Generate an image
result = client.images.generate(
     model=model_deployment,
     prompt=input_text,
     n=1
 )

json_response = json.loads(result.model_dump_json())
image_url = json_response["data"][0]["url"]
```

6. Note that the code in the remainder of the **main** function passes the image URL and a filename to a provided function, which downloads the generated image and saves it as a .png file.

7. Use the **CTRL+S** command to save your changes to the code file and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Run the client application

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *–tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.

3. In the cloud shell command line pane, enter the following command to run the app:

Code Copy

```
python dalle-client.py
```

4. When prompted, enter a request for an image, such as `Create an image of a robot eating pizza`. After a moment or two, the app should confirm that the image has been saved.

5. Try a few more prompts. When you're finished, enter `quit` to exit the program.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

6. To download and view the images that were generated by your app, use the cloud shell **download** command - specifying the .png file that was generated:

Code Copy

```
download ./images/image_1.png
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file.

Summary

In this exercise, you used Azure AI Foundry and the Azure OpenAI SDK to create a client application uses a DALL-E model to generate images.

Clean up

If you've finished exploring DALL-E, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Extract information from multimodal content

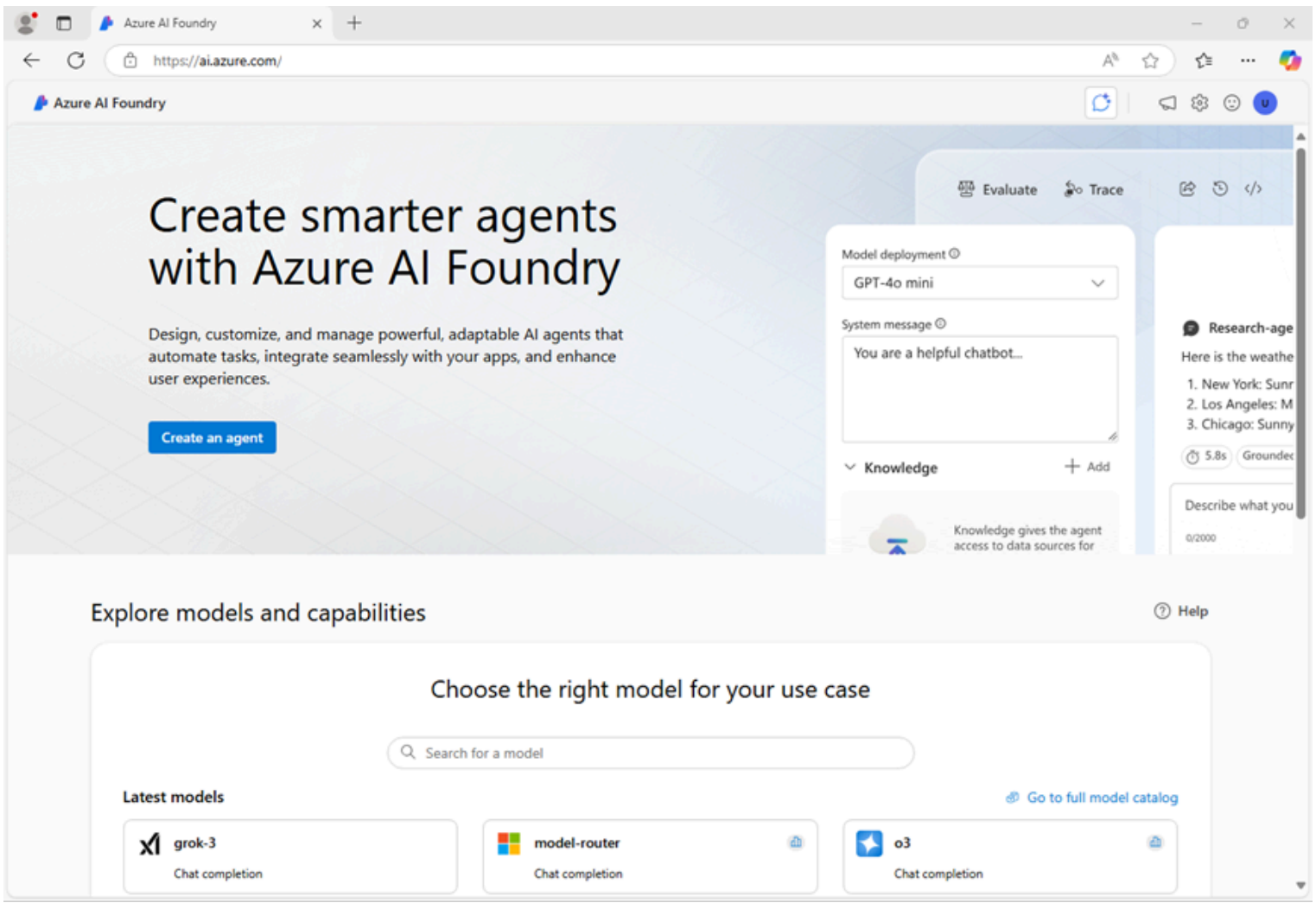
In this exercise, you use Azure Content Understanding to extract information from a variety of content types; including an invoice, an images of a slide containing charts, an audio recording of a voice messages, and a video recording of a conference call.

This exercise takes approximately **40** minutes.

Create an Azure AI Foundry hub and project

The features of Azure AI Foundry we're going to use in this exercise require a project that is based on an Azure AI Foundry *hub* resource.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: Select one of the following locations (*At the time of writing, Azure AI Content understanding is only available in these regions*):
        - Australia East
        - Sweden Central
        - West US

> **Note**: If you're working in an Azure subscription in which policies are used to restrict allowable resource names, you may need to use the link at the bottom of the **Create a new project** dialog box to create the hub using the Azure portal.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

4. Wait for your project to be created.

Download content

The content you're going to analyze is in a .zip archive. Download it and extract it in a local folder.

1. In a new browser tab, download content.zip from
   ```
   https://github.com/microsoftlearning/mslearn-ai-information-extraction/raw/main/Labfiles/content/content.zip
   ```
   and save it in a local folder.
2. Extract the downloaded *content.zip* file and view the files it contains. You'll use these files to build various Content Understanding analyzers in this exercise.

> **Note**: If you're only interested in exploring analysis of a specific modality (documents, images, video, or audio), you can skip to the relevant task below. For the best experience, go through each task to learn how to extract information from different types of content.

Extract information from invoice documents

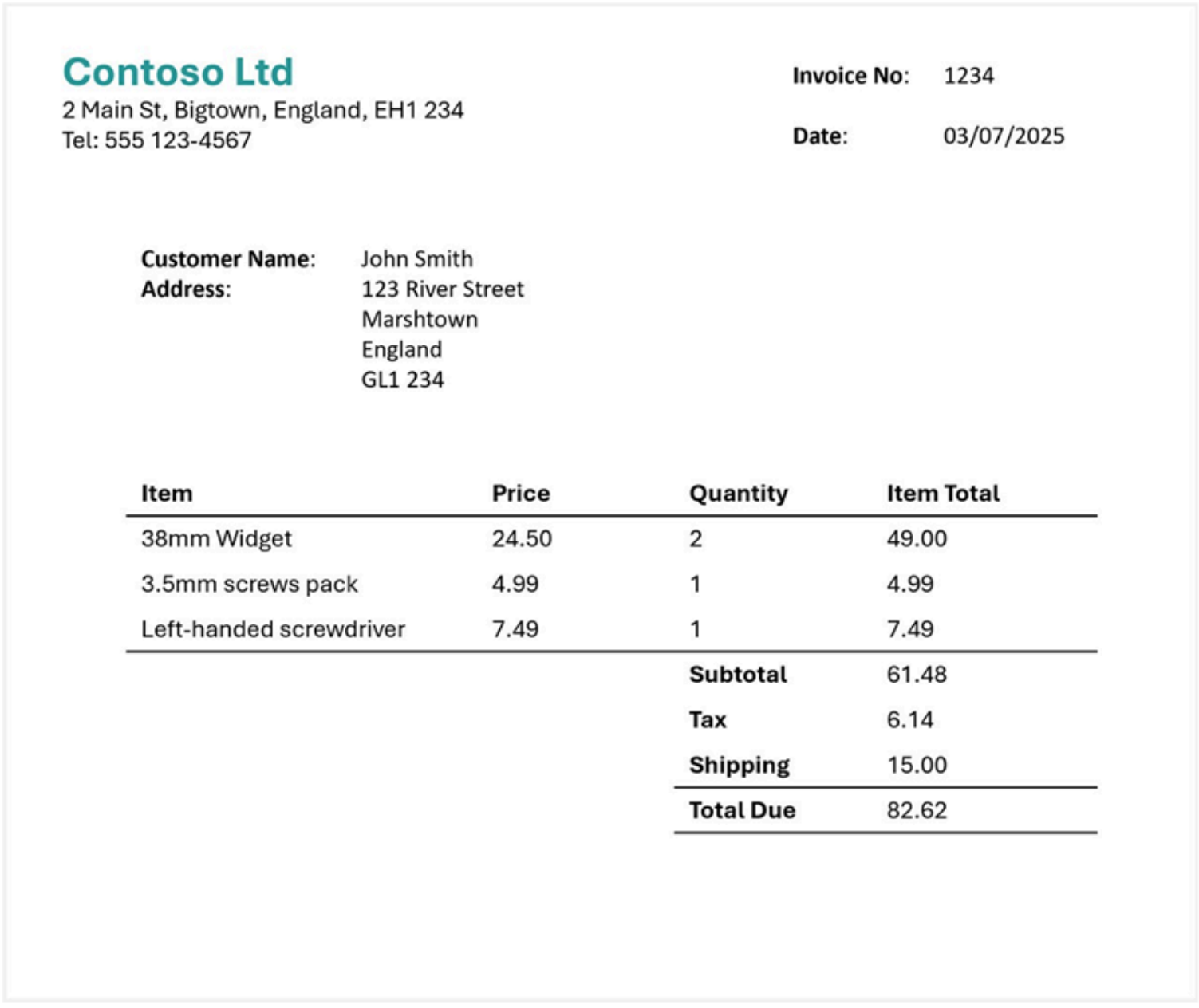
You are going to build an Azure AI Content Understanding analyzer that can extract information from invoices. You'll start by defining a schema based on a sample invoice.

Define a schema for invoice analysis

1. In the browser tab containing the home page for your Azure AI Foundry project; in the navigation pane on the left, select **Content Understanding**.
2. On the **Content Understanding** page, select the **Custom task** tab at the top.
3. On the Content Understanding custom task page, select **+ Create**, and create a task with the following settings:
   - **Task name**: `Invoice analysis`
   - **Description**: `Extract data from an invoice`
   - **Single file content analysis**: *Selected*
   - **Advanced settings**:
     - **Azure AI services connection**: *The Azure AI Services resource in your Azure AI Foundry hub*
     - **Azure Blob Storage account**: *The default storage account in your Azure AI Foundry hub*
4. Wait for the task to be created.

> **Tip**: If an error accessing storage occurs, wait a minute and try again. Permissions for a new hub may take a few minutes to propagate.

5. On the **Define schema** page, upload the **invoice-1234.pdf** file from the folder where you extracted content files. This file contains the following invoice:



Contoso Ltd
2 Main St, Bigtown, England, EH1 234
Tel: 555 123-4567

Invoice No: 1234
Date: 03/07/2025

Customer Name: John Smith
Address: 123 River Street
Marshtown
England
GL1 234

Item	Price	Quantity	Item Total
38mm Widget	24.50	2	49.00
3.5mm screws pack	4.99	1	4.99
Left-handed screwdriver	7.49	1	7.49
		Subtotal	61.48
		Tax	6.14
		Shipping	15.00
		Total Due	82.62

6. On the **Define schema** page, after uploading the invoice file, select the **Invoice data extraction** template and select **Create**.

   The *Invoice analysis* template includes common fields that are found in invoices. You can use the schema editor to delete any of the suggested fields that you don't need, and add any custom fields that you do.
7. In the list of suggested fields, select **BillingAddress**. This field is not needed for the invoice format you have uploaded, so use the **Delete field** (🗑) icon that appears in the selected field row to delete it.
8. Now delete the following suggested fields, which aren't needed for your invoice schema:

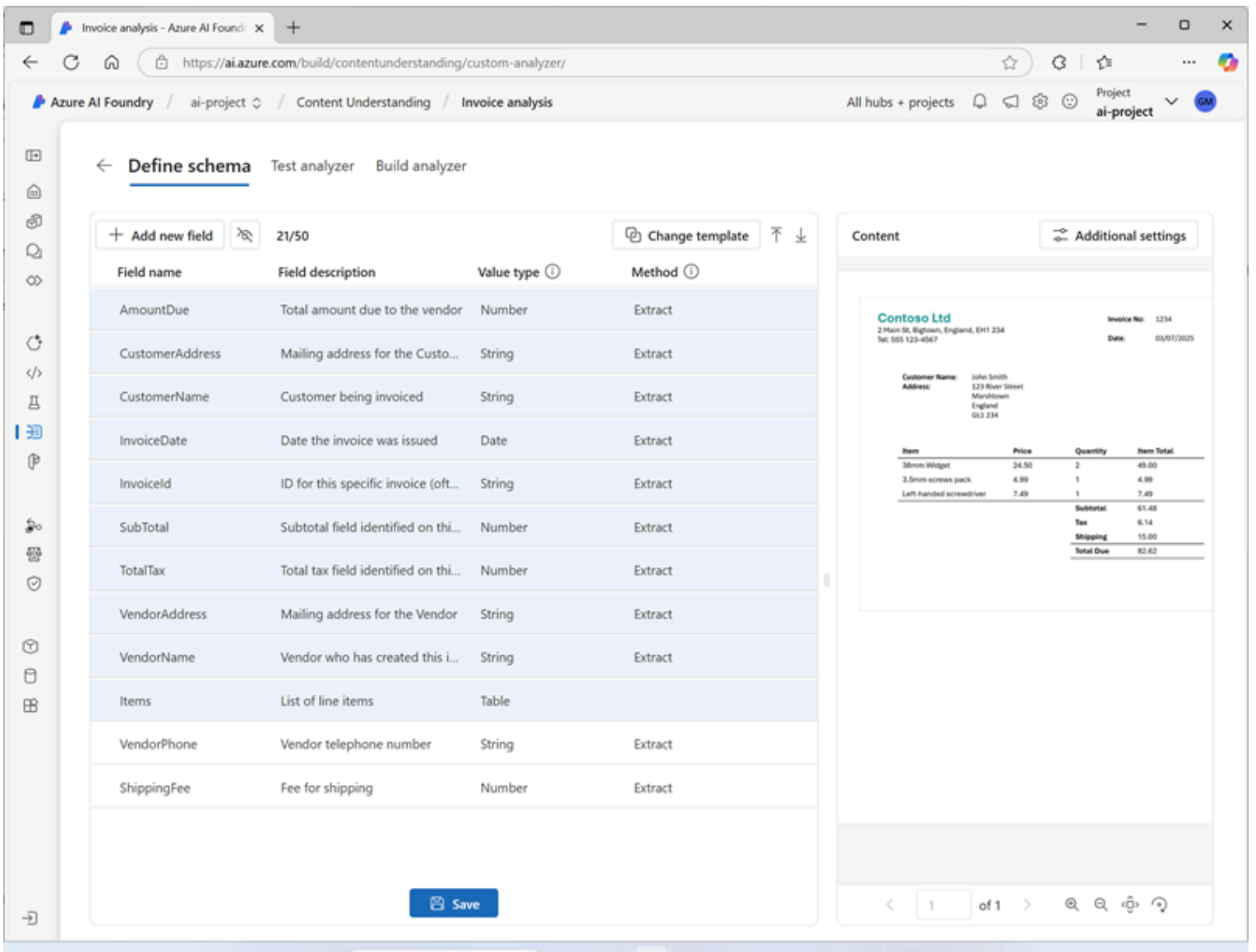
   - BillingAddressRecipient
   - CustomerAddressRecipient
   - CustomerId
   - CustomerTaxId
   - DueDate
   - InvoiceTotal
   - PaymentTerm
   - PreviousUnpaidBalance
   - PurchaseOrder
   - RemittanceAddress
   - RemittanceAddressRecipient
   - ServiceAddress
   - ServiceAddressRecipient
   - ShippingAddress
   - ShippingAddressRecipient
   - TotalDiscount
   - VendorAddressRecipient
   - VendorTaxId
   - TaxDetails

9. Use **+ Add new field** button to add the following fields, selecting **Save changes** (✓) for each new field:

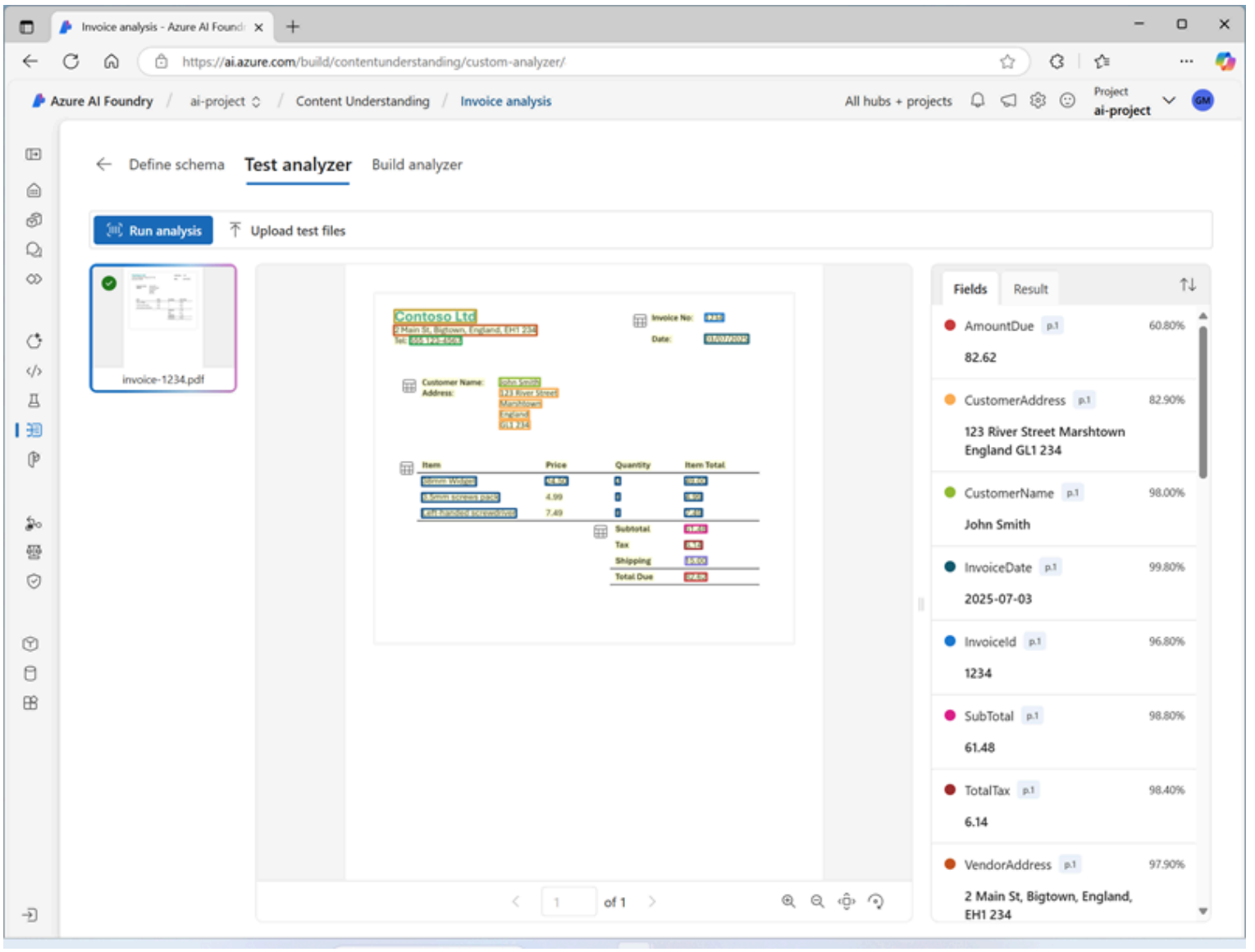
Field name	Field description	Value type	Method
`VendorPhone`	`Vendor telephone number`	String	Extract

Field name	Field description	Value type	Method
`ShippingFee`	`Fee for shipping`	Number	Extract

10. In the row for the **Items** field, note that this field is a *table* (it contains the collection of items in the invoice). Select it's **Edit** (▦) icon to open a new page with its subfields.
11. Remove the following subfields from the **Items** table:
    - Date
    - ProductCode
    - Unit
    - TaxAmount
    - TaxRate
12. Use the **OK** button to confirm the changes and return to the top-level of the invoice schema.
13. Verify that your completed schema looks like this, and select **Save**.



14. On the **Test Analyzer** page, if analysis does not begin automatically, select **Run analysis**. Then wait for analysis to complete.
15. Review the analysis results, which should look similar to this:



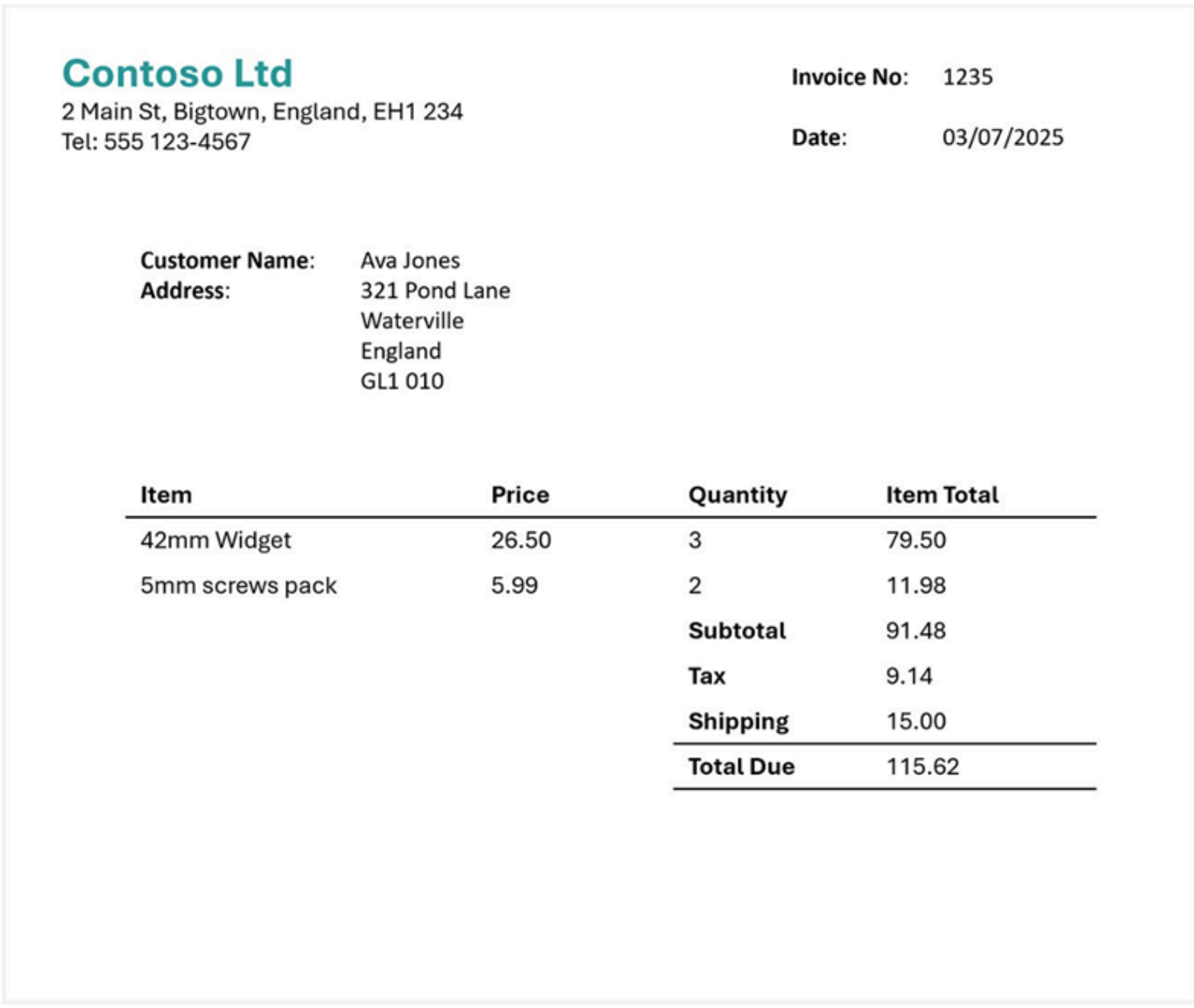
16. View the details of the fields that were identified in the **Fields** pane.

Build and test an analyzer for invoices

Now that you have trained a model to extract fields from invoices, you can build an analyzer to use with similar documents.

1. Select the **Analyzer list** page, and then select **+ Build analyzer** and build a new analyzer with the following properties (typed exactly as shown here):
    - **Name**: `invoice-analyzer`
    - **Description**: `Invoice analyzer`
2. Wait for the new analyzer to be ready (use the **Refresh** button to check).
3. When the analyzer has been built, select the **invoice-analyzer** link. The fields defined in the analyzer's schema will be displayed.
4. In the **invoice-analyzer** page, select the **Test** tab.
5. Use the **+ Upload test files** button to upload **invoice-1235.pdf** from the folder where you extracted the content files, and click on **Run analysis** to extract field data from the invoice.

The invoice being analyzed looks like this:



Contoso Ltd
2 Main St, Bigtown, England, EH1 234
Tel: 555 123-4567

Invoice No: 1235
Date: 03/07/2025

Customer Name: Ava Jones
Address: 321 Pond Lane
Waterville
England
GL1 010

Item	Price	Quantity	Item Total
42mm Widget	26.50	3	79.50
5mm screws pack	5.99	2	11.98
		Subtotal	91.48
		Tax	9.14
		Shipping	15.00
		Total Due	115.62

6. Review the **Fields** pane, and verify that the analyzer extracted the correct fields from the test invoice.
7. Review the **Results** pane to see the JSON response that the analyzer would return to a client application.
8. On the **Code example** tab, view the sample code that you could use to develop a client application that uses the Content Understanding REST interface to call your analyzer.
9. Close the **invoice-analyzer** page.

Extract information from a slide image

You are going to build an Azure AI Content Understanding analyzer that can extract information from a slide containing charts.

Define a schema for image analysis

1. In the browser tab containing the home page for your Azure AI Foundry project; in the navigation pane on the left, select **Content Understanding**.
2. On the **Content Understanding** page, select the **Custom task** tab at the top.
3. On the Content Understanding custom task page, select **+ Create**, and create a task with the following settings:
    - **Task name**: `Slide analysis`
    - **Description**: `Extract data from an image of a slide`
    - **Single file content analysis**: *Selected*
    - **Advanced settings**:
        - **Azure AI services connection**: *The Azure AI Services resource in your Azure AI Foundry hub*
        - **Azure Blob Storage account**: *The default storage account in your Azure AI Foundry hub*
4. Wait for the task to be created.

> **Tip**: If an error accessing storage occurs, wait a minute and try again. Permissions for a new hub may take a few minutes to propagate.

5. On the **Define schema** page, upload the **slide-1.jpg** file from the folder where you extracted content files. Then select the **Image analysis** template and select **Create**.

The *Image analysis* template doesn't include any predefined fields. You must define fields to describe the information you want to extract.

6. Use **+ Add new field** button to add the following fields, selecting **Save changes** (✓) for each new field:

Field name	Field description	Value type	Method
`Title`	`Slide title`	String	Generate
`Summary`	`Summary of the slide`	String	Generate
`Charts`	`Number of charts on the slide`	Integer	Generate

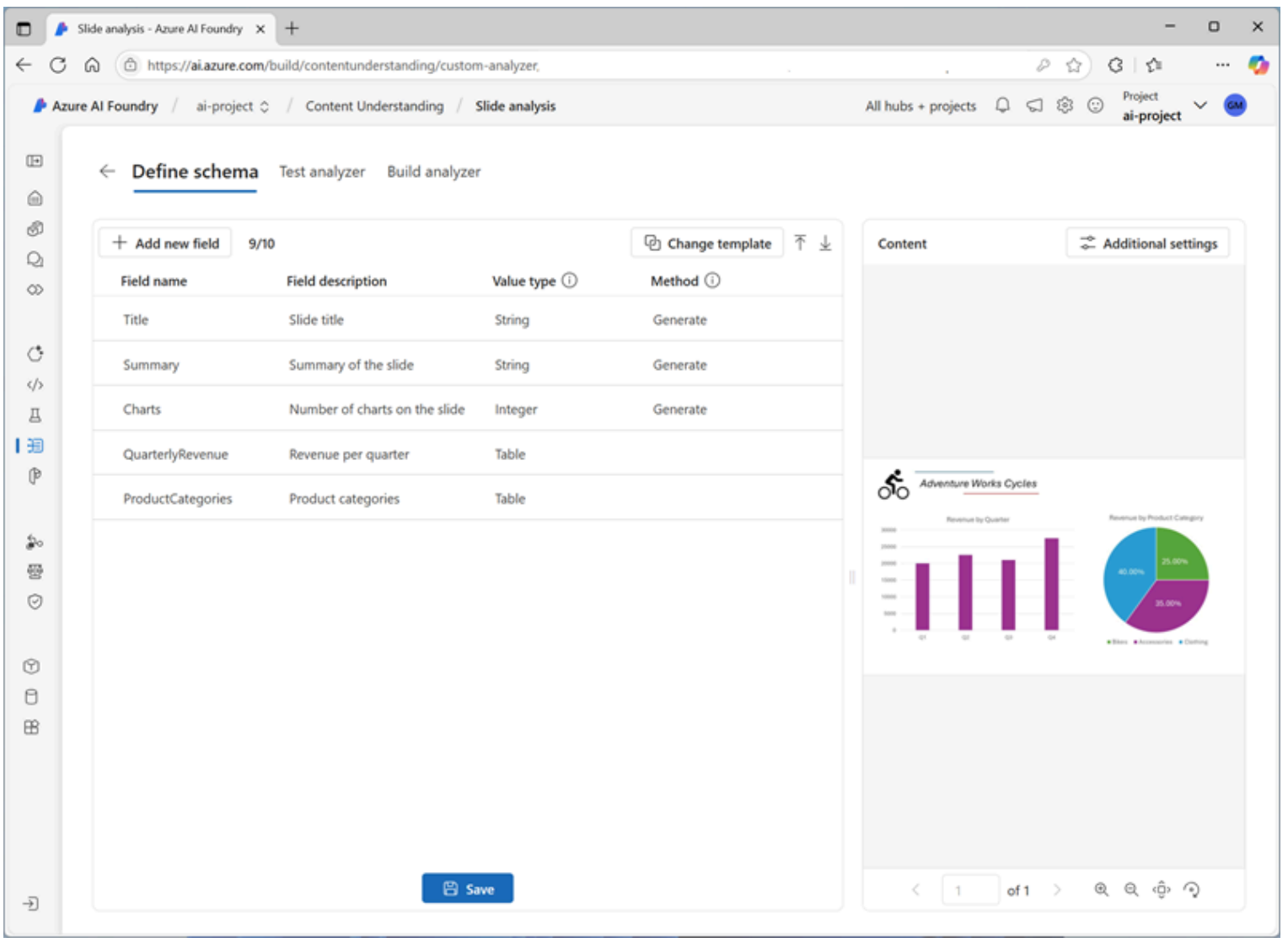
7. Use **+ Add new field** button to add a new field named `QuarterlyRevenue` with the description `Revenue per quarter` with the value type **Table**, and save the new field (✓). Then, in the new page for the table subfields that opens, add the following subfields:

Field name	Field description	Value type	Method
`Quarter`	`Which quarter?`	String	Generate
`Revenue`	`Revenue for the quarter`	Number	Generate

8. Select **Back** (the arrow icon near the **Add new subfield** button) or ✓ **OK** to return to the top level of your schema, and use **+ Add new field** button to add a new field named `ProductCategories` with the description `Product categories` with the value type **Table**, and save the new field (✓). Then, in the new page for the table subfields that opens, add the following subfields:

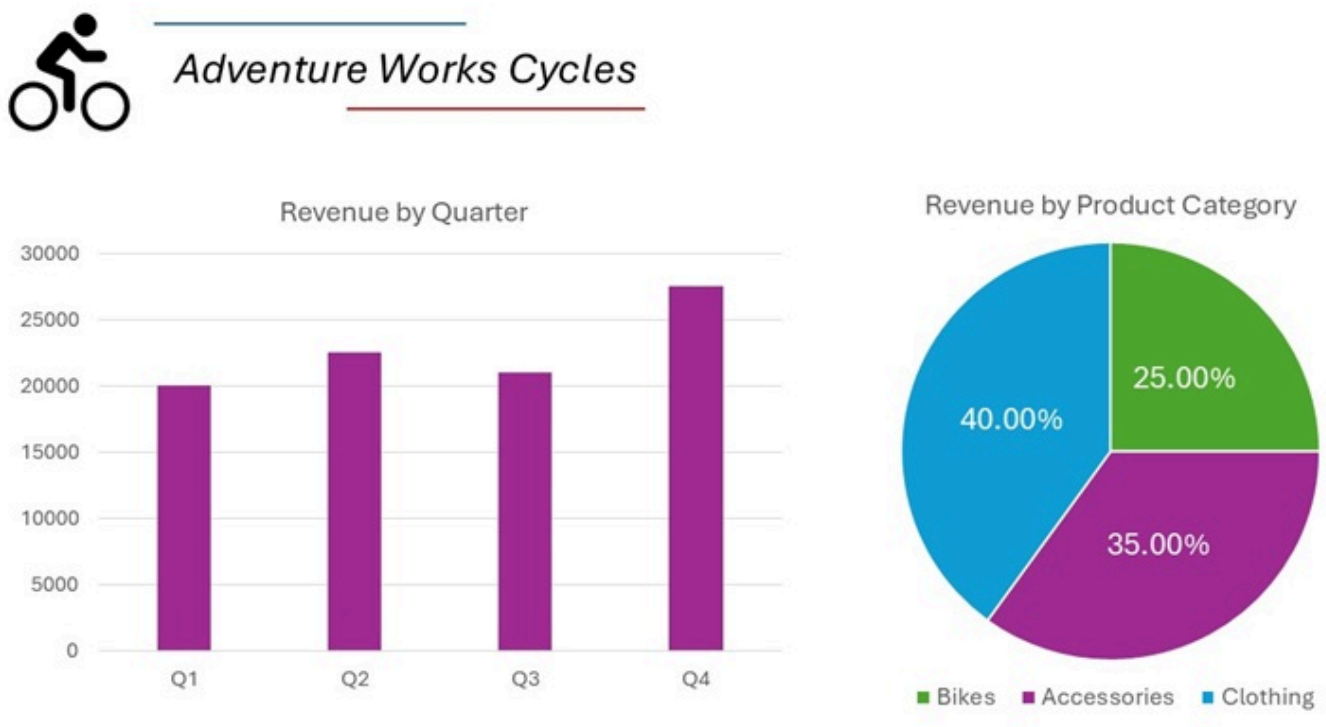
Field name	Field description	Value type	Method
`ProductCategory`	`Product category name`	String	Generate
`RevenuePercentage`	`Percentage of revenue`	Number	Generate

9. Select **Back** (the arrow icon near the **Add new subfield** button) or ✓ **OK** to return to the top level of your schema, and verify that it looks like this. Then select **Save**.

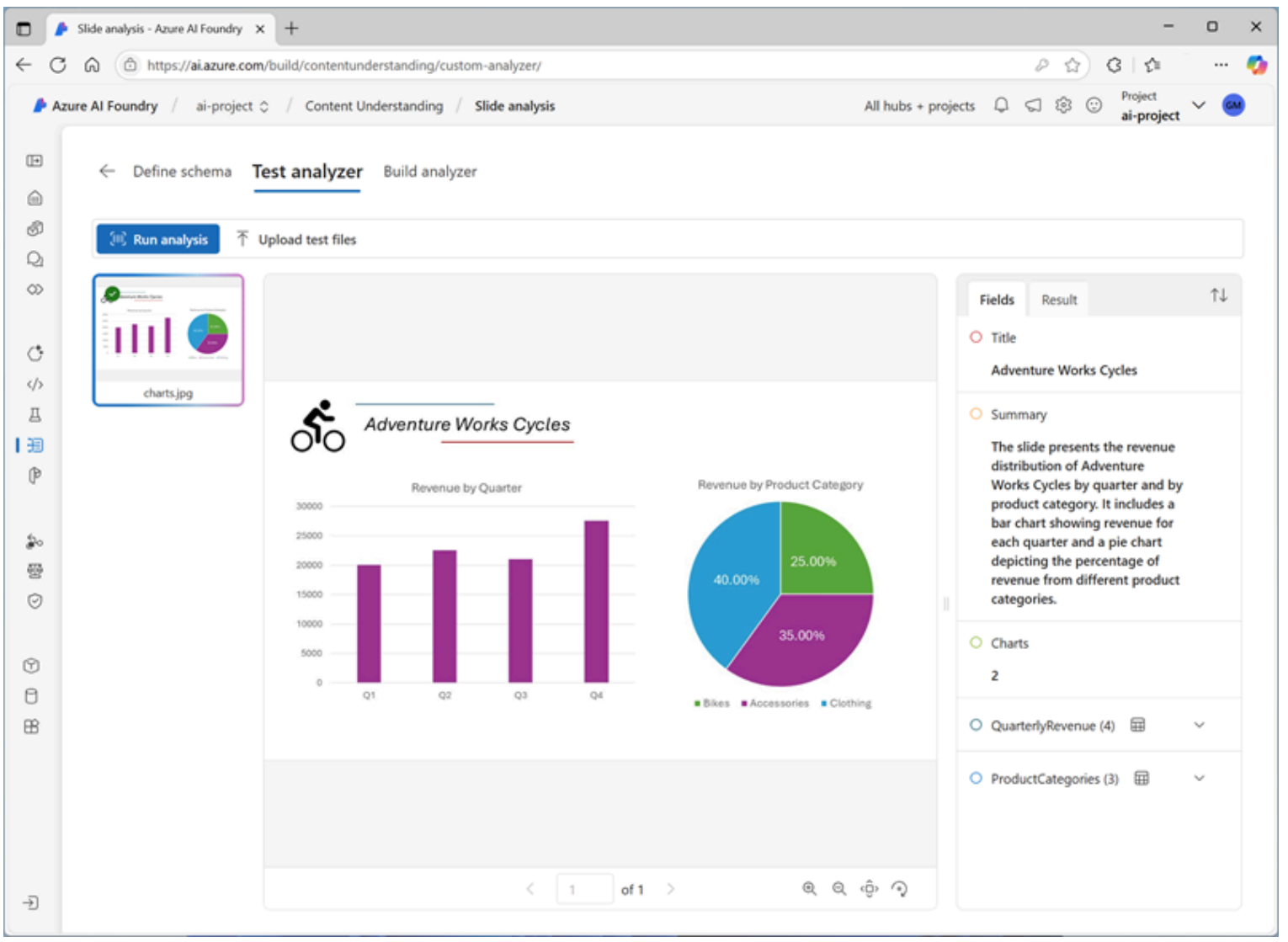


10. On the **Test Analyzer** page, if analysis does not begin automatically, select **Run analysis**. Then wait for analysis to complete.

The slide being analyzed looks like this:



11. Review the analysis results, which should look similar to this:



12. View the details of the fields that were identified in the **Fields** pane, expanding the **QuarterlyRevenue** and **ProductCategories** fields to see the subfield values.

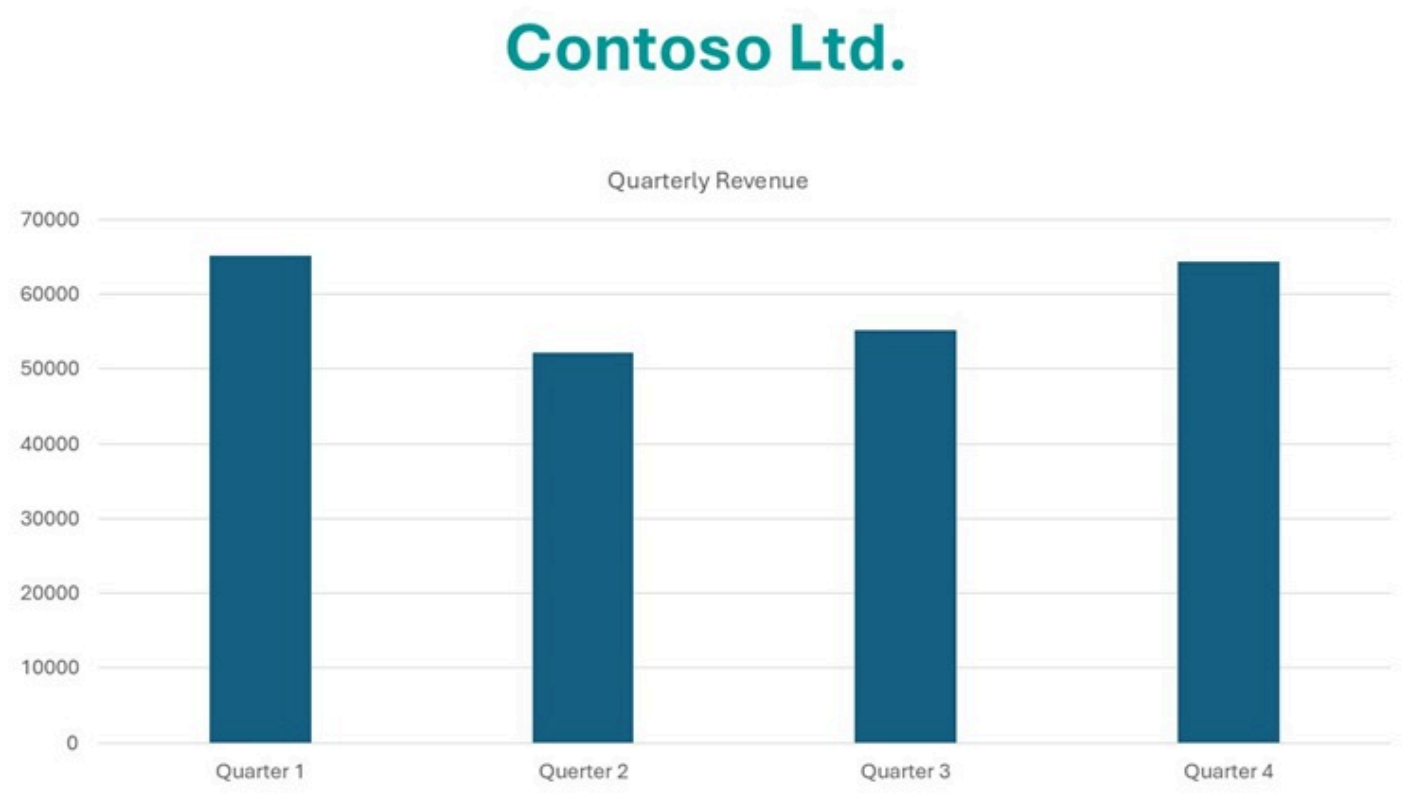
Build and test an analyzer

Now that you have trained a model to extract fields from slides, you can build an analyzer to use with similar slide images.

1. Select the **Analyzer list** page, and then select **+ Build analyzer** and build a new analyzer with the following properties (typed exactly as shown here):
    - **Name**: `slide-analyzer`
    - **Description**: `Slide image analyzer`
2. Wait for the new analyzer to be ready (use the **Refresh** button to check).
3. When the analyzer has been built, select the **slide-analyzer** link. The fields defined in the analyzer's schema will be displayed.
4. In the **slide-analyzer** page, select the **Test** tab.

5. Use the **+ Upload test files** button to upload **slide-2.jpg** from the folder where you extracted the content files, and click on **Run analysis** to extract field data from the image.

   The slide being analyzed looks like this:



6. Review the **Fields** pane, and verify that the analyzer extracted the correct fields from the slide image.

   > **Note**: Slide 2 doesn't include a breakfown by product category, so the product category revenue data is not found.

7. Review the **Results** pane to see the JSON response that the analyzer would return to a client application.
8. On the **Code example** tab, view the sample code that you could use to develop a client application that uses the Content understanding REST interface to call your analyzer.
9. Close the **slide-analyzer** page.

Extract information from a voicemail audio recording

You are going to build an Azure AI Content Understanding analyzer that can extract information from an audio recording of a voicemail message.

Define a schema for audio analysis

1. In the browser tab containing the home page for your Azure AI Foundry project; in the navigation pane on the left, select **Content Understanding**.
2. On the **Content Understanding** page, select the **Custom task** tab at the top.
3. On the Content Understanding custom task page, select **+ Create**, and create a task with the following settings:
   - **Task name**: `Voicemail analysis`
   - **Description**: `Extract data from a voicemail recording`
   - **Single file content analysis**: *Selected*
   - **Advanced settings**:
     - **Azure AI services connection**: *The Azure AI Services resource in your Azure AI Foundry hub*
     - **Azure Blob Storage account**: *The default storage account in your Azure AI Foundry hub*
4. Wait for the task to be created.

   > **Tip**: If an error accessing storage occurs, wait a minute and try again. Permissions for a new hub may take a few minutes to propagate.

5. On the **Define schema** page, upload the **call-1.mp3** file from the folder where you extracted content files. Then select the **Speech transcript analysis** template and select **Create**.

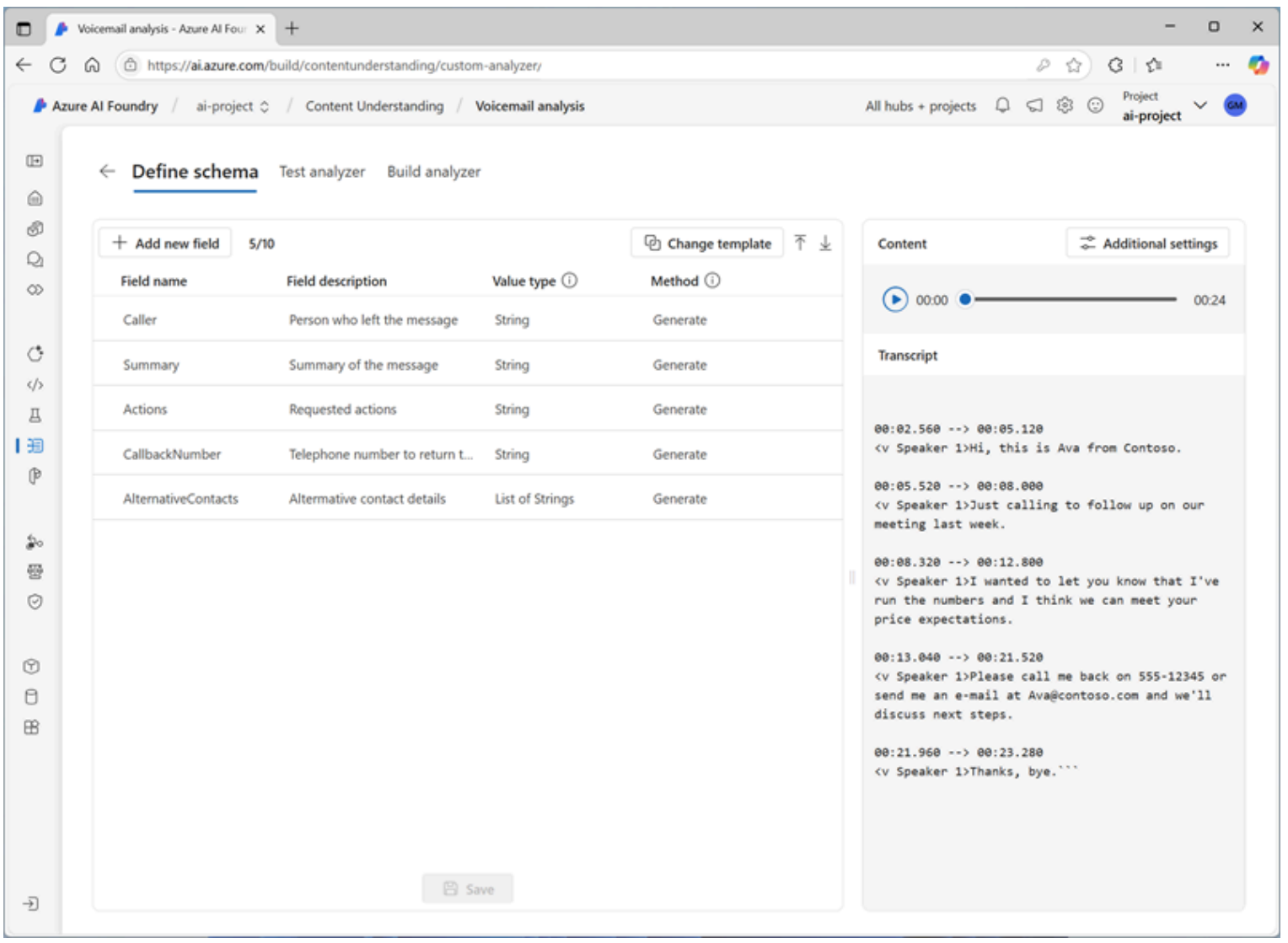
6. In the **Content** pane on the right, select **Get transcription preview** to see a transcription of the recorded message.

   The *Speech transcript analysis* template doesn't include any predefined fields. You must define fields to describe the information you want to extract.

7. Use **+ Add new field** button to add the following fields, selecting **Save changes** (✓) for each new field:

Field name	Field description	Value type	Method
`Caller`	`Person who left the message`	String	Generate
`Summary`	`Summary of the message`	String	Generate
`Actions`	`Requested actions`	String	Generate
`CallbackNumber`	`Telephone number to return the call`	String	Generate
`AlternativeContacts`	`Alternative contact details`	List of Strings	Generate

8. Verify that your schema looks like this. Then select **Save**.



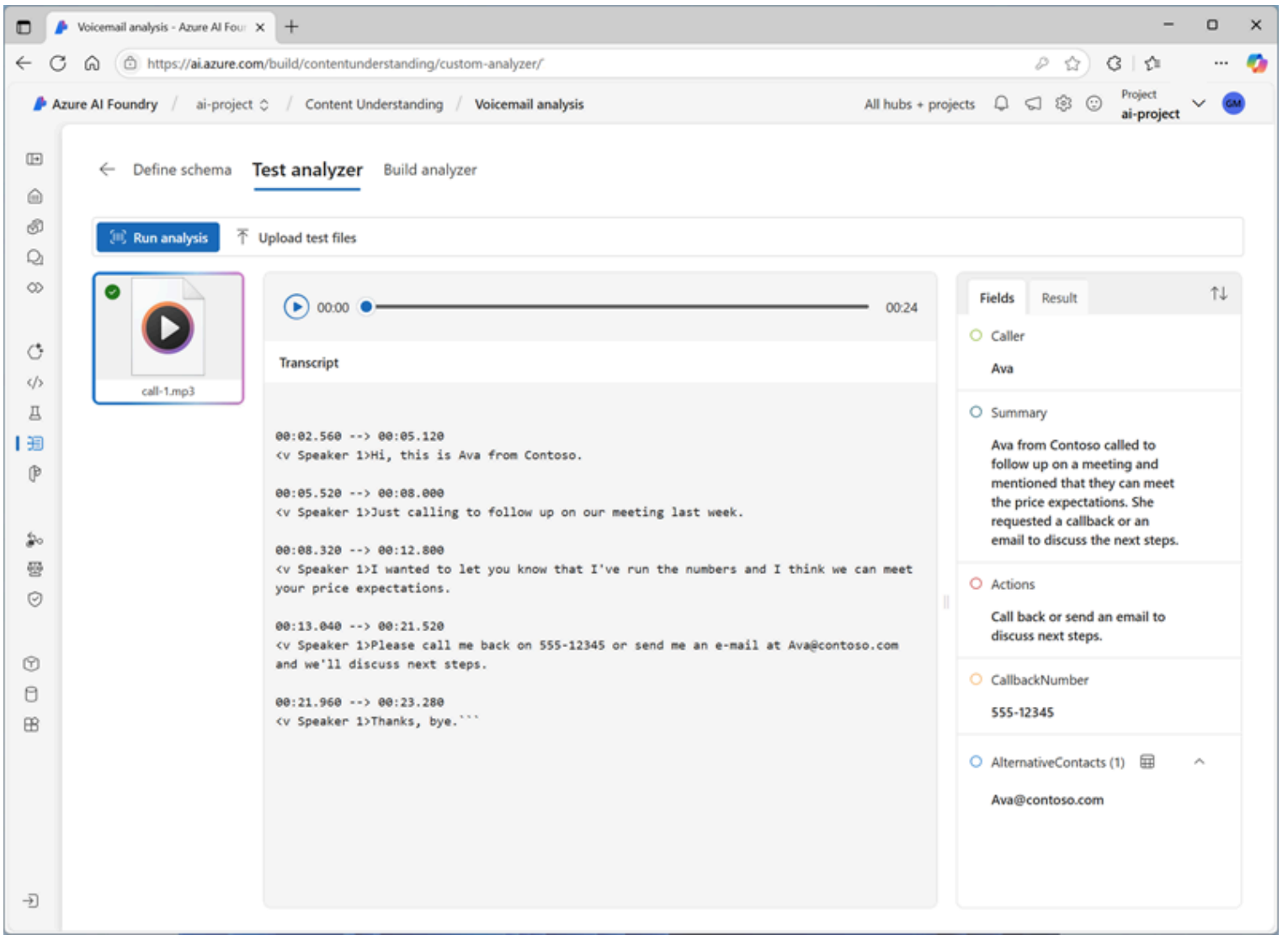
9. On the **Test Analyzer** page, if analysis does not begin automatically, select **Run analysis**. Then wait for analysis to complete.

   Audio analysis can take some time. While you're waiting, you can play the audio file below:

   0:00

   Note: This audio was generated using AI.

10. Review the analysis results, which should look similar to this:



11. View the details of the fields that were identified in the **Fields** pane, expanding the **AlternativeContacts** field to see the listed values.

Build and test an analyzer

Now that you have trained a model to extract fields from voice messages, you can build an analyzer to use with similar audio recordings.

1. Select the **Analyzer list** page, and then select **+ Build analyzer** and build a new analyzer with the following properties (typed exactly as shown here):
    - **Name**: `voicemail-analyzer`
    - **Description**: `Voicemail audio analyzer`
2. Wait for the new analyzer to be ready (use the **Refresh** button to check).
3. When the analyzer has been built, select the **voicemail-analyzer** link. The fields defined in the analyzer's schema will be displayed.
4. In the **voicemail-analyzer** page, select the **Test** tab.
5. Use the **+ Upload test files** button to upload **call-2.mp3** from the folder where you extracted the content files, and click on **Run analysis** to extract field data from the audio file.

   Audio analysis can take some time. While you're waiting, you can play the audio file below:

   0:00

   Note: This audio was generated using AI.
6. Review the **Fields** pane, and verify that the analyzer extracted the correct fields from the voice message.
7. Review the **Results** pane to see the JSON response that the analyzer would return to a client application.
8. On the **Code example** tab, view the sample code that you could use to develop a client application that uses the Content understanding REST interface to call your analyzer.
9. Close the **voicemail-analyzer** page.

Extract information from a video conference recording

You are going to build an Azure AI Content Understanding analyzer that can extract information from an video recording of a conference call.

Define a schema for video analysis

1. In the browser tab containing the home page for your Azure AI Foundry project; in the navigation pane on the left, select **Content Understanding**.
2. On the **Content Understanding** page, select the **Custom task** tab at the top.
3. On the Content Understanding custom task page, select **+ Create**, and create a task with the following settings:
    - **Task name**: `Conference call video analysis`
    - **Description**: `Extract data from a video conference recording`
    - **Single file content analysis**: *Selected*
    - **Advanced settings**:
        - **Azure AI services connection**: *The Azure AI Services resource in your Azure AI Foundry hub*
        - **Azure Blob Storage account**: *The default storage account in your Azure AI Foundry hub*
4. Wait for the task to be created.

> **Tip**: If an error accessing storage occurs, wait a minute and try again. Permissions for a new hub may take a few minutes to propagate.

5. On the **Define schema** page, upload the **meeting-1.mp4** file from the folder where you extracted content files. Then select the **Video analysis** template and select **Create**.
6. In the **Content** pane on the right, select **Get transcription preview** to see a transcription of the recorded message.

   The *Video analysis* template extracts data for the video. It doesn't include any predefined fields. You must define fields to describe the information you want to extract.

7. Use **+ Add new field** button to add the following fields, selecting **Save changes** (✓) for each new field:

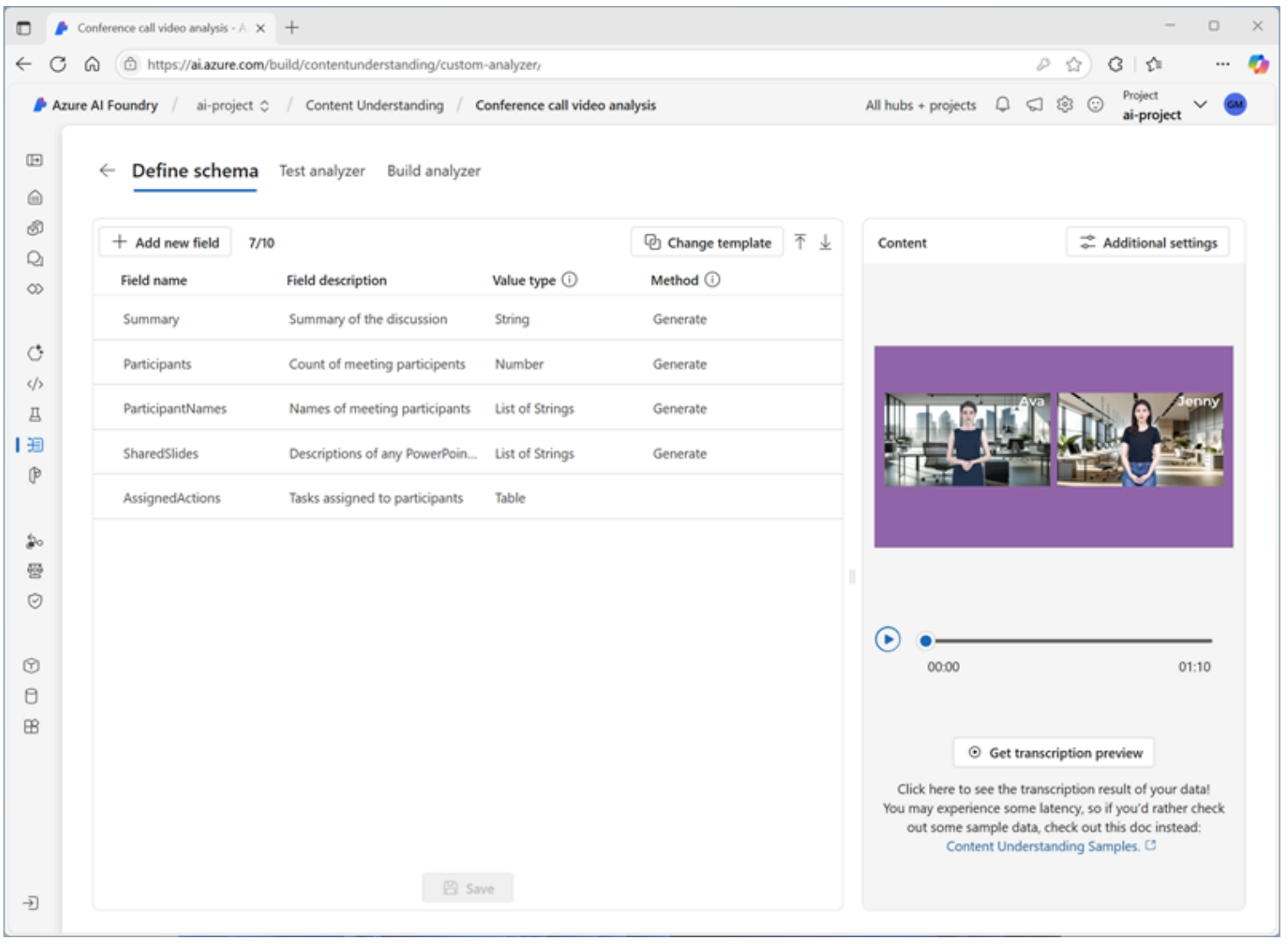
Field name	Field description	Value type	Method
`Summary`	`Summary of the discussion`	String	Generate
`Participants`	`Count of meeting participants`	Integer	Generate
`ParticipantNames`	`Names of meeting participants`	List of Strings	Generate
`SharedSlides`	`Descriptions of any PowerPoint slides presented`	List of Strings	Generate
`AssignedActions`	`Tasks assigned to participants`	Table	

8. When you enter the **AssignedActions** field, in the table of subfields that appears, create the following subfields:

Field name	Field description	Value type	Method
`Task`	`Description of the task`	String	Generate
`AssignedTo`	`Who the task is assigned to`	String	Generate

9. Select **Back** (the arrow icon near the **Add new subfield** button) or ✓ **OK** to return to the top level of your schema, and verify that it looks like this. Then select **Save**.

10. Verify that your schema looks like this. Then select **Save**.



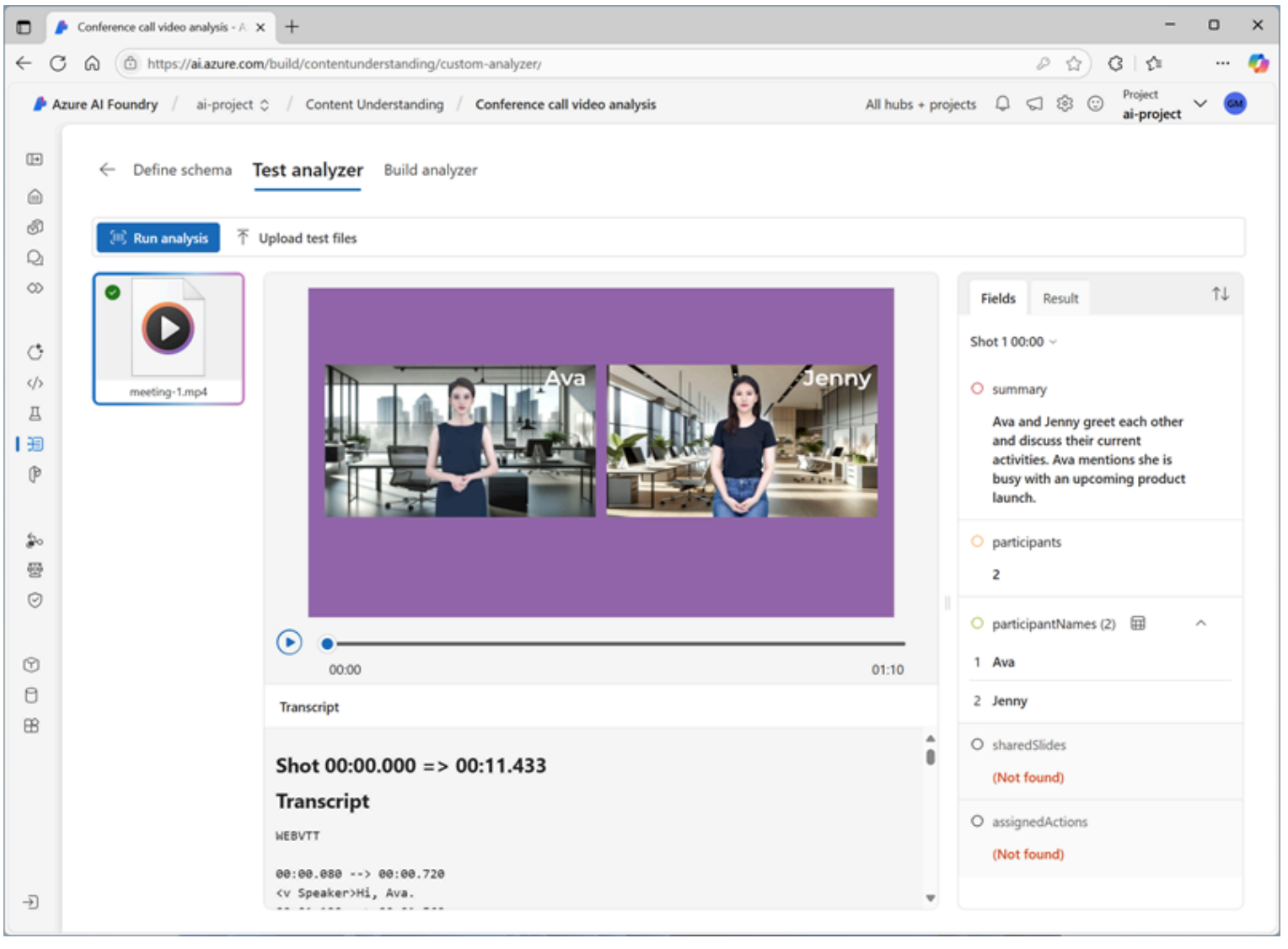
11. On the **Test Analyzer** page, if analysis does not begin automatically, select **Run analysis**. Then wait for analysis to complete.

    Video analysis can take some time. While you're waiting, you can view the video below:

    0:00

    Note: This video was generated using AI.

12. When analysis is complete, review the results, which should look similar to this:



13. In the **Fields** pane, view the extracted data for the video, including the fields you added. View the field values that were generated, expanding list and table fields as necessary.

Build and test an analyzer

Now that you have trained a model to extract fields from conference call recordings, you can build an analyzer to use with similar videos.

1. Select the **Analyzer list** page, and then select **+ Build analyzer** and build a new analyzer with the following properties (typed exactly as shown here):
    - **Name**: `conference-call-analyzer`
    - **Description**: `Conference call video analyzer`
2. Wait for the new analyzer to be ready (use the **Refresh** button to check).
3. When the analyzer has been built, select the **conference-call-analyzer** link. The fields defined in the analyzer's schema will be displayed.
4. In the **conference-call-analyzer** page, select the **Test** tab.
5. Use the **Upload test files** button to upload **meeting-2.mp4** from the folder where you extracted the content files, and run the analysis to extract field data from the audio file.

    Video analysis can take some time. While you're waiting, you can view the video below:

    0:00

    Note: This video was generated using AI.
6. Review the **Fields** pane, and view the fields that the analyzer extracted for the conference call video.
7. Review the **Results** pane to see the JSON response that the analyzer would return to a client application.
8. Close the **conference-call-analyzer** page.

Clean up

If you've finished working with the Content Understanding service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. In the Azure AI Foundry portal, navigate to your hub, in the overview page, select your project and delete it.
2. In the Azure portal, delete the resource group you created in this exercise.

Develop a Content Understanding client application

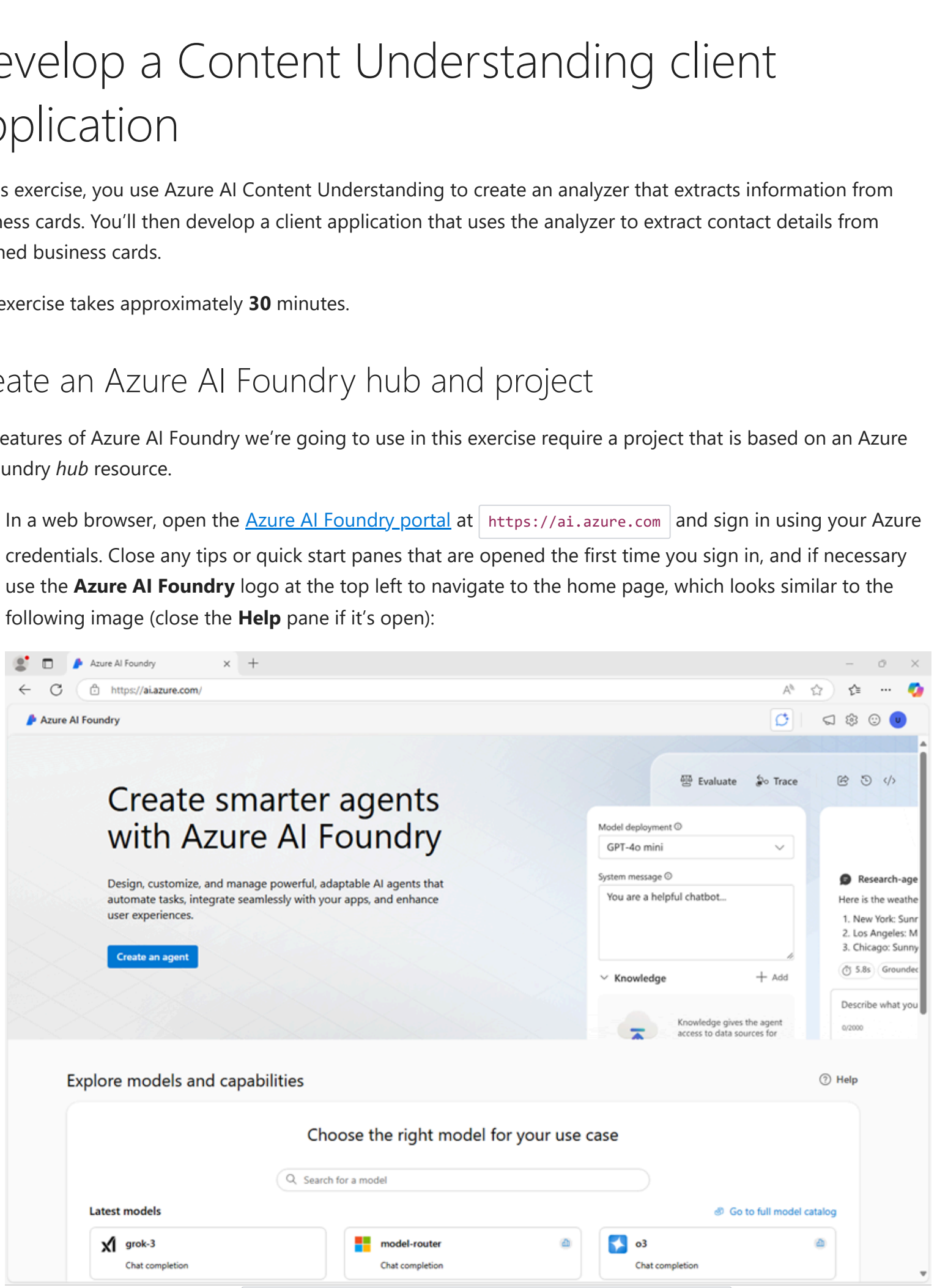
In this exercise, you use Azure AI Content Understanding to create an analyzer that extracts information from business cards. You'll then develop a client application that uses the analyzer to extract contact details from scanned business cards.

This exercise takes approximately **30** minutes.

Create an Azure AI Foundry hub and project

The features of Azure AI Foundry we're going to use in this exercise require a project that is based on an Azure AI Foundry *hub* resource.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):



2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Hub name**: A valid name for your hub
    - **Location**: Choose one of the following locations:*
        - Australia East
        - Sweden Central
        - West US

> *At the time of writing, Azure AI Content understanding is only available in these regions.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

4. Wait for your project to be created, and then navigate to your project overview page.

Use the REST API to create a Content Understanding analyzer

You're going to use the REST API to create an analyzer that can extract information from images of business cards.

1. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

   > **Tip**: Resize the pane so you can work mostly in the cloud shell but still see the keys and endpoint in the Azure portal page - you'll need to copy them into your code.

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-info -f
   git clone https://github.com/microsoftlearning/mslearn-ai-information-extraction mslearn-ai-info
   ```

   > **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

5. After the repo has been cloned, navigate to the folder containing the code files for your app:

   Code Copy

   ```
   cd mslearn-ai-info/Labfiles/content-app
   ls -a -l
   ```

   The folder contains two scanned business card images as well as the Python code files you need to build your app.

6. In the cloud shell command-line pane, enter the following command to install the libraries you'll use:

   Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt
```

7. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

The file is opened in a code editor.

8. In the code file, replace the **YOUR_ENDPOINT** and **YOUR_KEY** placeholders with the your Azure AI services endpoint and either of its keys (copied from the Azure portal), and ensure that **ANALYZER_NAME** is set to `business-card-analyzer`.

9. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

> **Tip**: You can maximize the cloud shell pane now.

10. In the cloud shell command line, enter the following command to view the **biz-card.json** JSON file that has been provided:

Code Copy

```
cat biz-card.json
```

Scroll the cloud shell pane to view the JSON in the file, which defines an analyzer schema for a business card.

11. When you've viewed the JSON file for the analyzer, enter the following command to edit the **create-analyzer.py** Python code file that has been provided:

Code Copy

```
code create-analyzer.py
```

The Python code file is opened in a code editor.

12. Review the code, which:
    - Loads the analyzer schema from **biz-card.json** file.
    - Retrieves the endpoint, key, and analyzer name from the environment configuration file.
    - Calls a function named **create_analyzer**, which is currently not implemented

13. In the **create_analyzer** function, find the comment **Create a Content Understanding analyzer** and add the following code (being careful to maintain the correct indentation):

Code Copy

```
# Create a Content Understanding analyzer
print (f"Creating {analyzer}")

# Set the API version
CU_VERSION = "2025-05-01-preview"

# initiate the analyzer creation operation
headers = {
     "Ocp-Apim-Subscription-Key": key,
     "Content-Type": "application/json"}

url = f"{endpoint}/contentunderstanding/analyzers/{analyzer}?api-version={CU_VERSION}"

# Delete the analyzer if it already exists
response = requests.delete(url, headers=headers)
print(response.status_code)
time.sleep(1)

# Now create it
response = requests.put(url, headers=headers, data=(schema))
print(response.status_code)

# Get the response and extract the callback URL
callback_url = response.headers["Operation-Location"]

# Check the status of the operation
time.sleep(1)
result_response = requests.get(callback_url, headers=headers)

# Keep polling until the operation is no longer running
status = result_response.json().get("status")
while status == "Running":
     time.sleep(1)
     result_response = requests.get(callback_url, headers=headers)
     status = result_response.json().get("status")

result = result_response.json().get("status")
print(result)
if result == "Succeeded":
     print(f"Analyzer '{analyzer}' created successfully.")
else:
     print("Analyzer creation failed.")
     print(result_response.json())
```

14. Review the code you added, which:
    - Creates appropriate headers for the REST requests
    - Submits an HTTP *DELETE* request to delete the analyzer if it already exists.
    - Submits an HTTP *PUT* request to initiate the creation of the analyzer.
    - Checks the response to retrieve the *Operation-Location* callback URL.
    - Repeatedly submits an HTTP *GET* request to the callback URL to check the operation status until it is no longer running.
    - Confirms success (or failure) of the operation to the user.

> **Note**: The code includes some deliberate time delays to avoid exceeding the request rate limit foe the service.

15. Use the **CTRL+S** command to save the code changes, but keep the code editor pane open in case you need to correct any errors in the code. Resize the panes so you can clearly see the command line pane.

16. In the cloud shell command line pane, enter the following command to run the Python code:

```
python create-analyzer.py
```

17. Review the output from the program, which should hopefully indicate that the analyzer has been created.

Use the REST API to analyze content

Now that you've created an analyzer, you can consume it from a client application through the Content Understanding REST API.

1. In the cloud shell command line, enter the following command to edit the **read-card.py** Python code file that has been provided:

```
code read-card.py
```

   The Python code file is opened in a code editor:

2. Review the code, which:
   - Identifies the image file to be analyzed, with a default of **biz-card-1.png**.
   - Retrieves the endpoint and key for your Azure AI Services resource from the project (using the Azure credentials from the current cloud shell session to authenticate).
   - Calls a function named **analyze_card**, which is currently not implemented

3. In the **analyze_card** function, find the comment **Use Content Understanding to analyze the image** and add the following code (being careful to maintain the correct indentation):

```
# Use Content Understanding to analyze the image
print (f"Analyzing {image_file}")

# Set the API version
CU_VERSION = "2025-05-01-preview"

# Read the image data
with open(image_file, "rb") as file:
     image_data = file.read()

## Use a POST request to submit the image data to the analyzer
print("Submitting request...")
headers = {
     "Ocp-Apim-Subscription-Key": key,
     "Content-Type": "application/octet-stream"}
url = f'{endpoint}/contentunderstanding/analyzers/{analyzer}:analyze?api-version=
{CU_VERSION}'
response = requests.post(url, headers=headers, data=image_data)

# Get the response and extract the ID assigned to the analysis operation
print(response.status_code)
response_json = response.json()
id_value = response_json.get("id")

# Use a GET request to check the status of the analysis operation
print ('Getting results...')
time.sleep(1)
result_url = f'{endpoint}/contentunderstanding/analyzerResults/{id_value}?api-version=
{CU_VERSION}'
result_response = requests.get(result_url, headers=headers)
print(result_response.status_code)

# Keep polling until the analysis is complete
status = result_response.json().get("status")
while status == "Running":
     time.sleep(1)
     result_response = requests.get(result_url, headers=headers)
     status = result_response.json().get("status")

# Process the analysis results
if status == "Succeeded":
     print("Analysis succeeded:\n")
     result_json = result_response.json()
     output_file = "results.json"
     with open(output_file, "w") as json_file:
         json.dump(result_json, json_file, indent=4)
         print(f"Response saved in {output_file}\n")

     # Iterate through the fields and extract the names and type-specific values
     contents = result_json["result"]["contents"]
     for content in contents:
         if "fields" in content:
             fields = content["fields"]
             for field_name, field_data in fields.items():
                 if field_data['type'] == "string":
                     print(f"{field_name}: {field_data['valueString']}")
                 elif field_data['type'] == "number":
                     print(f"{field_name}: {field_data['valueNumber']}")
```

```
        elif field_data['type'] == "integer":
            print(f"{field_name}: {field_data['valueInteger']}")
        elif field_data['type'] == "date":
            print(f"{field_name}: {field_data['valueDate']}")
        elif field_data['type'] == "time":
            print(f"{field_name}: {field_data['valueTime']}")
        elif field_data['type'] == "array":
            print(f"{field_name}: {field_data['valueArray']}")
```

4. Review the code you added, which:
    - Reads the contents of the image file
    - Sets the version of the Content Understanding REST API to be used
    - Submits an HTTP *POST* request to your Content Understanding endpoint, instructing the to analyze the image.
    - Checks the response from the operation to retrieve an ID for the analysis operation.
    - Repeatedly submits an HTTP *GET* request to your Content Understanding endpoint to check the operation status until it is no longer running.
    - If the operation has succeeded, saves the JSON response, and then parses the JSON and displays the values retrieved for each type-specific field.

> **Note**: In our simple business card schema, all of the fields are strings. The code here illustrates the need to check the type of each field so that you can extract values of different types from a more complex schema.

5. Use the **CTRL+S** command to save the code changes, but keep the code editor pane open in case you need to correct any errors in the code. Resize the panes so you can clearly see the command line pane.

6. In the cloud shell command line pane, enter the following command to run the Python code:

Code Copy

```
python read-card.py biz-card-1.png
```

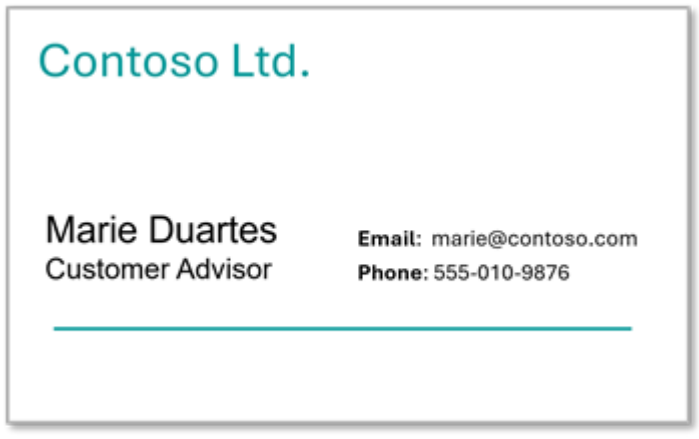
7. Review the output from the program, which should show the values for the fields in the following business card:

8. Use the following command to run the program with a different business card:

Code Copy

```
python read-card.py biz-card-2.png
```

9. Review the results, which should reflect the values in this business card:



10. In the cloud shell command line pane, use the following command to view the full JSON response that was returned:

```
cat results.json
```

Scroll to view the JSON.

Clean up

If you've finished working with the Content Understanding service, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. In the Azure portal, delete the resources you created in this exercise.

Analyze forms with prebuilt Azure AI Document Intelligence models

In this exercise, you'll set up an Azure AI Foundry project with all the necessary resources for document analysis. You'll use both the Azure AI Foundry portal and the Python SDK to submit forms to that resource for analysis.

While this exercise is based on Python, you can develop similar applications using multiple language-specific SDKs; including:

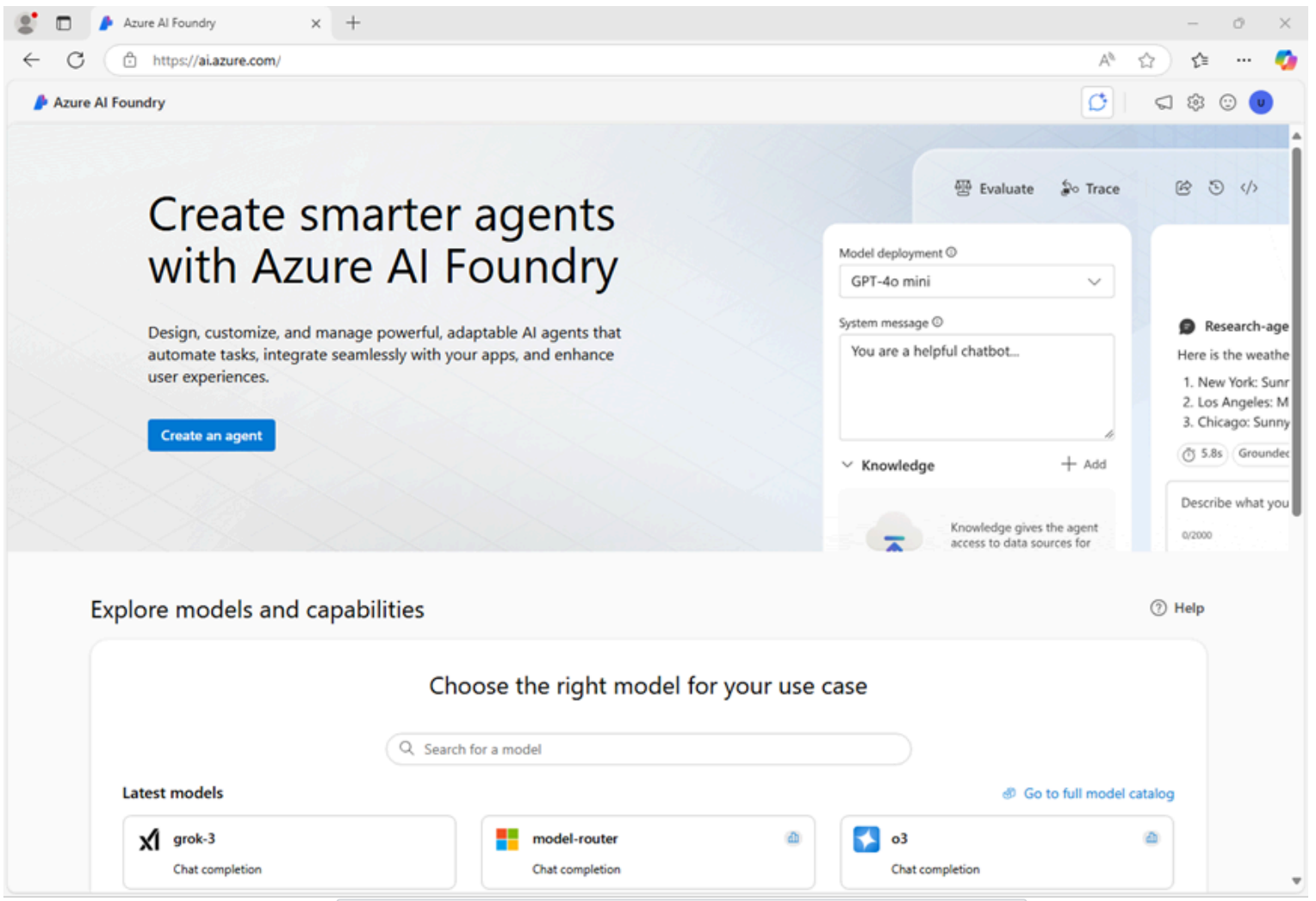
- Azure AI Document Intelligence client library for Python
- Azure AI Document Intelligence client library for Microsoft .NET
- Azure AI Document Intelligence client library for JavaScript

This exercise takes approximately **30** minutes.

Create an Azure AI Foundry project

Let's start by creating an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image (close the **Help** pane if it's open):

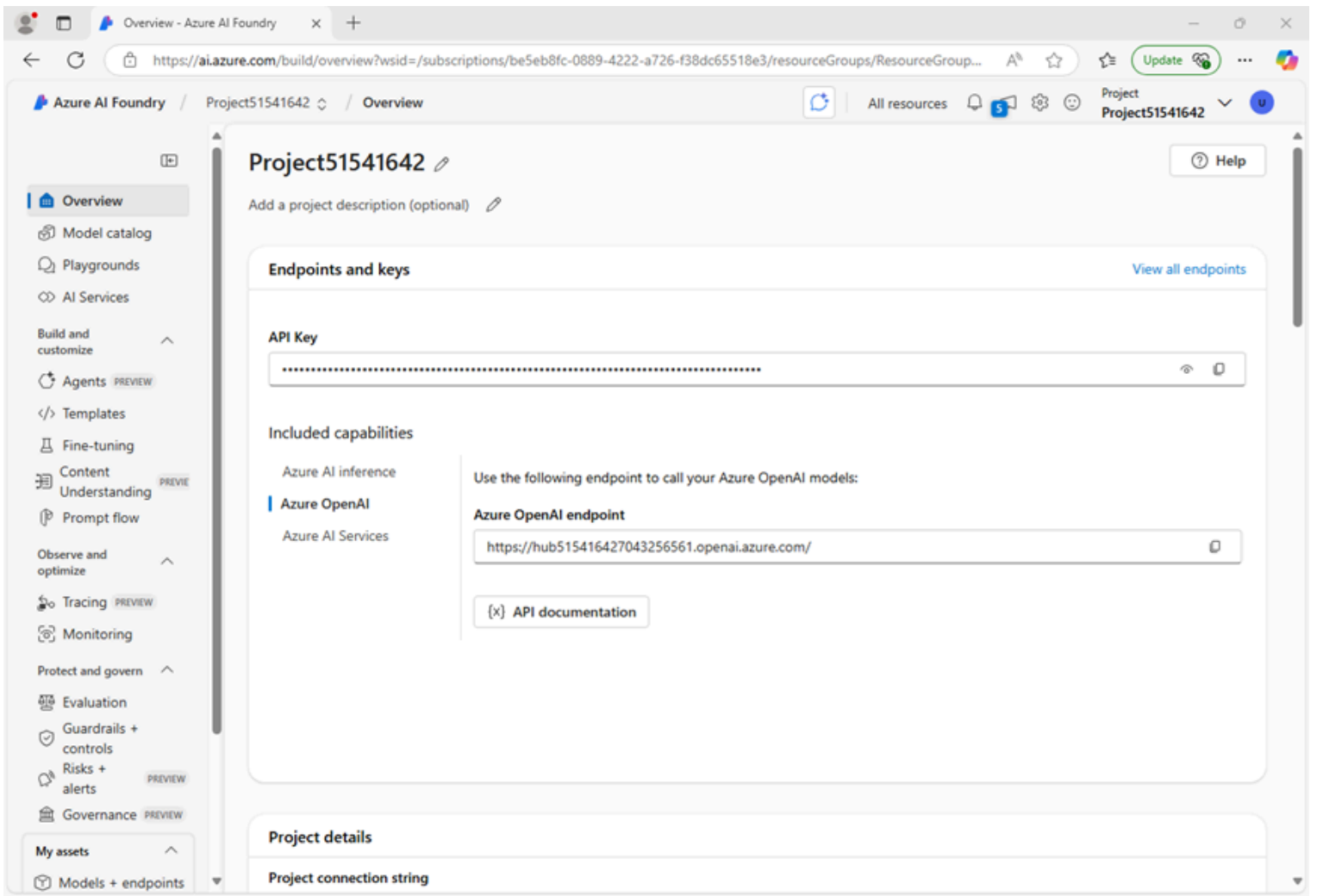


2. In the browser, navigate to `https://ai.azure.com/managementCenter/allResources` and select **Create new**. Then choose the option to create a new **AI hub resource**.
3. In the **Create a project** wizard, enter a valid name for your project, and select the option to create a new hub. Then use the **Rename hub** link to specify a valid name for your new hub, expand **Advanced options**, and specify the following settings for your project:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Region**: *Any available region*

> **Note**: If you're working in an Azure subscription in which policies are used to restrict allowable resource names, you may need to use the link at the bottom of the **Create a new project** dialog box to create the hub using the Azure portal.

> **Tip**: If the **Create** button is still disabled, be sure to rename your hub to a unique alphanumeric value.

4. Wait for your project to be created.
5. When your project is created, close any tips that are displayed and review the project page in Azure AI Foundry portal, which should look similar to the following image:

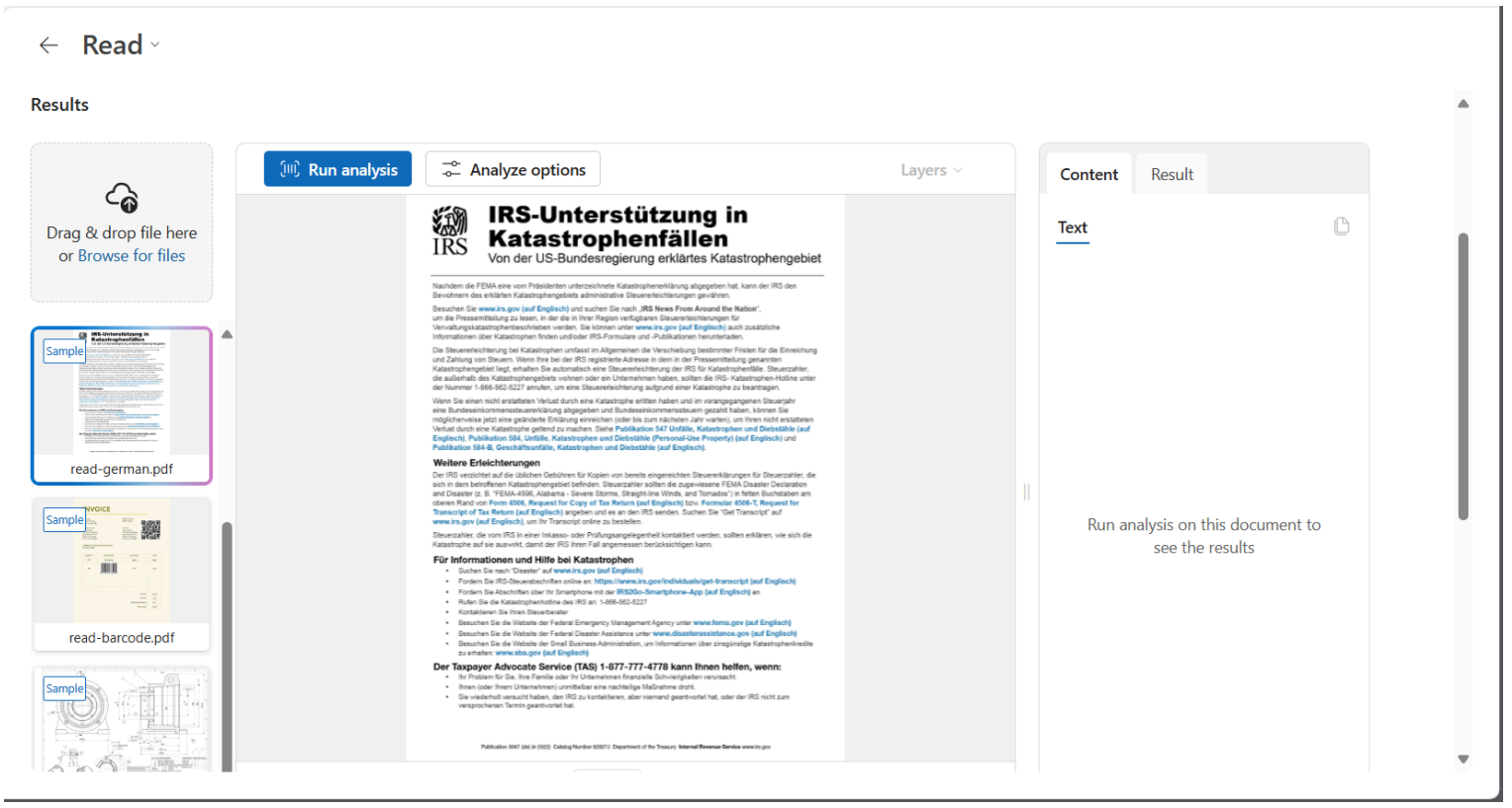


Use the Read model

Let's start by using the **Azure AI Foundry** portal and the Read model to analyze a document with multiple languages:

1. In the navigation panel on the left, select **AI Services**.
2. In the **Azure AI Services** page, select the **Vision + Document** tile.
3. In the **Vision + Document** page, verify that the **Document** tab is selected, then select the **OCR/Read** tile.

   In the **Read** page, the Azure AI Services resource created with your project should already be connected.
4. In the list of documents on the left, select **read-german.pdf**.



5. At the top toolbar, select **Analyze options**, then enable the **Language** check-box (under **Optional detection**) in the **Analyze options** pane and select **Save**.
6. At the top-left, select **Run Analysis**.
7. When the analysis is complete, the text extracted from the image is shown on the right in the **Content** tab. Review this text and compare it to the text in the original image for accuracy.
8. Select the **Result** tab. This tab displays the extracted JSON code.

Prepare to develop an app in Cloud Shell

Now let's explore the app that uses the Azure Document Intelligence service SDK. You'll develop your app using Cloud Shell. The code files for your app have been provided in a GitHub repo.

This is the invoice that your code will analyze.

CONTOSO LTD.

INVOICE

Contoso Headquarters
123 456th St
New York, NY, 10001

INVOICE: INV-100
DATE: 11/15/2019
DUE DATE: 12/15/2019
CUSTOMER NAME: MICROSOFT CORPORATION
CUSTOMER ID: CID-12345

Microsoft Corp
123 Other St,
Redmond WA, 98052

BILL TO:
Microsoft Finance
123 Bill St,
Redmond WA, 98052

SHIP TO:
Microsoft Delivery
123 Ship St,
Redmond WA, 98052

SERVICE ADDRESS:
Microsoft Services
123 Service St,
Redmond WA, 98052

SALESPERSON	P.O. NUMBER	REQUISITIONER	SHIPPED VIA	F.O.B. POINT	TERMS
	PO-3333				

QUANTITY	DESCRIPTION	UNIT PRICE	TOTAL
1	Test for 23 fields	1	$100.00
		SUBTOTAL	$100.00
		SALES TAX	$10.00
		TOTAL	$110.00

1. In the Azure AI Foundry portal, view the **Overview** page for your project.

2. In the **Endpoints and keys** area, select the **Azure AI Services** tab, and note the **API Key** and **Azure AI Services endpoint**. You'll use these credentials to connect to your Azure AI Services in a client application.
3. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.
4. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

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

5. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

6. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-info -f
   git clone https://github.com/microsoftlearning/mslearn-ai-information-extraction mslearn-ai-info
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

   Now follow the steps for your chosen programming language.

7. After the repo has been cloned, navigate to the folder containing the code files:

   Code Copy

   ```
   cd mslearn-ai-info/Labfiles/prebuilt-doc-intelligence/Python
   ```

8. In the cloud shell command line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-formrecognizer==3.3.3
   ```

9. Enter the following command to edit the configuration file that has been provided:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.
10. In the code file, replace the **YOUR_ENDPOINT** and **YOUR_KEY** placeholders with your Azure AI services endpoint and its API key (copied from the Azure AI Foundry portal).
11. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to use the Azure Document Intelligence service

Now you're ready to use the SDK to evaluate the pdf file.

1. Enter the following command to edit the app file that has been provided:

```
code document-analysis.py
```

   The file is opened in a code editor.

2. In the code file, find the comment **Import the required libraries** and add the following code:

```
# Add references
from azure.core.credentials import AzureKeyCredential
from azure.ai.formrecognizer import DocumentAnalysisClient
```

3. Find the comment **Create the client** and add the following code (being careful to maintain the correct indentation level):

```
# Create the client
document_analysis_client = DocumentAnalysisClient(
     endpoint=endpoint, credential=AzureKeyCredential(key)
)
```

4. Find the comment **Analyze the invoice** and add the following code:

```
# Analyse the invoice
poller = document_analysis_client.begin_analyze_document_from_url(
     fileModelId, fileUri, locale=fileLocale
)
```

5. Find the comment **Display invoice information to the user**and add the following code:

```
# Display invoice information to the user
receipts = poller.result()

for idx, receipt in enumerate(receipts.documents):

     vendor_name = receipt.fields.get("VendorName")
     if vendor_name:
         print(f"\nVendor Name: {vendor_name.value}, with confidence
{vendor_name.confidence}.")

     customer_name = receipt.fields.get("CustomerName")
     if customer_name:
         print(f"Customer Name: '{customer_name.value}, with confidence
{customer_name.confidence}.")


     invoice_total = receipt.fields.get("InvoiceTotal")
     if invoice_total:
         print(f"Invoice Total: '{invoice_total.value.symbol}{invoice_total.value.amount},
 with confidence {invoice_total.confidence}.")
```

6. In the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes. Keep the code editor open in case you need to fix any errors in the code, but resize the panes so you can see the command line pane clearly.

7. In the command line pane, enter the following command to run the application.

Code Copy

```
python document-analysis.py
```

The program displays the vendor name, customer name, and invoice total with confidence levels. Compare the values it reports with the sample invoice you opened at the start of this section.

Clean up

If you’re done with your Azure resource, remember to delete the resource in the Azure portal (`https://portal.azure.com`) to avoid further charges.

Analyze forms with custom Azure AI Document Intelligence models

Suppose a company currently requires employees to manually purchase order sheets and enter the data into a database. They would like you to utilize AI services to improve the data entry process. You decide to build a machine learning model that will read the form and produce structured data that can be used to automatically update a database.

Azure AI Document Intelligence is an Azure AI service that enables users to build automated data processing software. This software can extract text, key/value pairs, and tables from form documents using optical character recognition (OCR). Azure AI Document Intelligence has pre-built models for recognizing invoices, receipts, and business cards. The service also provides the capability to train custom models. In this exercise, we will focus on building custom models.

While this exercise is based on Python, you can develop similar applications using multiple language-specific SDKs; including:

- Azure AI Document Intelligence client library for Python
- Azure AI Document Intelligence client library for Microsoft .NET
- Azure AI Document Intelligence client library for JavaScript

This exercise takes approximately **30** minutes.

Create a Azure AI Document Intelligence resource

To use the Azure AI Document Intelligence service, you need a Azure AI Document Intelligence or Azure AI Services resource in your Azure subscription. You'll use the Azure portal to create a resource.

1. In a browser tab, open the Azure portal at `https://portal.azure.com`, signing in with the Microsoft account associated with your Azure subscription.
2. On the Azure portal home page, navigate to the top search box and type **Document Intelligence** and then press **Enter**.
3. On the **Document Intelligence** page, select **Create**.
4. On the **Create Document Intelligence** page, create a new resource with the following settings:
    - **Subscription**: Your Azure subscription.
    - **Resource group**: Create or select a resource group
    - **Region**: Any available region
    - **Name**: A valid name for your Document Intelligence resource
    - **Pricing tier**: Free F0 (*if you don't have a Free tier available, select* Standard S0).
5. When the deployment is complete, select **Go to resource** to view the resource's **Overview** page.

Prepare to develop an app in Cloud Shell

You'll develop your text translation app using Cloud Shell. The code files for your app have been provided in a GitHub repo.

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

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

2. Size the cloud shell pane so you can see both the command line console and the Azure portal. You'll need to use the the split bar to switch as you switch between the two panes.

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

4. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-info -f
   git clone https://github.com/microsoftlearning/mslearn-ai-information-extraction mslearn-ai-info
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

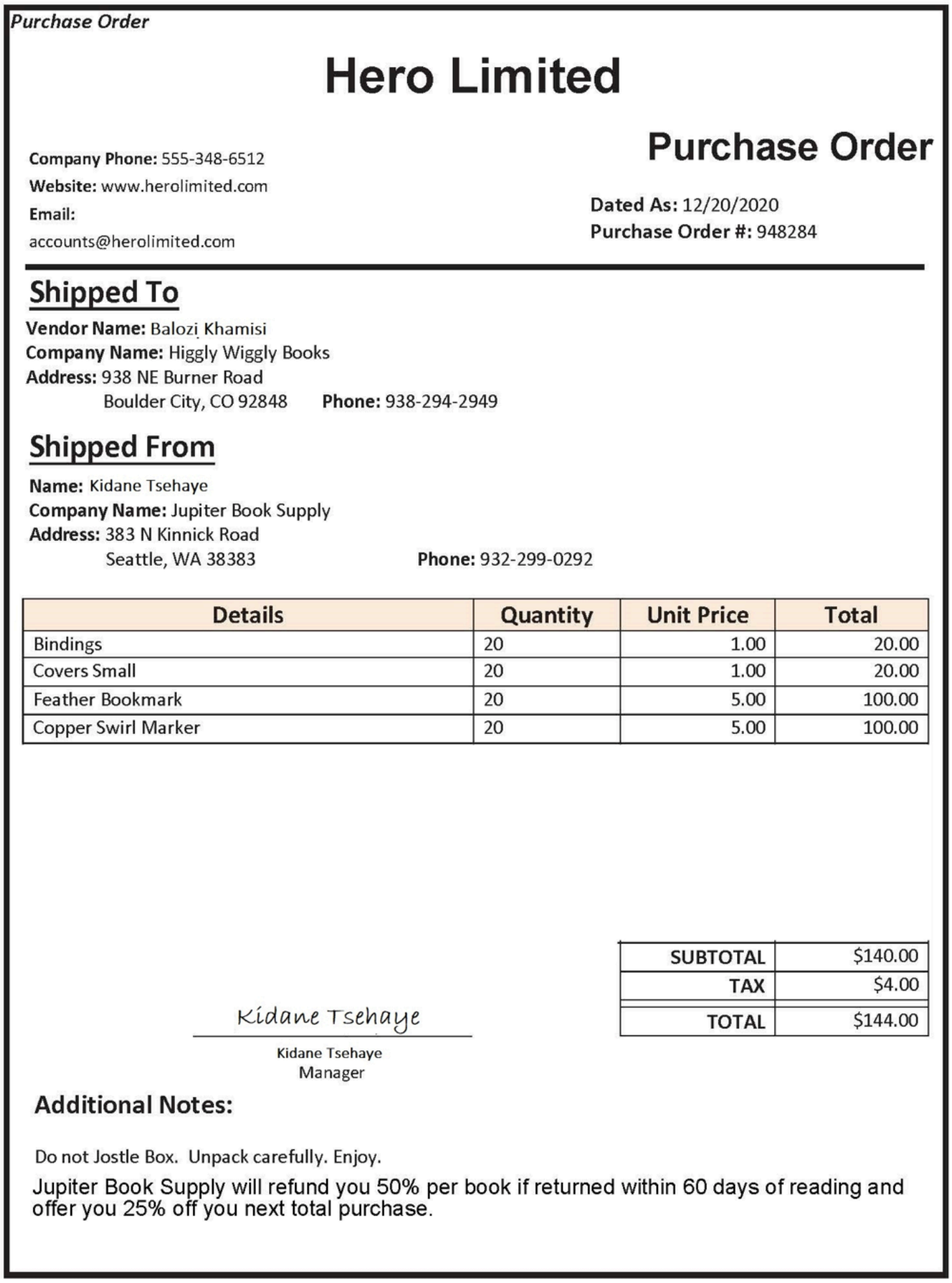
5. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-info/Labfiles/custom-doc-intelligence
   ```

Gather documents for training

You'll use the sample forms such as this one to train a test a model:



Purchase Order

Hero Limited

Company Phone: 555-348-6512
Website: www.herolimited.com
Email:
accounts@herolimited.com

Purchase Order

Dated As: 12/20/2020
Purchase Order #: 948284

Shipped To

Vendor Name: Balozi Khamisi
Company Name: Higgly Wiggly Books
Address: 938 NE Burner Road
Boulder City, CO 92848 **Phone:** 938-294-2949

Shipped From

Name: Kidane Tsehaye
Company Name: Jupiter Book Supply
Address: 383 N Kinnick Road
Seattle, WA 38383 **Phone:** 932-299-0292

Details	Quantity	Unit Price	Total
Bindings	20	1.00	20.00
Covers Small	20	1.00	20.00
Feather Bookmark	20	5.00	100.00
Copper Swirl Marker	20	5.00	100.00

SUBTOTAL	$140.00
TAX	$4.00
TOTAL	$144.00

Kidane Tsehaye

Kidane Tsehaye
Manager

Additional Notes:

Do not Jostle Box. Unpack carefully. Enjoy.

Jupiter Book Supply will refund you 50% per book if returned within 60 days of reading and offer you 25% off you next total purchase.

1. In the command line, run `ls ./sample-forms` to list the content in the **sample-forms** folder. Notice there are files ending in **.json** and **.jpg** in the folder.

   You will use the **.jpg** files to train your model.

   The **.json** files have been generated for you and contain label information. The files will be uploaded into your blob storage container alongside the forms.
2. In the **Azure portal** and navigate to your resource's **Overview** page if you're not already there. Under the *Essentials* section, note the **Resource group**, **Subscription ID**, and **Location**. You will need these values in subsequent steps.

3. Run the command `code setup.sh` to open **setup.sh** in a code editor. You will use this script to run the Azure command line interface (CLI) commands required to create the other Azure resources you need.
4. In the **setup.sh** script, review the commands. The program will:

   - Create a storage account in your Azure resource group
   - Upload files from your local *sampleforms* folder to a container called *sampleforms* in the storage account
   - Print a Shared Access Signature URI

5. Modify the **subscription_id**, **resource_group**, and **location** variable declarations with the appropriate values for the subscription, resource group, and location name where you deployed the Document Intelligence resource.

   > **Important**: For your **location** string, be sure to use the code version of your location. For example, if your location is "East US", the string in your script should be `eastus`. You can see that version is the **JSON View** button on the right side of the **Essentials** tab of your resource group in Azure portal.

   If the **expiry_date** variable is in the past, update it to a future date. This variable is used when generating the Shared Access Signature (SAS) URI. In practice, you will want to set an appropriate expiry date for your SAS. You can learn more about SAS here.

6. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

7. Enter the following commands to make the script executable and to run it:

   Code Copy

   ```
   chmod +x ./setup.sh
   ./setup.sh
   ```

8. When the script completes, review the displayed output.
9. In the Azure portal, refresh your resource group and verify that it contains the Azure Storage account just created. Open the storage account and in the pane on the left, select **Storage browser**. Then in Storage Browser, expand **Blob containers** and select the **sampleforms** container to verify that the files have been uploaded from your local **custom-doc-intelligence/sample-forms** folder.

Train the model using Document Intelligence Studio

Now you will train the model using the files uploaded to the storage account.

1. Open a new browser tab, and navigate to the Document Intelligence Studio at `https://documentintelligence.ai.azure.com/studio`.
2. Scroll down to the **Custom models** section and select the **Custom extraction model** tile.
3. If prompted, sign in with your Azure credentials.
4. If you are asked which Azure AI Document Intelligence resource to use, select the subscription and resource name you used when you created the Azure AI Document Intelligence resource.
5. Under **My Projects**, Create a new project with the following configuration:
   - **Enter project details**:
     - **Project name**: A valid name for your project
   - **Configure service resource**:
     - **Subscription**: Your Azure subscription
     - **Resource group**: The resource group where you deployed your Document Intelligence resource
     - **Document intelligence resource** Your Document Intelligence resource (select the *Set as default* option and use the default API version)
   - **Connect training data source**:
     - **Subscription**: Your Azure subscription
     - **Resource group**: The resource group where you deployed your Document Intelligence resource
     - **Storage account**: The storage account that was created by the setup script (select the *Set as default* option, select the `sampleforms` blob container, and leave the folder path blank)
6. When your project is created, on the top right of the page, select **Train** to train your model. Use the following configurations:
   - **Model ID**: A valid name for your model (*you'll need the model ID name in the next step*)

- **Build Mode**: Template.

7. Select **Go to Models**.
8. Training can take some time. Wait until the status is **succeeded**.

Test your custom Document Intelligence model

1. Return to the browser tab containing the Azure Portal and cloud shell. In the command line, run the following command to change to the folder containing the application code files:

```
cd Python
```

2. Install the Document Intelligence package by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-formrecognizer==3.3.3
```

3. Enter the following command to edit the configuration file that has been provided:

```
code .env
```

4. In the pane containing the Azure portal, on the **Overview** page for your Document Intelligence resource, select **Click here to manage keys** to see the endpoint and keys for your resource. Then edit the configuration file with the following values:
   - Your Document Intelligence endpoint
   - Your Document Intelligence key
   - The Model ID you specified when training your model
5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.
6. Open the code file for your client application (`code Program.cs` for C#, `code test-model.py` for Python) and review the code it contains, particularly that the image in the URL refers to the file in this GitHub repo on the web. Close the file without making any changes.
7. In the command line, and enter the following command to run the program:

```
python test-model.py
```

8. View the output and observe how the output for the model provides field names like `Merchant` and `CompanyPhoneNumber` .

Clean up

If you're done with your Azure resource, remember to delete the resource in the Azure portal to avoid further charges.

More information

For more information about the Document Intelligence service, see the Document Intelligence documentation.

Create an knowledge mining solution

In this exercise, you use AI Search to index a set of documents maintained by Margie's Travel, a fictional travel agency. The indexing process involves using AI skills to extract key information to make them searchable, and generating a knowledge store containing data assets for further analysis.

While this exercise is based on Python, you can develop similar applications using multiple language-specific SDKs; including:

- Azure AI Search client library for Python
- Azure AI Search client library for Microsoft .NET
- Azure AI Search client library for JavaScript

This exercise takes approximately **40** minutes.

Create Azure resources

The solution you will create for Margie's Travel requires multiple resources in your Azure subscription. In this exercise, you'll create them directly in the Azure portal. You could also create them by using a script, or an ARM or BICEP template; or you could create an Azure AI Foundry project that includes an Azure AI Search resource.

> **Important**: Your Azure resources should be created in the same location!

Create an Azure AI Search resource

1. In a web browser, open the Azure portal at `https://portal.azure.com`, and sign in using your Azure credentials.
2. Select the **+ Create a resource** button, search for `Azure AI Search`, and create an **Azure AI Search** resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*
    - **Service name**: *A valid name for your search resource*
    - **Location**: *Any available location*
    - **Pricing tier**: Free
3. Wait for deployment to complete, and then go to the deployed resource.
4. Review the **Overview** page on the blade for your Azure AI Search resource in the Azure portal. Here, you can use a visual interface to create, test, manage, and monitor the various components of a search solution; including data sources, indexes, indexers, and skillsets.

Create a storage account

1. Return to the home page, and then create a **Storage account** resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *The same resource group as your Azure AI Search and Azure AI Services resources*
    - **Storage account name**: *A valid name for your storage resource*
    - **Region**: *The same region as your Azure AI Search resource*
    - **Primary service**: Azure Blob Storage or Azure Data Lake Storage Gen 2
    - **Performance**: Standard
    - **Redundancy**: Locally-redundant storage (LRS)
2. Wait for deployment to complete, and then go to the deployed resource.

> **Tip**: Keep the storage account portal page open - you will use it in the next procedure.

Upload documents to Azure Storage

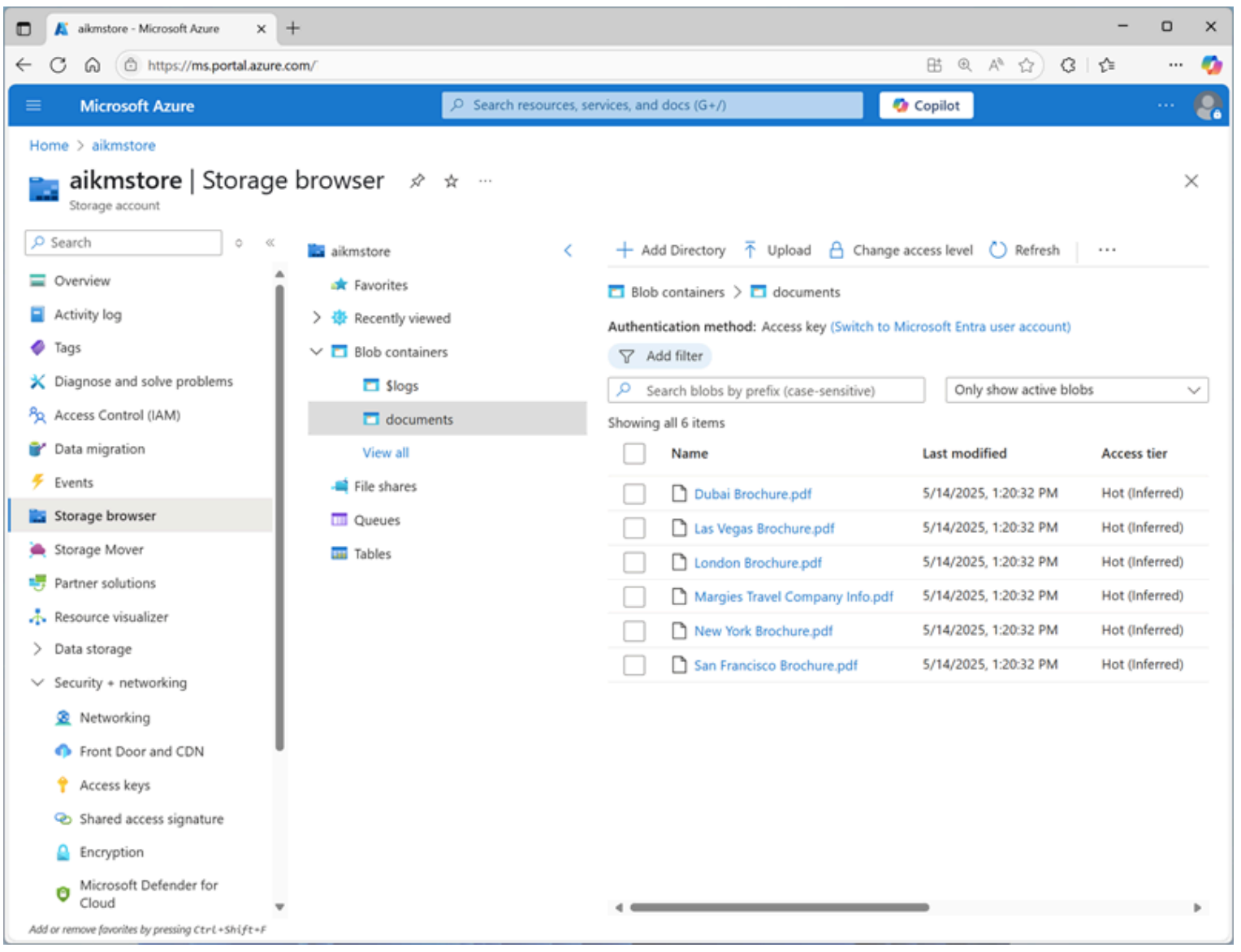
Your knowledge mining solution will extract information from travel brochure documents in an Azure Storage blob container.

1. In a new browser tab, download documents.zip from
   ```
   https://github.com/microsoftlearning/mslearn-ai-information-extraction/raw/main/Labfiles/knowledge/documents.zip
   ```
   and save it in a local folder.
2. Extract the downloaded *documents.zip* file and view the travel brochure files it contains. You'll extract and index information from these files.
3. In the browser tab containing the Azure portal page for your storage account, in the navigation pane on the left, select **Storage browser**.
4. In the storage browser, select **Blob containers**.

   Currently, your storage account should contain only the default **$logs** container.
5. In the toolbar, select **+ Container** and create a new container with the following settings:
   - **Name**: `documents`
   - **Anonymous access level**: Private (no anonymous access)*

> **Note**: *Unless you enabled the option to allow anonymous container access when creating your storage account, you won't be able to select any other setting!

6. Select the **documents** container to open it, and then use the **Upload** toolbar button to upload the .pdf files you extracted from **documents.zip** previously into the root of the container, as shown here:



Create and run an indexer

Now that you have the documents in place, you can create an indexer to extract information from them.

1. In the Azure portal, browse to your Azure AI Search resource. Then, on its **Overview** page, select **Import data**.
2. On the **Connect to your data** page, in the **Data Source** list, select **Azure Blob Storage**. Then complete the data store details with the following values:

- **Data Source**: Azure Blob Storage
- **Data source name**: `margies-documents`
- **Data to extract**: Content and metadata
- **Parsing mode**: Default
- **Subscription**: *Your Azure subscription*
- **Connection string**:
    - Select **Choose an existing connection**
    - Select your storage account
    - Select the **documents** container
- **Managed identity authentication**: None
- **Container name**: documents
- **Blob folder**: *Leave this blank*
- **Description**: `Travel brochures`

3. Proceed to the next step (**Add cognitive skills**), which has three expandable sections to complete.

4. In the **Attach Azure AI Services** section, select **Free (limited enrichments)***.

> **Note**: *The free Azure AI Services resource for Azure AI Search can be used to index a maximum of 20 documents. In a real solution, you should create an Azure AI Services resource in your subscription to enable AI enrichment for a larger number of documents.

5. In the **Add enrichments** section:

    - Change the **Skillset name** to `margies-skillset`.
    - Select the option **Enable OCR and merge all text into merged_content field**.
    - Ensure that the **Source data field** is set to **merged_content**.
    - Leave the **Enrichment granularity level** as **Source field**, which is set the entire contents of the document being indexed; but note that you can change this to extract information at more granular levels, like pages or sentences.
    - Select the following enriched fields:

Cognitive Skill	Parameter	Field name
Text Cognitive Skills		
Extract people names		people
Extract location names		locations
Extract key phrases		keyphrases
Image Cognitive Skills		
Generate tags from images		imageTags
Generate captions from images		imageCaption

    Double-check your selections (it can be difficult to change them later).

6. In the **Save enrichments to a knowledge store** section:

    - Select only the following checkboxes (an error will be displayed, you'll resolve that shortly):
        - **Azure file projections**:
            - Image projections
        - **Azure table projections**:
            - Documents
                - Key phrases
        - **Azure blob projections**:

        - Document
    - Under **Storage account connection string** (beneath the error messages):
        - Select **Choose an existing connection**
        - Select your storage account
        - Select the **documents** container (*this is only required to select the storage account in the browse interface - you'll specify a different container name for the extracted knowledge assets!*)
    - Change the **Container name** to `knowledge-store`.
7. Proceed to the next step (**Customize target index**), where you'll specify the fields for your index.
8. Change the **Index name** to `margies-index`.
9. Ensure that the **Key** is set to **metadata_storage_path**, leave the **Suggester name** blank, and ensure **Search mode** is **analyzingInfixMatching**.
10. Make the following changes to the index fields, leaving all other fields with their default settings (**IMPORTANT**: you may need to scroll to the right to see the entire table):

Field name	Retrievable	Filterable	Sortable	Facetable	Searchable
metadata_storage_size	✓	✓	✓		
metadata_storage_last_modified	✓	✓	✓		
metadata_storage_name	✓	✓	✓		✓
locations	✓	✓			✓
people	✓	✓			✓
keyphrases	✓	✓			✓

    Double-check your selections, paying particular attention to ensure that the correct **Retrievable**, **Filterable**, **Sortable**, **Facetable**, and **Searchable** options are selected correctly for each field (it can be difficult to change them later).
11. Proceed to the next step (**Create an indexer**), where you'll create and schedule the indexer.
12. Change the **Indexer name** to `margies-indexer`.
13. Leave the **Schedule** set to **Once**.
14. Select **Submit** to create the data source, skillset, index, and indexer. The indexer is run automatically and runs the indexing pipeline, which:
    - Extracts the document metadata fields and content from the data source
    - Runs the skillset of cognitive skills to generate additional enriched fields
    - Maps the extracted fields to the index.
    - Saves the extracted data assets to the knowledge store.
15. In the navigation pane on the left, under **Search management** view the **Indexers** page, which should show the newly created **margies-indexer**. Wait a few minutes, and click **↻ Refresh** until the **Status** indicates **Success**.

Search the index

Now that you have an index, you can search it.

1. Return to the **Overview** page for your Azure AI Search resource, and on the toolbar, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter `*` (a single asterisk), and then select **Search**.

    This query retrieves all documents in the index in JSON format. Examine the results and note the fields for each document, which contain document content, metadata, and enriched data extracted by the cognitive skills you selected.
3. In the **View** menu, select **JSON view** and note that the JSON request for the search is shown, like this:

Code Copy

```
{
  "search": "*",
  "count": true
}
```

4. The results include a **@odata.count** field at the top of the results that indicates the number of documents returned by the search.

5. Modify the JSON request to include the **select** parameter as shown here:

Code Copy

```
{
  "search": "*",
  "count": true,
  "select": "metadata_storage_name,locations"
}
```

This time the results include only the file name and any locations mentioned in the document content. The file name is in the **metadata_storage_name** field, which was extracted from the source document. The **locations** field was generated by an AI skill.

6. Now try the following query string:

Code Copy

```
{
  "search": "New York",
  "count": true,
  "select": "metadata_storage_name,keyphrases"
}
```

This search finds documents that mention “New York” in any of the searchable fields, and returns the file name and key phrases in the document.

7. Let’s try one more query:

Code Copy

```
{
    "search": "New York",
    "count": true,
    "select": "metadata_storage_name,keyphrases",
    "filter": "metadata_storage_size lt 380000"
}
```

This query returns the filename and key phrases for any documents mentioning “New York” that are smaller than 380,000 bytes in size.

Create a search client application

Now that you have a useful index, you can use it from a client application. You can do this by consuming the REST interface, submitting requests and receiving responses in JSON format over HTTP; or you can use the software development kit (SDK) for your preferred programming language. In this exercise, we’ll use the SDK.

> **Note**: You can choose to use the SDK for either **C#** or **Python**. In the steps below, perform the actions appropriate for your preferred language.

Get the endpoint and keys for your search resource

1. In the Azure portal, close the search explorer page and return to the **Overview** page for your Azure AI Search resource.

   Note the **Url** value, which should be similar to **https://*your_resource_name*.search.windows.net**. This is the endpoint for your search resource.

2. In the navigation pane on the left, expand **Settings** and view the **Keys** page.

   Note that there are two **admin** keys, and a single **query** key. An *admin* key is used to create and manage search resources; a *query* key is used by client applications that only need to perform search queries.

   *You will need the endpoint and **query** key for your client application.*

Prepare to use the Azure AI Search SDK

1. Use the **[>_]** button to the right of the search bar at the top of the Azure portal to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in. Initially, you'll need to see both the cloud shell and the Azure portal (so you can find and copy the endpoint and key you'll need).

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-info -f
   git clone https://github.com/microsoftlearning/mslearn-ai-information-extraction mslearn-ai-info
   ```

   > **Tip**: As you enter commands into the cloudshell, the output may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-info/Labfiles/knowledge/python
   ls -a -l
   ```

5. Install the Azure AI Search SDK and Azure identity packages by running the following commands:

   Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-identity azure-search-documents==11.5.1
```

6. Run the following command to edit the configuration file for your app:

Code Copy

```
code .env
```

The configuation file is opened in a code editor.

7. Edit the configuration file to replace the following placeholder values:
    - **your_search_endpoint** (*replace with the endpoint for your Azure AI Search resource*)
    - **your_query_key** *(replace with the query key for your Azure AI Search resource*)
    - **your_index_name** (*replace with the name of your index, which should be* `margies-index`)
8. When you've updated the placeholders, use the **CTRL+S** command to save the file and then use the **CTRL+Q** command to close it.

> **Tip**: Now that you've copied the endpoint and key from the Azure portal, you might want to maximize the cloud shell pane to make it easier to work in.

9. Run the following command to open the code file for your app:

Code Copy

```
code search-app.py
```

The code file is opened in a code editor.

10. Review the code, and note that it performs the following actions:
    - Retrieves the configuration settings for your Azure AI Search resource and index from the configuration file you edited.
    - Creates a **SearchClient** with the endpoint, key, and index name to connect to your search service.
    - Prompts the user for a search query (until they enter "quit")
    - Searches the index using the query, returning the following fields (ordered by metadata_storage_name):
        - metadata_storage_name
        - locations
        - people
        - keyphrases
    - Parses the search results that are returned to display the fields returned for each document in the result set.
11. Close the code editor pane (*CTRL+Q*), keeping the cloud shell command line console pane open
12. Enter the following command to run the app:

Code Copy

```
python search-app.py
```

13. When prompted, enter a query such as `London` and view the results.
14. Try another query, such as `flights` .
15. When you're finished testing the app, enter `quit` to close it.

16. Close the Cloud shell, returning to the Azure portal.

View the knowledge store

After you have run an indexer that uses a skillset to create a knowledge store, the enriched data extracted by the indexing process is persisted in the knowledge store projections.

View object projections

The *object* projections defined in the Margie's Travel skillset consist of a JSON file for each indexed document. These files are stored in a blob container in the Azure Storage account specified in the skillset definition.

1. In the Azure portal, view the Azure Storage account you created previously.
2. Select the **Storage browser** tab (in the pane on the left) to view the storage account in the storage explorer interface in the Azure portal.
3. Expand **Blob containers** to view the containers in the storage account. In addition to the **documents** container where the source data is stored, there should be two new containers: **knowledge-store** and **margies-skillset-image-projection**. These were created by the indexing process.
4. Select the **knowledge-store** container. It should contain a folder for each indexed document.
5. Open any of the folders, and then select the **objectprojection.json** file it contains and use the **Download** button on the toolbar to download and open it. Each JSON file contains a representation of an indexed document, including the enriched data extracted by the skillset as shown here (formatted to make it easier to read).

Code Copy

```
{
    "metadata_storage_content_type": "application/pdf",
    "metadata_storage_size": 388622,
    "<more_metadata_fields>": "...",
    "key_phrases":[
        "Margie’s Travel",
        "Margie's Travel",
        "best travel experts",
        "world-leading travel agency",
        "international reach"
        ],
    "locations":[
        "Dubai",
        "Las Vegas",
        "London",
        "New York",
        "San Francisco"
        ],
    "image_tags":[
        "outdoor",
        "tree",
        "plant",
        "palm"
        ],
    "more fields": "..."
}
```

The ability to create *object* projections like this enables you to generate enriched data objects that can be incorporated into an enterprise data analysis solution.

View file projections

The *file* projections defined in the skillset create JPEG files for each image that was extracted from the documents during the indexing process.

1. In the *Storage browser* interface in the Azure portal, select the **margies-skillset-image-projection** blob container. This container contains a folder for each document that contained images.
2. Open any of the folders and view its contents - each folder contains at least one *.jpg file.
3. Open any of the image files, and download and view it to see the image.

The ability to generate *file* projections like this makes indexing an efficient way to extract embedded images from a large volume of documents.

View table projections

The *table* projections defined in the skillset form a relational schema of enriched data.

1. In the *Storage browser* interface in the Azure portal, expand **Tables**.
2. Select the **margiesSkillsetDocument** table to view data. This table contains a row for each document that was indexed:
3. View the **margiesSkillsetKeyPhrases** table, which contains a row for each key phrase extracted from the documents.

The ability to create *table* projections enables you to build analytical and reporting solutions that query a relational schema. The automatically generated key columns can be used to join the tables in queries - for example to return all of the key phrases extracted from a specific document.

Clean-up

Now that you've completed the exercise, delete all the resources you no longer need. Delete the Azure resources:

1. In the Azure portal, select **Resource groups**.
2. Select the resource group you don't need, then select **Delete resource group**.

More information

To learn more about Azure AI Search, see the Azure AI Search documentation.