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.

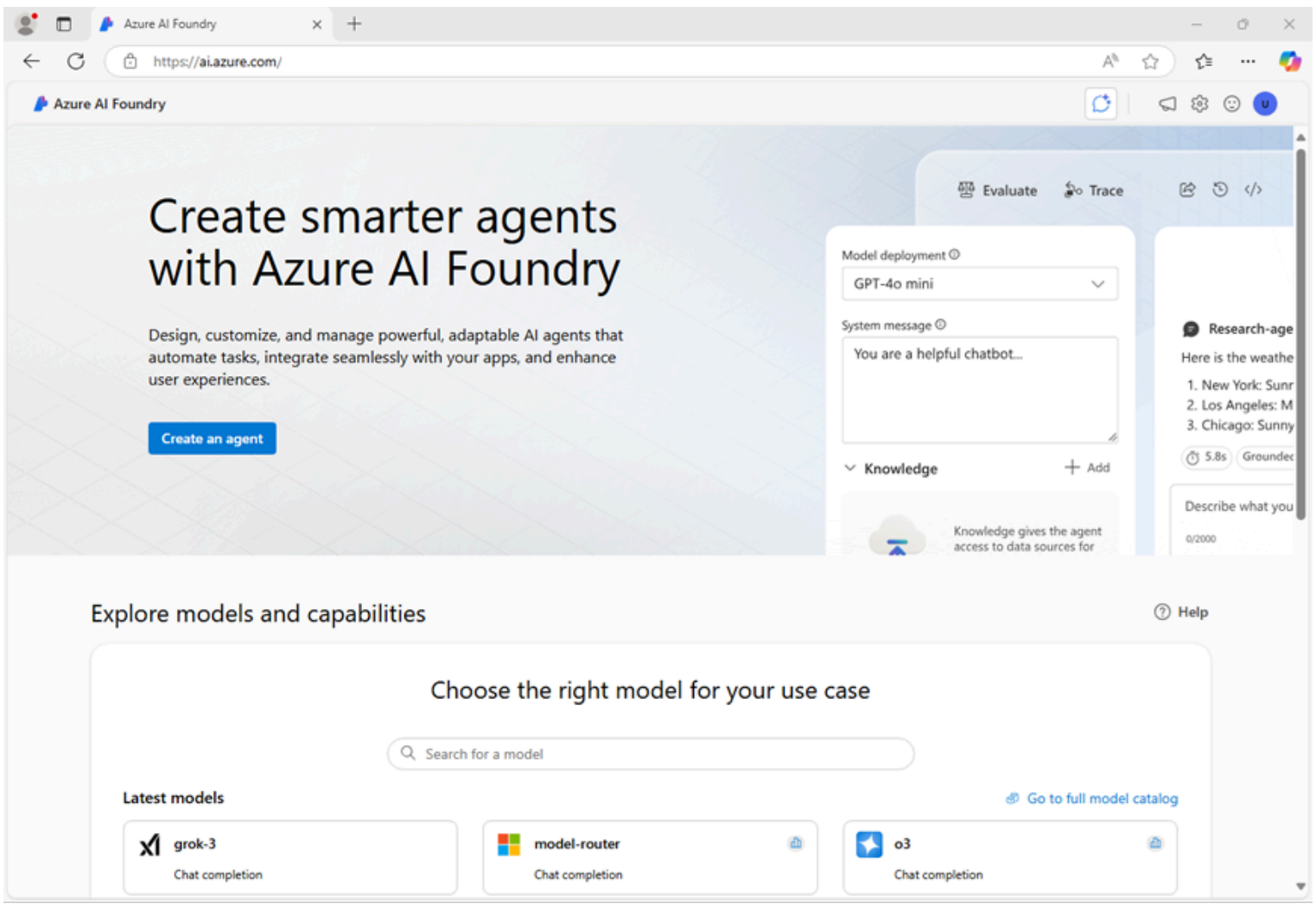
This exercise takes approximately **30** minutes.

> **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*****

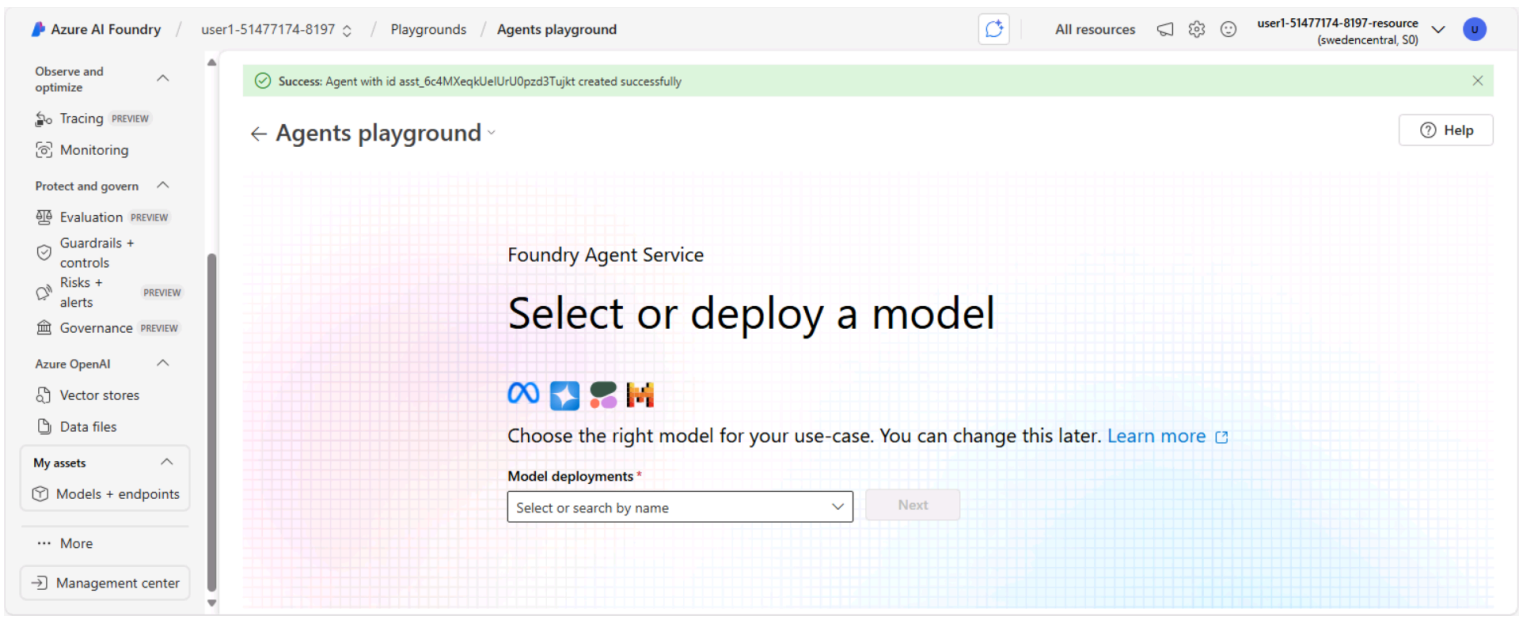
    > * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

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.

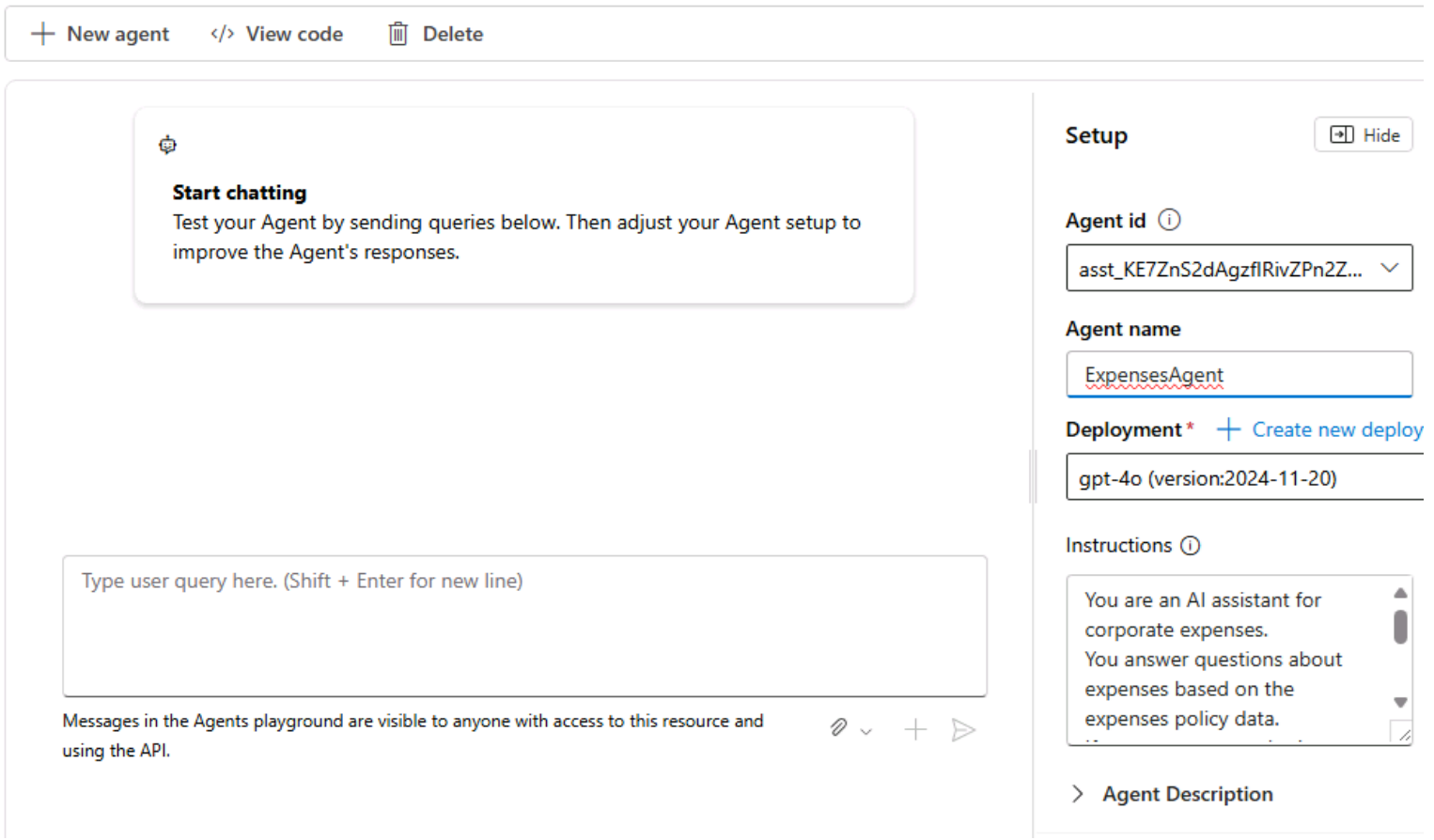
Create your agent

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

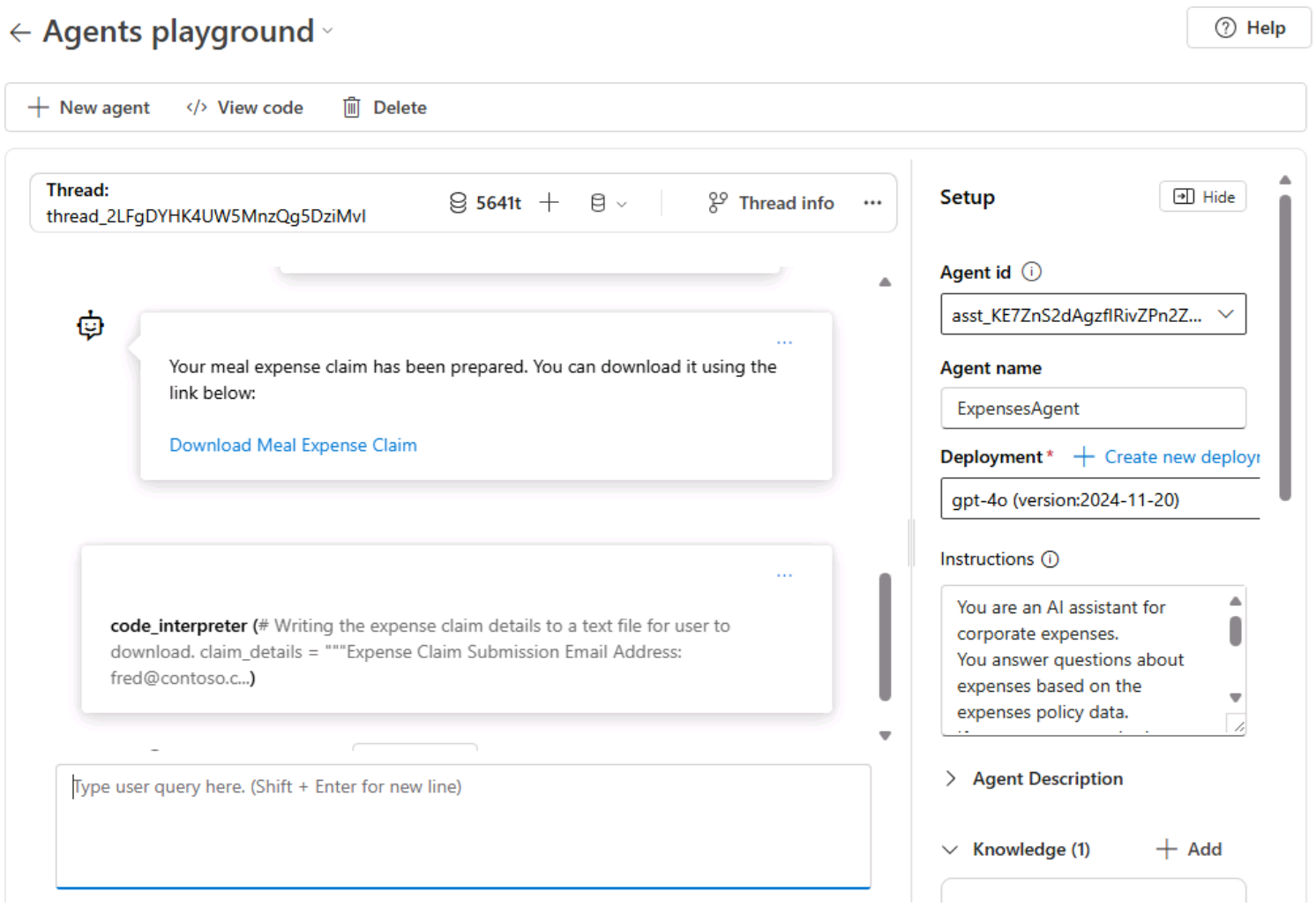
Now that you've created an agent, you can test it in the playground chat.

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.

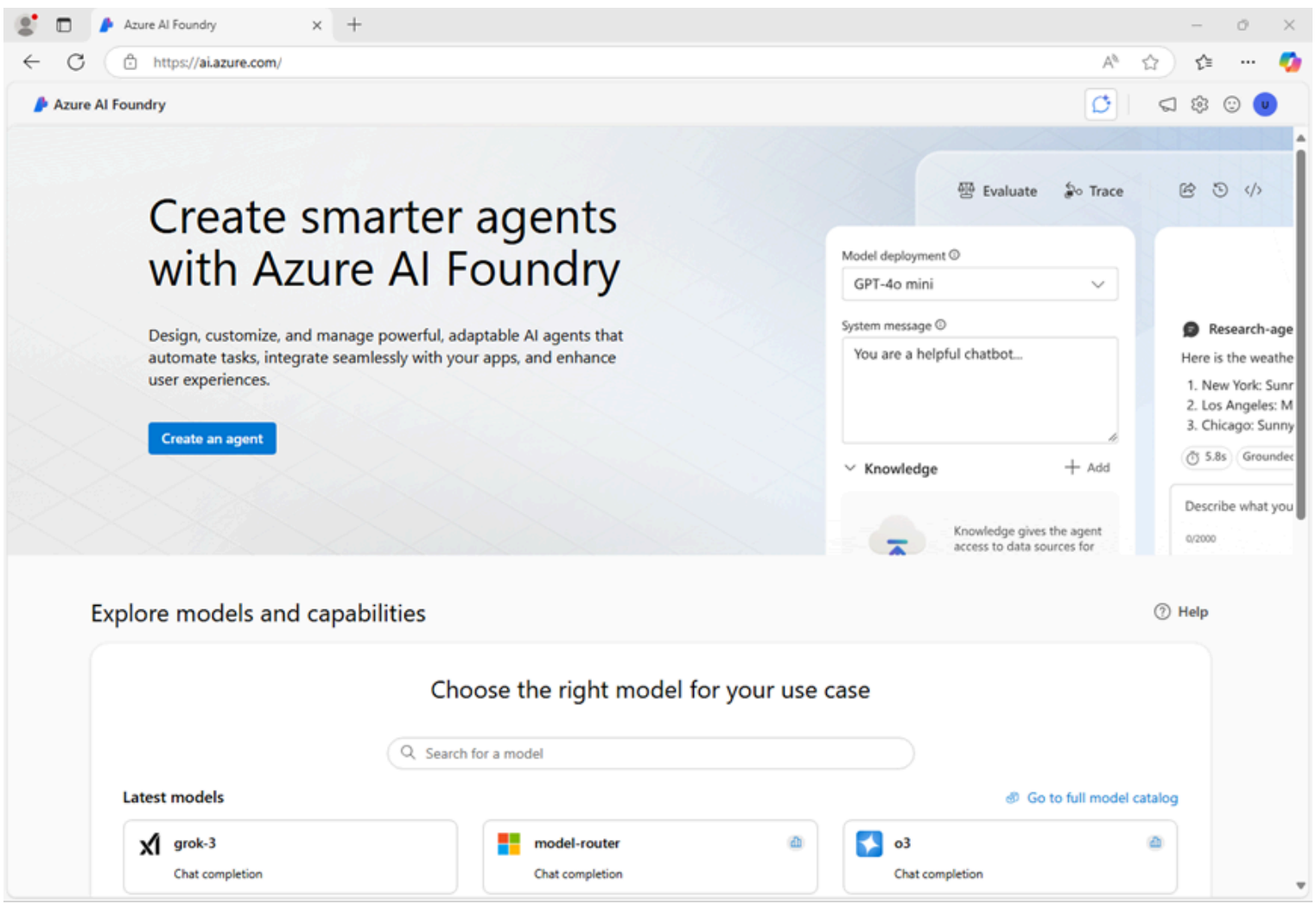
This exercise should take approximately **30** minutes to complete.

> **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.

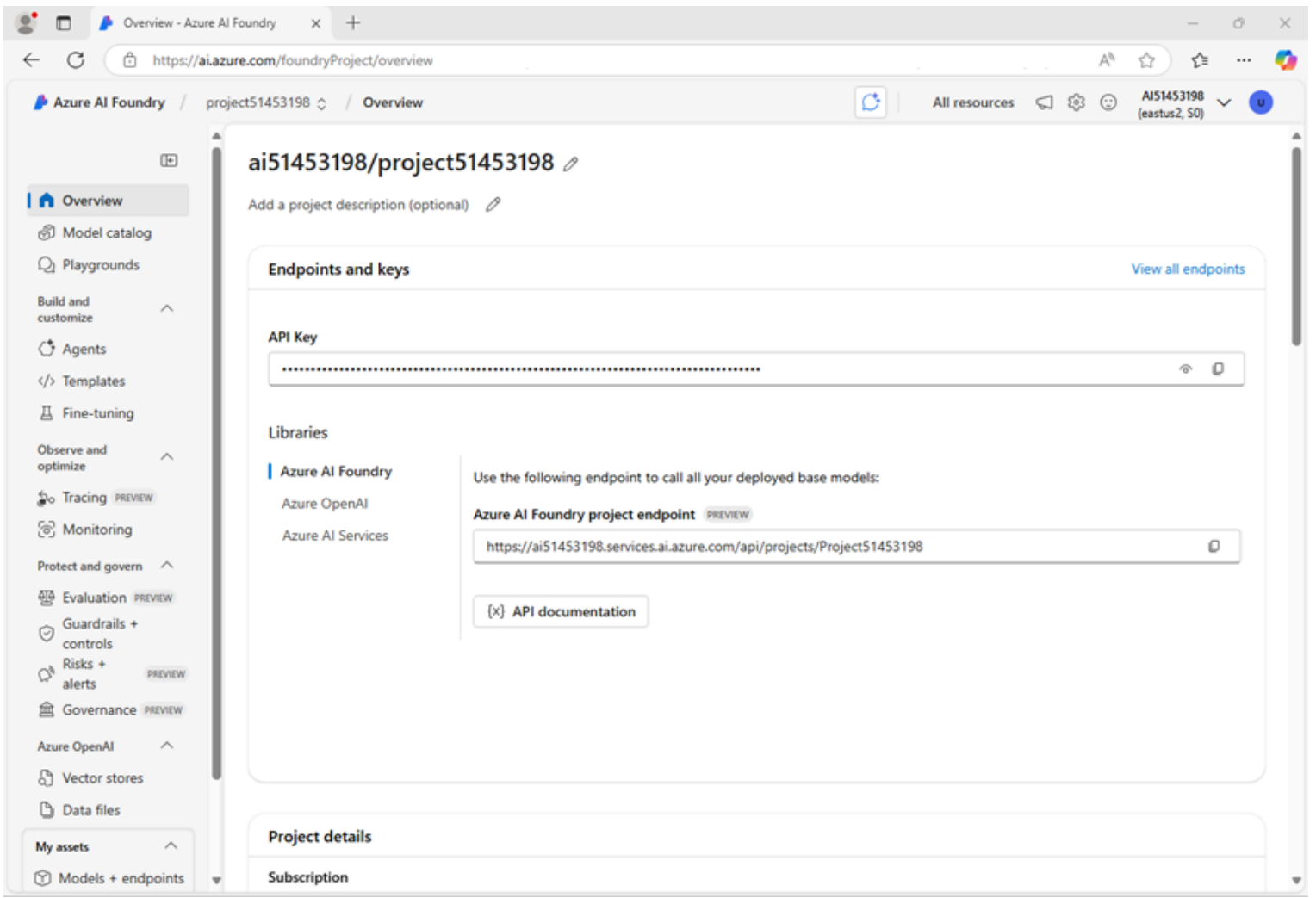
5. Select **Create** and wait for your project to be created.

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.”

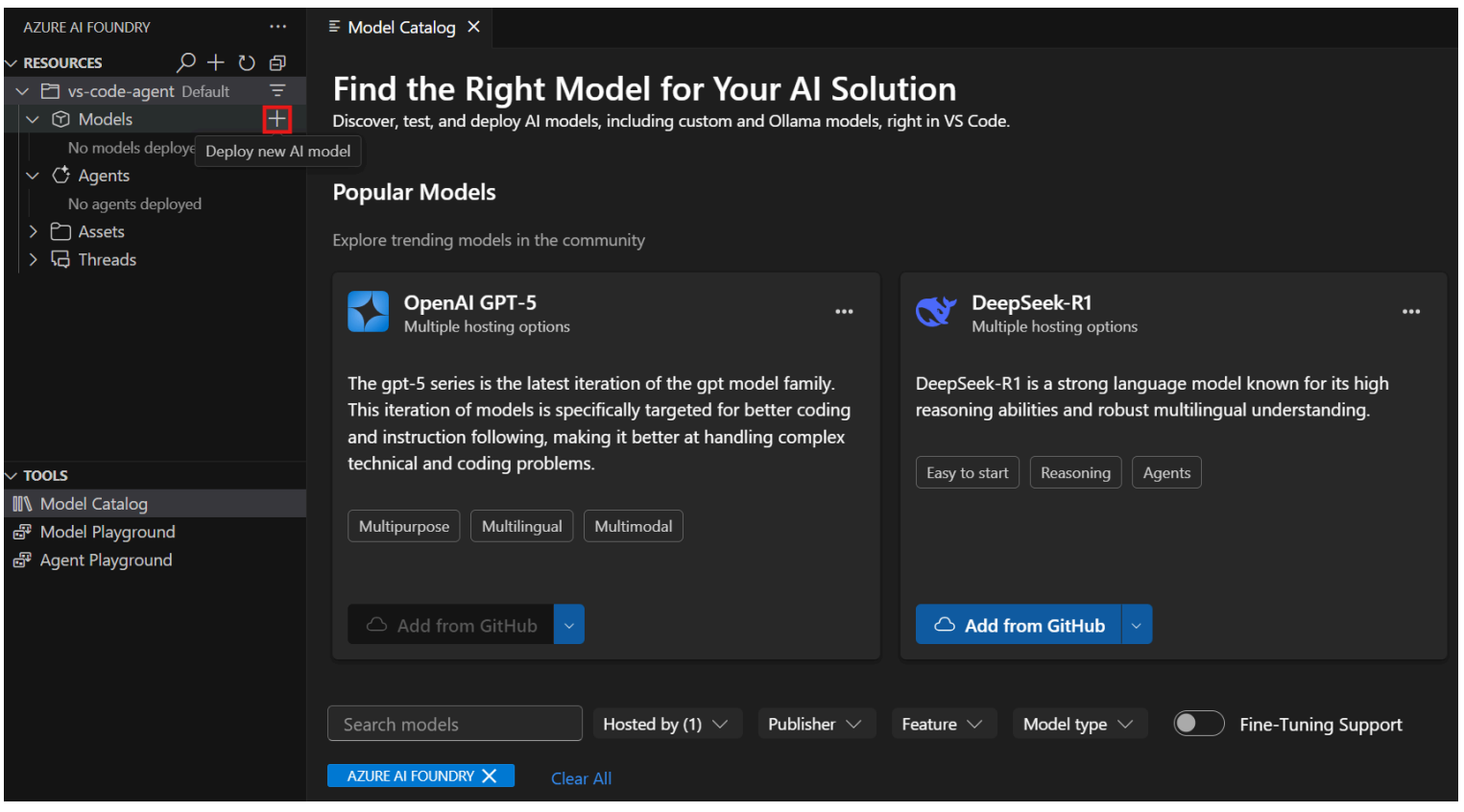
Deploy a model

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).

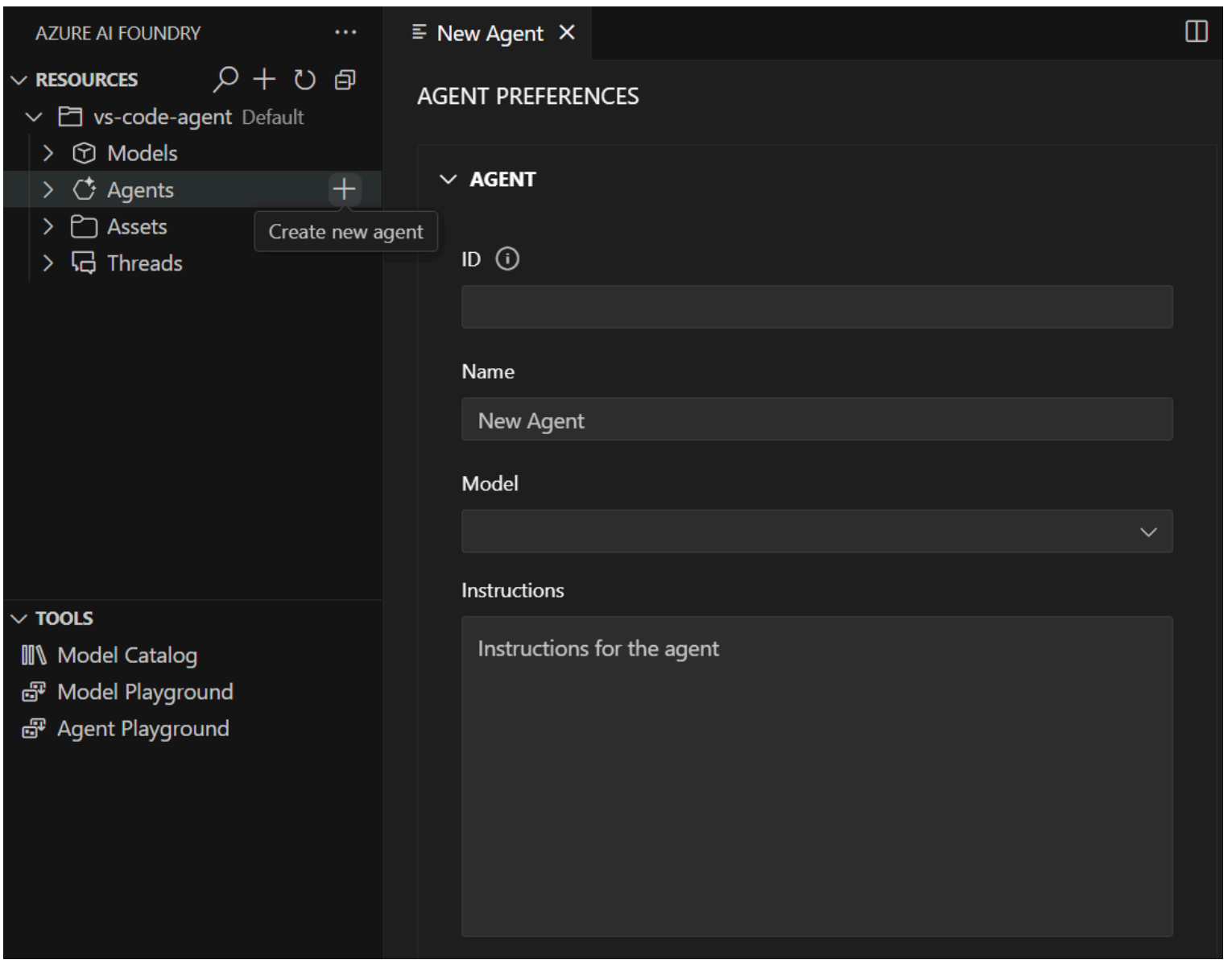


3. Select **Deploy in Azure** next to the gpt-4o model.
4. Configure the deployment settings:
    - **Deployment name**: Enter a name like “gpt-4o-deployment”
    - **Deployment type**: Select **Global Standard** (or **Standard** if Global Standard is not available)
    - **Model version**: Leave as 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:

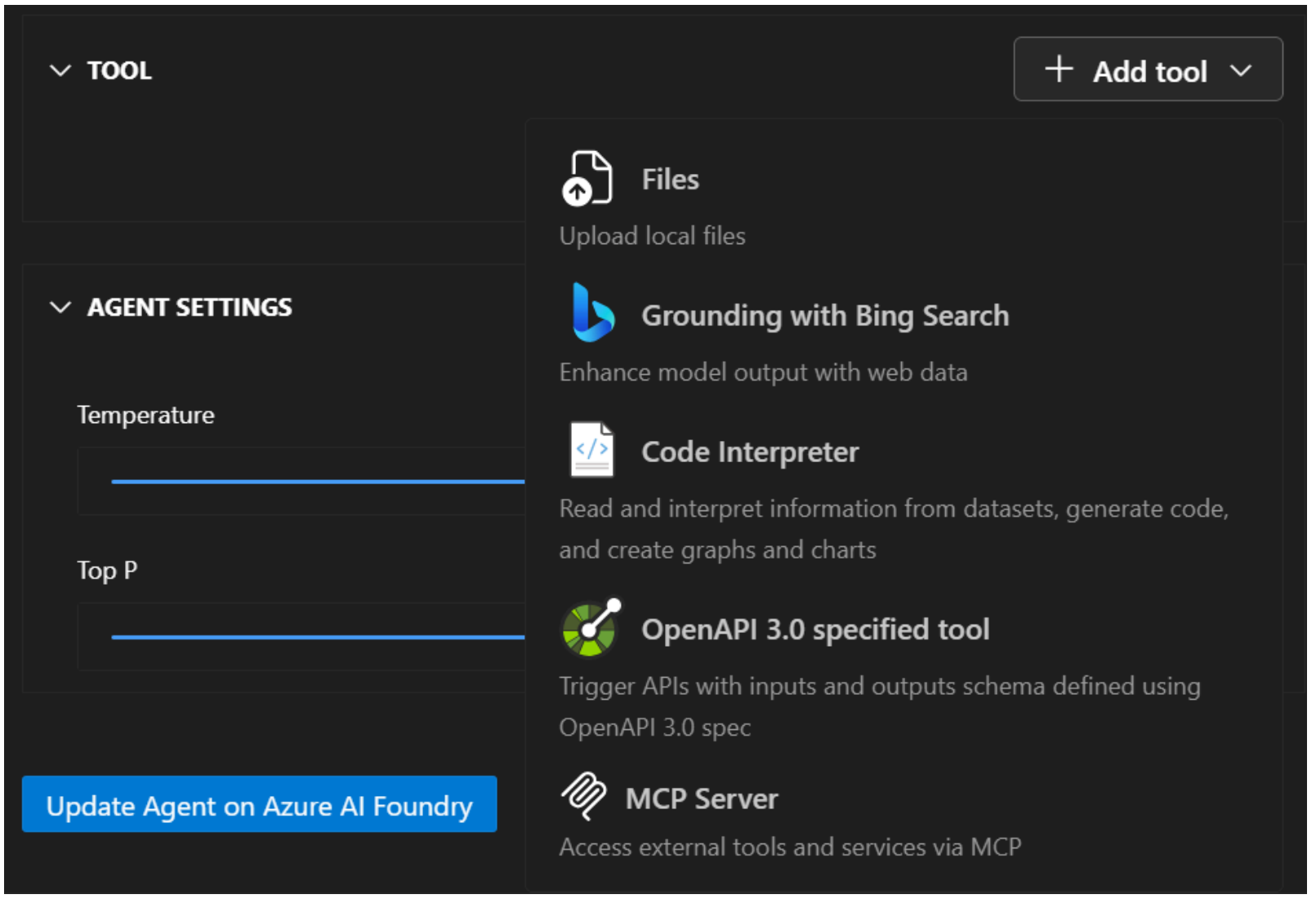
```
You are an AI agent that helps users research information from various sources. Use the available tools to access up-to-date information and provide comprehensive responses based on external data sources.
```

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.

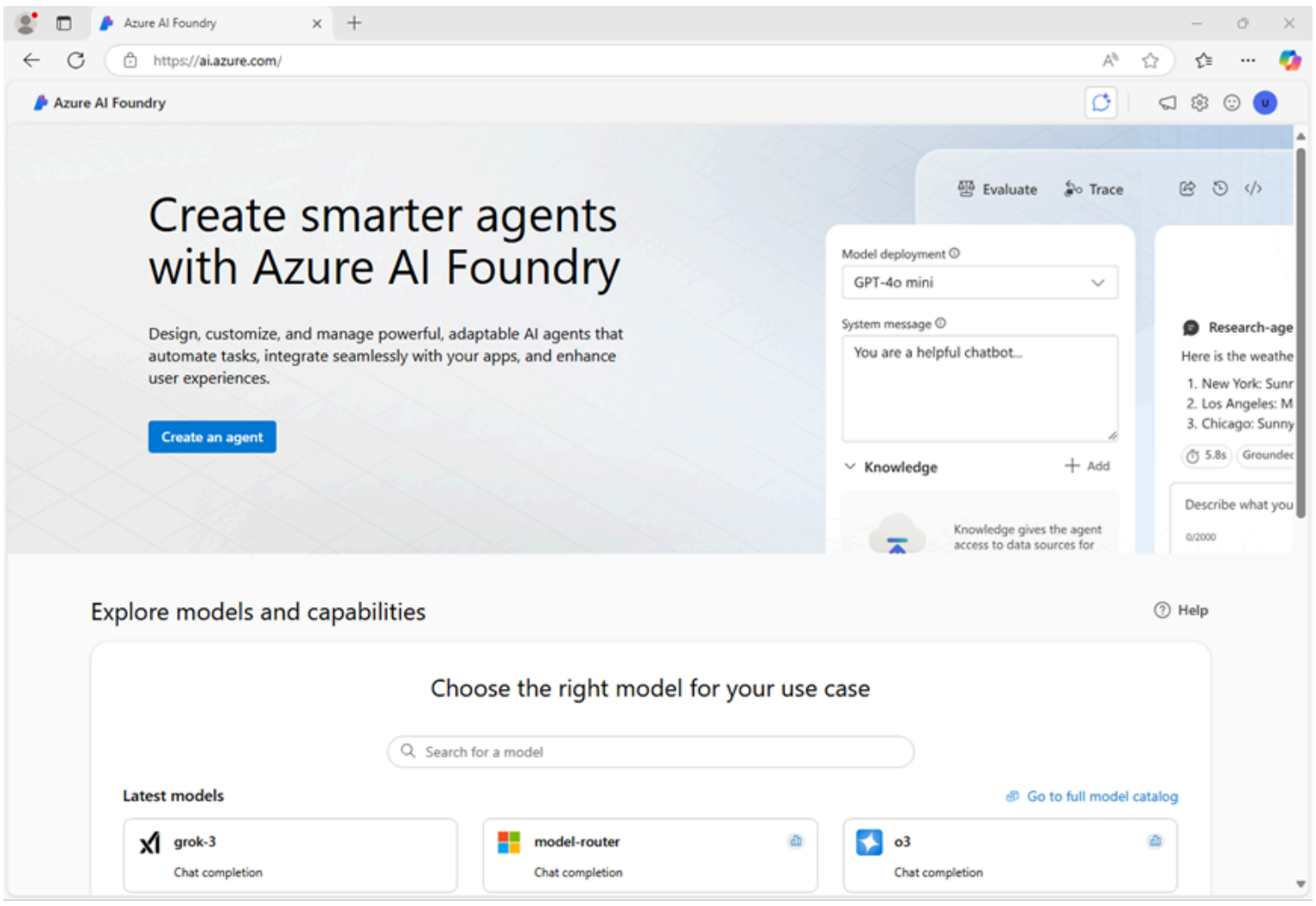
This exercise should take approximately **30** minutes to complete.

> **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):



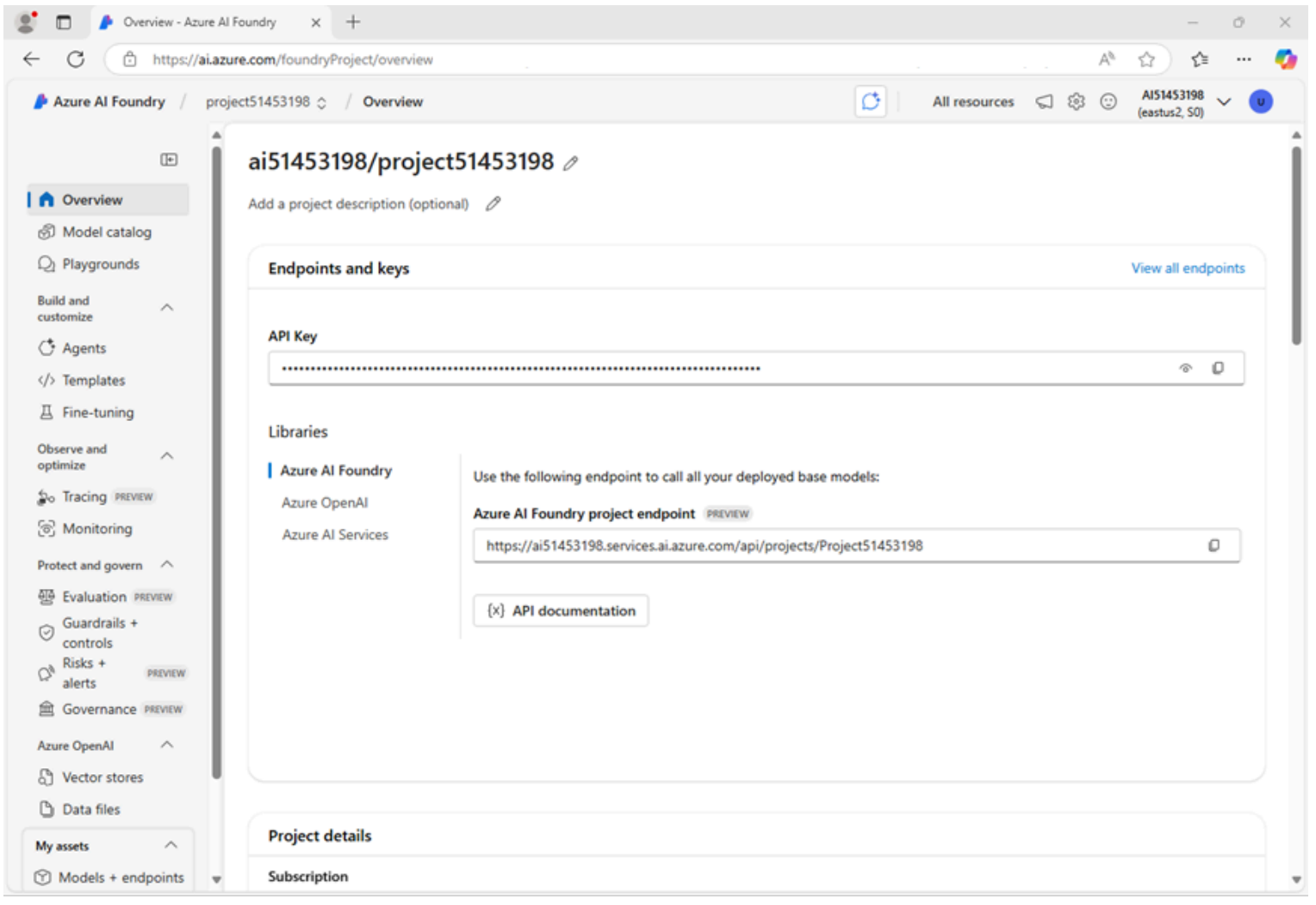
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **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.

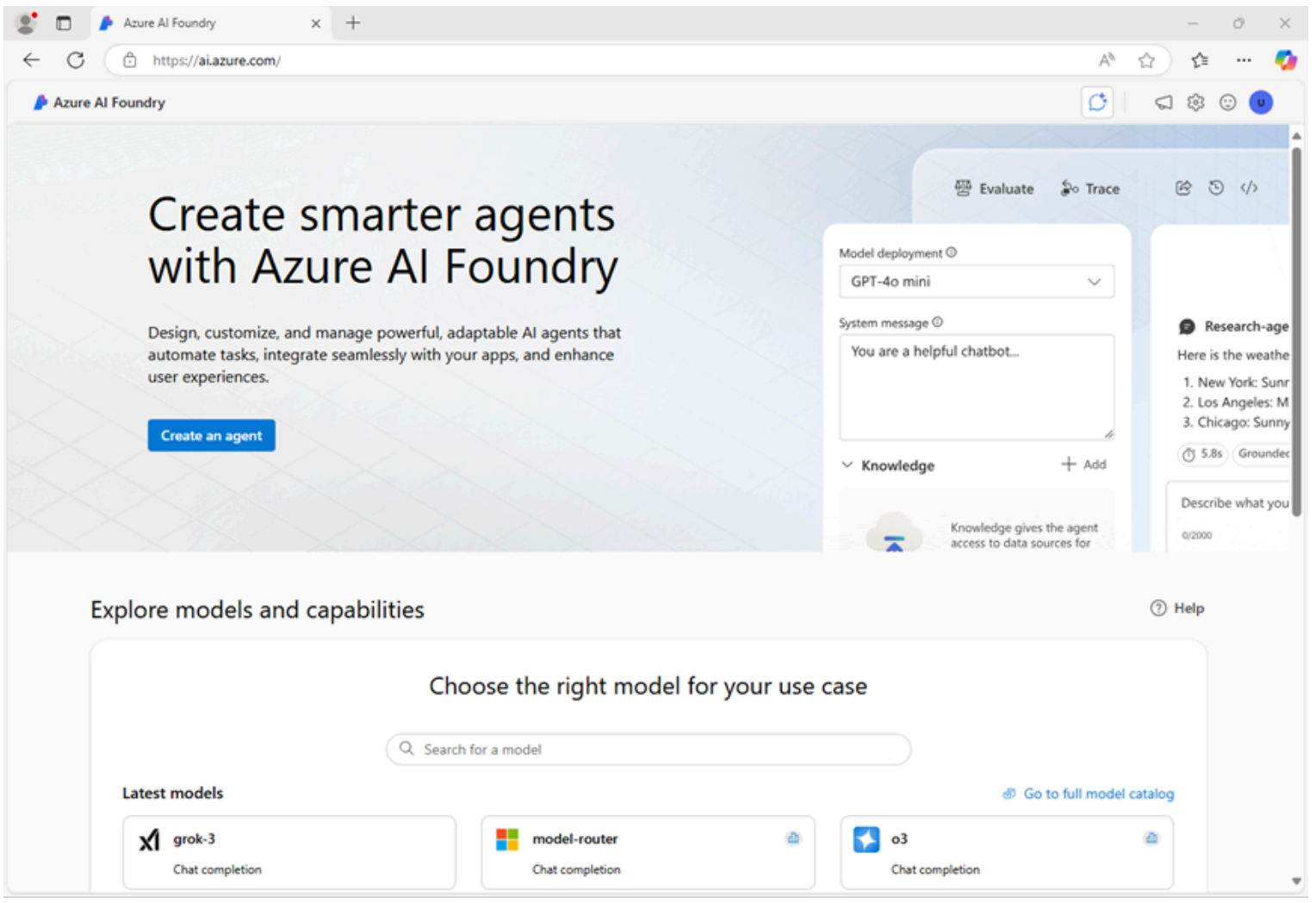
This exercise should take approximately **30** minutes to complete.

> **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):



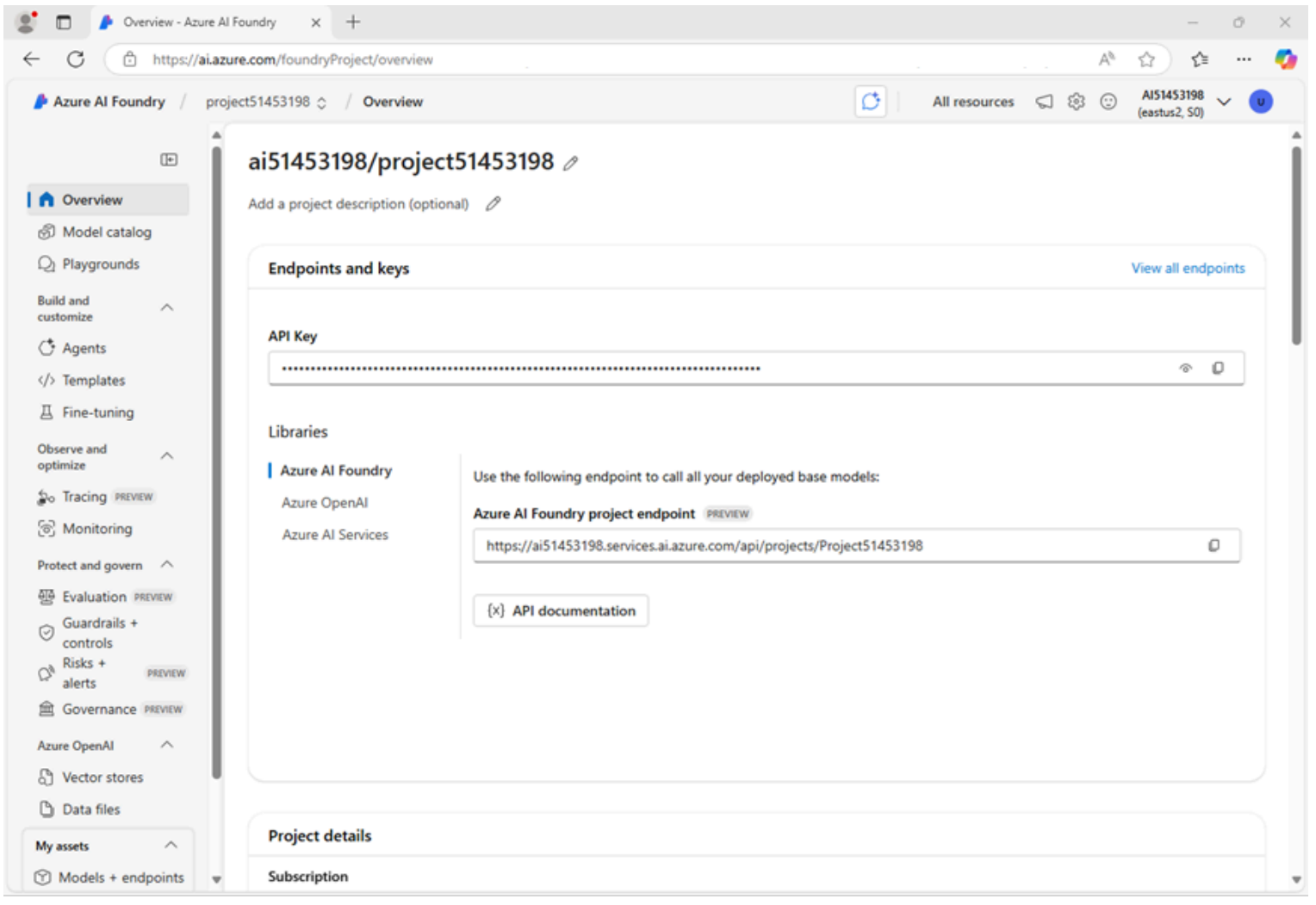
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **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.

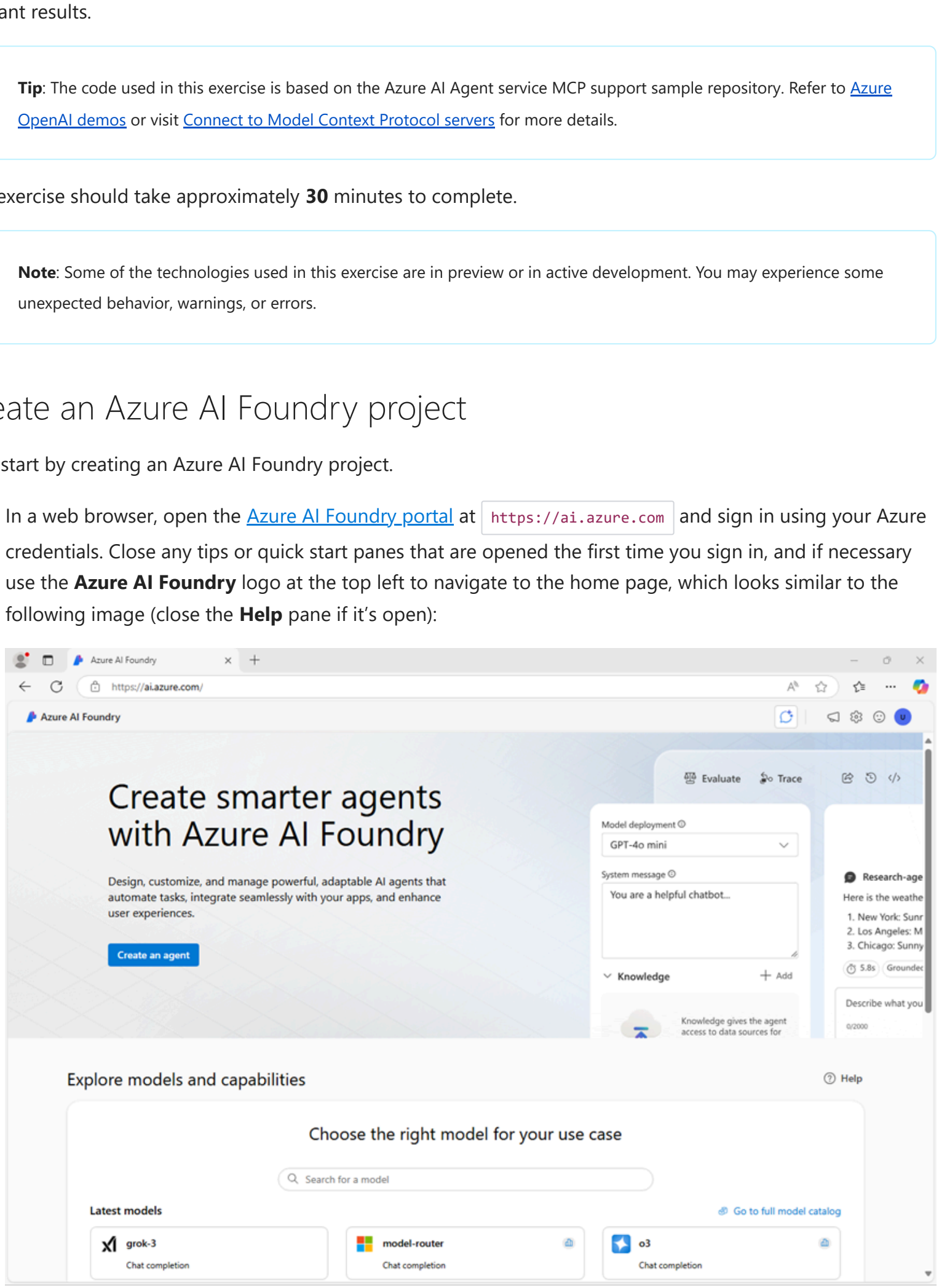
This exercise should take approximately **30** minutes to complete.

> **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: **

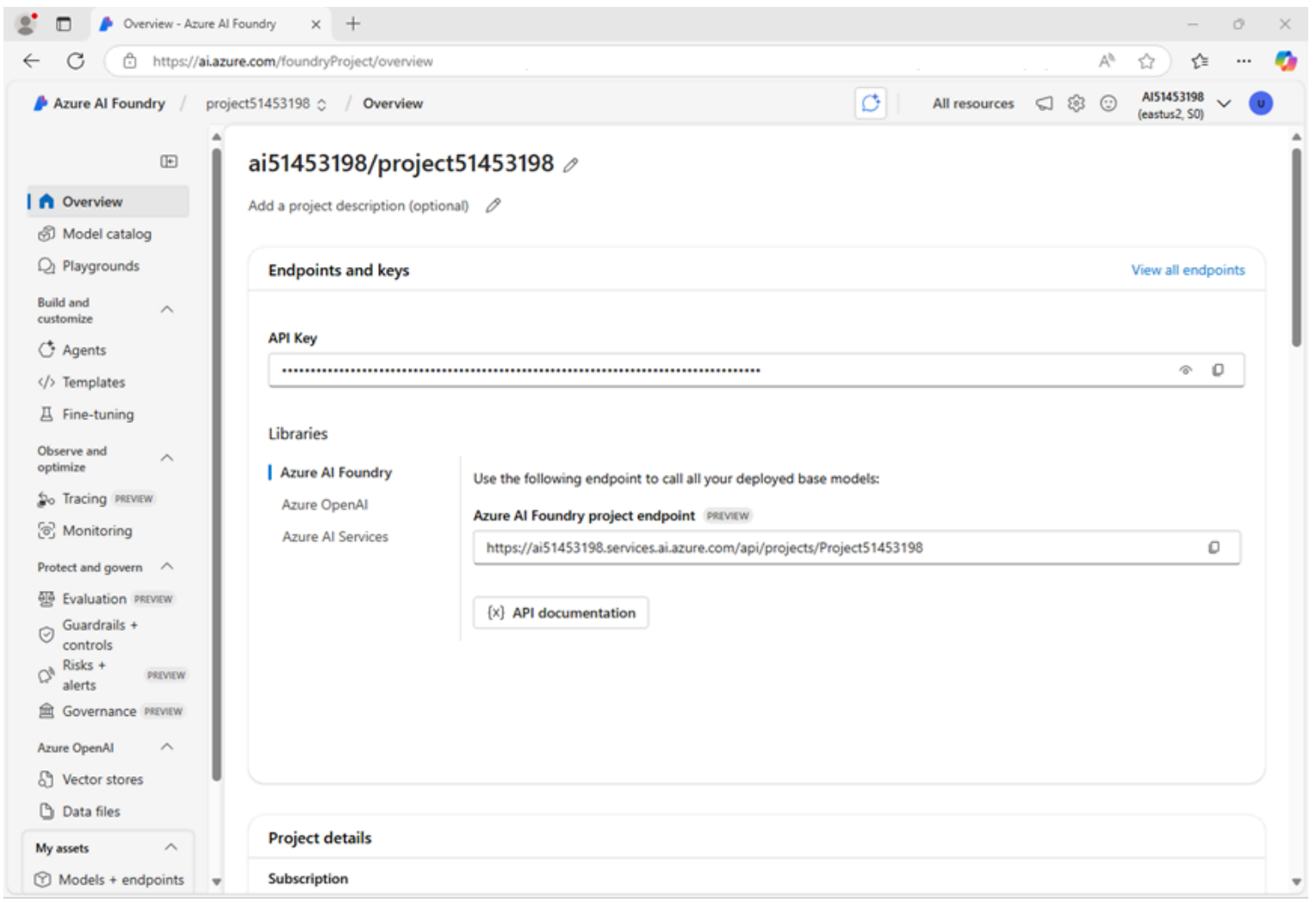
- West US 2
- West US
- Norway East
- Switzerland North
- UAE North
- South India

> * 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.

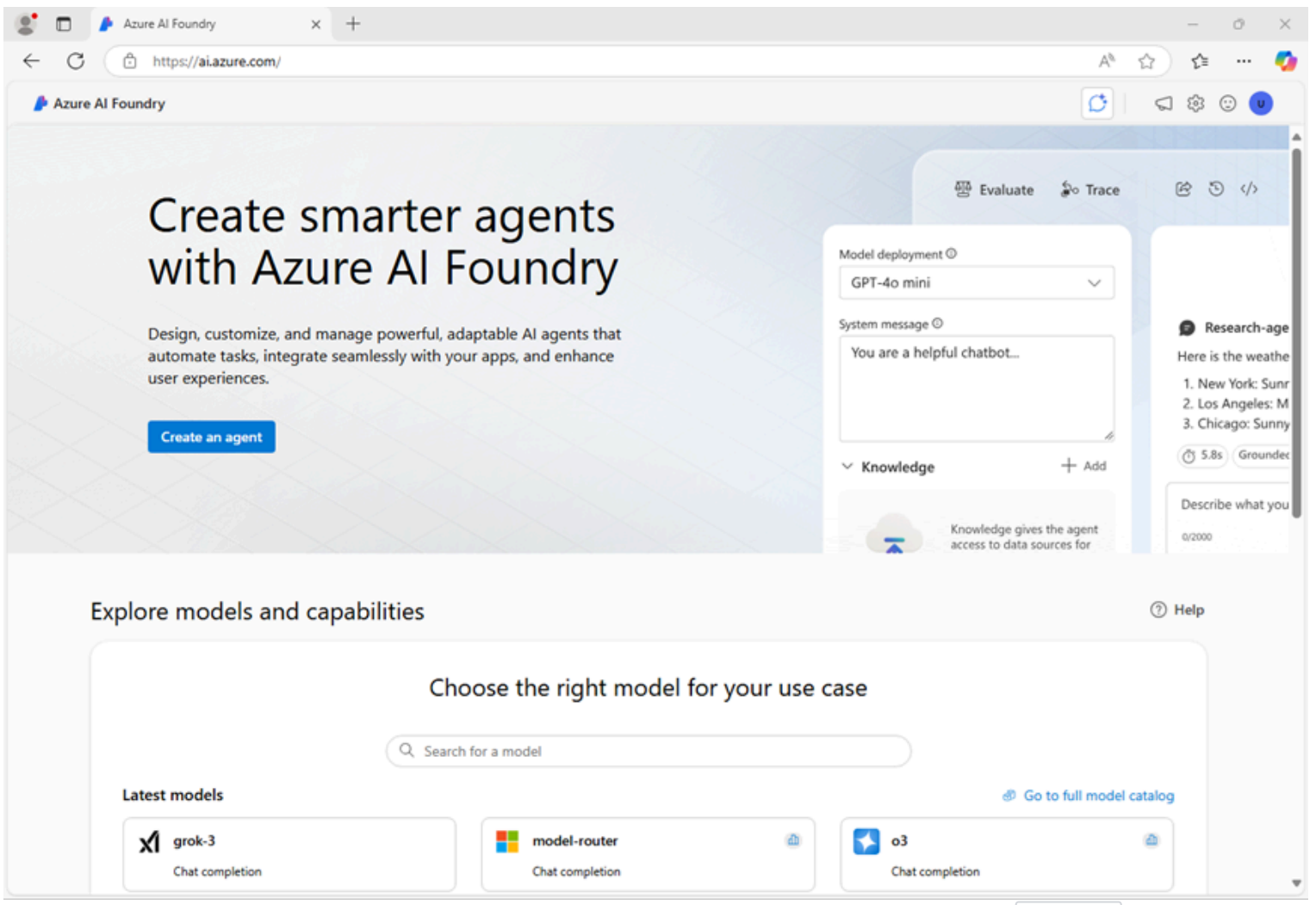
This exercise should take approximately **30** minutes to complete.

> **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):

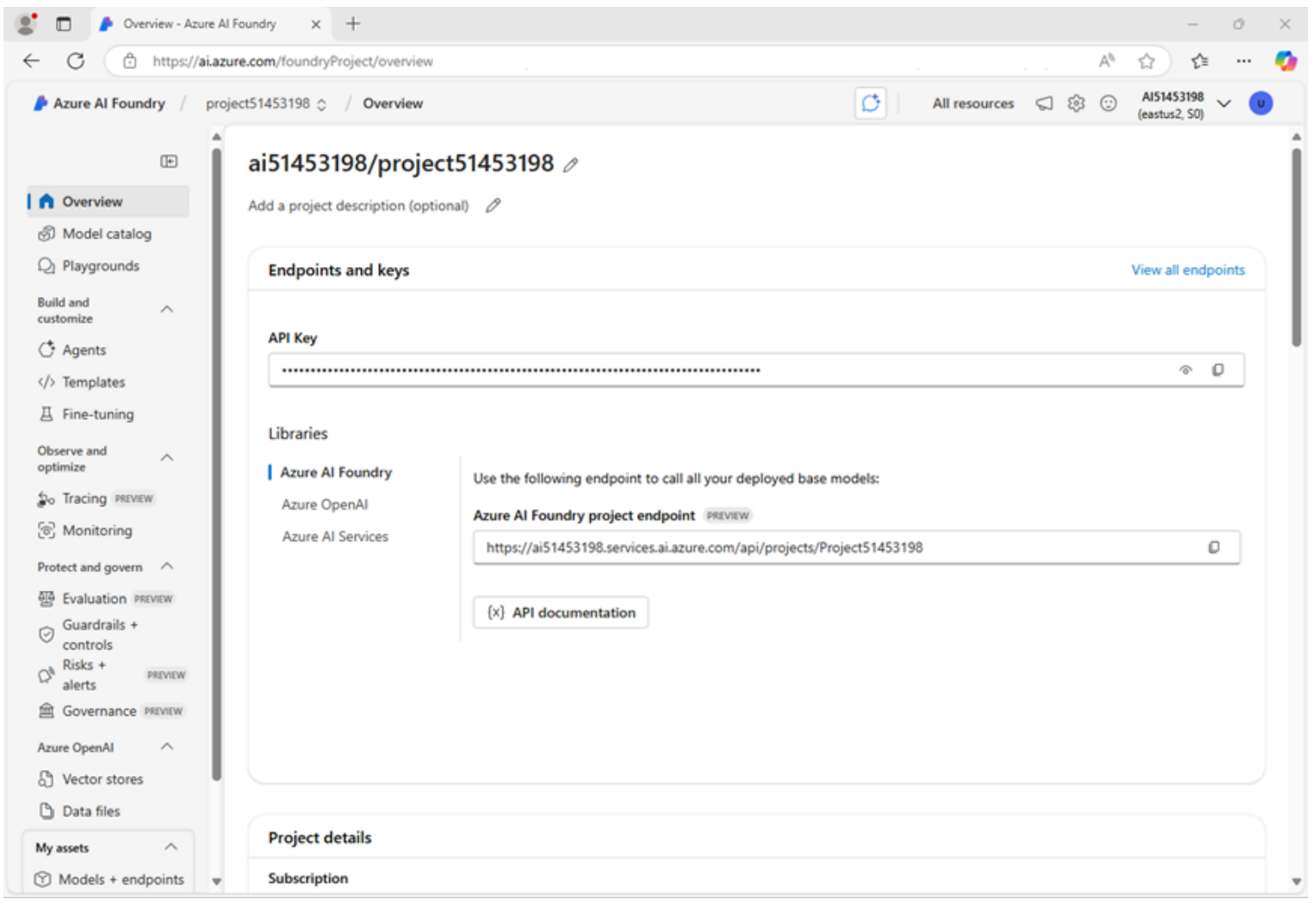


2. On the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted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

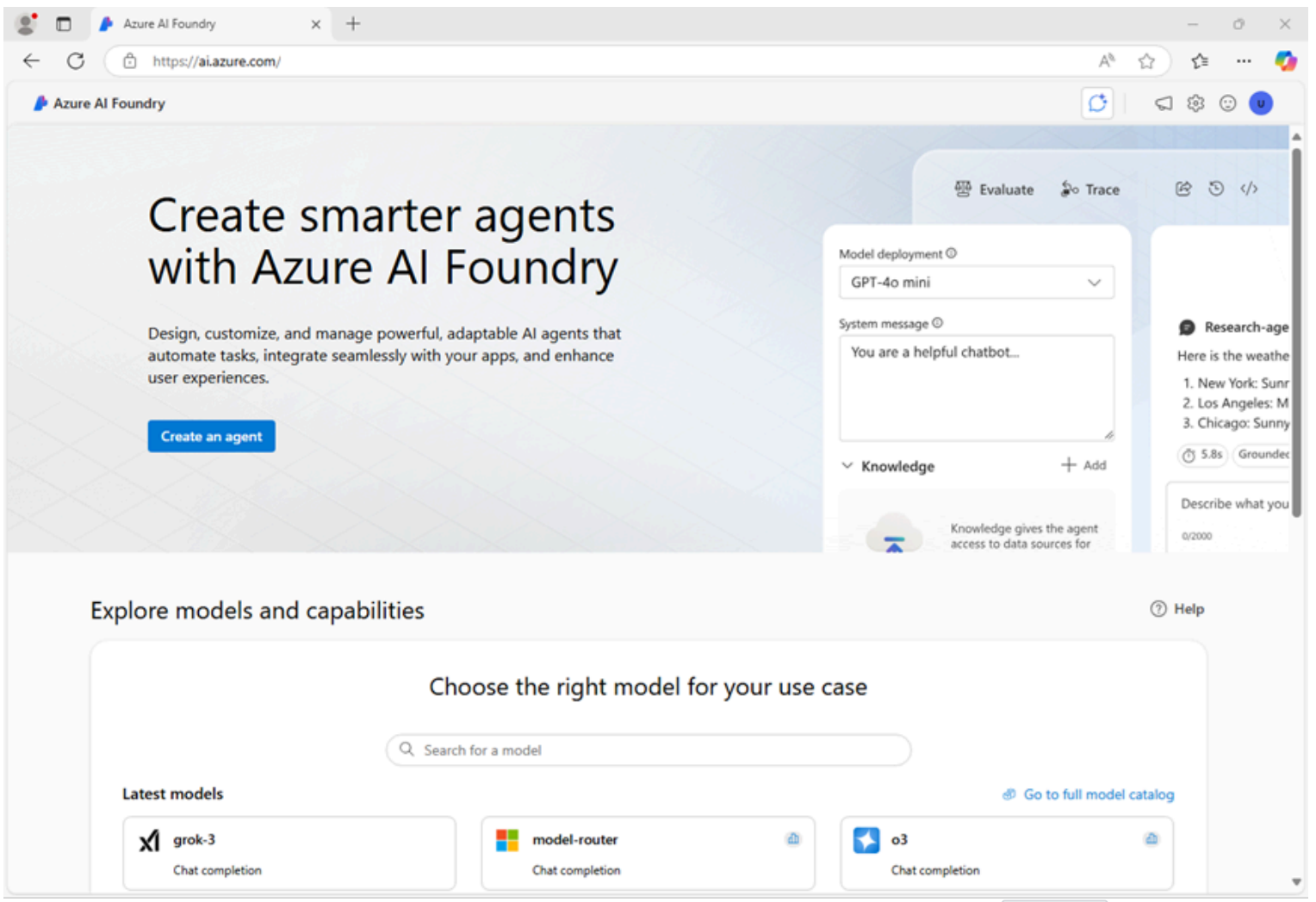
This exercise should take approximately **30** minutes to complete.

> **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):

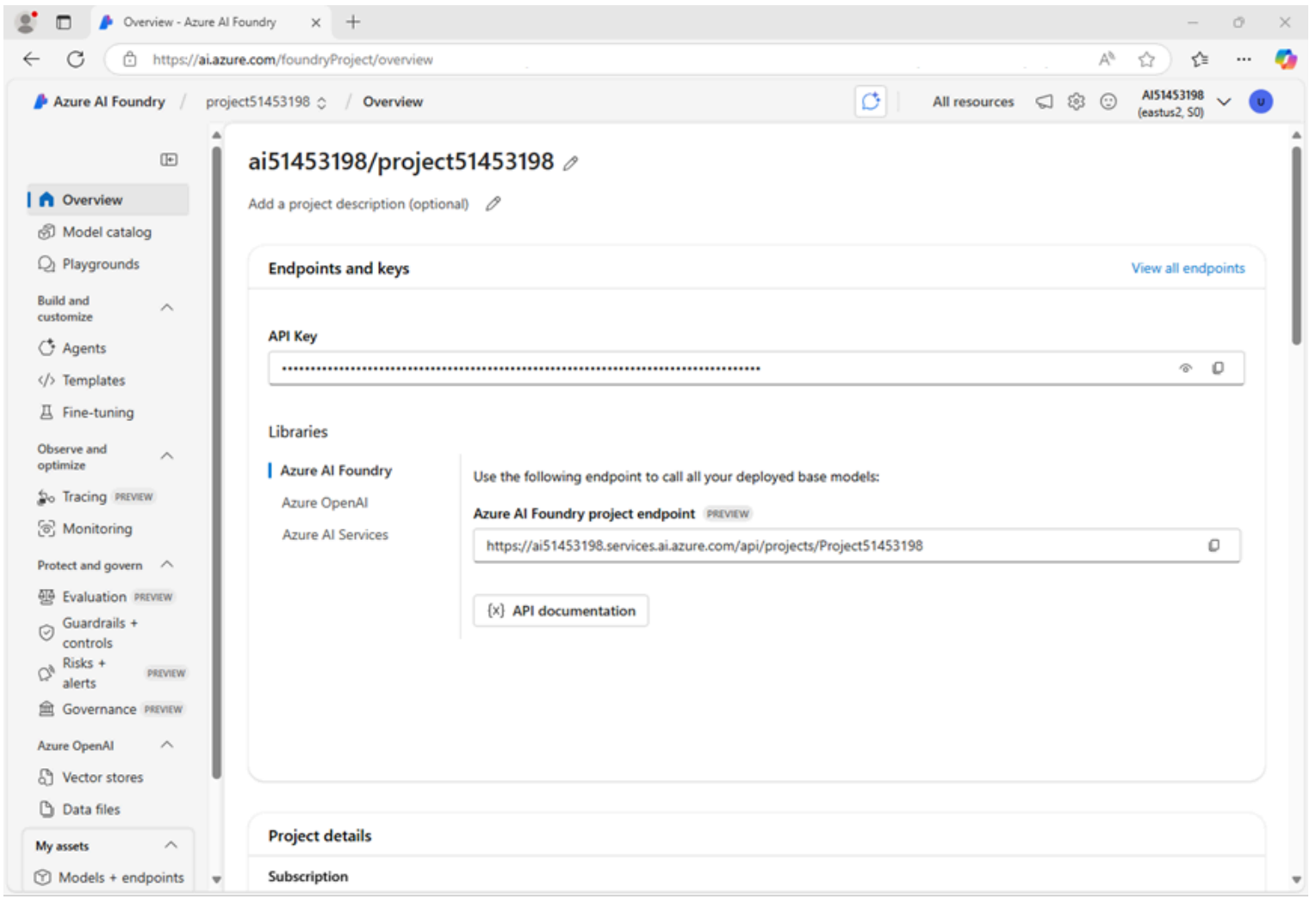


2. In the home page, in the **Explore models and capabilities** section, search for the `gpt-4o` model; which we'll use in our project.
3. In the search results, select the **gpt-4o** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted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

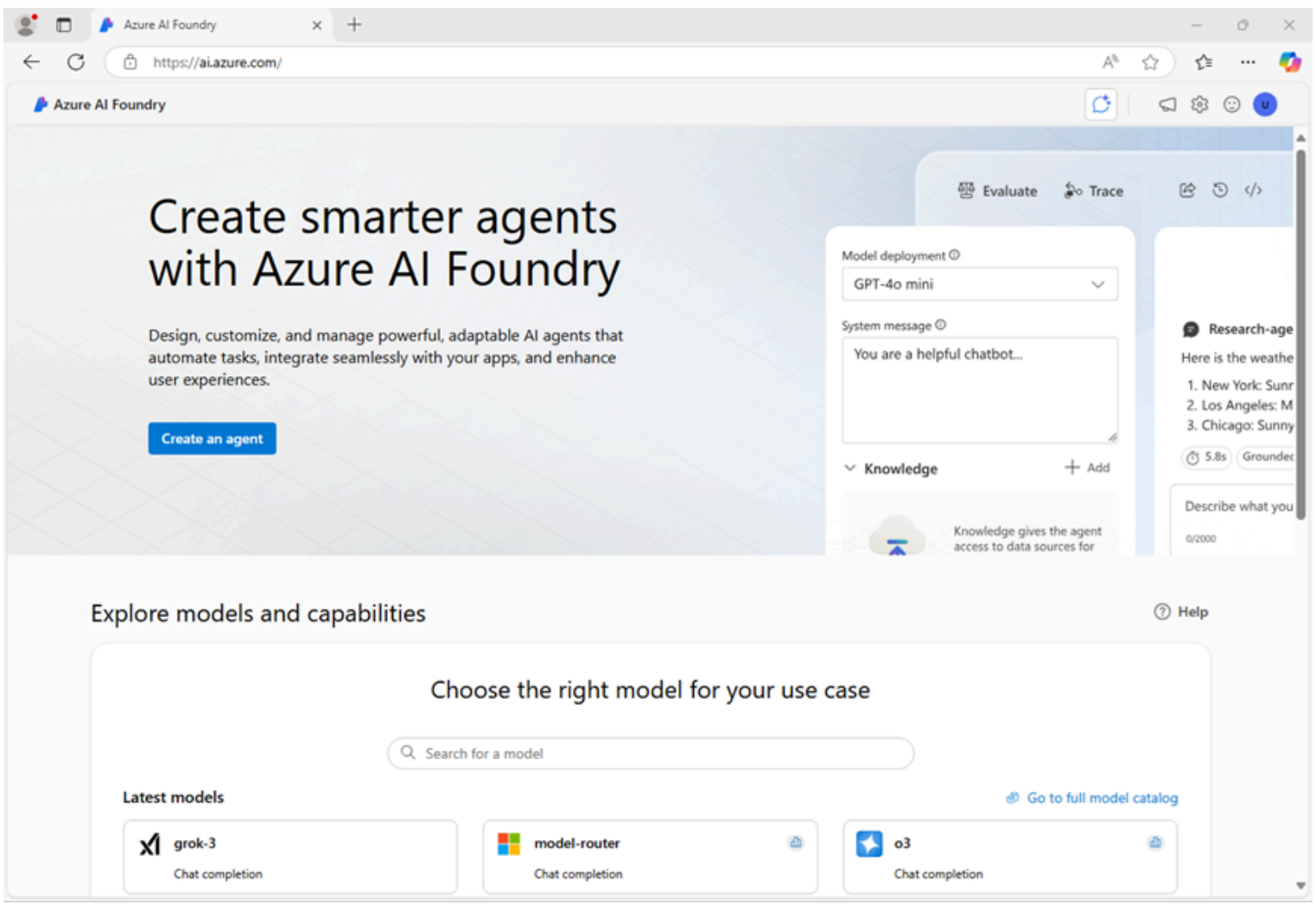
This exercise should take approximately **30** minutes to complete.

> **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):



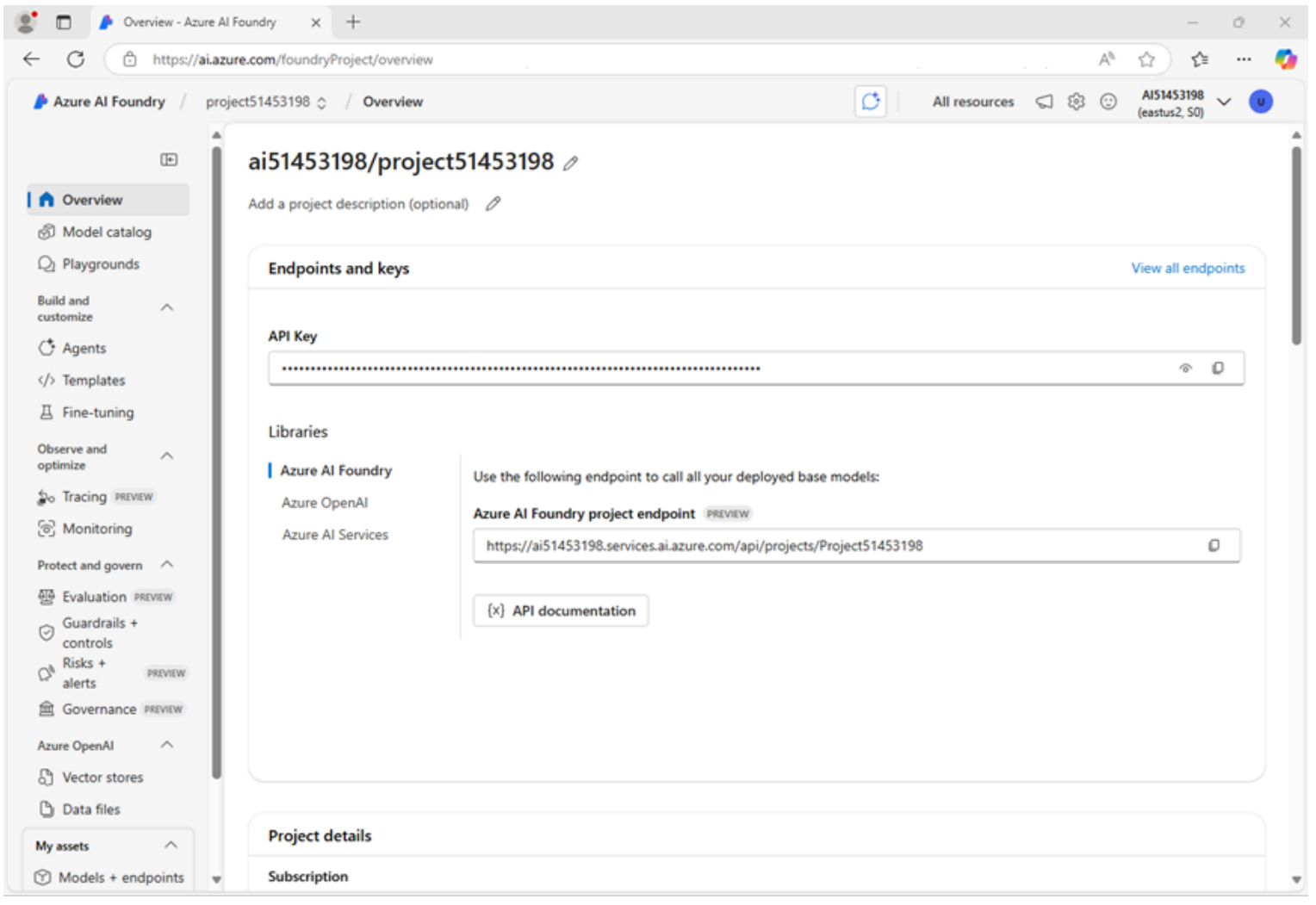
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*
    - **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.