

[Create an Azure AI Foundry project](#)

[Develop an agent that uses function tools](#)

[Clean up](#)

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.

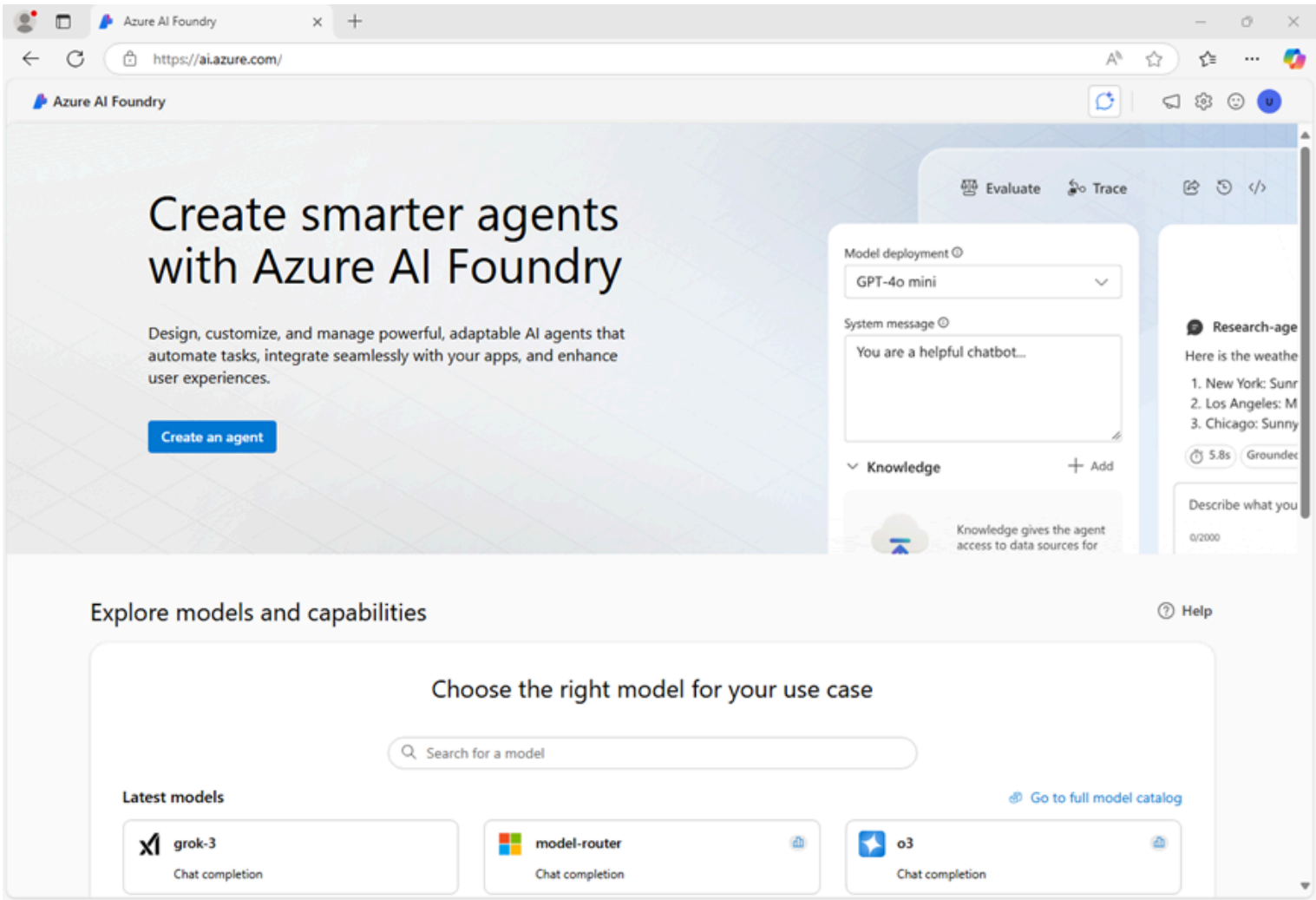
This exercise should take approximately **30** minutes to complete.

! **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.

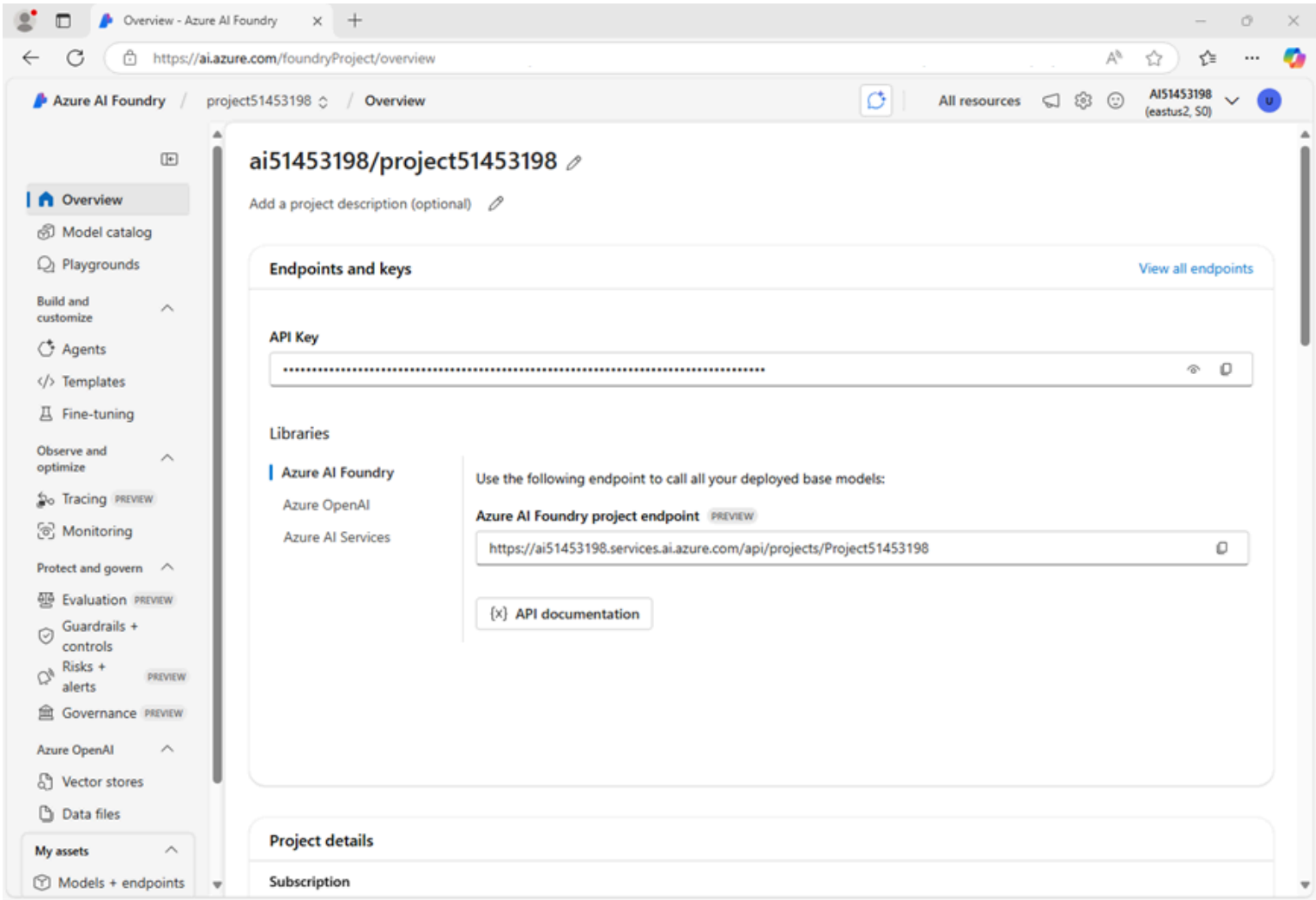
1. In a web browser, open the [Azure AI Foundry portal](#) at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, 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](https://portal.azure.com) 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):

CodeCopy

```
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.

CodeCopy

```
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:

CodeCopy

```
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:

CodeCopy

```
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:

CodeCopy

```
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.

CodeCopy

```
# 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('-', '')[0:6]
    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):

CodeCopy

```
# 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.

CodeCopy

```
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:

CodeCopy

```
# 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.

CodeCopy

```
# 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.

CodeCopy

```
# 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.

CodeCopy

```
# 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.

CodeCopy

```
# 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.

CodeCopy

```
# 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

CodeCopy

```
# 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.


CodeCopy


```
# 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.

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.