# Develop an Azure AI agent with the Semantic Kernel SDK

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

> **Tip**: The code used in this exercise is based on the for Semantic Kernel SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET and Java. Refer to Supported Semantic Kernel languages for details.

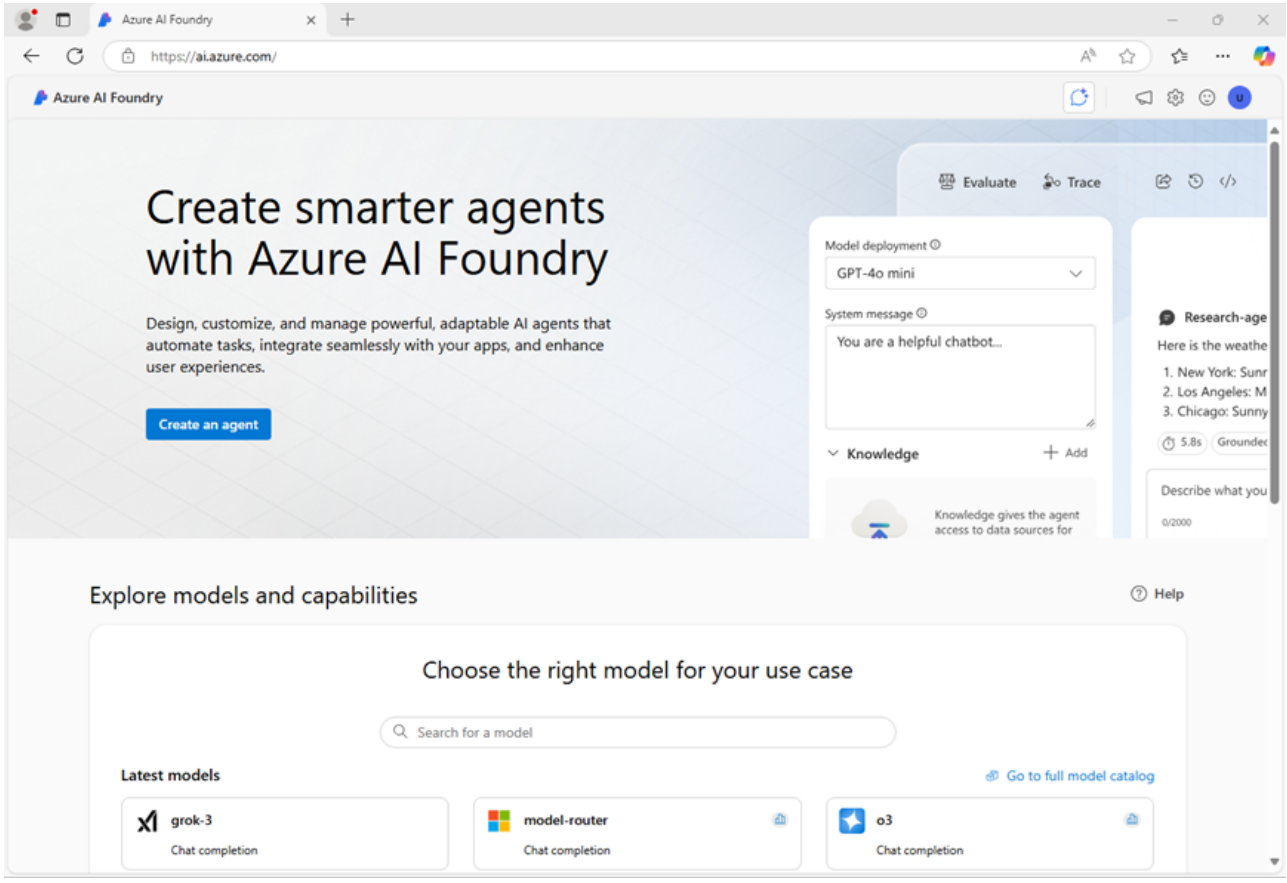
This exercise should take approximately **30** minutes to complete.

> **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.

## 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 Services supported location****

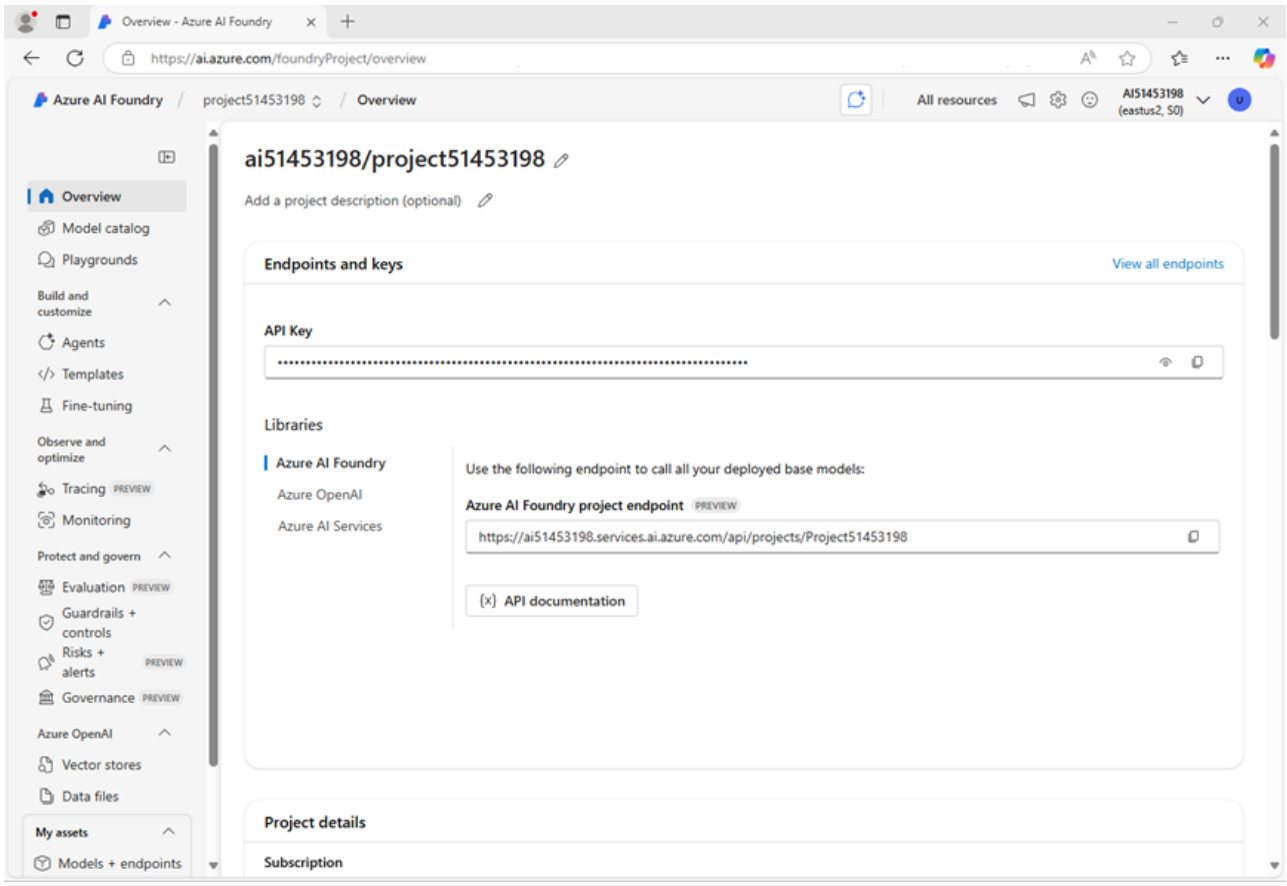
    > * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region.

6. Select **Create** and wait for your project, including the gpt-4 model deployment you selected, to be created.

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

8. In the **Setup** pane, note the name of your model deployment; which should be **gpt-4o**. You can confirm this by viewing the deployment in the **Models and endpoints** page (just open that page in the navigation pane on the left).

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



# Create 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):

   ```
   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.

   ```
   cd ai-agents/Labfiles/04-semantic-kernel/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:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install python-dotenv azure-identity semantic-kernel --upgrade
```

> **Note**: Installing *semantic-kernel* autmatically installs a semantic kernel-compatible version of *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 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 semantic-kernel.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
    - An **EmailPlugin** class that includes a kernel function named **send_email**; which will be used by your agent to simulate the functionality used to send an email.

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:

```

```

```
# Add references
from dotenv import load_dotenv
from azure.identity.aio import DefaultAzureCredential
from semantic_kernel.agents import AzureAIAgent, AzureAIAgentSettings,
AzureAIAgentThread
from semantic_kernel.functions import kernel_function
from typing import Annotated
```

4. Near the bottom of the file, find the comment **Create a Plugin for the email functionality**, and add the following code to define a class for a plugin containing a function that your agent will use to send email (plug-ins are a way to add custom functionality to Semantic Kernel agents)

```
# Create a Plugin for the email functionality
class EmailPlugin:
    """A Plugin to simulate email functionality."""

    @kernel_function(description="Sends an email.")
    def send_email(self,
                   to: Annotated[str, "Who to send the email to"],
                   subject: Annotated[str, "The subject of the email."],
                   body: Annotated[str, "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 new **EmailPlugin** class code, in the **create_expense_claim** function, find the comment **Get configuration settings**, and add the following code to load the configuration file and create an **AzureAIAgentSettings** object (which will automatically include the Azure AI Agent settings from the configuration).

   (Be sure to maintain the indentation level)

```
# Get configuration settings
load_dotenv()
ai_agent_settings = AzureAIAgentSettings()
```

6. Find the comment **Connect to the Azure AI Foundry project**, and add the following code to connect to your Azure AI Foundry project using the Azure credentials you're currently signed in with.

   (Be sure to maintain the indentation level)

```
# Connect to the Azure AI Foundry project
async with (
```

```
    DefaultAzureCredential(
        exclude_environment_credential=True,
        exclude_managed_identity_credential=True) as creds,
    AzureAIAgent.create_client(
        credential=creds
    ) as project_client,
):
```

7. Find the comment **Define an Azure AI agent that sends an expense claim email**, and add the following code to create an Azure AI Agent definition for your agent.

   (Be sure to maintain the indentation level)

```
# Define an Azure AI agent that sends an expense claim email
expenses_agent_def = await project_client.agents.create_agent(
    model= ai_agent_settings.model_deployment_name,
    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."""
)
```

8. Find the comment **Create a semantic kernel agent**, and add the following code to create a semantic kernel agent object for your Azure AI agent, and includes a reference to the **EmailPlugin** plugin.

   (Be sure to maintain the indentation level)

```
# Create a semantic kernel agent
expenses_agent = AzureAIAgent(
    client=project_client,
    definition=expenses_agent_def,
    plugins=[EmailPlugin()]
)
```

9. 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):

```
# Use the agent to process the expenses data
# If no thread is provided, a new thread will be
# created and returned with the initial response
thread: AzureAIAgentThread | None = None
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 expenses_agent.get_response(prompt_messages,
thread=thread)
    # Display the response
    print(f"\n# {response.name}:\n{response}")
except Exception as e:
    # Something went wrong
    print (e)
finally:
    # Cleanup: Delete the thread and agent
    await thread.delete() if thread else None
    await project_client.agents.delete_agent(expenses_agent.id)
```

10. 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**).

11. 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.

```
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 semantic-kernel.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 Azure AI Agent Service SDK and Semantic Kernel to create an agent.

## 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.