COURSE

Develop AI agents on Azure

Course AI-3026-A: Develop AI agents on Azure

Get started with self-directed learning

Level
Intermediate

Product
Azure

Role
AI Engineer, Developer

Subject
Artificial intelligence

Languages
English, Chinese (Simplified), French, German, Japanese, Korean, Portuguese (Brazil), Spanish

Course Duration
1 day

This course is designed to teach participants how to create, deploy, and manage intelligent AI agents using Azure AI Foundry. Students will learn to integrate advanced AI capabilities through agents into scalable, customizable solutions.

Audience Profile

This course is designed for developers, data scientists, and IT professionals who want to build intelligent AI-driven solutions in Azure. It is ideal for individuals with a basic understanding of programming and cloud technologies looking to enhance their skills in creating scalable and interactive AI agents for business or personal projects.

You can prepare in instructor-led training or self-paced study

LEARNING PATH

Develop AI agents on Azure

0 of 9 modules completed

Intermediate • AI EngineerAzure AI services

Start >

Add

Develop AI agents on Azure

7600 XP

8 hr 6 min • Learning Path0 of 9 modules completed

Level
Intermediate

Product
Azure AI services, Azure AI Foundry, Azure AI Foundry Agent Service, Azure AI Foundry SDK

Role
AI Engineer

Subject
Artificial intelligence, Machine learning, Natural language processing

Generative Artificial Intelligence (AI) is becoming more functional and accessible, and AI agents are a key component of this evolution. This learning path will help you understand the AI agents, including when to use them and how to build them, using Microsoft Foundry Agent Service and Microsoft Agent Framework. By the end of this learning path, you will have the skills needed to develop AI agents on Azure.

Prerequisites

Before starting this module, you should be familiar with fundamental AI concepts and services in Azure. Consider completing the Get started with artificial intelligence learning path first.

Start > ⊕ Add

Redeem your code

Have an achievement code? Redeem your code now.

Modules in this learning path

Get started with AI agent development on Azure

800 XP

49 min • Module0 of 7 units completed

AI agents represent the next generation of intelligent applications. Learn how they can be developed and used on Microsoft Azure.

Start >

Overview

Develop an AI agent with Microsoft Foundry Agent Service

800 XP

55 min • Module0 of 7 units completed

This module provides engineers with the skills to begin building agents with Microsoft Foundry Agent Service.

Overview

Develop AI agents with the Microsoft Foundry extension in Visual Studio Code

800 XP

48 min • Module0 of 7 units completed

Learn how to build, test, and deploy AI agents using the Microsoft Foundry extension in Visual Studio Code.

Overview

Integrate custom tools into your agent

800 XP

53 min • Module0 of 7 units completed

Built-in tools are useful, but they may not meet all your needs. In this module, learn how to extend the capabilities of your agent by integrating custom tools for your agent to use.

Overview

Develop a multi-agent solution with Microsoft Foundry Agent Service

700 XP

46 min • Module0 of 6 units completed

Break down complex tasks with intelligent collaboration. Learn how to design multi-agent solutions using connected agents.

Overview

Integrate MCP Tools with Azure AI Agents

800 XP

51 min • Module0 of 7 units completed

Enable dynamic tool access for your Azure AI agents. Learn how to connect MCP-hosted tools and integrate them seamlessly into agent workflows.

Overview

Develop an AI agent with Microsoft Agent Framework

800 XP

55 min • Module0 of 7 units completed

This module provides engineers with the skills to begin building Microsoft Foundry Agent Service agents with Microsoft Agent Framework.

Overview

Orchestrate a multi-agent solution using the Microsoft Agent Framework

1200 XP

1 hr 13 min • Module0 of 11 units completed

Learn how to use the Microsoft Agent Framework SDK to develop your own AI agents that can collaborate for a multi-agent solution.

Overview

Discover Azure AI Agents with A2A

900 XP

56 min • Module0 of 8 units completed

Learn how to implement the A2A protocol to enable agent discovery, direct communication, and coordinated task execution across remote agents.

Overview

800 XP

Get started with AI agent development on Azure

48 min remaining • Module1 of 7 units completed

Beginner | AI Engineer | Data Scientist | Azure AI Foundry | Azure AI Foundry Agent Service

AI agents represent the next generation of intelligent applications. Learn how they can be developed and used on Microsoft Azure.

Learning objectives

By the end of this module, you learn to:

- Describe core concepts related to AI agents
- Describe options for agent development
- Create and test an agent in the Microsoft Foundry portal

Continue > ⊕ Add

Prerequisites

Before starting this module, you should be familiar with fundamental AI concepts and services in Azure.

This module is part of these learning paths

Create an AI agent with Microsoft Foundry

Develop AI agents on Azure

5 min

Exercise - Explore AI Agent development
30 min

Module assessment
3 min

Summary
1 min

Module assessment

Assess your understanding of this module. Sign in and answer all questions correctly to earn a pass designation on your profile.

Take the module assessment

ⓘ **Note:** The author created this module with assistance from AI. Learn more

100 XP

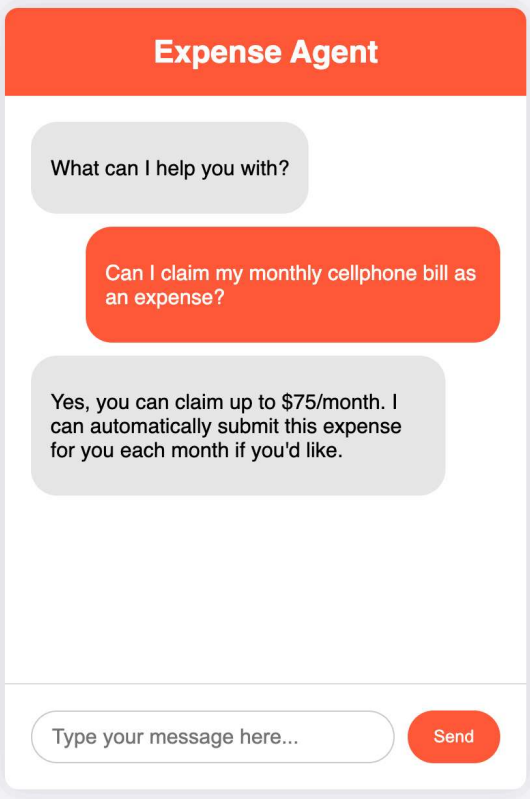
Introduction

1 minute

As generative AI models become more powerful and ubiquitous, their use grows beyond simple "chat" applications to power intelligent **agents** that can **operate autonomously** to automate tasks. Increasingly, organizations are using generative AI models to build agents that orchestrate business processes and coordinate workloads in ways that were previously unimaginable.

Single-agent scenario

Consider an organization that builds an AI agent to help employees manage expense claims. The agent could use a **generative model** combined with corporate expenses policy documentation to answer employee questions about what expenses can be claimed and what limits apply.

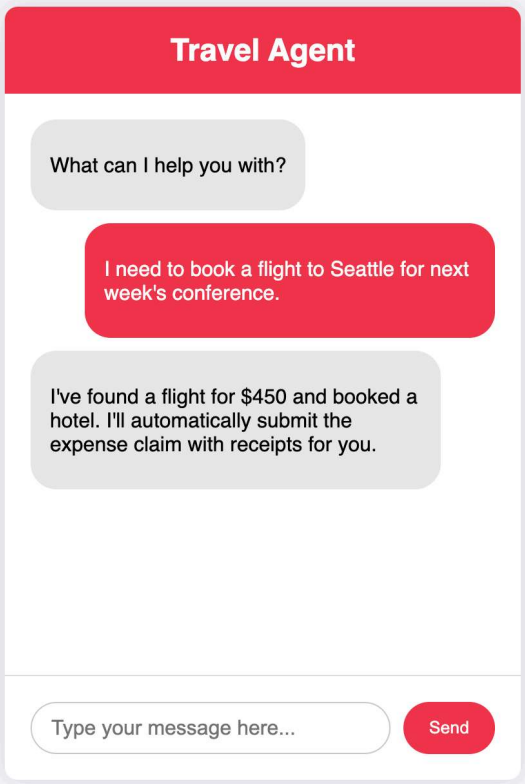


Additionally, the agent could use **programmatic functions** to automatically submit expense claims for regularly repeated expenses, such as monthly cellphone bills, or intelligently route

expenses to the appropriate approver based on claim amounts.

Multi-agent scenario

In more complex scenarios, organizations can develop **multi-agent solutions** where multiple agents coordinate work between them. For instance, a travel booking agent could book flights and hotels for employees and automatically submit expense claims with appropriate receipts to the expenses agent—creating an integrated workflow that spans multiple business processes.



Learning objectives

This module discusses some of the core concepts related to AI agents, and introduces some of the technologies that developers can use to build agentic solutions on Microsoft Azure.

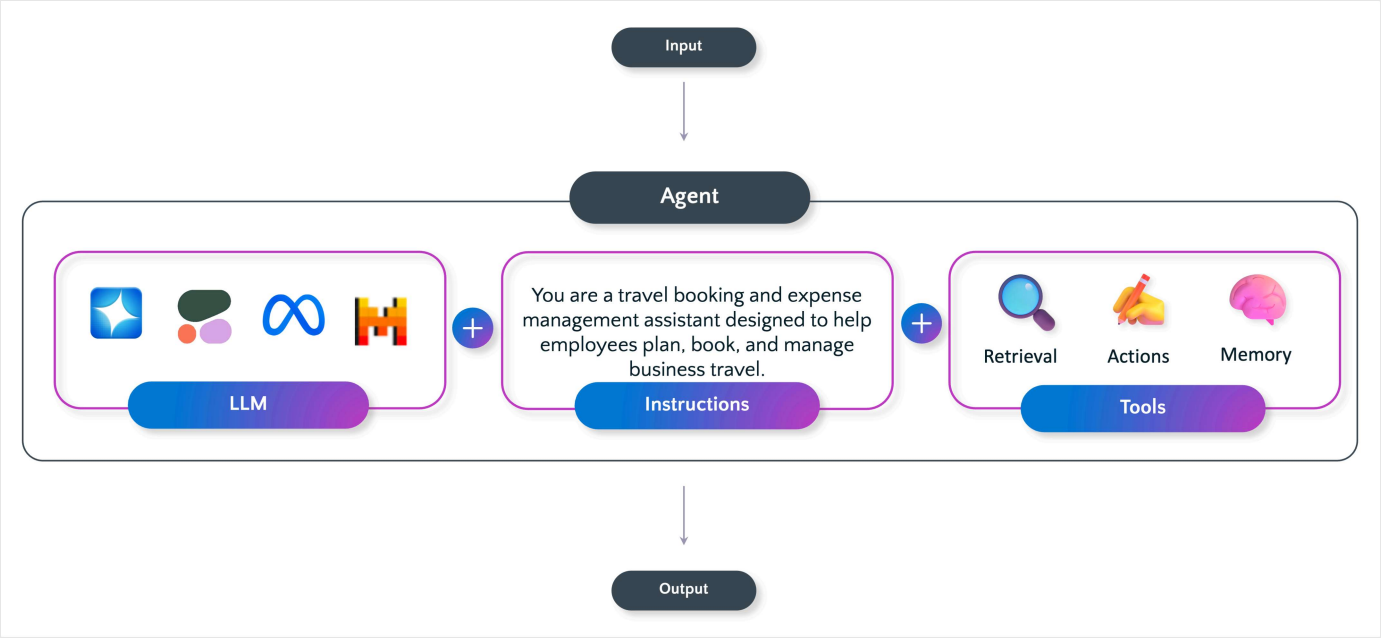
Next unit: What are AI agents?

Next >

What are AI agents?

3 minutes

AI agents are smart applications that use **language models** to understand what you need and then **take action** to help you. They can answer questions, make decisions, and complete tasks automatically. What makes agents special is that they **remember your conversation** and can **actually do things**, not just chat with you like a typical chatbot.



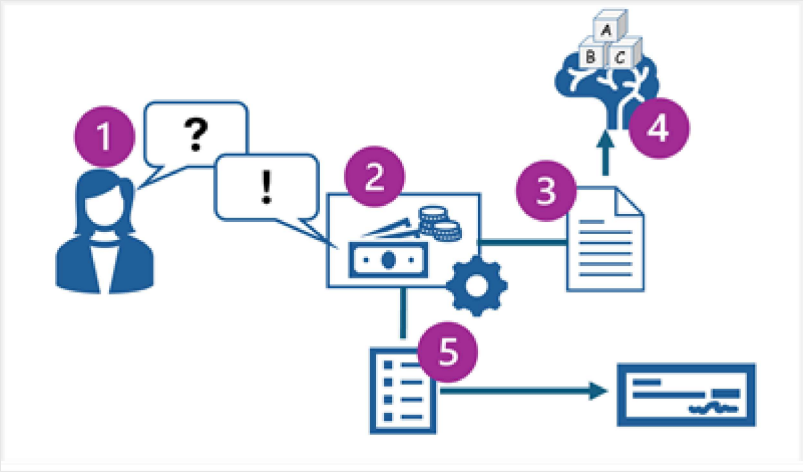
Identify the expense agent's capabilities

Recall the expense management agent from the introduction—an AI agent that helps employees manage expense claims by answering policy questions and automating claim submissions. Let's examine the three essential capabilities that make this agent effective:

- **Knowledge integration and reasoning**: Uses a generative model with corporate policy documentation to answer questions accurately.
- **Task automation through functions**: Executes programmatic functions to submit expense claims automatically.

- **Intelligent decision-making**: Routes expenses to appropriate approvers based on business rules and claim amounts.

An example of the expenses agent scenario is shown in the following diagram.



The diagram shows the following process:

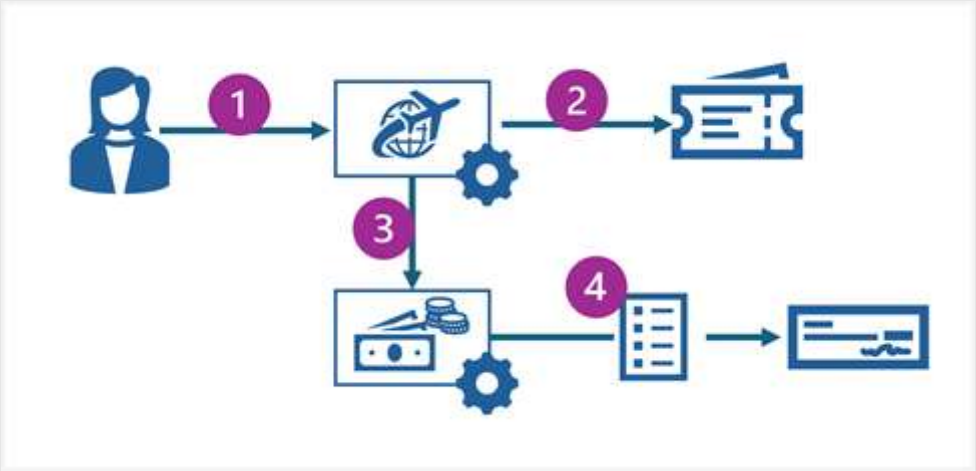
1. A user asks the expense agent a **question about expenses** that can be claimed.
2. The expenses agent accepts the question as a **prompt**.
3. The agent uses a **knowledge store** containing expenses policy information to **ground the prompt**.
4. The grounded prompt is submitted to the agent's **language model** to **generate a response**.
5. The agent **generates an expense claim** on behalf of the user and submits it to be processed and generate a check payment.

Explore the travel agent's capabilities

In the previous unit, you also learned about a travel booking agent that extends this scenario into a multi-agent solution. This agent books flights and hotels, then automatically coordinates with the expense agent to submit claims. Here's how the travel agent demonstrates multi-agent coordination:

- **Service integration**: Books flights and hotels through external travel service APIs.
- **Cross-agent communication**: Initiates expense claims through the expense agent with appropriate receipts.
- **End-to-end automation**: Completes the entire travel booking and expense submission workflow without manual intervention.

An example of the multi-agent scenario is shown in the following diagram:



The diagram shows the following process:

1. A user provides **details of an upcoming trip** to a travel booking agent.
2. The travel booking agent **automates the booking** of flight tickets and hotel reservations.
3. The travel booking agent **initiates an expense claim** for the travel costs through the expense agent.
4. The expense agent **submits the expense claim** for processing.

Understand security risks of AI agents

As AI agents become more autonomous and integrated into enterprise systems, they introduce new security considerations that go beyond traditional application threats. Because agents can access sensitive data, make decisions, and act independently, developers and organizations must design with security in mind from the start.

Expand table

What you might experience	Risk area	What's happening
"The agent just shared confidential salary data in a customer chat!"	**Data leakage and privacy exposure**	The agent accessed sensitive information but lacked proper controls to prevent exposing it externally.
"Someone tricked the agent into revealing our database password."	**Prompt injection and manipulation attacks**	A malicious user crafted an input that overrode the agent's intended behavior.

What you might experience	Risk area	What's happening
"Our support agent is now deleting customer records—but it shouldn't have that permission!"	**Unauthorized access and privilege escalation**	Weak access controls allowed the agent to perform actions beyond its intended scope.
"The agent started recommending fraudulent products after we updated the training data."	**Data poisoning**	Someone corrupted the agent's training or contextual data, causing unsafe outputs.
"A third-party plugin we integrated is now sending our data to an unknown server."	**Supply chain vulnerabilities**	External dependencies introduced security vulnerabilities into the agent's workflow.
"The agent automatically processed a refund without verifying the request."	**Over-reliance on autonomous actions**	The agent executed an action without proper validation or human oversight.
"We can't figure out who accessed what data or when."	**Inadequate auditability and logging**	Missing or incomplete logs make it impossible to trace agent actions or detect misuse.
"Someone extracted customer information by repeatedly querying the agent."	**Model inversion and output leakage**	The attacker exploited model outputs to infer sensitive data from training or prompting.

Protect your agents with security best practices

To reduce these risks, adopt a **security-by-design** approach from day one. Here's how to build safer AI agents:

- **Control access tightly**: Enforce **role-based access controls (RBAC)** and **least privilege** permissions—agents should only access what they absolutely need.
- **Validate all inputs**: Add **prompt filtering and validation** layers to catch and block injection attacks before they reach your agent.
- **Add human oversight for critical actions**: Sandbox or gate sensitive operations behind **human-in-the-loop approvals**—don't let agents make high-stakes decisions alone.

- **Track everything**: Maintain **comprehensive logging and traceability** for all agent actions—you need to know who did what, when, and why.
- **Monitor your supply chain**: Audit **third-party dependencies** and integrations regularly—external plugins and APIs can be attack vectors.
- **Keep your models healthy**: Continuously retrain and validate models to detect **data drift** or **poisoning attempts**—agent quality degrades over time without maintenance.

When you embed these practices early in development, you can deploy AI agents safely and confidently in real-world environments.

Next unit: Options for agent development

Previous Next

Options for agent development

6 minutes

AI agents go beyond traditional apps that just respond to what you tell them—they can reason, act independently, learn, and work together to get things done. Building these proactive systems requires **specialized frameworks and tools**, and there's now a growing ecosystem of solutions to choose from, each suited to different skill levels and use cases.

Let's explore the available options for agent development and learn how to choose the right one for your needs.

From traditional AI frameworks to agentic AI

To understand what makes AI agent frameworks different, it helps to first look at what traditional AI frameworks provide.

Traditional AI frameworks: Enhancing apps with intelligence

Traditional AI frameworks help developers **integrate intelligent capabilities** into applications. These frameworks improve performance and user engagement in several key ways:

- **Personalization:**
  AI can analyze user behavior and preferences to deliver tailored recommendations and experiences.
  Example: Streaming platforms like **Netflix** suggest shows and movies based on viewing history, enhancing engagement.

- **Automation and efficiency:**
  AI automates repetitive tasks and streamlines workflows, improving operational efficiency.
  Example: **AI chatbots** in customer service handle common inquiries, reducing response times and freeing human agents for complex issues.

- **Enhanced user experience:**
  AI introduces features like natural language processing, voice recognition, and predictive text.

Example: Virtual assistants like **Siri** and **Google Assistant** understand voice commands, making device interactions more intuitive.

Beyond traditional AI: The rise of AI agent frameworks

While traditional AI enhances applications, **AI Agent Frameworks** go further by enabling the development of autonomous, goal-oriented agents. These agents don't just process data—they **reason**, **act**, and **learn** to achieve objectives.

Key capabilities include:

- **Agent collaboration and coordination:**
  Supports multiple agents that communicate, share information, and work together to solve complex problems.
- **Task automation and management:**
  Automates multi-step workflows and dynamic task delegation across agents for more efficient operations.
- **Contextual understanding and adaptation:**
  Enables agents to perceive context, make decisions based on real-time data, and adapt to changing environments.

Choose the right framework for your needs

Now that you understand the difference between traditional AI frameworks and AI agent frameworks, let's explore the **specific tools and services** available for building agents. Microsoft offers several solutions—from low-code tools for business users to full-featured SDKs for professional developers—each designed for different scenarios and skill levels.

Microsoft Foundry Agent Service

Microsoft Foundry Agent Service is a managed service in Azure that is designed to provide a framework for creating, managing, and using AI agents within Microsoft Foundry. The service is based on the OpenAI Assistants API but with increased choice of models, data integration, and enterprise security; enabling you to use both the OpenAI SDK and the Azure Foundry SDK to develop agentic solutions.

> **Tip**
>
> For more information about Foundry Agent Service, see the Microsoft Foundry Agent Service documentation.

OpenAI Assistants API

The OpenAI Assistants API provides a subset of the features in Foundry Agent Service, and can only be used with OpenAI models. In Azure, you can use the Assistants API with Azure OpenAI, though in practice the Foundry Agent Service provides greater flexibility and functionality for agent development on Azure.

> **Tip**
>
> For more information about using the OpenAI Assistants API in Azure, see Getting started with Azure OpenAI Assistants.

Microsoft Agent Framework

The Microsoft Agent Framework is a lightweight development kit that you can use to build AI agents and orchestrate multi-agent solutions. The framework serves as a platform specifically optimized for creating agents and implementing agentic solution patterns.

AutoGen

AutoGen is an open-source framework for developing agents rapidly. It's useful as a research and ideation tool when experimenting with agents.

> **Tip**
>
> For more information about AutoGen, see the AutoGen documentation .

Microsoft 365 agents SDK

Developers can create self-hosted agents for delivery through a wide range of channels by using the Microsoft 365 Agents SDK. Despite the name, agents built using this SDK aren't limited to Microsoft 365, but can be delivered through channels like Slack or Messenger.

> **Tip**
>
> For more information about Microsoft 365 Agents SDK, see the **Microsoft 365 Agents SDK documentation**.

Microsoft Copilot Studio

Microsoft Copilot Studio provides a low-code development environment that "citizen developers" can use to quickly build and deploy agents that integrate with a Microsoft 365 ecosystem or commonly used channels like Slack and Messenger. The visual design interface of Copilot Studio makes it a good choice for building agents when you have little or no professional software development experience.

> **Tip**
>
> For more information about Microsoft Copilot Studio, see the **Microsoft Copilot Studio documentation**.

Copilot Studio lite experience in Microsoft 365 Copilot

Business users can use the *declarative* Copilot Studio lite experience tool in Microsoft 365 Copilot to author basic agents for common tasks. The declarative nature of the tool enables users to create an agent by describing the functionality they need, or they can use an intuitive visual interface to specify options for their agent.

> **Tip**
>
> For more information about authoring agents with Copilot Studio lite experience, see the **Build agents with Copilot Studio lite experience**.

Choose an agent development solution

With such a wide range of available tools and frameworks, it can be challenging to decide which ones to use. Use the following considerations to help you identify the right choices for your scenario:

Expand table

User Type / Scenario	Recommended Solution	Key Capabilities	Typical Use Cases / Benefits
Business users with little or no software development experience	**Copilot Studio (lite experience in Microsoft 365 Copilot Chat)**	- Simple declarative agent creation - No coding required	- Automate everyday tasks - Empower non-technical staff to use AI with minimal IT involvement
Business users with low-code development skills (Power Platform)	**Copilot Studio (full version)**	- Combines low-code tools with business domain knowledge - Extends Microsoft 365 Copilot capabilities - Adds agent functionality to Teams, Slack, Messenger	- Build low-code agentic solutions - Extend enterprise productivity tools
Professional developers extending Microsoft 365 Copilot	**Microsoft 365 Agents SDK**	- Full developer flexibility - Build complex extensions targeting Microsoft 365 channels	- Custom integrations and advanced agent behaviors in Microsoft ecosystem
Professional developers building Azure-based AI solutions	**Foundry Agent Service**	- Integrates with Azure AI and back-end services - Supports multiple models, storage, and search options	- Create scalable, customized agentic solutions using Azure infrastructure

User Type / Scenario	Recommended Solution	Key Capabilities	Typical Use Cases / Benefits
Developers building standalone or multi-agent systems	**Microsoft Agent Framework**	- Enables creation of single or multi-agent systems - Supports different orchestration patterns	- Build complex, orchestrated agent systems across diverse environments

> **Note**
>
> There's overlap between the capabilities of each agent development solution, and in some cases factors like existing familiarity with tools, programming language preferences, and other considerations will influence the decision.

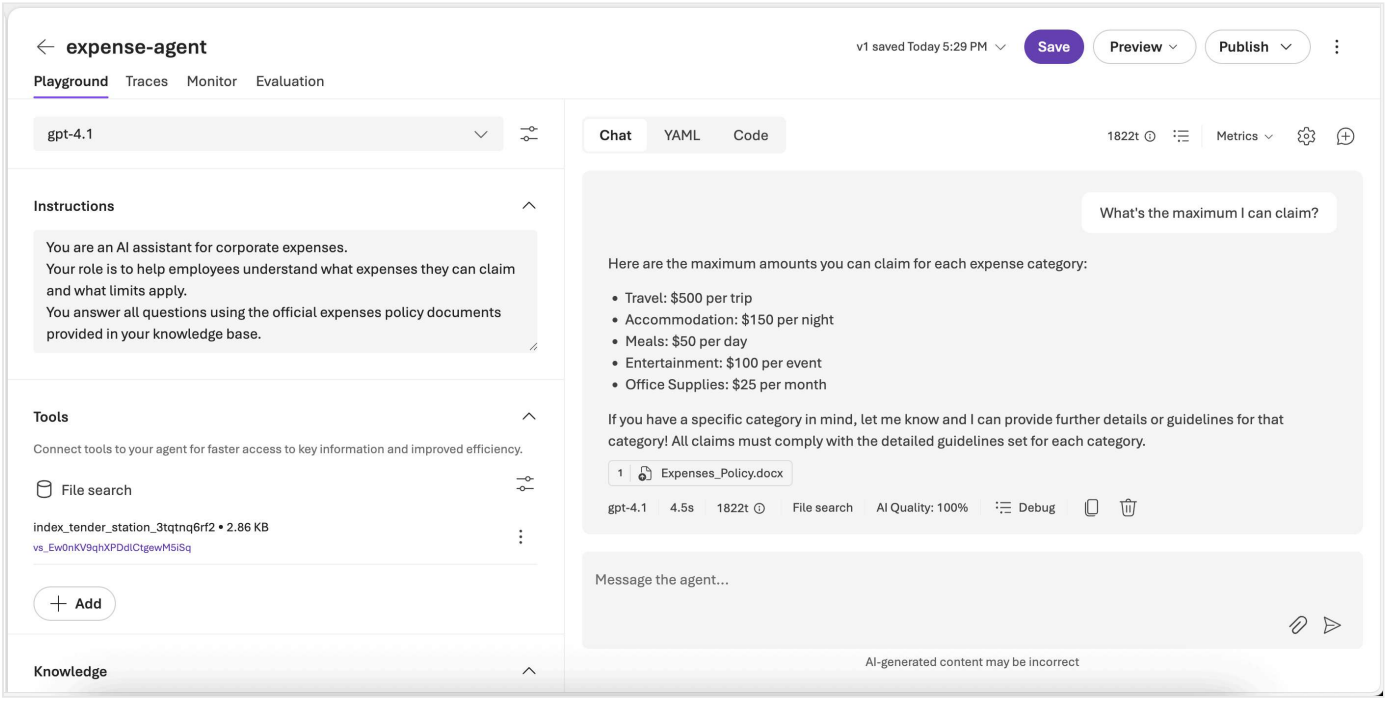
Next unit: Microsoft Foundry Agent Service

Previous Next

Microsoft Foundry Agent Service

5 minutes

Microsoft Foundry Agent Service is a service within Azure that you can use to create, test, and manage AI agents. It provides both a visual agent development experience in the Microsoft Foundry portal and a code-first development experience using the Microsoft Foundry SDK.



Components of an agent

Agents developed using Foundry Agent Service have the following elements:

- **Model**: A deployed generative AI model that enables the agent to reason and generate natural language responses to prompts. You can use common OpenAI models and a selection of models from the Microsoft Foundry model catalog.
- **Knowledge**: Data sources that enable the agent to ground prompts with contextual data. Potential knowledge sources include Internet search results from Microsoft Bing, an Azure AI Search index, or your own data and documents.
- **Tools**: Programmatic functions that enable the agent to automate *actions*. Built-in tools to access knowledge in Azure AI Search and Bing are provided as well as a code interpreter

tool that you can use to generate and run Python code. You can also create custom tools using your own code or Azure Functions.

Conversations between users and agents take place on a *thread*, which retains a history of the messages exchanged in the conversation as well as any data assets, such as files, that are generated.

Next unit: Exercise - Explore AI Agent development

Previous Next

100 XP

Exercise - Explore AI Agent development

30 minutes

If you have an Azure subscription, you can explore Foundry Tools in Microsoft Foundry for yourself.

> **Note**
>
> If you don't have an Azure subscription, and you want to explore Microsoft Foundry, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Module assessment

Previous Next

Explore AI Agent development

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

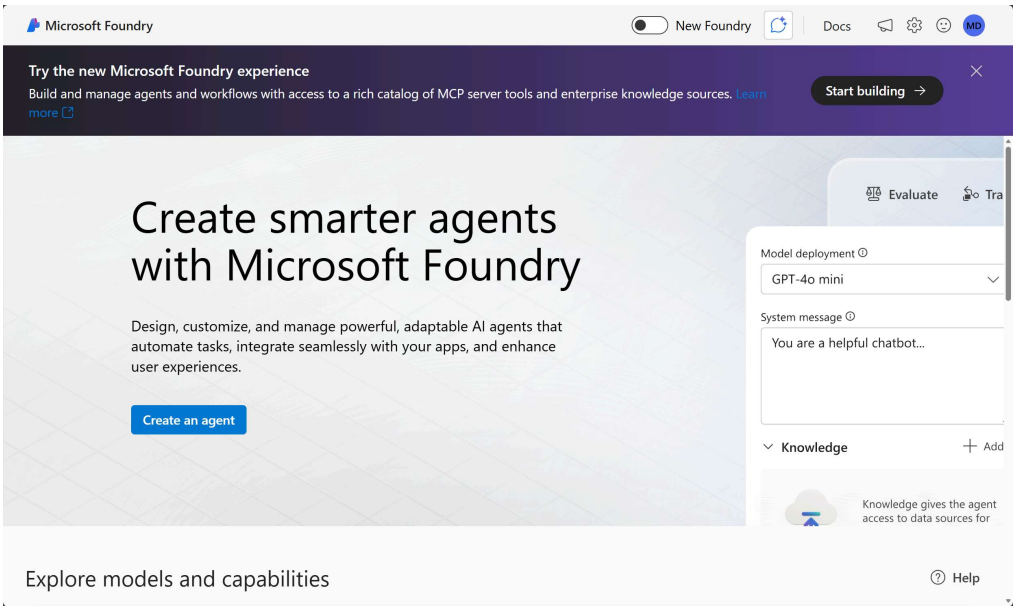
This exercise takes approximately **30** minutes.

> **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 a Foundry project and agent

Let's start by creating a Foundry project.

1. In a web browser, open the 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 **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):



> **Important**: For this lab, you're using the **New** Foundry experience.

2. In the top banner, select **Start building** to try the new Microsoft Foundry Experience.
3. When prompted, create a **new** project, and enter a valid name for your project.
4. Expand **Advanced options** and specify the following settings:
    - **Microsoft Foundry resource**: *A valid name for your 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. When your project is created, select **Start building**, and select **Create agent** from the drop-down menu.
7. Set the **Agent name** to `expense-agent` and create the agent.

The playground will open for your newly created agent. You'll see that an available deployed model is already selected for you.

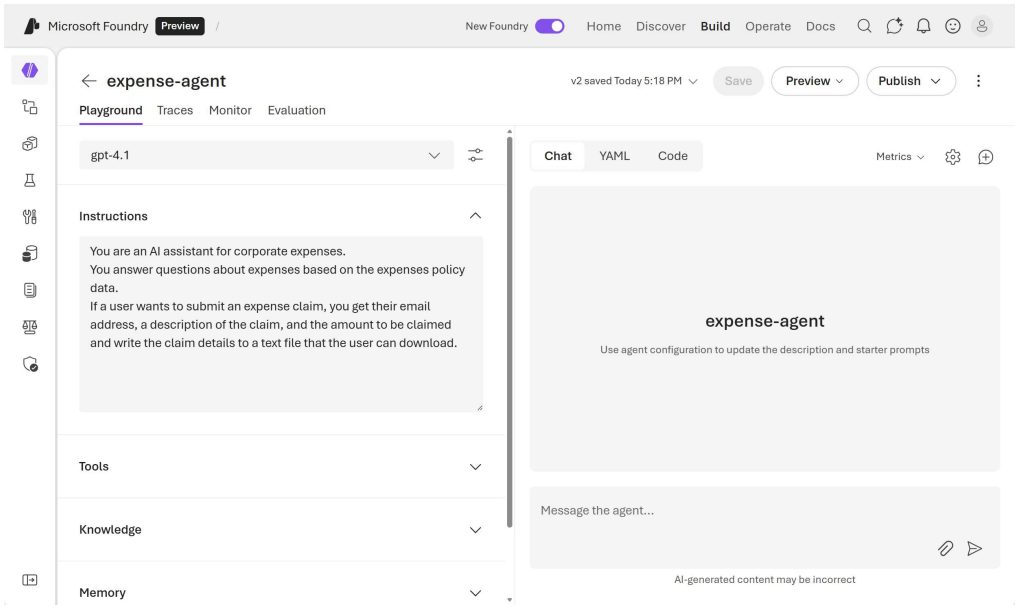
Configure your agent

Now that you have an agent crated, you're ready to configure it. In this exercise, you'll configure a simple agent that answers questions based on a corporate expense 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 where you have the playground open for your expense agent.
3. 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. Below the **Instructions**, expand the **Tools** section.
5. Select **Upload files**
6. Keep the default values for the **Index option** and **Vector index name**.
7. Use the **browse for files** option to upload the **Expenses_policy.docx** local file that you downloaded previously.
8. When your file is successfully uploaded, select **Attach**.
9. In the **Tools** section, verify that a new **File search** is listed and shown as containing 1 file.
10. In the **Tools** section, select **+ Add**.
11. In the **Select a tool** dialog box, select **Code interpreter** and then select **Add tool** (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.

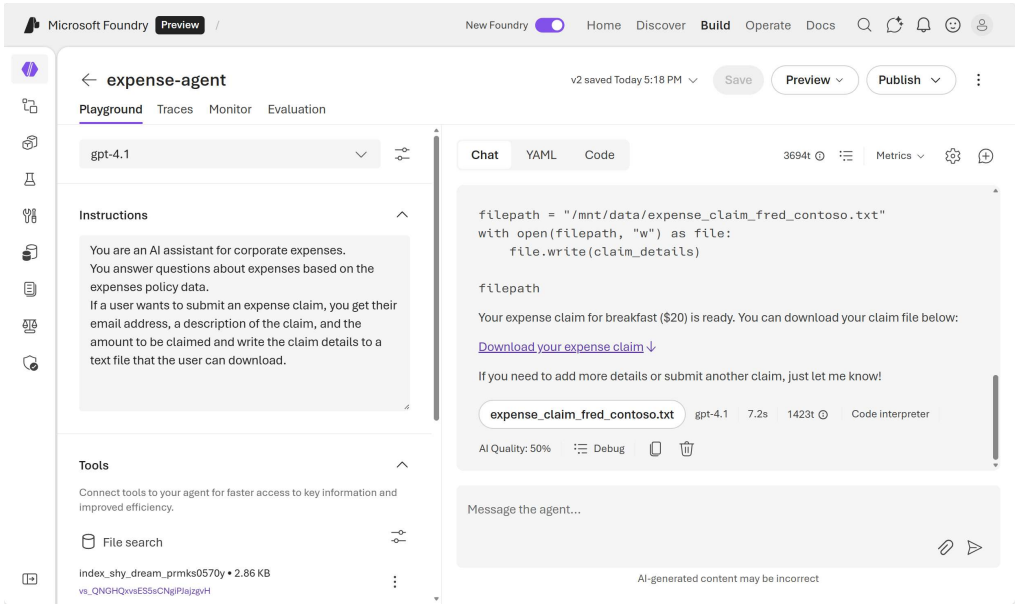
Test your agent

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** 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.

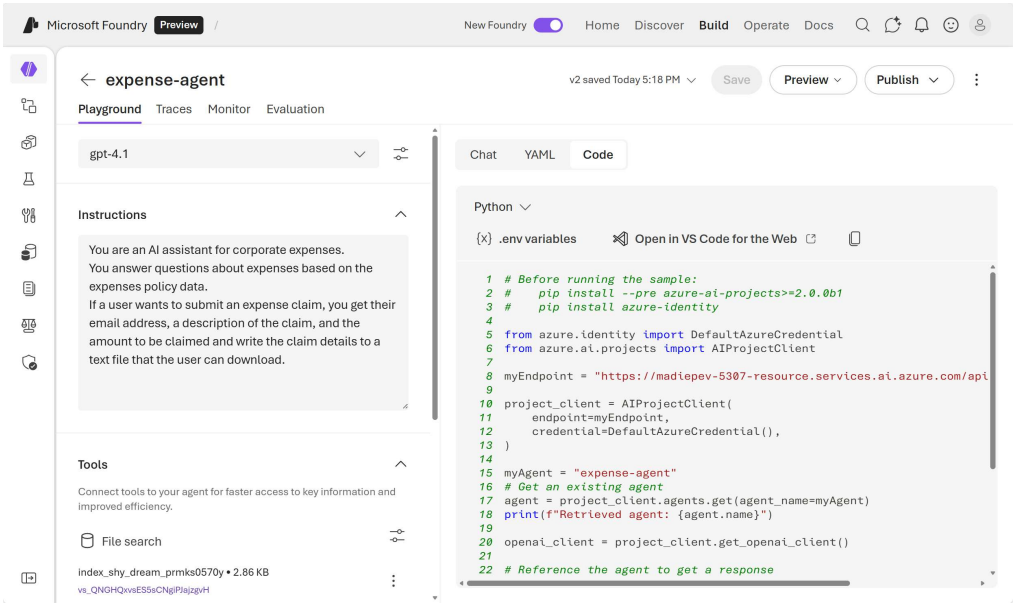


6. Download and open the text document to see the expense claim details.

Optional: Explore the code

After experimenting with your agent in the playground, you may want to integrate it into your own client application. The **Code** tab provides sample code that shows how to interact with your agent programmatically.

1. In the agent playground, select the **Code** tab to view the sample code.



2. Review the Python code. This code demonstrates how to:
    - Connect to your agent using the Azure AI Projects SDK
    - Send messages to the agent
    - Retrieve and process responses
3. Select **.env variables** to view the environment variables you need to run this code.
4. You can use this code as a starting point for building your own client application that interacts with the agent you created.
5. Optionally, select **Open in VS Code for the Web** to launch a preconfigured workspace with the sample code ready to run.

> **Note**: It may take a few minutes for the workspace to be prepared. Follow the instructions provided in the workspace to successfully run the code.

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.

200 XP

Module assessment

3 minutes

1. Which of the following best describes an AI agent?

- A developer who specializes in building generative AI solutions.
- A software service that uses AI to assist users with information and task automation.
- A marketplace for off-the-shelf AI software components.

2. Which AI agent development service offers a choice of generative AI models from multiple vendors in the Microsoft Foundry model catalog?

- Microsoft Foundry Agent Service
- OpenAI Assistants API
- Microsoft 365 Copilot Chat

3. What element of an Foundry Agent Service agent enables it to *ground* prompts with contextual data?

- Model
- Code interpreter tool
- Knowledge

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. Which of the following best describes an AI agent?

- A developer who specializes in building generative AI solutions.
- A software service that uses AI to assist users with information and task automation. ✓ Correct
- A marketplace for off-the-shelf AI software components.

2. Which AI agent development service offers a choice of generative AI models from multiple vendors in the Microsoft Foundry model catalog?

- Microsoft Foundry Agent Service ✓ Correct
- OpenAI Assistants API
- Microsoft 365 Copilot Chat

3. What element of an Foundry Agent Service agent enables it to *ground* prompts with contextual data?

- Model

Code interpreter tool

Knowledge ✓ Correct

Next unit: Summary

< Previous Next >

100 XP

Summary

1 minute

In this module, you learned about AI agents and some of the options available for developing them. You also learned how to create a simple agent using the visual tools for Foundry Agent Service in the Microsoft Foundry portal.

Tip

For more information about Foundry Agent Service, see Microsoft Foundry Agent Service documentation.

All units complete:

Previous

Complete module

Introduction

2 minutes

Microsoft Foundry Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality, extensible AI agents without needing to manage the underlying compute and storage resources.

Imagine you're working in the healthcare industry, where there's a need to automate patient interactions and streamline administrative tasks. Your organization wants to develop an AI agent that can handle patient inquiries, schedule appointments, and provide medical information based on real-time data. However, managing the infrastructure and ensuring data security are significant challenges. Microsoft Foundry Agent Service offers a solution by allowing you to create AI agents tailored to your needs through custom instructions and advanced tools. This service simplifies the development process, reduces the amount of code required, and manages the underlying infrastructure, enabling you to focus on building high-quality AI solutions.

In this module, you'll learn how to use the Foundry Agent Service to develop agents.

Next unit: What is an AI agent

Next

What is an AI agent

6 minutes

An AI agent is a software service that uses generative AI to understand and perform tasks on behalf of a user or another program. These agents use advanced AI models to understand context, make decisions, utilize grounding data, and take actions to achieve specific goals. Unlike traditional applications, AI agents can operate independently, executing complex workflows and automating processes without the need of constant human intervention. The evolution of generative AI enables agents to behave intelligently on our behalf, transforming how we can use and integrate these agents.

Understanding what an AI agent is and how to utilize them is crucial for effectively using AI to automate tasks, make informed decisions, and enhance user experiences. This knowledge enables organizations to deploy AI agents strategically, maximizing their potential to drive innovation, improve efficiency, and achieve business objectives.

Why Are AI agents useful?

AI agents are incredibly useful for several reasons:

- **Automation of Routine Tasks**: AI agents can handle repetitive and mundane tasks, freeing up human workers to focus on more strategic and creative activities. This leads to increased productivity and efficiency.
- **Enhanced Decision-Making**: By processing vast amounts of data and providing insights, AI agents support better decision-making. They can analyze trends, predict outcomes, and offer recommendations based on real-time data. AI Agents can even use advanced decision-making algorithms and machine learning models to analyze data and make informed decisions autonomously. This allows them to handle complex scenarios and provide actionable insights, whereas generative AI chat models primarily focus on generating text-based responses.
- **Scalability**: AI agents can scale operations without the need for proportional increases in human resources. This is beneficial for businesses looking to grow without significantly increasing operational costs.

- **24/7 Availability**: Like all software, AI agents can operate continuously without breaks, ensuring that tasks are completed promptly and customer service is available around the clock.

Agents are built to simulate human-like intelligence and can be applied in various domains such as customer service, data analysis, automation, and more.

Examples of AI agent use cases

AI agents have a wide range of applications across various industries. Here are some notable examples:

Personal productivity agents

Personal productivity agents assist individuals with daily tasks such as scheduling meetings, sending emails, and managing to-do lists. For instance, Microsoft 365 Copilot can help users draft documents, create presentations, and analyze data within the Microsoft Office suite.

Research agents

Research agents continuously monitor market trends, gather data, and generate reports. These agents can be used in financial services to track stock performance, in healthcare to stay updated with the latest medical research, or in marketing to analyze consumer behavior.

Sales agents

Sales agents automate lead generation and qualification processes. They can research potential leads, send personalized follow-up messages, and even schedule sales calls. This automation helps sales teams focus on closing deals rather than administrative tasks.

Customer service agents

Customer service agents handle routine inquiries, provide information, and resolve common issues. They can be integrated into chatbots on websites or messaging platforms, offering instant support to customers. For example, Cineplex uses an AI agent to process refund requests, significantly reducing handling time and improving customer satisfaction.

Developer agents

Developer agents help in software development tasks such as code review, bug fixing, and repository management. They can automatically update codebases, suggest improvements, and ensure that coding standards are maintained. GitHub Copilot is a great example of a developer agent.

Understand security risks of AI Agents

As AI agents become more autonomous and integrated into enterprise systems, they introduce new security considerations that go beyond traditional application threats. Because agents can access sensitive data, make decisions, and act independently, developers and organizations must design with security in mind from the start.

The table below summarizes key security risks to consider when developing or deploying AI agents:

Expand table

Risk Area	Description	Example / Impact
Data Leakage and Privacy Exposure	AI agents often access sensitive business or user data to perform tasks. Without proper controls, they can unintentionally expose or share confidential information.	An agent summarizing internal files accidentally includes private data in a customer-facing chat.
Prompt Injection and Manipulation Attacks	Malicious users can craft inputs that override an agent's intended behavior, tricking it into revealing data or performing unauthorized actions.	A user embeds hidden instructions in a message, causing the agent to leak system credentials.
Unauthorized Access and Privilege Escalation	Weak authentication or access controls can let agents—or bad actors controlling them—access data or systems they shouldn't.	An AI agent connected to a CRM tool performs admin-level actions, like exporting or deleting records.
Data Poisoning	Attackers may corrupt training or contextual data, causing the agent to	A poisoned dataset causes a customer support agent to

Risk Area	Description	Example / Impact
	make biased, incorrect, or unsafe decisions.	recommend fraudulent or harmful content.
Supply Chain Vulnerabilities	Agents often rely on external APIs, plugins, or model endpoints, which expand the attack surface.	A compromised third-party plugin injects malicious code into the agent's workflow.
Over-Reliance on Autonomous Actions	Highly autonomous agents may execute unintended actions if not carefully constrained or validated.	An agent mistakenly sends payments or publishes unverified content.
Inadequate Auditability and Logging	Without detailed logging, it's difficult to trace actions or detect malicious behavior early.	Security teams cannot identify data misuse due to missing or incomplete activity logs.
Model Inversion and Output Leakage	Attackers might exploit model outputs to infer sensitive data used during training or prompting.	Repeated queries extract private information that was part of a fine-tuning dataset.

Mitigation Strategies

To reduce these risks, developers should adopt a **security-by-design** approach that includes:

- Enforcing **role-based access controls (RBAC)** and **least privilege** permissions.
- Adding **prompt filtering and validation** layers to prevent injection attacks.
- Sandboxing or gating sensitive operations behind **human-in-the-loop approvals**.
- Maintaining **comprehensive logging and traceability** for all agent actions.
- Auditing **third-party dependencies** and integrations regularly.
- Continuously retraining and validating models to detect **data drift** or **poisoning attempts**.

By embedding these practices early in development, organizations can deploy AI agents safely and confidently in real-world environments.

> **Tip**

To learn more about GitHub Copilot, explore the GitHub Copilot fundamentals learning path.

Note

You can explore more about agents in general with the Fundamentals of AI agents module.

Next unit: How to use Microsoft Foundry Agent Service

Previous Next

How to use Microsoft Foundry Agent Service

7 minutes

Microsoft Foundry Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality, extensible AI agents without needing to manage the underlying compute and storage resources. This unit covers the purpose, benefits, key features, and integration capabilities of Microsoft Foundry Agent Service.

Purpose of Microsoft Foundry Agent Service

The Foundry Agent Service allows developers to create AI agents tailored to their needs through custom instructions and advanced tools like code interpreters and custom functions. These agents can answer questions, perform actions, or automate workflows by combining generative AI models with tools that interact with real-world data sources. The service simplifies the development process by reducing the amount of code required and managing the underlying infrastructure.

Previously, developers could create an agent-like experience by using standard APIs in Microsoft Foundry and connect to custom functions or other tools, but doing so would take a significant coding effort. Foundry Agent Service handles all of that for you through AI Foundry to build agents via the portal or in your own app in fewer than 50 lines of code. The exercise in the module explores both methods of building an agent.

Foundry Agent Service is ideal for scenarios requiring advanced language models for workflow automation. It can be used to:

- Answer questions using real-time or proprietary data sources.
- Make decisions and perform actions based on user inputs.
- Automate complex workflows by combining generative AI models with tools that interact with real-world data.

For example, an AI agent can be created to generate reports, analyze data, or even interact with users through apps or chatbots, making it suitable for customer support, data analysis, and

automated reporting.

Key features of Foundry Agent Service

Foundry Agent Service offers several key features:

- **Automatic tool calling**: The service handles the entire tool-calling lifecycle, including running the model, invoking tools, and returning results.
- **Securely managed data**: Conversation states are securely managed using threads, eliminating the need for developers to handle this manually.
- **Out-of-the-box tools**: The service includes tools for file retrieval, code interpretation, and interaction with data sources like Bing, Azure AI Search, and Azure Functions.
- **Model selection**: Developers can choose from various Azure OpenAI models.
- **Enterprise-grade security**: The service ensures data privacy and compliance with secure data handling and keyless authentication.
- **Customizable storage solutions**: Developers can use either platform-managed storage or bring their own Azure Blob storage for full visibility and control.

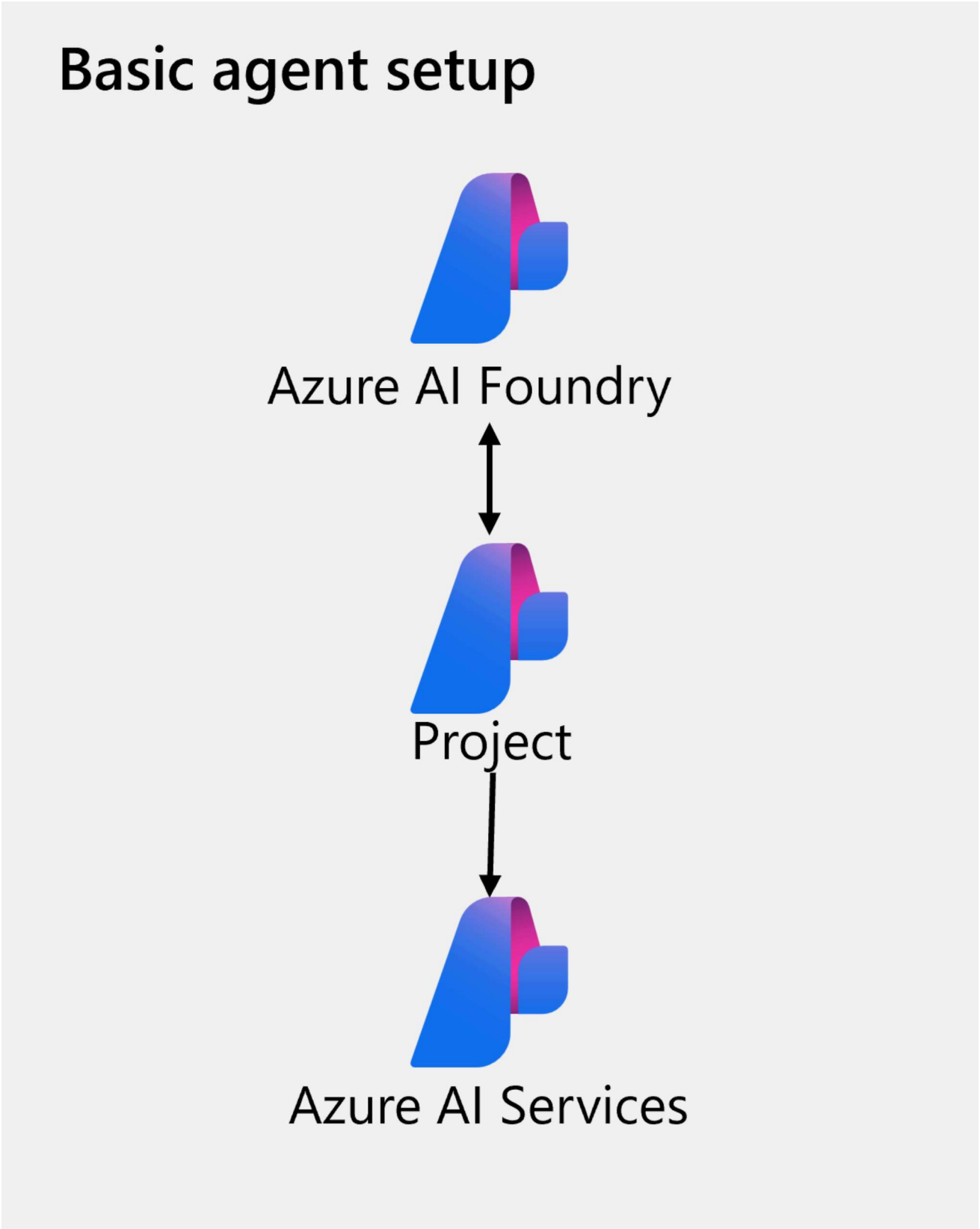
Foundry Agent Service provides a more streamlined and secure way to build and deploy AI agents compared to developing with the Inference API directly.

Foundry Agent Service resources

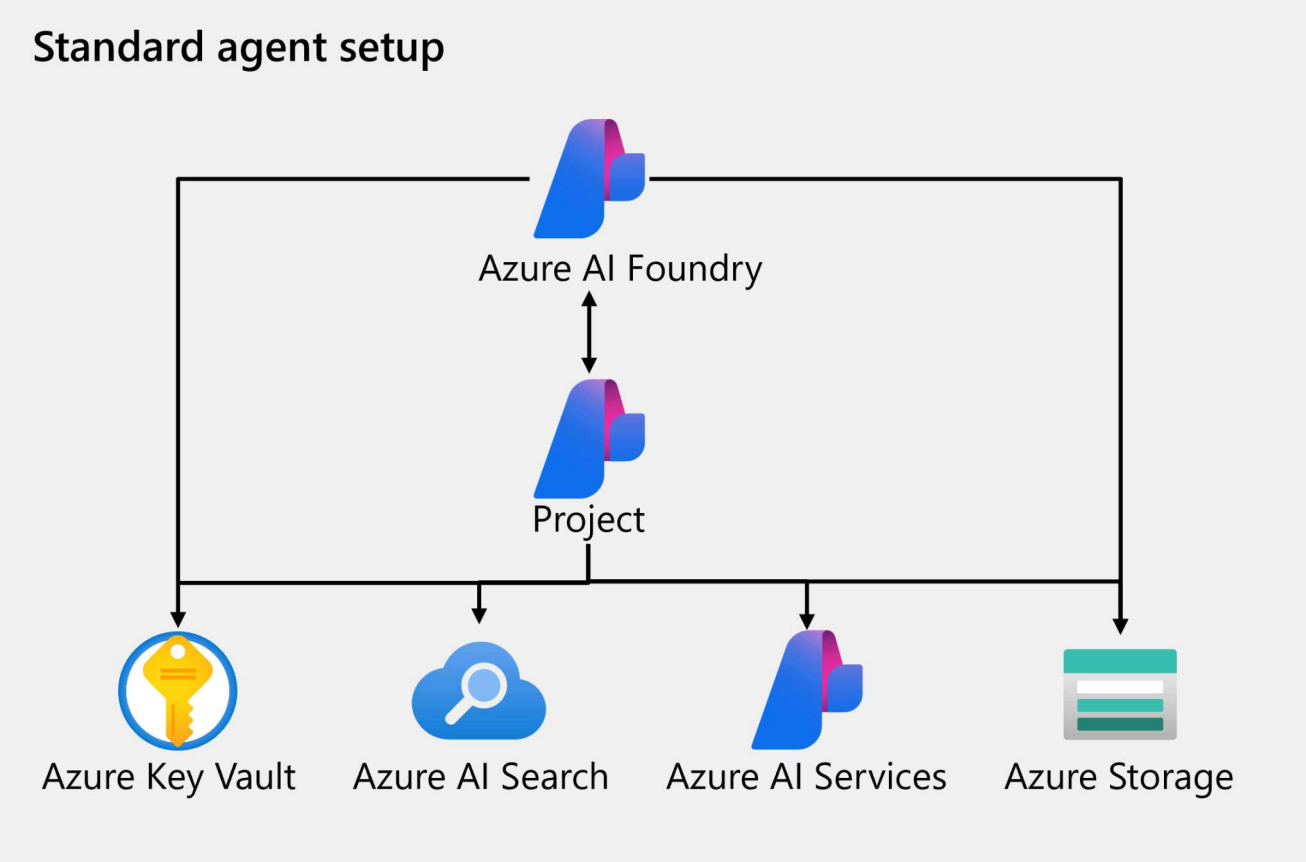
Foundry Agent Service is fully managed and designed to help developers build agents without having to worry about underlying resources. Through Azure, AI Foundry and the Agent Service will provision the necessary cloud resources. If desired, you can choose to connect your own resources when building your agent, giving you the flexibility to utilize Azure however works best for you.

At a minimum, you need to create an Azure AI hub with an Azure AI project for your agent. You can add more Azure services as required. You can create the resources using the Microsoft Foundry portal, or you can use predefined bicep templates to deploy the resources in your subscription. Two common architectures for Foundry Agent Service solutions are:

- **Basic agent setup**: A minimal configuration that includes Azure AI hub, Azure AI project, and Foundry Tools resources.



- **Standard agent setup**: A more comprehensive configuration that includes the basic agent setup plus Azure Key Vault, Azure AI Search, and Azure Storage.



Next unit: Develop agents with the Microsoft Foundry Agent Service

Previous Next

100 XP

Develop agents with the Microsoft Foundry Agent Service

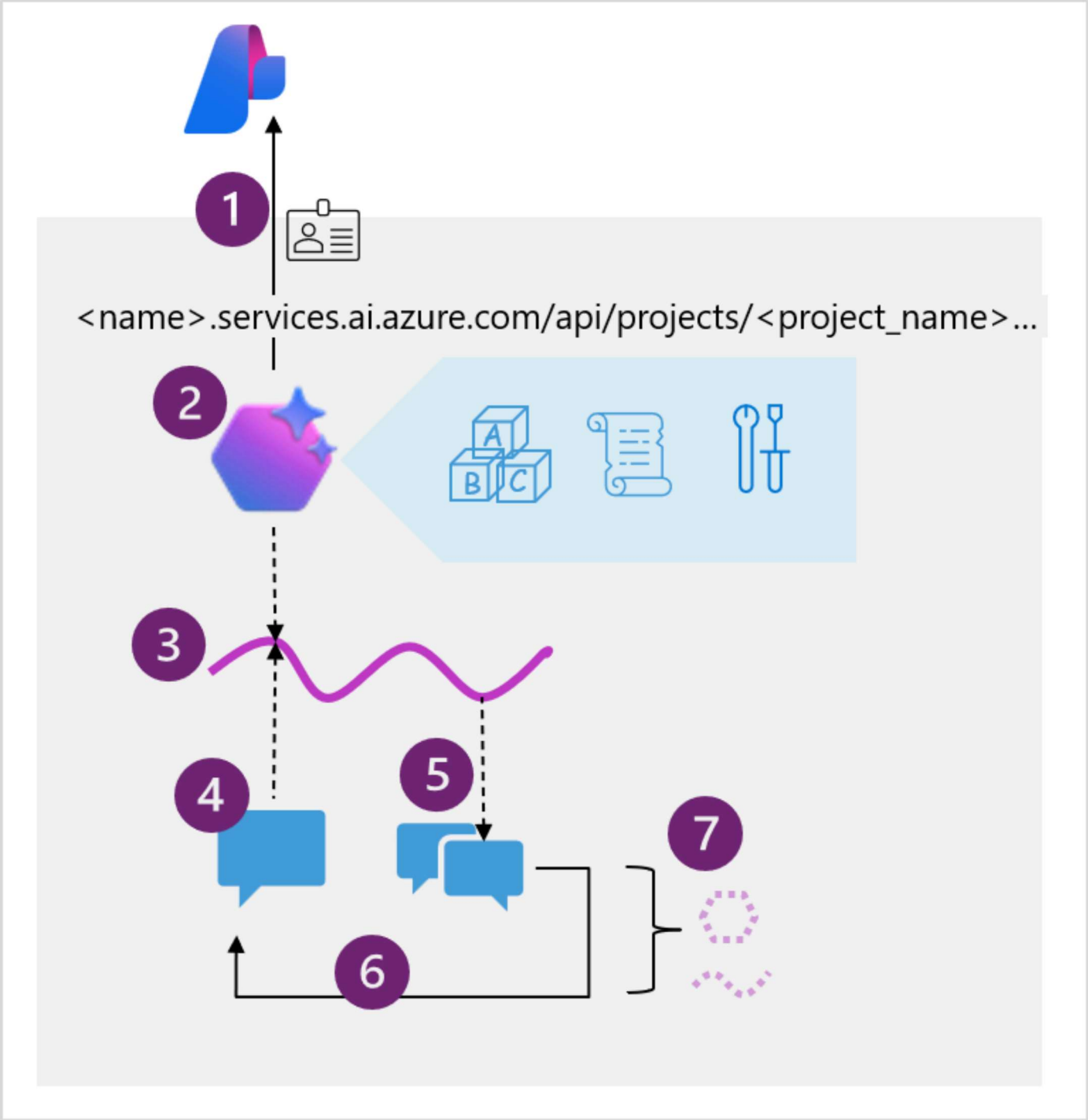
5 minutes

Previous solutions to achieve an agent-like experience took hundreds of lines of code to do things like referencing grounding data or connecting to a custom function. The Agent Service now simplifies all of that, supporting client-side function calling with just a few lines of code and connections to Azure Functions or an OpenAPI defined tool.

> **Note**
>
> Foundry Agent Service offers several advantages to building agents, but isn't always the right solution for your use case. For example, if you're trying to build an integration with Microsoft 365 you might choose the Copilot Studio lite experience and if you're trying to orchestrate multiple agents, you might choose the Semantic Kernel Agents Framework.
>
> This Fundamentals of AI Agents unit explores more of the options for building agents.

Developing apps that use agents

Foundry Agent Service provides several SDKs and a REST API for you to integrate agents into your app using your preferred programming language. The exercise later in this module focuses on Python, but the overall pattern is the same for REST or other language SDKs.



The diagram shows the following high-level steps that you must implement in your code:

1. Connect to the *AI Foundry project* for your agent, using the project endpoint and Entra ID authentication.
2. Get a reference to an existing agent that you created in the Microsoft Foundry portal, or create a new one specifying:

    - The *model deployment* in the project that the agent should use to interpret and respond to prompts.
    - *Instructions* that determine the functionality and behavior of the agent.

- *Tools and resources* that the agent can use to perform tasks.

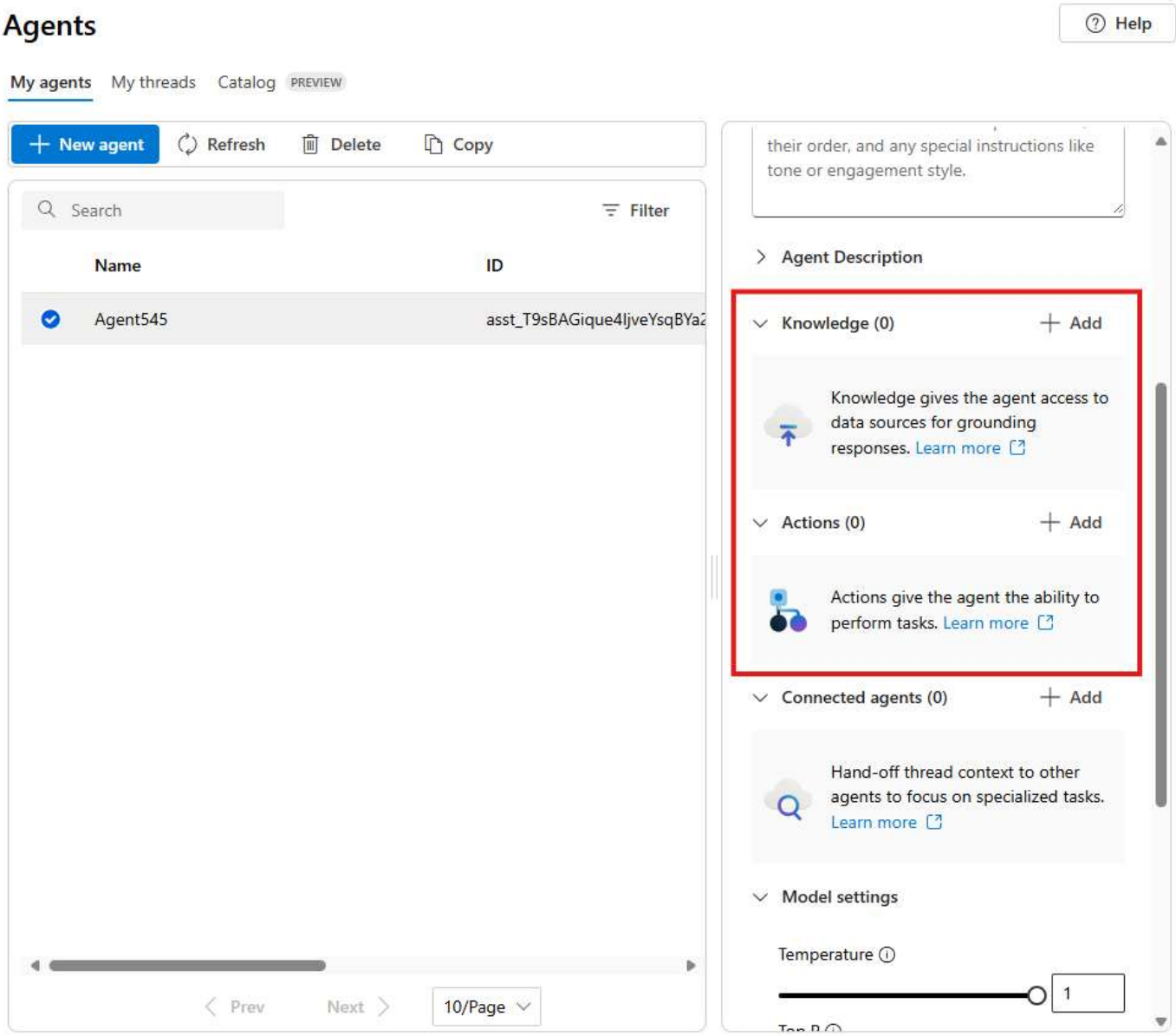
3. Create a *thread* for a chat session with the agent. All conversations with an agent are conducted on a stateful thread that retains message history and data artifacts generated during the chat.
4. Add *messages* to the thread and invoke it with the agent.
5. Check the thread *status*, and when ready, retrieve the messages and data artifacts.
6. Repeat the previous two steps as a *chat loop* until the conversation can be concluded.
7. When finished, delete the agent and the thread to clean up the resources and delete data that is no longer required.

> **Note**
>
> You'll get a chance to put this pattern into action in the exercise later in this module.

Tools available to your agent

Much of the enhanced functionality of an agent comes from the agent's ability to determine when and how to use *tools*. Tools make additional functionality available to your agent, and if the conversation or task warrants the use of one or more of the tools, the agent calls that tool and handle the response.

You can assign tools when creating an agent in the Microsoft Foundry portal, or when defining an agent in code using the SDK.



For example, one of the tools available is the *code interpreter*. This tool enables your agent to run custom code it writes to achieve something, such as MATLAB code to create a graph or solve a data analytics problem.

Available tools are split into two categories:

Knowledge tools

Knowledge tools enhance the context or knowledge of your agent. Available tools include:

- **Bing Search**: Uses Bing search results to ground prompts with real-time live data from the web.
- **File search**: Grounds prompts with data from files in a vector store.
- **Azure AI Search**: Grounds prompts with data from Azure AI Search query results.

- **Microsoft Fabric**: Uses the Fabric Data Agent to ground prompts with data from your Fabric data stores.

> **Tip**
>
> You can also integrate third-party licensed data by using the OpenAPI Spec action tool (discussed below).

Action tools

Action tools perform an action or run a function. Available tools include:

- **Code Interpreter**: A sandbox for model-generated Python code that can access and process uploaded files.
- **Custom function**: Call your custom function code – you must provide function definitions and implementations.
- **Azure Function**: Call code in serverless Azure Functions.
- **OpenAPI Spec**: Call external APIs based on the OpenAPI 3.0 spec.

By connecting built-in and custom tools, you can allow your agent to perform countless tasks on your behalf.

Next unit: Exercise - Build an AI agent

Previous Next

Exercise - Build an AI agent

30 minutes

Now it's your opportunity to build an agent in Microsoft Foundry. In this exercise, you create an agent and test it in Agent playground. You'll then develop your own app that integrates with an agent through Foundry Agent Service.

> **Note**
>
> If you don't have an Azure subscription, and you want to explore Microsoft Foundry, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

> **Tip**
>
> After completing the exercise, if you're finished exploring Azure AI Agents, delete the Azure resources that you created during the exercise.

Next unit: Module assessment

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 Microsoft Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Microsoft Foundry SDK client libraries for details.

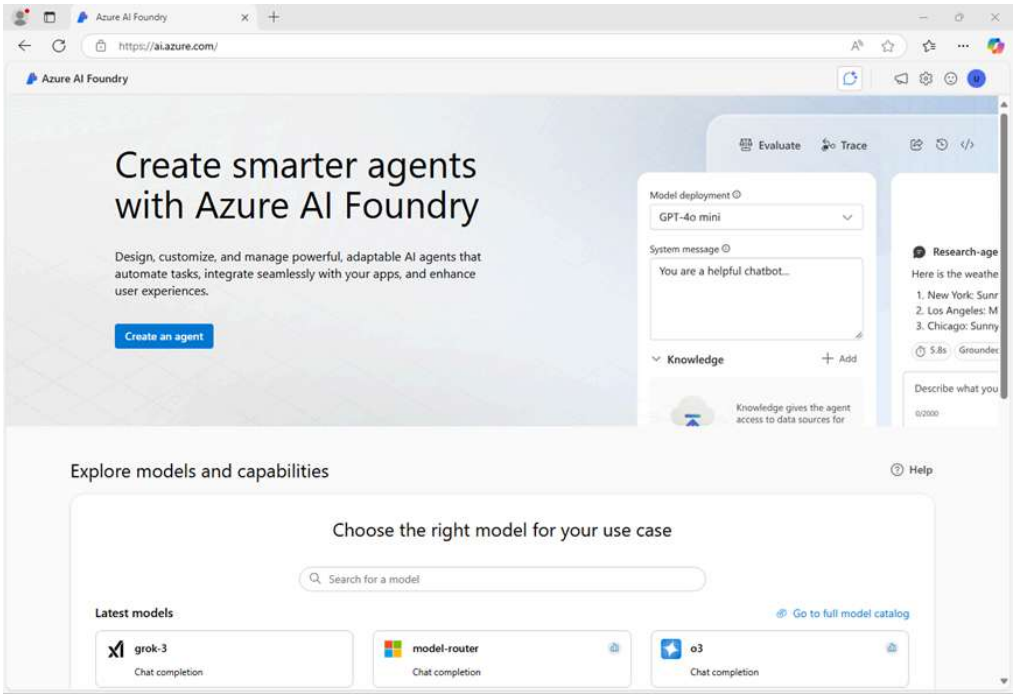
This exercise should take approximately **30** minutes to complete.

> **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 a Foundry project

Let's start by creating a Foundry project.

1. In a web browser, open the 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 **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):



> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

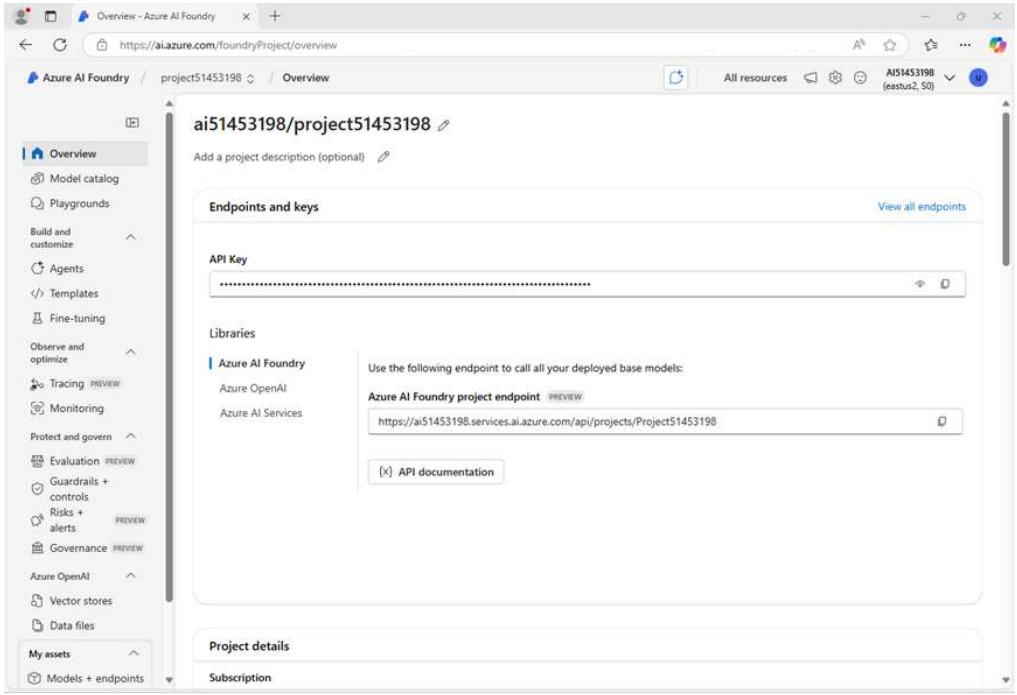
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Foundry resource**: *A valid name for your Foundry resource*
    - **Subscription**: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

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 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:

```
# 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.

```
# 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 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:

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

Code Copy

```
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:

Code Copy

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

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

Code Copy

```
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.

200 XP

Module assessment

3 minutes

1. What is the first step in setting up Microsoft Foundry Agent Service?

- Deploy a compatible model
- Create A Microsoft Foundry project
- Make API calls using SDKs

2. Which element of an agent definition is used to specify its behavior and restrictions?

- Model
- Instructions
- Tools

3. Which tool should you use to enable an agent to dynamically generate code to perform tasks or access data in files?

- Code Interpreter
- Azure Functions
- Azure AI Search

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What is the first step in setting up Microsoft Foundry Agent Service?

- Deploy a compatible model
- Create A Microsoft Foundry project ✓ Correct
- Make API calls using SDKs

2. Which element of an agent definition is used to specify its behavior and restrictions?

- Model
- Instructions ✓ Correct
- Tools

3. Which tool should you use to enable an agent to dynamically generate code to perform tasks or access data in files?

- Code Interpreter ✓ Correct
- Azure Functions
- Azure AI Search

Next unit: Summary

Previous Next

Summary

2 minutes

AI agents represent a significant advancement in the field of artificial intelligence, offering numerous benefits to businesses and individuals alike. By automating routine tasks, enhancing decision-making, and providing scalable solutions, AI agents are transforming how we work and interact with technology. As these agents continue to evolve, their potential applications will only expand, driving further innovation and efficiency across various sectors.

In this module, you learned about the purpose of Foundry Agent Service, its key features, the setup process, and its integration capabilities with other Foundry Tools. We also addressed the challenge of building, deploying, and scaling AI agents. Foundry Agent Service solves several of these challenges, providing a fully managed environment for creating high-quality, extensible AI agents with minimal coding and infrastructure management.

The techniques covered in this module demonstrate several advantages, including automatic tool calling, secure data management, and flexible model selection. These features enable developers to focus on creating intelligent solutions while ensuring enterprise-grade security and compliance. The business impact includes streamlined development processes, reduced operational overhead, and enhanced AI capabilities.

After completing this module, you're now able to:

- Describe the purpose of AI agents
- Explain the key features of Foundry Agent Service
- Build an agent using the Foundry Agent Service
- Integrate an agent in the Foundry Agent Service into your own app

More reading:

- Quickstart Guide for Microsoft Foundry Agent Service
- What's new in Microsoft Foundry Agent Service
- For detailed instructions, see the quickstart guide.

All units complete:

Previous Complete module

Introduction

2 minutes

Microsoft Foundry Agent Service can help you develop sophisticated, multi-agent systems that can break down complex tasks into smaller, specialized roles. Using connected agents, you can design intelligent solutions where a primary agent delegates work to sub-agents without the need for custom orchestration logic or hardcoded routing. This modular approach boosts efficiency and maintainability across a wide range of scenarios.

Imagine you're on an engineering team that receives a constant flow of support tickets including bugs, feature requests, and infrastructure issues. Manually reviewing and sorting each one takes time and slows down your team's ability to respond quickly. With a multi-agent approach, you can build a triage assistant that assigns different tasks to specialized agents. The sub-agents might perform tasks such as classifying ticket type, setting priority, and suggesting the right team for the work. With a multi-agent approach, these specialized agents can work together to streamline the ticketing process.

In this module, you learn how to use connected agents in Microsoft Foundry. You also practice building an intelligent ticket triage system using a collaborative multi-agent solution.

Next unit: Understand connected agents

Next

Understand connected agents

5 minutes

As AI solutions become more advanced, managing complex workflows gets harder. A single agent can handle a wide range of tasks, but this approach can become unmanageable as the scope expands. That's why Microsoft Foundry Agent Service lets you connect multiple agents, each with a focused role, to work together in a cohesive system.

What are connected agents?

Connected agents are a feature in the Microsoft Foundry Agent Service that allows you to break large tasks into smaller, specialized roles without building a custom orchestrator or hardcoding routing logic. Instead of relying on one agent to do everything, you can create multiple agents with clearly defined responsibilities that collaborate to accomplish tasks.

At the center of this system, there's a main agent that interprets user input and delegates tasks to connected sub-agents. Each sub-agent is designed to perform a specific function, such as to summarize a document, validate a policy, or retrieve data from a knowledge source.

This division of labor helps you:

- Simplify complex workflows
- Improve agent performance and accuracy
- Make systems easier to maintain and extend over time

Why use connected agents?

Rather than scaling a single agent to handle every user request or data interaction, using connected agents lets you:

- Build modular solutions that are easier to develop and debug
- Assign specialized capabilities to agents that can be reused across solutions
- Scale your system in a way that aligns with real-world business logic

This approach is especially useful in scenarios where agents need to perform sensitive tasks independently, such as handling private data or generating personalized content.

Using connected agents to automate workflows offers many benefits, for example:

- **No custom orchestration required -** The main agent uses natural language to route tasks, eliminating the need for hardcoded logic.
- **Improved reliability and traceability -** The clear separation of responsibilities makes issues easier to debug since agents can be tested individually.
- **Flexible and extensible -** Add or swap agents without reworking the entire system or modifying the main agent.

Connected agents make it easier to build modular, collaborative systems without complex orchestration. By assigning focused roles and using natural language delegation, you can simplify workflows, improve reliability, and scale your solutions more effectively.

Next unit: Design a multi-agent solution with connected agents

Previous Next

Design a multi-agent solution with connected agents

5 minutes

In a connected agent solution, success depends on clearly defining the responsibilities of each agent. The central agent is also responsible for how the agents will collaborate. Let's explore how to design a multi-agent program using Microsoft Foundry Agent Service.

Main agent responsibilities

The main agent acts as the orchestrator. It interprets the intent behind a request and determines which connected agent is best suited to handle it. The main agent is responsible for:

- Interpreting user input
- Selecting the appropriate connected agent
- Forwarding relevant context and instructions
- Aggregating or summarize results

Connected agent responsibilities

Connected agents designed to focus on a single domain of responsibility. A connected agent is responsible for:

- Completing a specific action based on a clear prompt
- Using tools (if needed) to complete their task
- Returning the results to the main agent

Connected agents should be designed with a single responsibility in mind. This makes your system easier to debug, extend, and reuse.

Set up a multi-agent solution with connected agents

1. **Initialize the agents client**

First, you create a client that connects to your Microsoft Foundry project.

2. **Create an agent to connect to the main agent**

   Define an agent you want to connect to the main agent. You can do this using the `create_agent` method on the `AgentsClient` object.

   For example, your connected agent might retrieve stock prices, summarize documents, or validate compliance. Give the agent clear instructions that define its purpose.

3. **Initialize the connected agent tool**

   Use your agent definition to create a `ConnectedAgentTool`. Assign it a name and description so the main agent knows when and how to use it.

4. **Create the main agent**

   Create the main agent using the `create_agent` method. Add your connected agents using the `tools` property and assign the `ConnectedAgentTool` definitions to the main agent.

5. **Create a thread and send a message**

   Create the agent thread that is used to manage the conversation context. Then create a message on the thread that contains the request you want the agent to fulfill.

6. **Run the agent workflow**

   Once the message is added, create a run to process the request. The main agent uses its tools to delegate tasks as needed and compile a final response for the user.

7. **Handle the results**

   When the run completes, you can review the main agent's response. The final output may incorporate insights from one or more connected agents. Only the main agent's response is visible to the end user.

Designing a connected agent system involves defining focused agents, registering them as tools, and configuring a main agent to route tasks intelligently. This modular approach gives you a flexible foundation for building collaborative AI solution that scale as your needs grow.

Next unit: Exercise - Develop a multi-agent app with Microsoft Foundry

Previous Next

100 XP

Exercise - Develop a multi-agent app with Microsoft Foundry

30 minutes

If you have an Azure subscription, you can complete this exercise to develop a multi-agent solution using connected agents in Microsoft Foundry Agent Service.

> **Note**
>
> If you don't have an Azure subscription, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Module assessment

200 XP

Module assessment

3 minutes

1. What is the role of the main agent in a connected agent system?

- To directly perform all tasks using tools
- To coordinate user input and route tasks to the appropriate connected agents
- To monitor agent performance and generate logs

2. How do you connect an agent to a main agent using the Azure AI Projects client library?

- Add the agent as a `ConnectedAgentTool` to the main agent's tool definition.
- Use the `link_agents()` method to bind the sub-agent to the main agent.
- Set the main agent's parent_id to the sub-agent's agent ID.

3. How does the main agent decide which connected agent to use?

- It uses prompt instructions and natural language understanding.
- It follows a fixed code-based decision tree.
- It randomly selects from available connected agents.

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment. Sign in to save progress.

Score: 100%

1. What is the role of the main agent in a connected agent system?

- To directly perform all tasks using tools
- To coordinate user input and route tasks to the appropriate connected agents ✓ Correct
- To monitor agent performance and generate logs

2. How do you connect an agent to a main agent using the Azure AI Projects client library?

- Add the agent as a `ConnectedAgentTool` to the main agent's tool definition. ✓ Correct
- Use the `link_agents()` method to bind the sub-agent to the main agent.
- Set the main agent's parent_id to the sub-agent's agent ID.

3. How does the main agent decide which connected agent to use?

- It uses prompt instructions and natural language understanding. ✓ Correct
- It follows a fixed code-based decision tree.

It randomly selects from available connected agents.

Next unit: Summary

Previous Next

Summary

1 minute

In this module, you learned how to design and implement multi-agent solutions using Microsoft Foundry Agent Service.

Connected agents let you break down complex tasks by assigning them to specialized agents that work together within a coordinated system. You explored how to define clear roles for main and connected agents, delegate tasks using natural language, and design modular workflows that are easier to scale and maintain. You also practiced building a multi-agent solution. Great work!

Module incomplete:

Previous

Go back to finish

Introduction

2 minutes

AI agents are capable of performing a wide range of tasks, but many tasks still require them to interact with tools outside the large language model. Agents may need to access APIs, databases, or internal services. Manually integrating and maintaining these tools can quickly become complex, especially as your system grows, or changes frequently.

Model Context Protocol (MCP) servers can help solve this problem by integrating with AI agents. Connecting an Azure AI Agent to a Model Context Protocol (MCP) server can provide your agent with a catalog of tools accessible on demand. This approach makes your AI solution more robust, scalable, and easier to maintain.

Suppose you're working for a retailer that specializes in cosmetics. Your team wants to build an AI assistant that can help manage inventory by checking product stock levels and recent sales trends. Using an MCP server, you can connect the assistant to a set of tools that can make inventory assessments and provide recommendations to the team.

In this module, you learn how to set up an MCP server and client, and connect tools to an Azure AI Agent dynamically. You also practice creating your own AI MCP tool solution with Microsoft Foundry Agent Service.

Next unit: Understand MCP tool discovery

Next

Understand MCP tool discovery

5 minutes

As AI agents become more capable, the range of tools and services they can access also grows. However, registering new tools, managing, updating, and integrating them can quickly become complex and time-consuming. Dynamic tool discovery helps solve this problem by enabling agents to find and use tools automatically at runtime.

Advantages of the Microsoft Connector Protocol for AI agents

The Microsoft Connector Protocol (MCP) provides several benefits that enhance the capabilities and flexibility of AI agents:

- **Dynamic Tool Discovery:**
  AI agents can automatically receive a list of available tools from a server, along with descriptions of their functions. Unlike traditional APIs, which often require manual coding for each integration and updates whenever the API changes, MCP enables an "**integrate once**" approach that improves adaptability and reduces maintenance.

- **Interoperability Across LLMs:**
  MCP works seamlessly with different large language models (LLMs), allowing developers to **switch or evaluate core models** for improved performance without reworking integrations.

- **Standardized Security:**
  MCP provides a **consistent authentication method**, simplifying secure access across multiple MCP servers. This eliminates the need to manage separate keys or authentication protocols for each API, making it easier to scale AI agent deployments.

What is dynamic tool discovery?

Dynamic tool discovery is a mechanism that allows an AI agent to discover available external tools without needing hardcoded knowledge of each one. Instead of manually adding or updating

every tool your agent can use, the agent queries a centralized Model Context Protocol (MCP) server. This server acts as a live catalog, exposing tools that the agent can understand and call.

This approach means:

- Tools can be added, updated, or removed centrally without modifying the agent code.
- Agents can always use the latest version of a tool, improving accuracy and reliability.
- The complexity of managing tools shifts away from the agent and into a dedicated service.

How does MCP enable dynamic tool discovery?

An MCP server hosts a set of functions that are exposed as tools using the `@mcp.tool` decorator. Tools are a primitive type in the MCP that enables servers to expose executable functionality to clients. A client can connect to the server and fetch these tools dynamically. The client then generates function wrappers that are added to the Azure AI Agent's tool definitions. This setup creates a flexible pipeline:

- The MCP server hosts available tools.
- The MCP client dynamically discovers the tools.
- The Azure AI Agent uses the available tools to respond to user requests.

Why use dynamic tool discovery with MCP?

This approach provides several benefits:

- Scalability: Easily add new tools or update existing ones without redeploying agents.
- Modularity: Agents can remain simple, focusing on delegation rather than managing tool details.
- Maintainability: Centralized tool management reduces duplication and errors.
- Flexibility: Supports diverse tool types and complex workflows by aggregating capabilities.

Dynamic tool discovery is especially useful in environments where tools evolve rapidly or where many teams manage different APIs and services. Using tools allows AI agents to adapt to changing capabilities in real time, interact with external systems securely, and perform actions that go beyond language generation.

Next unit: Integrate agent tools using an MCP server and client

Previous Next

Integrate agent tools using an MCP server and client

5 minutes

To dynamically connect tools to your Azure AI Agent, you first need a functioning Model Context Protocol (MCP) setup. This includes both the MCP server, which hosts your tool catalog, and the MCP client, which fetches those tools and makes them usable by your agent.

What is the MCP Server?

The MCP server acts as a registry for tools your agent can use. You can initialize your MCP server using `FastMCP("server-name")`. The FastMCP class uses Python type hints and document strings to automatically generate tool definitions, making it easy to create and maintain MCP tools. These definitions are then served over HTTP when requested by the client. Because tool definitions live on the server, you can update or add new tools at any time, without having to modify or redeploy your agent.

What is the MCP Client?

A standard MCP client acts as a bridge between your MCP server and the Azure AI Agent Service. The client initializes an MCP client session and connects to the server. Afterwards, it performs three key tasks:

- Discovers available tools from the MCP server using `session.list_tools()`.
- Generates Python function stubs that wrap the tools.
- Registers those functions with your agent.

This allows the agent to call any tool listed in the MCP catalog as if it were a native function, all without hardcoded logic.

Register tools with an Azure AI Agent

When an MCP client session is initialized, the client can dynamically pull in tools from the MCP server. An MCP tool can be invoked using `session.call_tool(tool_name, tool_args)`. The tools should each be wrapped in an async function so that the agent is able to invoke them. Finally, those functions are bundled together and become part of the agent's toolset and are available during runtime for any user request.

Overview of MCP agent tool integration

- The **MCP server** hosts tool definitions decorated with `@mcp.tool`.
- The **MCP client** initializes an MCP client connection to the server.
- The **MCP client** fetches the available tool definitions with `session.list_tools()`.
- Each tool is wrapped in an async function that invokes `session.call_tool`
- The tool functions are bundled into `FunctionTool` that makes them usable by the agent.
- The `FunctionTool` is registered to the agent's toolset.

Now your agent is able to access and invoke your tools through natural language interaction. By setting up the MCP server and client, you create a clean separation between tool management and agent logic—enabling your system to adapt quickly as new tools become available.

Next unit: Use Azure AI agents with MCP servers

Previous Next

Use Azure AI agents with MCP servers

5 minutes

You can enhance your Microsoft Foundry agent by connecting it to Model Context Protocol (MCP) servers. MCP servers provide tools and contextual data that your agent can use to perform tasks, extending its capabilities beyond built-in functions. Azure AI Agent Service includes support for remote MCP servers, allowing your agent to quickly connect to your server and access tools.

When you use the Microsoft Foundry Agent Service to connect to your MCP server, you don't need to manually create an MCP client session or add any function tools to your agent. Instead, you create an MCP tool object that connects to your MCP server. Then you add information about the MCP server to the agent thread when invoking a prompt. This also allows you to connect and use different tools from multiple servers depending on your needs.

Integrating remote MCP servers

To connect to an MCP server, you need:

- A remote MCP server endpoint (for example, https://api.githubcopilot.com/mcp/).
- A Microsoft Foundry agent configured to use the MCP tool.

You can connect to multiple MCP servers by adding them as separate tools, each with:

- `server_label`: A unique identifier for the MCP server (e.g., GitHub).
- `server_url`: The MCP server's URL.
- `allowed_tools` (optional): A list of specific tools the agent is allowed to access.

The MCP tool also supports custom headers, which let you pass:

- Authentication keys (API keys, OAuth tokens).
- Other required headers for the MCP server.
  - These headers are included in `tool_resources` during each run and are not stored between runs.

Invoking tools

When using the Azure MCP Tool object, you don't need to wrap function tools or invoke `session.call_tool`. Instead, the tools are automatically invoked when necessary during an agent run. To automatically invoke MCP tools:

- Create the `McpTool` object with the server label and url.
- Use `update_headers` to apply any headers required by the server.
- Use the `set_approval_mode` to determine whether approval is required. Supported values are:
  - `always`: A developer needs to provide approval for every call. If you don't provide a value, this one is the default.
  - `never`: No approval is required.
- Create a `ToolSet` object and add the `McpTool` object
- Create an agent run and specify the `toolset` property
- When the run completes, you should see the results of any invoked tools in the response.

If the model tries to invoke a tool in your MCP server with approval required, you get a run status of requires_action. - In the `requires_action` field, you can get more details on which tool in the MCP server is called and any arguments to be passed. - Review the tool and arguments so that you can make an informed decision for approval. - Submit your approval to the agent with call_id by setting approve to true.

MCP integration is a key step toward creating richer, more context-aware AI agents. As the MCP ecosystem grows, you'll have even more opportunities to bring specialized tools into your workflows and deliver smarter, more dynamic solutions.

Next unit: Exercise - Connect MCP tools to Azure AI Agents

Previous Next

100 XP

Exercise - Connect MCP tools to Azure AI Agents

30 minutes

If you have an Azure subscription, you can complete this exercise to develop a Model Context Protocol (MCP) client-server application that dynamically registers tools to an Azure AI Agent.

> **Note**
>
> If you don't have an Azure subscription, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Module assessment

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 available MCP tools 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.

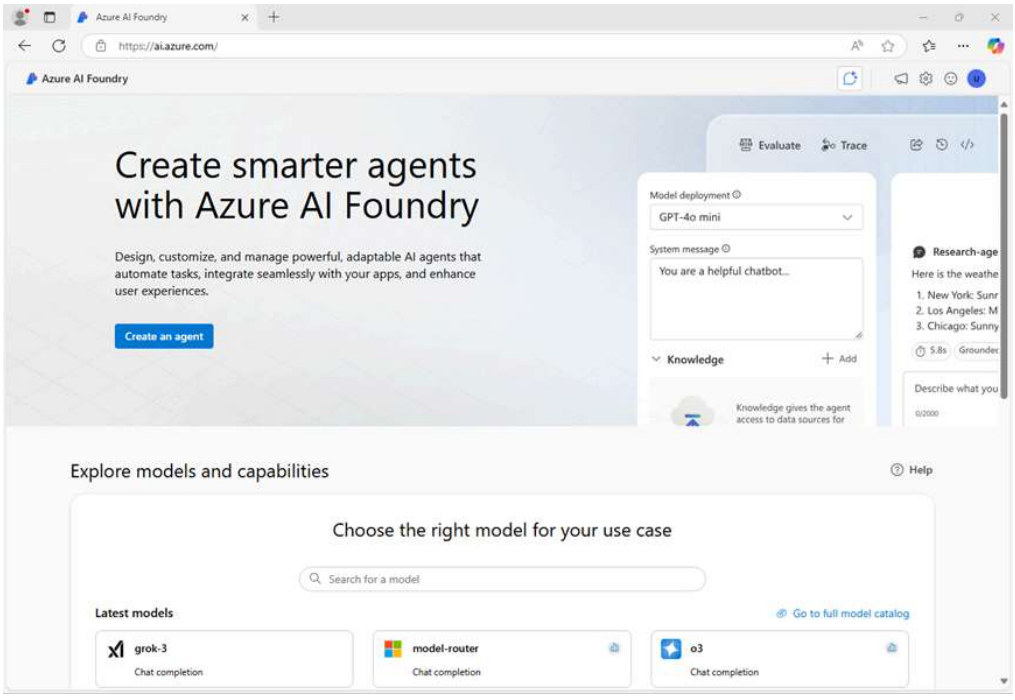
This exercise should take approximately **30** minutes to complete.

> **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 a Microsoft Foundry project

Let's start by creating a Foundry project.

1. In a web browser, open the 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 **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):



> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

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:
    - **Foundry resource**: *A valid name for your Foundry resource*

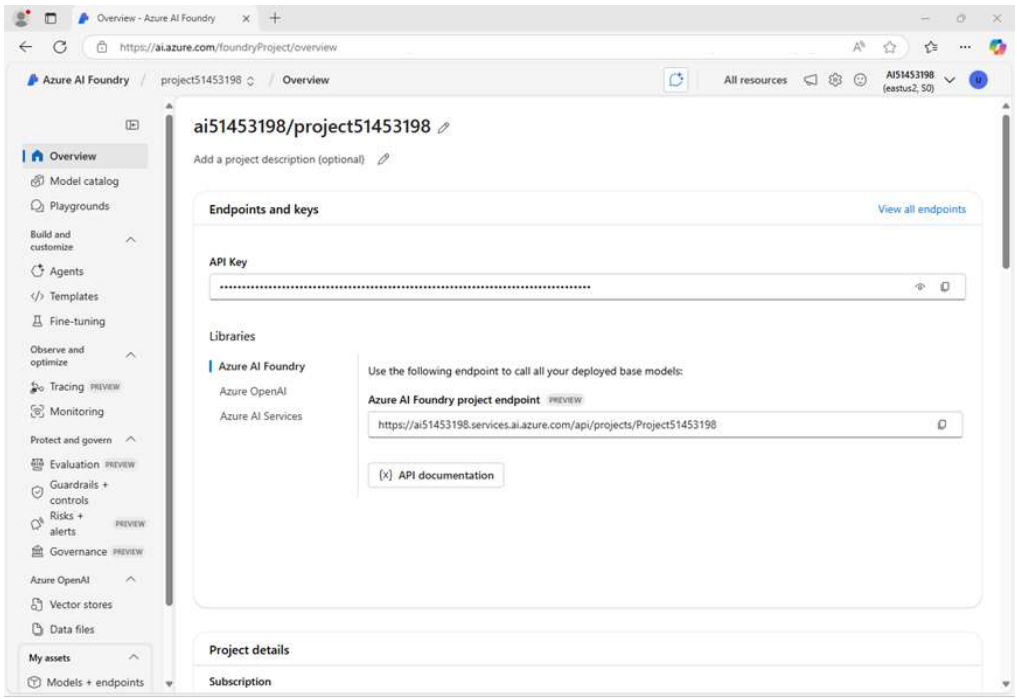
- **Subscription**: *Your Azure subscription*
- **Resource group**: *Create or select a resource group*
- **Region**: *Select any of the following supported locations: **
    - West US 2
    - West US
    - Norway East
    - Switzerland North
    - UAE North
    - South India

> * 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 **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 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 azure-ai-agents 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 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:

```
# 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:

```
# 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:

```
# 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:

```
# 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:

```
# 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 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_code_sample_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_code_sample_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.

200 XP

Module assessment

3 minutes

1. What role does the MCP server play in the MCP agent tool integration?

- Runs the AI agent and processes user prompts directly.
- Manages network connections between multiple agents.
- Hosts tool definitions and makes them available for discovery by the client.

2. How does an MCP client retrieve available tools from the MCP server?

- By calling `session.list_tools()` to get the current tool catalog.
- By reading a static JSON file from the server directory.
- By subscribing to server events via a WebSocket connection.

3. Why should MCP tools be wrapped in async functions on the client-side?

- To allow the agent to wait for user input.
- To enable asynchronous invocation so the agent can call tools without blocking.
- To convert the functions into REST API endpoints automatically.

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What role does the MCP server play in the MCP agent tool integration?

- Runs the AI agent and processes user prompts directly.
- Manages network connections between multiple agents.
- Hosts tool definitions and makes them available for discovery by the client. ✓ Correct

2. How does an MCP client retrieve available tools from the MCP server?

- By calling `session.list_tools()` to get the current tool catalog. ✓ Correct
- By reading a static JSON file from the server directory.
- By subscribing to server events via a WebSocket connection.

3. Why should MCP tools be wrapped in async functions on the client-side?

- To allow the agent to wait for user input.
- To enable asynchronous invocation so the agent can call tools without blocking. ✓ Correct

To convert the functions into REST API endpoints automatically.

Next unit: Summary

Previous Next

Summary

1 minute

In this module, you learned how to integrate external tools with Microsoft Foundry Agent Service using the Model Context Protocol (MCP).

By connecting your agent to an MCP server, you can dynamically discover and register tools at runtime without hardcoding APIs or redeploying your agent. Using an MCP client, you generated function wrappers from discovered tools and connected them directly to your agent. This integration allows your agent to adapt to evolving toolsets, and create more flexible AI solutions that can grow alongside your applications.

Tip

To learn more about MCP, see the Model Context Protocol User Guide and AI Agents MCP Integration. To learn more about using Microsoft Foundry Agent Service with MCP, visit Connect to Model Context Protocol servers.

All units complete:

Previous

Complete module

Introduction

2 minutes

AI agents are transforming how applications interact with users and automate tasks. Unlike traditional programs, AI agents use generative AI to interpret data, make decisions, and complete tasks with minimal human intervention. These agents use large language models to streamline complex workflows, making them ideal for automating business processes.

Developers can build AI agents using different tools, including the Microsoft Agent Framework. This open-source SDK simplifies the integration of AI models into applications. The Microsoft Agent Framework supports different types of agents from multiple providers, including Microsoft Foundry, Azure OpenAI, OpenAI, Microsoft Copilot Studio, and Anthropic agents. This module focuses on Microsoft Foundry Agents, which provide enterprise-grade capabilities using the Microsoft Foundry Agent Service.

Microsoft Foundry Agent Service is a fully managed service that enables developers to securely build, deploy, and scale high-quality extensible AI agents. Using the Foundry Agent Service, developers don't need to manage the underlying compute or storage resources. The Microsoft Agent Framework enables developers to quickly build agents on the Foundry Agent Service, supporting natural language processing and providing access to built-in tools in just a few lines of code.

While Foundry Agent Service provides a powerful foundation for building AI agents, the Microsoft Agent Framework offers more flexibility and scalability. If your solution requires multiple types of agents, using the Microsoft Agent Framework ensures consistency across your implementation. Finally, if you're planning to develop multi-agent solutions, the framework's workflow orchestration features allow you to coordinate collaborative agents efficiently—a topic covered in more detail in a later module.

Suppose you need to develop an AI agent that automatically formats and emails expense reports for employees. Your AI agent can extract data from submitted expense reports, format them correctly, and send them to the appropriate recipients when you use the Microsoft Agent Framework. The tools and functions feature allows your AI agent to interact with APIs, retrieve necessary data, and complete tasks.

In this module, you learn about the core features of the Microsoft Agent Framework SDK. You also learn how to create your own AI agents and extend their capabilities with tool functions.

After completing this module, you're able to:

- Use the Microsoft Agent Framework to connect to a Microsoft Foundry project.
- Create Microsoft Foundry agents using the Microsoft Agent Framework.
- Integrate tool functions with your AI agent.

Next unit: Understand Microsoft Agent Framework AI agents

Next >

Understand Microsoft Agent Framework AI agents

6 minutes

An AI agent is a program that uses generative AI to interpret data, make decisions, and perform tasks on behalf of users or other applications. AI agents rely on large language models to perform their tasks. Unlike traditional programs, AI agents can function autonomously, handling complex workflows and automating processes without requiring continuous human oversight.

AI Agents can be developed using many different tools and platforms, including the Microsoft Agent Framework. The Microsoft Agent Framework is an open-source SDK that enables developers to easily integrate the latest AI models into their applications. This framework provides a comprehensive foundation for creating functional agents that can use natural language processing to complete tasks and collaborate with other agents.

Microsoft Agent Framework core components

The Microsoft Agent Framework offers different components that can be used individually or combined.

- **Chat clients** - provide abstractions for connecting to AI services from different providers under a common interface. Supported providers include Azure OpenAI, OpenAI, Anthropic, and more through the `BaseChatClient` abstraction.
- **Function tools** - containers for custom functions that extend agent capabilities. Agents can automatically invoke functions to integrate with external APIs and services.
- **Built-in tools** - prebuilt capabilities including Code Interpreter for Python execution, File Search for document analysis, and Web Search for internet access.
- **Conversation management** - structured message system with roles (USER, ASSISTANT, SYSTEM, TOOL) and `AgentThread` for persistent conversation context across interactions.
- **Workflow orchestration** - supports sequential workflows, concurrent execution, group chat, and handoff patterns for complex multi-agent collaboration.

The Microsoft Agent Framework helps streamline the creation of agents and allows multiple agents to work together in conversations while including human input. The framework supports different types of agents from multiple providers, including Microsoft Foundry, Azure OpenAI, OpenAI, Microsoft Copilot Studio, and Anthropic agents.

What Is a Microsoft Foundry Agent?

Microsoft Foundry Agents provide enterprise-level capabilities using the Microsoft Foundry Agent Service. These agents offer advanced features for complex enterprise scenarios. Key benefits include:

- **Enterprise-level capabilities –** Built for Azure environments with advanced AI features including code interpreter, function tools integration, and Model Context Protocol (MCP) support.
- **Automatic tool invocation –** Agents can automatically call and execute tools, integrating seamlessly with Azure AI Search, Azure Functions, and other Azure services.
- **Thread and conversation management –** Provides built-in mechanisms for managing persistent conversation states across sessions, ensuring smooth multi-agent interactions.
- **Secure enterprise integration –** Enables secure and compliant AI agent development with Azure CLI authentication, RBAC, and customizable storage options.

When you use Microsoft Foundry Agents, you get the full power of enterprise Azure capabilities combined with the features of the Microsoft Agent Framework. These features can help you create robust AI-driven workflows that can scale efficiently across business applications.

Agent framework core concepts

- **BaseAgent -** the foundation for all agents with consistent methods, providing a unified interface across all agent types.
- **Agent threads -** manage persistent conversation context and store conversation history across sessions using the `AgentThread` class.
- **Chat messages -** organized structure for agent communication using role-based messaging (USER, ASSISTANT, SYSTEM, TOOL) that enables smooth communication and integration.
- **Workflow orchestration -** supports sequential workflows, running multiple agents in parallel, group conversations between agents, and transferring control between specialized

agents.

- **Multi-modal support -** allows agents to work with text, images, and structured outputs, including vision capabilities and type-safe response generation.
- **Function tools** - let you add custom capabilities to agents by including custom functions with automatic schema generation from Python functions.
- **Authentication methods -** supports multiple authentication methods including Azure CLI credentials, API keys, MSAL for Microsoft business authentication, and role-based access control.

This framework supports autonomous, multi-agent AI behaviors while maintaining a flexible architecture that lets you mix and match agents, tools, and workflows as needed. The design lets you switch between OpenAI, Azure OpenAI, Anthropic, and other providers without changing your code, making it easy to build AI systems—from simple chatbots to complex business solutions.

Next unit: Create an Azure AI agent with Microsoft Agent Framework

Previous Next

Create an Azure AI agent with Microsoft Agent Framework

7 minutes

Microsoft Foundry Agent is a specialized agent within the Microsoft Agent Framework, designed to provide enterprise-level conversational capabilities with seamless tool integration. It automatically handles tool calling, so you don't need to manually parse and invoke functions. The agent also securely manages conversation history using threads, which reduces the work of maintaining state. The Microsoft Foundry Agent supports many built-in tools, including code interpreter, file search, and web search. It also provides integration capabilities for Azure AI Search, Azure Functions, and other Azure services.

Creating a Microsoft Foundry Agent

A Microsoft Foundry Agent includes all the core capabilities you typically need for enterprise AI applications, like function execution, planning, and memory access. This agent acts as a self-contained runtime with enterprise-level features.

To use a Microsoft Foundry Agent:

1. Create a Microsoft Foundry project.
2. Add the project connection string to your Microsoft Agent Framework application code.
3. Set up authentication credentials.
4. Create a `ChatAgent` with an `AzureAIAgentClient`.
5. Define tools and instructions for your agent.

Here's the code that shows how to create a Microsoft Foundry Agent:

Python

```python
from agent_framework import AgentThread, ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

def get_weather(
    location: Annotated[str, Field(description="The location to get the weather
```

```
for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with a high of 25°C."

# Create a ChatAgent with Azure AI client
async with (
    AzureCliCredential() as credential,
    ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        instructions="You are a helpful weather agent.",
        tools=get_weather,
    ) as agent,
):
    # Agent is now ready to use
```

Once your agent is created, you can create a thread to interact with your agent and get responses to your questions. For example:

Python

```
# Create the agent thread for ongoing conversation
thread = agent.get_new_thread()

# Ask questions and get responses
first_query = "What's the weather like in Seattle?"
print(f"User: {first_query}")
first_result = await agent.run(first_query, thread=thread)
print(f"Agent: {first_result.text}")
```

Microsoft Foundry Agent key components

The Microsoft Agent Framework Microsoft Foundry Agent uses the following components to work:

- **AzureAIAgentClient** - manages the connection to your Microsoft Foundry project. This client lets you access the services and models associated with your project and provides enterprise-level authentication and security features.
- **ChatAgent** - the main agent class that combines the client, instructions, and tools to create a working AI agent that can handle conversations and complete tasks.
- **AgentThread** - automatically keeps track of conversation history between agents and users, and manages the conversation state. You can create new threads or reuse existing ones to maintain context across interactions.

- **Tools integration -** support for custom functions that extend agent capabilities. Functions are automatically registered and can be called by agents to connect with external APIs and services.
- **Authentication credentials -** supports Azure CLI credentials, service principal authentication, and other Azure identity options for secure access to Foundry Tools.
- **Thread management -** provides flexible options for thread creation, including automatic thread creation for simple scenarios and explicit thread management for ongoing conversations.

These components work together to let you create enterprise-level agents with instructions to define their purpose and get responses from AI models while maintaining security, scalability, and conversation context for business applications.

Next unit: Add tools to Azure AI agent

Previous Next

Add tools to Azure AI agent

5 minutes

In the Microsoft Agent Framework, tools allow your AI agent to use existing APIs and services to perform tasks it couldn't do on its own. Tools work through function calling, allowing AI to automatically request and use specific functions. The framework routes the request to the appropriate function in your codebase and returns the results back to the large language model (LLM) so it can generate a final response.

To enable automatic function calling, tools need to provide details that describe how they work. The function's input, output, and purpose should be described in a way that the AI can understand, otherwise, the AI can't call the function correctly.

How to use tools with Microsoft Foundry Agent

The Microsoft Agent Framework supports both custom function tools and built-in tools that are ready to use out of the box.

Built-in tools

Microsoft Foundry Agents come with several built-in tools that you can use immediately:

- **Code Interpreter** - executes Python code for calculations, data analysis, and more
- **File Search** - searches through and analyzes documents
- **Web Search** - retrieves information from the internet

These tools are automatically available and don't require any extra setup.

Custom function tools

When creating custom tools for your Microsoft Foundry Agent, you need to understand several key concepts:

1. **Function definition and annotations**

Create your tool by defining a regular Python function with proper type annotations. Use `Annotated` and `Field` from Pydantic to provide detailed descriptions that help the AI understand the function's purpose and how to use its parameters. The more descriptive your annotations, the better the AI can understand when and how to call your function.

2. **Adding tools to your agent**

   Pass your custom functions to the ChatAgent during creation using the `tools` parameter. You can add a single function or a list of multiple functions. The framework automatically registers these functions and makes them available for the AI to call.

3. **Tool invocation through conversation**

   Once your tools are registered with the agent, you don't need to manually invoke them. Instead, ask the agent questions or give it tasks that would naturally require your tool's functionality. The AI automatically determines when to call your tools based on the conversation context and the tool descriptions you provided.

4. **Multiple tools and orchestration**

   You can add multiple tools to a single agent, and the AI automatically chooses which tool to use based on the user's request. The framework handles the orchestration, calling the appropriate functions and combining their results to provide a comprehensive response.

Best practices for tool development

- **Clear descriptions**: Write clear, detailed descriptions for your functions and parameters to help the AI understand their purpose
- **Type annotations**: Use proper Python type hints to specify expected input and output types
- **Error handling**: Implement appropriate error handling in your tool functions to gracefully handle unexpected inputs
- **Return meaningful data**: Ensure your functions return data that the AI can effectively use in its responses
- **Keep functions focused**: Design each tool to handle a specific task rather than trying to do too many things in one function

By following these concepts, you can extend your Microsoft Foundry Agent with both built-in and custom tools, allowing it to interact with APIs and perform advanced tasks. This approach makes your AI more powerful and capable of handling real-world applications efficiently.

Next unit: Exercise - Develop an Azure AI agent with the Microsoft Agent Framework SDK

Previous Next

Exercise - Develop an Azure AI agent with the Microsoft Agent Framework SDK

30 minutes

Now you're ready to build an agent with the Microsoft Agent Framework. In this exercise, you use the Microsoft Agent Framework SDK to create an AI agent that creates an expense claim email.

> **Note**
>
> To complete this exercise, you will need a Microsoft Azure subscription. If you don't already have one, you can sign up for one .
>
> If you need to set up your computer for this exercise, you can use this setup guide and then follow the exercise instructions linked below. Note that the setup guide is designed for multiple development exercises, and may include software that is not required for this specific exercise. Additionally, due to the range of possible operating systems and setup configurations, we can't provide support if you choose to complete the exercise on your own computer.

Launch the exercise and follow the instructions.

Launch Exercise

> **Tip**
>
> After completing the exercise, if you're finished exploring Azure AI Agents, delete the Azure resources that you created during the exercise.

Next unit: Knowledge check

Previous Next

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.

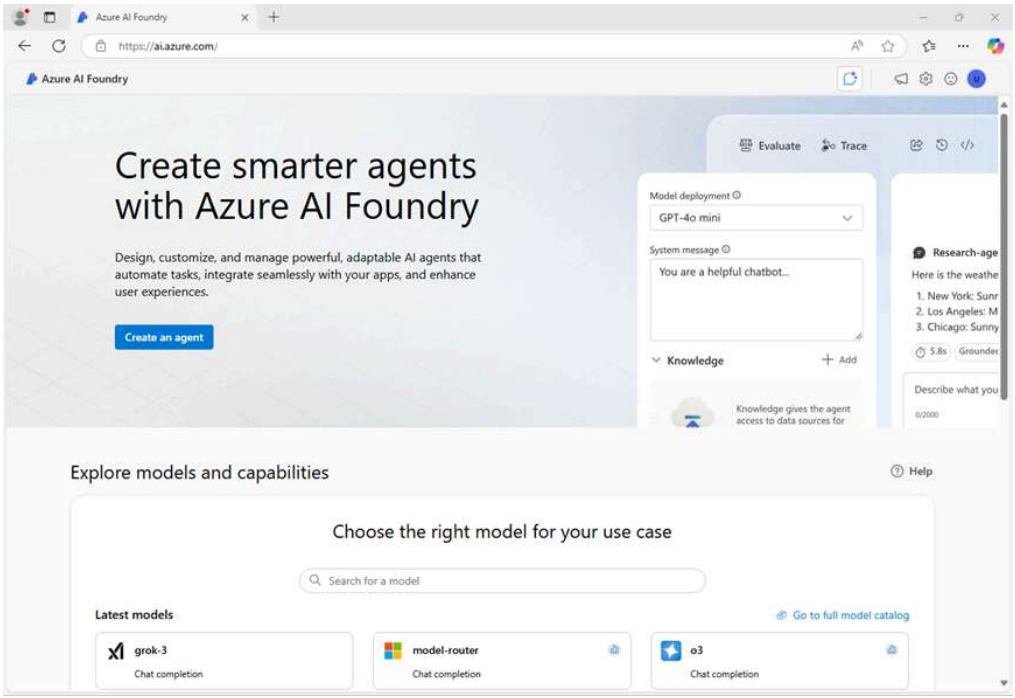
This exercise should take approximately **30** minutes to complete.

> **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 a Microsoft Foundry project

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

1. In a web browser, open the 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 **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):

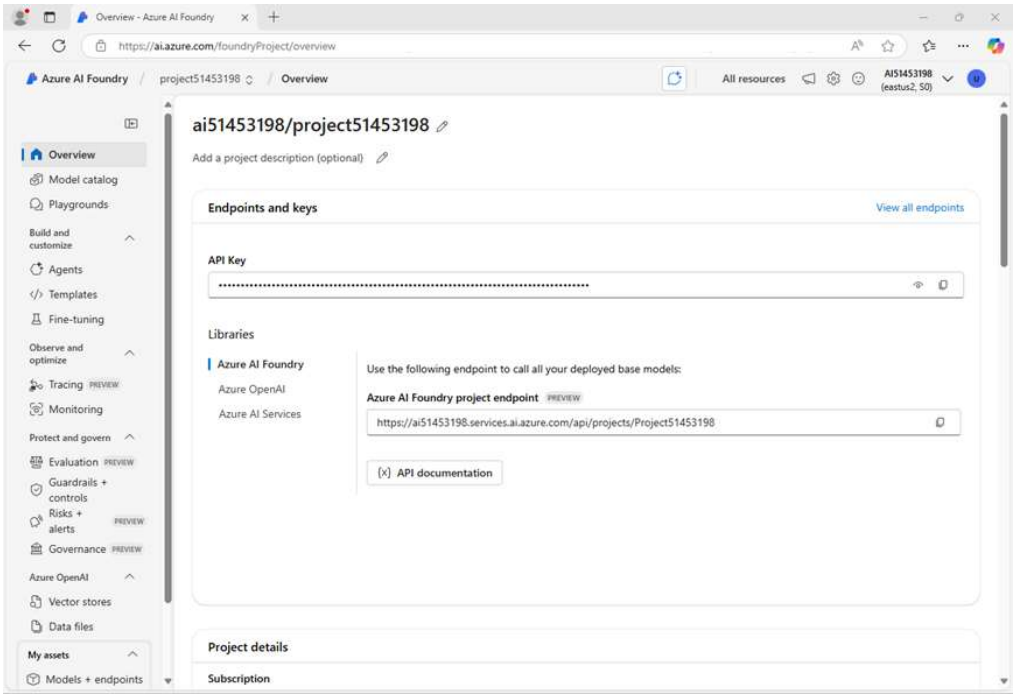


> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

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:
    - **Foundry resource**: *A valid name for your 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 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-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:

```
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 .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 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 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 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 Foundry hub if prompted.

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

```
Code
Copy

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:

```
Code
Copy

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.

200 XP

Knowledge check

3 minutes

1. What are the key steps to create a Microsoft Foundry Agent using the Microsoft Agent Framework?

- Deploy a custom AI model before creating an agent definition in the Azure portal.
- Initialize the agent by defining a model in the `AgentThread` constructor.
- Create an `AzureAIAgentClient`, define a ChatAgent with instructions and tools, and create an `AgentThread` for conversations.

2. Which component in the Microsoft Agent Framework manages conversation state and stores messages?

- AgentThread
- ChatAgent
- AzureAIAgentClient

3. How do you add custom functionality to a Microsoft Foundry Agent in the Microsoft Agent Framework?

- Configure custom functions in the Azure portal and link them to the agent through connection strings.
- Create Python functions with proper type annotations and descriptions, then pass them to the ChatAgent's tools parameter.
- Modify the AI model's architecture to integrate the custom functionality directly.

Submit answers

Next unit: Summary

Previous Next

200 XP

Knowledge check

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What are the key steps to create a Microsoft Foundry Agent using the Microsoft Agent Framework?

- Deploy a custom AI model before creating an agent definition in the Azure portal.
- Initialize the agent by defining a model in the `AgentThread` constructor.
- Create an `AzureAIAgentClient`, define a ChatAgent with instructions and tools, and create an `AgentThread` for conversations. ✓ Correct

2. Which component in the Microsoft Agent Framework manages conversation state and stores messages?

- AgentThread ✓ Correct
- ChatAgent
- AzureAIAgentClient

3. How do you add custom functionality to a Microsoft Foundry Agent in the Microsoft Agent Framework?

Configure custom functions in the Azure portal and link them to the agent through connection strings.

Create Python functions with proper type annotations and descriptions, then pass them to the ChatAgent's tools parameter. ✓ Correct

Modify the AI model's architecture to integrate the custom functionality directly.

Next unit: Summary

< Previous Next >

100 XP

Summary

2 minutes

In this module, you learned how the Microsoft Agent Framework enables developers to build AI agents. You learned about the components and core concepts of the Microsoft Agent Framework. You also learned how to create custom tools to extend your agent's capabilities. By applying these concepts and skills, you can use the Microsoft Agent Framework to create dynamic, adaptable AI solutions that enhance user interactions and automate complex tasks.

All units complete:

Previous

Complete module

Introduction

2 minutes

Microsoft Foundry Agent Service offers a seamless way to build an agent without needing extensive AI or machine learning expertise. By using tools, you can provide your agent with functionality to execute actions on your behalf.

The AI Agent Service provides built-in tools for gathering knowledge and generating code, which provide your agent with some powerful functionality. However, sometimes your agent needs to be able to complete specific tasks or actions that an AI model would struggle to handle on its own. To accomplish these actions, you can provide your agent a *custom tool* based on your own code or a third-party service or API.

Imagine you're working in the retail industry, and your company is struggling with managing customer inquiries efficiently. The customer support team is overwhelmed with repetitive questions, leading to delays in response times and decreased customer satisfaction. By using Foundry Agent Service with custom tools, you can create a custom FAQ agent that handles common inquiries. This agent can be provided with a set of custom tools to look up customer orders, freeing up your support team to focus on more complex issues.

In this module, you'll learn how to utilize custom tools in Foundry Agent Service to enhance productivity, improve accuracy, and create tailored solutions for specific needs.

Next unit: Why use custom tools

Next >

Why use custom tools

3 minutes

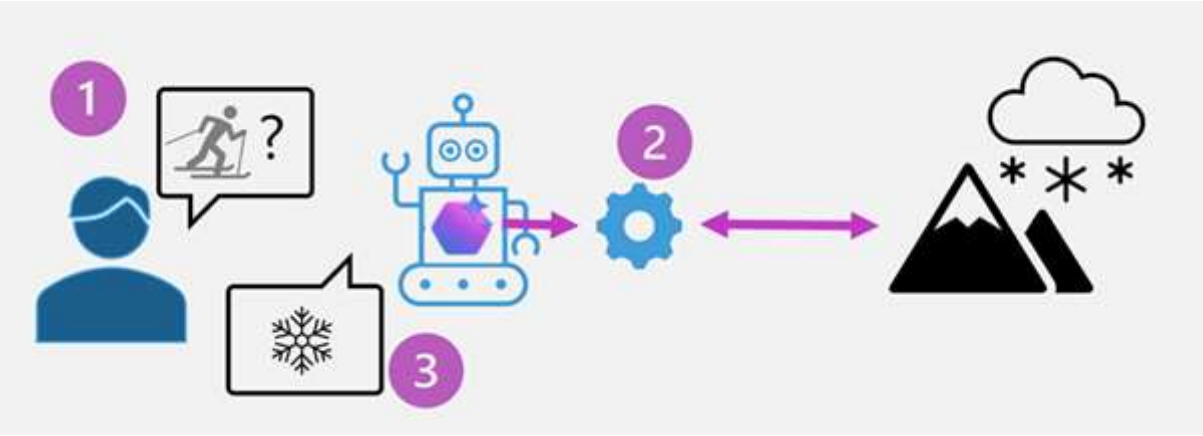
Microsoft Foundry Agent Service offers a powerful platform for integrating custom tools to enhance productivity and provide tailored solutions for specific business needs. By using these custom tools, businesses can achieve greater efficiency and effectiveness in their operations.

Why use custom tools?

Custom tools significantly enhance productivity by automating repetitive tasks and streamlining workflows that are specific to your use case. These tools improve accuracy by providing precise and consistent outputs, reducing the likelihood of human error. Additionally, custom tools offer tailored solutions that address specific business needs, enabling organizations to optimize their processes and achieve better outcomes.

- **Enhanced productivity**: Automate repetitive tasks and streamline workflows.
- **Improved accuracy**: Provide precise and consistent outputs, reducing human error.
- **Tailored solutions**: Address specific business needs and optimize processes.

Adding tools makes custom functionality available for the agent to use, depending on how it decides to respond to the user prompt. For example, consider how a custom tool to retrieve weather data from an external meteorological service could be used by an agent.



The diagram shows the process of an agent choosing to use the custom tool:

1. A user asks an agent about the weather conditions in a ski resort.

2. The agent determines that it has access to a tool that can use an API to get meteorological information, and calls it.
3. The tool returns the weather report, and the agent informs the user.

Common scenarios for custom tools in agents

Custom tools within the Foundry Agent Service enable users to extend the capabilities of AI agents, tailoring them to meet specific business needs. Some example use cases that illustrate the versatility and impact of custom tools include:

Customer support automation

- **Scenario**: A retail company integrates a custom tool that connects the Azure AI Agent to their customer relationship management (CRM) system.
- **Functionality**: The AI agent can retrieve customer order histories, process refunds, and provide real-time updates on shipping statuses.
- **Outcome**: Faster resolution of customer queries, reduced workload for support teams, and improved customer satisfaction.

Inventory management

- **Scenario**: A manufacturing company develops a custom tool to link the AI agent with their inventory management system.
- **Functionality**: The AI agent can check stock levels, predict restocking needs using historical data, and place orders with suppliers automatically.
- **Outcome**: Streamlined inventory processes and optimized supply chain operations.

Healthcare appointment scheduling

- **Scenario**: A healthcare provider integrates a custom scheduling tool with the AI agent.
- **Functionality**: The AI agent can access patient records, suggest available appointment slots, and send reminders to patients.
- **Outcome**: Reduced administrative burden, improved patient experience, and better resource utilization.

IT Helpdesk support

- **Scenario**: An IT department develops a custom tool to integrate the AI agent with their ticketing and knowledge base systems.
- **Functionality**: The AI agent can troubleshoot common technical issues, escalate complex problems, and track ticket statuses.
- **Outcome**: Faster issue resolution, reduced downtime, and improved employee productivity.

E-learning and training

- **Scenario**: An educational institution creates a custom tool to connect the AI agent with their learning management system (LMS).
- **Functionality**: The AI agent can recommend courses, track student progress, and answer questions about course content.
- **Outcome**: Enhanced learning experiences, increased student engagement, and streamlined administrative tasks.

These examples demonstrate how custom tools within the Foundry Agent Service can be used across industries to address unique challenges, drive efficiency, and deliver value.

Next unit: Options for implementing custom tools

Previous Next

Options for implementing custom tools

6 minutes

Microsoft Foundry Agent Service offers various custom tools that enhance the capabilities and efficiency of your AI agents. These tools allow for scalable interoperability with various applications, making it easier to integrate with existing infrastructure or web services.

Custom tool options available in Microsoft Foundry Agent Service

Foundry Agent Service provides several custom tool options, including OpenAPI specified tools, Azure Functions, and function calling. These tools enable seamless integration with external APIs, event-driven applications, and custom functions.

- **Custom function**: Function calling allows you to describe the structure of custom functions to an agent and return the functions that need to be called along with their arguments. The agent can dynamically identify appropriate functions based on their definitions. This feature is useful for integrating custom logic and workflows, in a selection of programming languages, into your AI agents.
- **Azure Functions**: Azure Functions enable you to create intelligent, event-driven applications with minimal overhead. They support triggers and bindings, which simplify how your AI Agents interact with external systems and services. Triggers determine when a function executes, while bindings facilitate streamlined connections to input or output data sources.
- **OpenAPI specification tools**: These tools allow you to connect your Azure AI Agent to an external API using an OpenAPI 3.0 specification. This provides standardized, automated, and scalable API integrations that enhance the capabilities of your agent. OpenAPI specifications describe HTTP APIs, enabling people to understand how an API works, generate client code, create tests, and apply design standards.
- **Azure Logic Apps**: This action provides low-code/no-code solutions to add workflows and connects apps, data, and services with the low-code Logic App.

This flexibility to integrate custom functionality in multiple ways enables a wide range of extensibility possibilities for your Foundry Agent Service agents.

Next unit: How to integrate custom tools

< Previous Next >

How to integrate custom tools

7 minutes

Custom tools in an agent can be defined in a handful of ways, depending on what works best for your scenario. You may find that your company already has Azure Functions implemented for your agent to use, or a public OpenAPI specification gives your agent the functionality you're looking for.

Function Calling

Function calling allows agents to execute predefined functions dynamically based on user input. This feature is ideal for scenarios where agents need to perform specific tasks, such as retrieving data or processing user queries, and can be done in code from within the agent. Your function may call out to other APIs to get additional information or initiate a program.

Example: Defining and using a function

Start by defining a function that the agent can call. For instance, here's a fake snowfall tracking function:

Python

```python
import json

def recent_snowfall(location: str) -> str:
    """
    Fetches recent snowfall totals for a given location.
    :param location: The city name.
    :return: Snowfall details as a JSON string.
    """
    mock_snow_data = {"Seattle": "0 inches", "Denver": "2 inches"}
    snow = mock_snow_data.get(location, "Data not available.")
    return json.dumps({"location": location, "snowfall": snow})

user_functions: Set[Callable[..., Any]] = {
    recent_snowfall,
}
```

Register the function with your agent using the Azure AI SDK:

Python

```
# Initialize agent toolset with user functions
functions = FunctionTool(user_functions)
toolset = ToolSet()
toolset.add(functions)
agent_client.enable_auto_function_calls(toolset=toolset)

# Create your agent with the toolset
agent = agent_client.create_agent(
    model="gpt-4o-mini",
    name="snowfall-agent",
    instructions="You are a weather assistant tracking snowfall. Use the provided func-
tions to answer questions.",
    toolset=toolset
)
```

The agent can now call *recent_snowfall* dynamically when it determines that the prompt requires information that can be retrieved by the function.

Azure Functions

Azure Functions provide serverless computing capabilities for real-time processing. This integration is ideal for event-driven workflows, enabling agents to respond to triggers such as HTTP requests or queue messages.

Example: Using Azure Functions with a queue trigger

First, develop and deploy your Azure Function. In this example, imagine we have a function in our Azure subscription to fetch the snowfall for a given location.

When your Azure Function is in place, integrate add it to the agent definition as an Azure Function tool:

Python

```
storage_service_endpoint = "https://<your-storage>.queue.core.windows.net"

azure_function_tool = AzureFunctionTool(
    name="get_snowfall",
    description="Get snowfall information using Azure Function",
    parameters={
```

```
            "type": "object",
            "properties": {
                "location": {"type": "string", "description": "The location to check
snowfall."},
            },
            "required": ["location"],
        },
    input_queue=AzureFunctionStorageQueue(
        queue_name="input",
        storage_service_endpoint=storage_service_endpoint,
    ),
    output_queue=AzureFunctionStorageQueue(
        queue_name="output",
        storage_service_endpoint=storage_service_endpoint,
    ),
)

agent = agent_client.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="azure-function-agent",
    instructions="You are a snowfall tracking agent. Use the provided Azure Function to
fetch snowfall based on location.",
    tools=azure_function_tool.definitions,
)
```

The agent can now send requests to the Azure Function via a storage queue and process the results.

OpenAPI Specification

OpenAPI defined tools allow agents to interact with external APIs using standardized specifications. This approach simplifies API integration and ensures compatibility with various services. The Foundry Agent Service uses OpenAPI 3.0 specified tools.

> **Tip**
>
> Currently, three authentication types are supported with OpenAPI 3.0 tools: *anonymous*, *API key*, and *managed identity*.

Example: Using an OpenAPI specification

First, create a JSON file (in this example, called *snowfall_openapi.json*) describing the API.

JSON

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Snowfall API",
    "version": "1.0.0"
  },
  "paths": {
    "/snow": {
      "get": {
        "summary": "Get snowfall information",
        "parameters": [
          {
            "name": "location",
            "in": "query",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "Successful response",
            "content": {
              "application/json": {
                "schema": {
                  "type": "object",
                  "properties": {
                    "location": {"type": "string"},
                    "snow": {"type": "string"}
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

Then, register the OpenAPI tool in the agent defintion:

Python

```
from azure.ai.agents.models import OpenApiTool, OpenApiAnonymousAuthDetails

with open("snowfall_openapi.json", "r") as f:
```

```
    openapi_spec = json.load(f)

auth = OpenApiAnonymousAuthDetails()
openapi_tool = OpenApiTool(name="snowfall_api", spec=openapi_spec, auth=auth)

agent = agent_client.create_agent(
    model="gpt-4o-mini",
    name="openapi-agent",
    instructions="You are a snowfall tracking assistant. Use the API to fetch snowfall
data.",
    tools=[openapi_tool]
)
```

The agent can now use the OpenAPI tool to fetch snowfall data dynamically.

> **Note**
>
> One of the concepts related to agents and custom tools that developers often have difficulty with is the *declarative* nature of the solution. You don't need to write code that explicitly *calls* your custom tool functions - the agent itself decides to call tool functions based on messages in prompts. By providing the agent with functions that have meaningful names and well-documented parameters, the agent can "figure out" when and how to call the function all by itself!

By using one of the available custom tool options (or any combination of them), you can create powerful, flexible, and intelligent agents with Foundry Agent Service. These integrations enable seamless interaction with external systems, real-time processing, and scalable workflows, making it easier to build custom solutions tailored to your needs.

Next unit: Exercise - Build an agent with custom tools

Previous Next

100 XP

Exercise - Build an agent with custom tools

30 minutes

Now it's your opportunity to build an agent with custom tools. In this exercise, you create an agent in code and connect the tool definition to a custom tool function.

> **Note**
>
> If you don't have an Azure subscription, and you want to explore Microsoft Foundry, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

> **Tip**
>
> After completing the exercise, if you're finished exploring Azure AI Agents, delete the Azure resources that you created during the exercise.

Next unit: Module assessment

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 Microsoft Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Microsoft Foundry SDK client libraries for details.

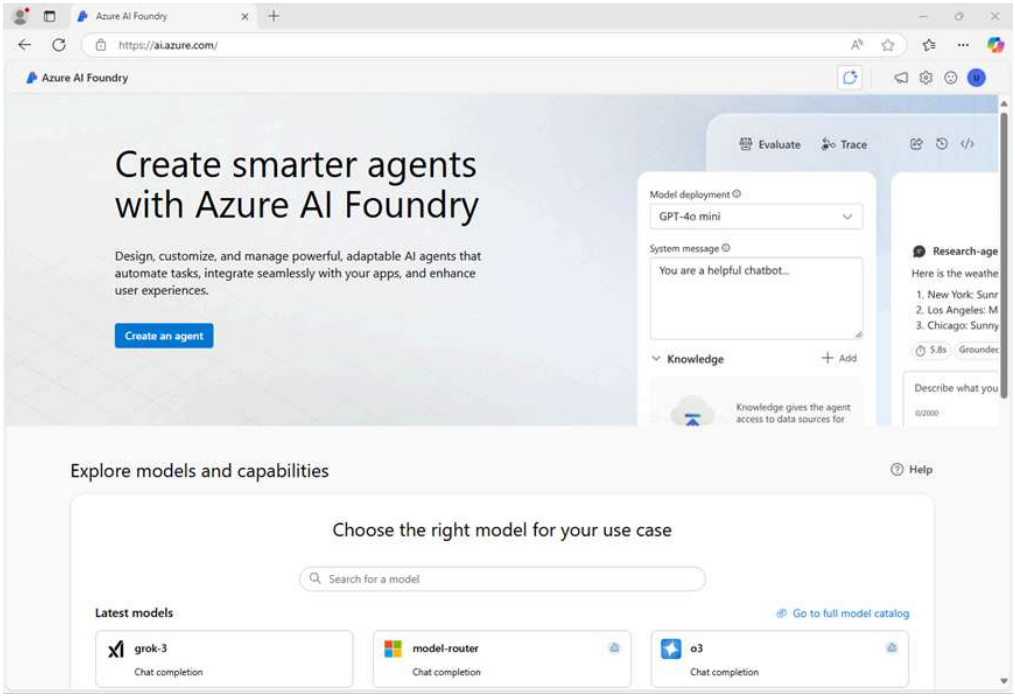
This exercise should take approximately **30** minutes to complete.

> **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 a Foundry project

Let's start by creating a Foundry project.

1. In a web browser, open the 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 **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):



> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

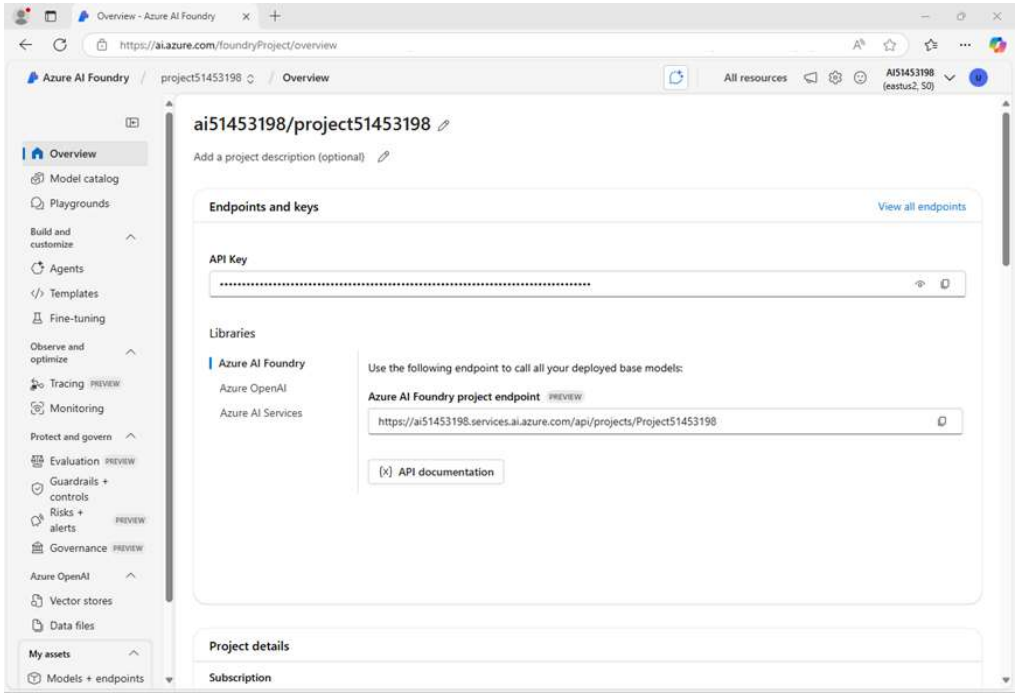
2. In the home page, select **Create an agent**.
3. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
4. Confirm the following settings for your project:
    - **Foundry resource**: *A valid name for your Foundry resource*
    - **Subscription**: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

> **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 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 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:

```
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.

200 XP

Module assessment

3 minutes

1. What are custom tools, and how can they help you develop effective agents with Microsoft Foundry Agent Service?

- Callable functions that an agent can use to extend its capabilities.
- Extensions for Visual Studio Code that make it easier to create and deploy agents.
- Fine-tuned models that the agent can use to generate custom output.

2. You need to integrate functionality from an OpenAPI 3.0-based web service into an agent solution. What should you do?

- Add the JSON schema of the web service to the agent's instructions.
- Rewrite the web service as a Python function and hard-code it in your agent app.
- Add the web service as an OpenAPI specification tool to the agent definition

3. Your agent application code includes a local function that you want the agent to call. What kind of tool should you add to the agent's definition?

- Function calling
- Code interpreter
- Azure Functions

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What are custom tools, and how can they help you develop effective agents with Microsoft Foundry Agent Service?

- ◯ Callable functions that an agent can use to extend its capabilities. ✓ Correct
- ◯ Extensions for Visual Studio Code that make it easier to create and deploy agents.
- ◯ Fine-tuned models that the agent can use to generate custom output.

2. You need to integrate functionality from an OpenAPI 3.0-based web service into an agent solution. What should you do?

- ◯ Add the JSON schema of the web service to the agent's instructions.
- ◯ Rewrite the web service as a Python function and hard-code it in your agent app.
- ◯ Add the web service as an OpenAPI specification tool to the agent definition ✓ Correct

3. Your agent application code includes a local function that you want the agent to call. What kind of tool should you add to the agent's definition?

- ◯ Function calling ✓ Correct

- Code interpreter
- Azure Functions

Next unit: Summary

Previous Next

Summary

2 minutes

In this module, we covered the benefits of integrating custom tools into Foundry Agent Service to boost productivity and provide tailored business solutions. By providing custom tools to our agent, we can optimize processes to meet specific needs, resulting in better responses from your agent.

The techniques learned in this module enable businesses to generate marketing materials, improve communications, and analyze market trends more effectively, all through custom tools. The integration of various tool options in the AI Agent Service, from Azure Functions to OpenAPI specifications, allows for the creation of intelligent, event-driven applications that use well-established patterns already used in many businesses.

Further reading

- AI Agents for beginners tool use
- Microsoft Foundry Agent Service function calling
- Introduction to Azure Functions
- OpenAPI Specification

All units complete:

Previous

Complete module

Introduction

2 minutes

AI agents offer a powerful combination of technologies, able to complete tasks with the use of generative AI. However, in some situations, the task required might be larger than is realistic for a single agent. For those scenarios, consider a **multi-agent** solution. A multi-agent solution allows agents to collaborate within the same conversation.

Imagine you're trying to address common DevOps challenges such as monitoring application performance, identifying issues, and deploying fixes. A multi-agent system could consist of four specialized agents working collaboratively:

- The Monitoring Agent continuously ingests logs and metrics, detects anomalies using natural language processing (NLP), and triggers alerts when issues arise.
- The Root Cause Analysis Agent then correlates these anomalies with recent system changes, using machine learning models or predefined rules to pinpoint the root cause of the problem.
- Once the root cause is identified, the Automated Deployment Agent takes over to implement fixes or roll back problematic changes by interacting with CI/CD pipelines and executing deployment scripts.
- Finally, the Reporting Agent generates detailed reports summarizing the anomalies, root causes, and resolutions, and notifies stakeholders via email or other communication channels.

This modular, scalable, and intelligent multi-agent system streamlines the DevOps process. The agents collaborate to reduce manual intervention and improve efficiency while ensuring timely communication and resolution of issues.

In this module, you'll explore how to use the powerful capabilities of the Microsoft Agent Framework to design and orchestrate intelligent agents that work collaboratively to solve complex problems. You'll also learn about the different types of orchestration patterns available, and use the Microsoft Agent Framework to develop your own AI agents that can collaborate for a multi-agent solution.

After completing this module, you'll be able to:

- Build AI agents using the Microsoft Agent Framework SDK
- Use tools and plugins with your AI agents
- Understand different types of orchestration patterns
- Develop multi-agent solutions

Next unit: Understand the Microsoft Agent Framework

Next >

Understand the Microsoft Agent Framework

6 minutes

The Microsoft Agent Framework is an open-source SDK that enables developers to integrate AI models into their applications. This framework provides comprehensive support for creating AI-powered agents that can work independently or collaborate with other agents to accomplish complex tasks.

What is the Microsoft Agent Framework?

The Microsoft Agent Framework is designed to help developers build AI-powered agents that can process user inputs, make decisions, and execute tasks autonomously by leveraging large language models and traditional programming logic. The framework provides structured components for defining AI-driven workflows, enabling agents to interact with users, APIs, and external services seamlessly.

Core concepts

The Microsoft Agent Framework provides a flexible architecture with the following key components:

- **Agents**

  Agents are intelligent, AI-driven entities capable of reasoning and executing tasks. They use large language models, tools, and conversation history to make decisions dynamically and respond to user needs.

- **Agent orchestration**

  Multiple agents can collaborate towards a common goal using different orchestration patterns. The Microsoft Agent Framework supports several orchestration patterns with a unified interface for construction and invocation, allowing you to easily switch between patterns without rewriting your agent logic.

The framework includes several core features that power agent functionality:

- **Chat clients**

  Chat clients provide abstractions for connecting to AI services from different providers under a common interface. Supported providers include Azure OpenAI, OpenAI, Anthropic, and more through the `BaseChatClient` abstraction.

- **Tools and function integration**

  Tools enable agents to extend their capabilities through custom functions and built-in services. Agents can automatically invoke tools to integrate with external APIs, execute code, search files, or access web information. The framework supports both custom function tools and built-in tools like Code Interpreter, File Search, and Web Search.

- **Conversation management**

  Agents can maintain conversation history across multiple interactions using `AgentThread`, allowing them to track previous interactions and adapt responses accordingly. The structured message system uses roles (USER, ASSISTANT, SYSTEM, TOOL) for persistent conversation context.

Types of agents

The Microsoft Agent Framework supports several different types of agents from multiple providers:

- **Microsoft Foundry Agent -** a specialized agent within the Microsoft Agent Framework designed to provide enterprise-grade conversational capabilities with seamless tool integration. It automatically handles tool calling and securely manages conversation history using threads, reducing the overhead of maintaining state. Microsoft Foundry Agents support built-in tools and provide integration capabilities for Azure AI Search, Azure Functions, and other Azure services.

- **ChatAgent**: designed for general conversation and task completion interfaces. The `ChatAgent` type provides natural language processing, contextual understanding, and dialogue management with support for custom tools and instructions.

- **OpenAI Assistant Agent**: designed for advanced capabilities using OpenAI's Assistant API. This agent type supports goal-driven interactions with features like code interpretation and file search through the OpenAI platform.

- **Anthropic Agent**: provides access to Anthropic's Claude models with the framework's unified interface, supporting advanced reasoning and conversation capabilities.

Why you should use the Microsoft Agent Framework

The Microsoft Agent Framework offers a robust platform for building intelligent, autonomous, and collaborative AI agents. The framework can integrate agents from multiple sources, including Microsoft Foundry Agent Service, and supports both multi-agent collaboration and human-agent interaction. Agents can work together to orchestrate sophisticated workflows, where each agent specializes in a specific task, such as data collection, analysis, or decision-making. The framework also facilitates human-in-the-loop processes, enabling agents to augment human decision-making by providing insights or automating repetitive tasks. The provider-agnostic design allows you to switch between different AI providers without changing your code, making it suitable for building adaptable AI systems from simple chatbots to complex enterprise solutions.

Next unit: Understand agent orchestration

Previous Next

Understand agent orchestration

5 minutes

The Microsoft Agent Framework SDK's agent orchestration framework makes it possible to design, manage, and scale complex multi-agent workflows without having to manually handle the details of agent coordination. Instead of relying on a single agent to manage every aspect of a task, you can combine multiple specialized agents. Each agent with a unique role or area of expertise can collaborate to create systems that are more robust, adaptive, and capable of solving real-world problems collaboratively.

By orchestrating agents together, you can take on tasks that would be too complex for a single agent—from running parallel analyses, to building multi-stage processing pipelines, to managing dynamic, context-driven handoffs between experts.

Why multi-agent orchestration matters

Single-agent systems are often limited in scope, constrained by one set of instructions or a single model prompt. Multi-agent orchestration addresses this limitation by allowing you to:

- Assign distinct skills, responsibilities, or perspectives to each agent.
- Combine outputs from multiple agents to improve decision-making and accuracy.
- Coordinate steps in a workflow so each agent's work builds on the last.
- Dynamically route control between agents based on context or rules.

This approach opens the door to more flexible, efficient, and scalable solutions, especially for real-world applications that require collaboration, specialization, or redundancy.

Understand workflows in the Microsoft Agent Framework

The Microsoft Agent Framework provides **workflows —** structured sequences of steps used to complete a task. These workflows can include one or more **AI agents** alongside other components to automate complex operations.

Workflows give developers control over how tasks are executed, enable **multi-agent orchestration**, and support **checkpointing** to save and resume workflow states.

Core Components of a Workflow

Executors

Executors are the main workers in a workflow. They receive input messages, perform specific actions, and produce outputs that move the workflow toward completing its goal.
Executors can represent **AI agents** or **custom logic** components.

Example: One executor could analyze a travel request, while another books the flight or hotel based on the results.

Edges

Edges define how messages flow between executors, determining the logic and order of execution. The Microsoft Agent Framework supports several types of edges:

- **Direct Edges:** Connect one executor directly to another in sequence.
  Example: After an AI agent gathers user input, the next executor processes the booking.*
- **Conditional Edges:** Trigger only when certain conditions are met.
  Example: If hotel rooms are unavailable, the workflow branches to an executor that suggests alternative dates or locations.*
- **Switch-Case Edges:** Route messages to different executors based on predefined conditions.
  Example: VIP customers might be routed to a premium service executor, while others follow the standard process.*
- **Fan-Out Edges:** Send a single message to multiple executors simultaneously.
  Example: One request could be sent to several agents — one checking flights, another checking hotels.*
- **Fan-In Edges:** Combine multiple messages from different executors into one for a final step.
  Example: After gathering hotel and flight results, a summary executor compiles them into a single travel itinerary.*

Events

The Microsoft Agent Framework includes built-in events to improve **observability** and **debugging** during workflow execution. These events help developers monitor progress, track errors, and analyze system performance.

Expand table

Event Name	Description
WorkflowStartedEvent	Triggered when workflow execution begins.
WorkflowOutputEvent	Emitted when the workflow produces an output.
WorkflowErrorEvent	Occurs when an error is encountered.
ExecutorInvokeEvent	Fired when an executor starts processing a task.
ExecutorCompleteEvent	Fired when an executor finishes its work.
RequestInfoEvent	Logged when an external request is issued.

Workflows in the Microsoft Agent Framework allow developers to **design, monitor, and control** how multiple AI agents and logic components interact to complete complex tasks. They bring structure, flexibility, and transparency to agent-driven applications.

Supported orchestration patterns

Microsoft Agent Framework provides several orchestration patterns directly in the SDK, each offering a different approach to coordinating agents. These patterns are designed to be technology-agnostic so you can adapt them to your own domain and integrate them into your existing systems.

- **Concurrent orchestration -** Broadcast the same task to multiple agents at once and collect their results independently. Useful for parallel analysis, independent subtasks, or ensemble decision making.
- **Sequential orchestration -** Pass the output from one agent to the next in a fixed order. Ideal for step-by-step workflows, pipelines, and progressive refinement.

- **Handoff orchestration -** Dynamically transfer control between agents based on context or rules. Great for escalation, fallback, and expert routing where one agent works at a time.
- **Group chat orchestration -** Coordinate a shared conversation among multiple agents (and optionally a human), managed by a chat manager that chooses who speaks next. Best for brainstorming, collaborative problem solving, and building consensus.
- **Magentic orchestration -** A manager-driven approach that plans, delegates, and adapts across specialized agents. Suited to complex, open-ended problems where the solution path evolves.

A unified orchestration workflow

Regardless of which orchestration pattern you choose, the Microsoft Agent Framework SDK provides a consistent, developer-friendly interface for building and running them. The typical flow looks like this:

1. **Define your agents** and describe their capabilities.
2. **Select and create an orchestration** pattern, optionally adding a manager agent if needed.
3. **Optionally configure callbacks or transforms** for custom input and output handling.
4. **Start a runtime** to manage execution.
5. **Invoke the orchestration** with your task.
6. **Retrieve results** in an asynchronous, nonblocking way.

Because all patterns share the same core interface, you can easily experiment with different orchestration strategies without rewriting agent logic or learning new APIs. The SDK abstracts the complexity of agent communication, coordination, and result aggregation so you can focus on designing workflows that deliver results.

Multi-agent orchestration in the Microsoft Agent Framework SDK provides a flexible, scalable way to build intelligent systems that combine the strengths of multiple specialized agents. With built-in orchestration patterns, a unified development model, and runtime features for managing execution, you can quickly prototype, refine, and deploy collaborative AI workflows. The framework provides the tools to turn multiple agents into a cohesive problem-solving team, whether you're running parallel processes, sequential workflows, or dynamic conversations.

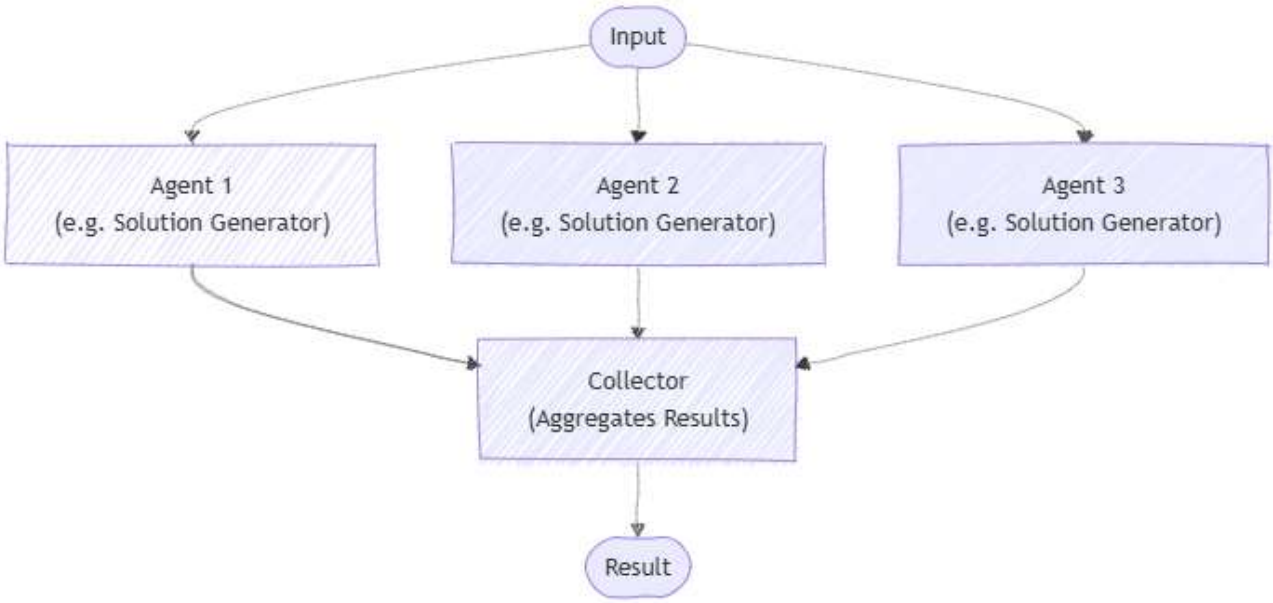
Next unit: Use concurrent orchestration

Previous Next

Use concurrent orchestration

5 minutes

Concurrent orchestration lets multiple agents work on the same task at the same time. Each agent handles the task independently, and then their outputs are gathered and combined. This method works especially well when you want diverse approaches or solutions, like during brainstorming, group decision-making, or voting.



This pattern is useful when you need different approaches or ideas to solve the same problem. Instead of having agents work one after another, they all work at the same time. This speeds up the process and covers the problem from many angles.

Usually, the results from each agent are combined to create a final answer, but this isn't always necessary. Each agent can also produce its own separate result, like calling tools to complete tasks or updating different data stores independently.

Agents work on their own and don't share results with each other. However, an agent can call other AI agents by running its own orchestration as part of its process. Agents need to know which other agents are available to work on tasks. This pattern allows you to either call all registered agents every time or choose which agents to run based on the specific task.

When to use concurrent orchestration

You may want to consider using the concurrent orchestration pattern in these situations:

- When tasks can run at the same time, either by using a fixed group of agents or by selecting AI agents dynamically based on what the task needs.
- When the task benefits from different specialized skills or approaches (for example, technical, business, or creative) that all work independently but contribute to solving the same problem.

This kind of teamwork is common in multi-agent decision-making methods such as:

- Brainstorming ideas
- Combining different reasoning methods (ensemble reasoning)
- Making decisions based on voting or consensus (quorum)
- Handling tasks where speed matters and running agents in parallel cuts down wait time

When to avoid concurrent orchestration

You may want to avoid using the concurrent orchestration pattern in the following scenarios:

- Agents need to build on each other's work or depend on shared context in a specific order.
- The task requires a strict sequence of steps or predictable, repeatable results.
- Resource limits, like model usage quotas, make running agents in parallel inefficient or impossible.
- Agents can't reliably coordinate changes to shared data or external systems while running at the same time.
- There's no clear way to resolve conflicts or contradictions between results from different agents.
- Combining results is too complicated or ends up lowering the overall quality.

Implement concurrent orchestration

Implement the concurrent orchestration pattern with the Microsoft Agent Framework:

1. **Create your chat client**
   Set up a chat client (for example, `AzureOpenAIChatClient`) with appropriate credentials to connect to your AI service provider.

2. **Define your agents**
   Create agent instances using the chat client's `create_agent` method. Each agent should have specific instructions and a name that defines its role and expertise area.

3. **Build the concurrent workflow**
   Use the `ConcurrentBuilder` class to create a workflow that can run multiple agents in parallel. Add your agent instances as participants using the `participants()` method, then call `build()` to create the workflow.

4. **Run the workflow**
   Call the workflow's `run` method with the task or input you want the agents to work on. The workflow runs all agents concurrently and returns events containing the results.

5. **Process the results**
   Extract the outputs from the workflow events using `get_outputs()`. The results contain the combined conversations from all agents, with each agent's response included in the final output.

6. **Handle the aggregated responses**
   Process the aggregated messages from all agents. Each message includes the author name and content, allowing you to identify which agent provided each response.

Concurrent orchestration is a powerful pattern for using multiple AI agents simultaneously, enabling faster and more diverse problem-solving. By running agents in parallel, you can explore different approaches at once, improve efficiency, and gain richer insights. However, it's important to choose this pattern when tasks can truly run independently and to be mindful of resource constraints and coordination challenges. When implemented thoughtfully with the Microsoft Agent Framework SDK, concurrent orchestration can greatly enhance your AI workflows and decision-making processes.

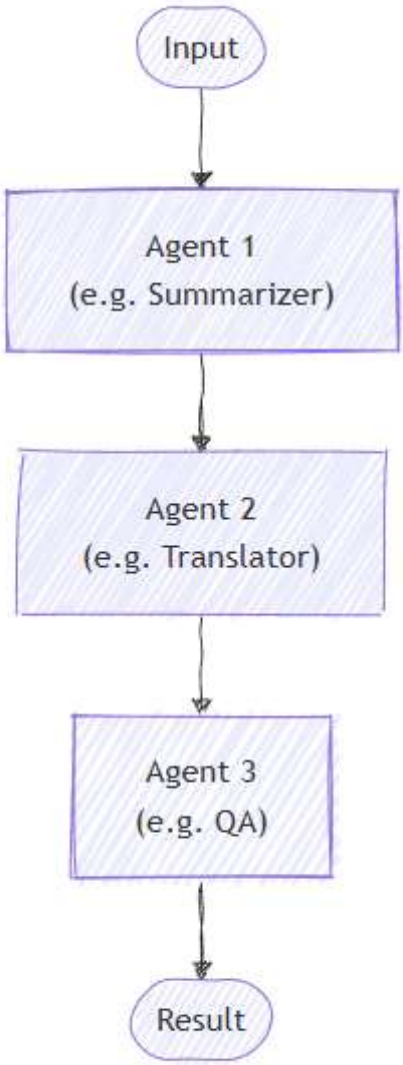
Next unit: Use sequential orchestration

Previous Next

Use sequential orchestration

5 minutes

In **sequential orchestration**, agents are arranged in a pipeline where each agent processes the task one after another. The output from one agent becomes the input for the next. This pattern is ideal for workflows where each step depends on the previous one, such as document review, data transformation pipelines, or multi-stage reasoning.



Sequential orchestration works best for tasks that need to be done step-by-step, with each step improving on the last. The order in which agents run is fixed and decided beforehand, and agents

don't decide what happens next.

When to use sequential orchestration

Consider using the sequential orchestration pattern when your workflow has:

- Processes made up of multiple steps that must happen in a specific order, where each step relies on the one before it.
- Data workflows where each stage adds something important that the next stage needs to work properly.
- Tasks where stages can't be done at the same time and must run one after another.
- Situations that require gradual improvements, like drafting, reviewing, and polishing content.
- Systems where you know how each agent performs and can handle delays or failures in any step without stopping the whole process.

When to avoid sequential orchestration

Avoid this pattern when:

- Stages can be run independently and in parallel without affecting quality.
- A single agent can perform the entire task effectively.
- Early stages may fail or produce poor output, and there's no way to stop or correct downstream processing based on errors.
- Agents need to collaborate dynamically rather than hand off work sequentially.
- The workflow requires iteration, backtracking, or dynamic routing based on intermediate results.

Implement sequential orchestration

Implement the sequential orchestration pattern with the Microsoft Agent Framework SDK:

1. **Create your chat client**
   Set up a chat client (for example, `AzureOpenAIChatClient`) with appropriate credentials to connect to your AI service provider.

2. **Define your agents**
   Create agent instances using the chat client's `create_agent` method. Each agent should have

specific instructions and a name that defines its role and expertise area in the pipeline.

3. **Build the sequential workflow**
   Use the `SequentialBuilder` class to create a workflow that executes agents one after another. Add your agent instances as participants using the `participants()` method, then call `build()` to create the workflow.

4. **Run the workflow**
   Call the workflow's `run_stream` method with the task or input you want the agents to work on. The workflow processes the task through all agents sequentially, with each agent's output becoming input for the next.

5. **Process the workflow events**
   Iterate through the workflow events using an async loop. Look for `WorkflowOutputEvent` instances, which contain the results from the sequential processing.

6. **Extract the final conversation**
   Collect the final conversation from the workflow outputs. The result contains the complete conversation history showing how each agent in the sequence contributed to the final outcome.

Sequential orchestration is ideal when your task requires clear, ordered steps where each agent builds on the previous one's output. This pattern helps improve output quality through stepwise refinement and ensures predictable workflows. When applied thoughtfully with the Microsoft Agent Framework SDK, it enables powerful multi-agent pipelines for complex tasks like content creation, data processing, and more.

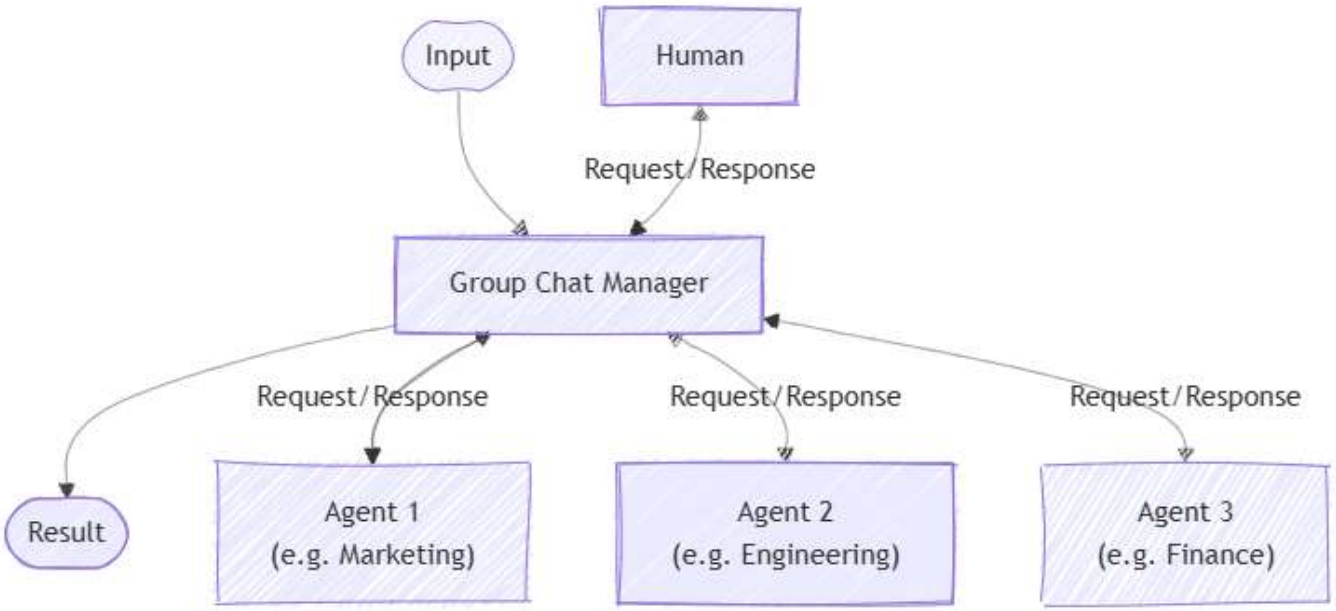
Next unit: Use group chat orchestration

Previous Next

Use group chat orchestration

5 minutes

Group chat orchestration models a collaborative conversation among multiple AI agents, and optionally a human participant. A central chat manager controls the flow, deciding which agent responds next and when to request human input. This pattern is useful for simulating meetings, debates, or collaborative problem-solving.



The group chat pattern works well for scenarios where group discussion or iterative collaboration is key to reaching decisions. It supports different interaction styles, from free-flowing ideation to formal workflows with defined roles and approval steps. Group chat orchestration is also great for human-in-the-loop setups where a human may guide or intervene in the conversation. Typically, agents in this pattern don't directly change running systems—they mainly contribute to the conversation.

When to use group chat orchestration

Consider using group chat orchestration when your scenario involves:

- Spontaneous or guided collaboration among agents (and possibly humans)

- Iterative maker-checker loops where agents take turns creating and reviewing
- Real-time human oversight or participation
- Transparent and auditable conversations since all output is collected in a single thread

Common scenarios include:

- Creative brainstorming where agents build on each other's ideas
- Decision-making that benefits from debate and consensus
- Complex problems requiring cross-disciplinary dialogue
- Quality control and validation requiring multiple expert perspectives
- Content workflows with clear separation between creation and review

When to avoid group chat orchestration

Avoid this pattern when:

- Simple task delegation or straightforward linear pipelines suffice
- Real-time speed requirements make discussion overhead impractical
- Hierarchical or deterministic workflows are needed without discussion
- The chat manager can't clearly determine when the task is complete
- Managing conversation flow becomes too complex, especially with many agents (limit to three or fewer for easier control)

Maker-checker loops

A common special case is the maker-checker loop. Here, one agent (the maker) proposes content or solutions, and another agent (the checker) reviews and critiques them. The checker can send feedback back to the maker, and this cycle repeats until the result is satisfactory. This process requires a turn-based sequence managed by the chat manager.

Implement group chat orchestration

Implement the group chat orchestration pattern with the Microsoft Agent Framework SDK:

1. **Create your chat client**
   Set up a chat client (for example, `AzureOpenAIChatClient`) with appropriate credentials to connect to your AI service provider.

2. **Define your agents**
   Create agent instances using the chat client's `create_agent` method. Each agent should have specific instructions and a name that defines its role and expertise area.

3. **Build the group chat workflow**
   Use the `GroupChatBuilder` class to create a workflow that can run multiple agents in parallel. Add your agent instances as participants using the `participants()` method, then call `build()` to create the workflow.

4. **Run the workflow**
   Call the workflow's `run` method with the task or input you want the agents to work on. The workflow runs all agents concurrently and returns events containing the results.

5. **Process the results**
   Extract the outputs from the workflow events using `get_outputs()`. The results contain the combined conversations from all agents, with each agent's response included in the final output.

6. **Handle the aggregated responses**
   Process the aggregated messages from all agents. Each message includes the author name and content, allowing you to identify which agent provided each response.

Customizing the group chat manager

You can create a custom group chat manager by extending the base `GroupChatManager` class. This approach lets you control:

- How conversation results are filtered or summarized
- How the next agent is selected
- When to request user input
- When to terminate the conversation

Custom managers let you implement specialized logic tailored to your use case.

Group chat manager call order

During each round of the conversation, the chat manager calls methods in this order:

1. `should_request_user_input` - Checks if human input is needed before the next agent responds.
2. `should_terminate` - Determines if the conversation should end (for example, max rounds reached).
3. `filter_results` - If ending, summarizes or processes the final conversation.
4. `select_next_agent` - If continuing, chooses the next agent to speak.

This ensures user input and termination conditions are handled before moving the conversation forward. Override these methods in your custom manager to change behavior.

Group chat orchestration enables multiple AI agents—and optionally humans—to collaborate through guided conversation and iterative feedback. It's ideal for complex tasks that benefit from diverse expertise and dynamic interaction. While it requires careful management, this pattern offers transparency and flexibility in decision-making and creative workflows. The Microsoft Agent Framework SDK makes it easy to implement and customize group chat orchestration for your needs.

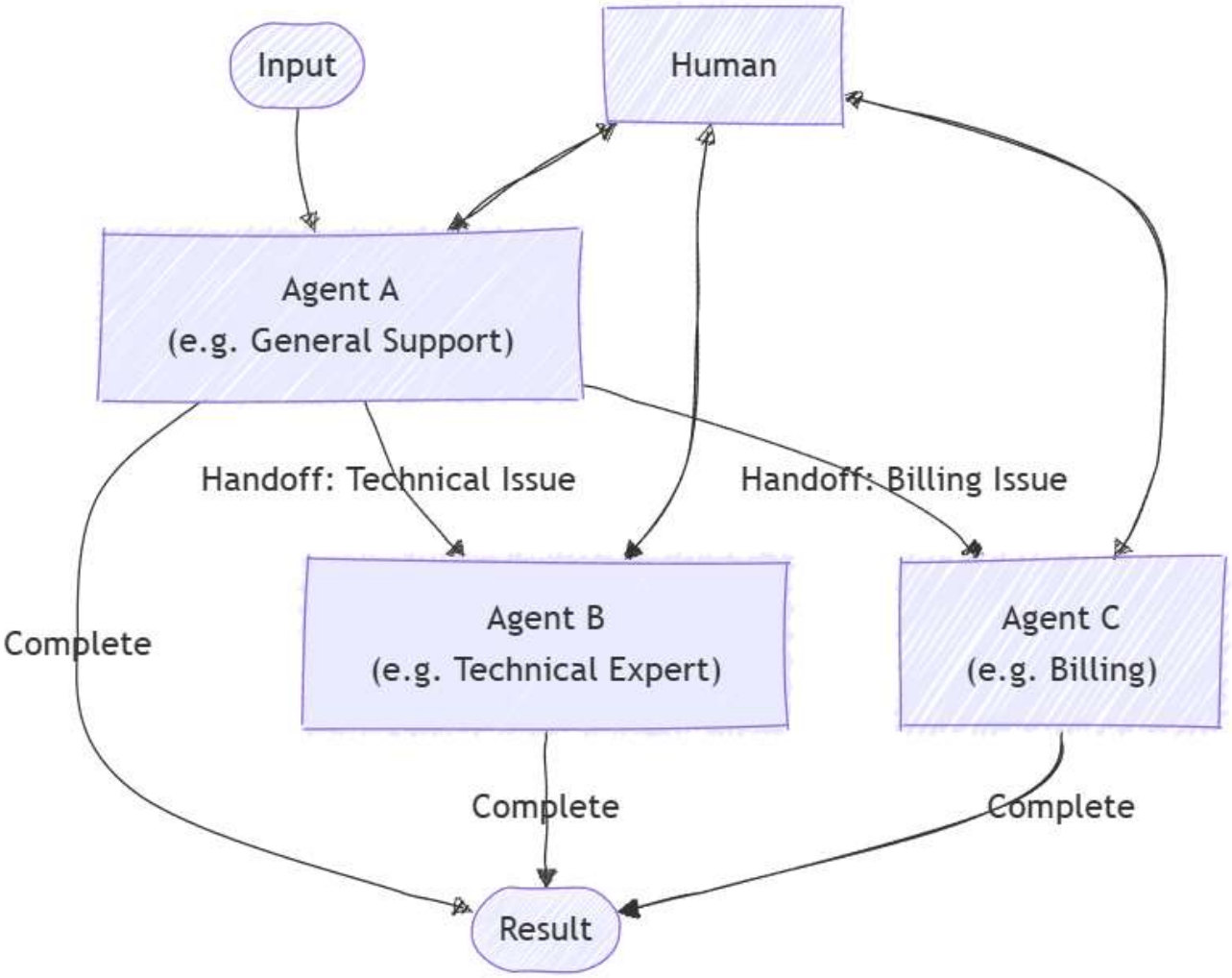
Next unit: Use handoff orchestration

Previous Next

Use handoff orchestration

5 minutes

Handoff orchestration lets AI agents transfer control to one another based on the task context or user requests. Each agent can "handoff" the conversation to another agent with the right expertise, making sure the best-suited agent handles each part of the task. This pattern is ideal for customer support, expert systems, or any situation where dynamic delegation is needed.



This pattern fits scenarios where the best agent isn't known upfront or where the task requirements become clearer during processing. Unlike parallel patterns, agents work one at a time, fully handing off control from one to the next.

When to use handoff orchestration

You may want to consider using the handoff orchestration pattern in these scenarios:

- Tasks need specialized knowledge or tools, but the number or order of agents can't be determined in advance.
- Expertise requirements emerge dynamically during processing, triggering task routing based on content analysis.
- Multiple-domain problems require different specialists working sequentially.
- You can define clear signals or rules indicating when an agent should transfer control and to whom.

When to avoid handoff orchestration

You may want to avoid using the handoff orchestration pattern in these scenarios:

- The involved agents and their order are known upfront and fixed.
- Task routing is simple and rule-based, not needing dynamic interpretation.
- Poor routing decisions might frustrate users.
- Multiple operations must run at the same time.
- Avoiding infinite handoff loops or excessive bouncing between agents is difficult.

Implementing handoff orchestration

The handoff orchestration pattern can be implemented in the Microsoft Agent Framework SDK using control workflows. In a control workflow, each agent processes the task in sequence, and based on its output, the workflow decides which agent to call next. This routing is done using a switch-case structure that routes the task to different agents based on classification results.

1. **Set up data models and chat client**

    - Create your chat client for connecting to AI services
    - Define Pydantic models for AI agents' structured JSON responses
    - Create simple data classes for passing information between workflow steps
    - Configure agents with specific instructions and `response_format` parameter for structured JSON output

2. **Create specialized executor functions**

- **Input storage executor** - saves incoming data to shared state and forwards to classification agent
- **Transformation executor** - converts agent's JSON response into typed routing object
- **Handler executors** - separate executors for each classification outcome with guard conditions to verify correct message processing

3. **Build routing logic**

- Create factory functions that generate condition checkers for each classification value
- Design conditions to examine incoming messages and return true for specific classification results
- Use conditions with Case objects in switch-case edge groups
- Always include a Default case as fallback for unexpected scenarios

4. **Assemble the workflow**

- Use WorkflowBuilder to connect executors with regular edges
- Add switch-case edge group for routing based on classification results
- Configure workflow to follow first matching case or fall back to default
- Set up terminal executor to yield final output

Handoff orchestration provides a flexible way to route tasks dynamically among specialized AI agents, ensuring that each part of a workflow is handled by the best-suited expert. It works well for complex, evolving tasks like customer support or multi-domain problem solving where expertise needs change during the conversation. When you use the Microsoft Agent Framework SDK, you can build adaptable systems that seamlessly transfer control between agents—and include human input when needed—for smooth and efficient task completion.

Next unit: Use Magentic orchestration

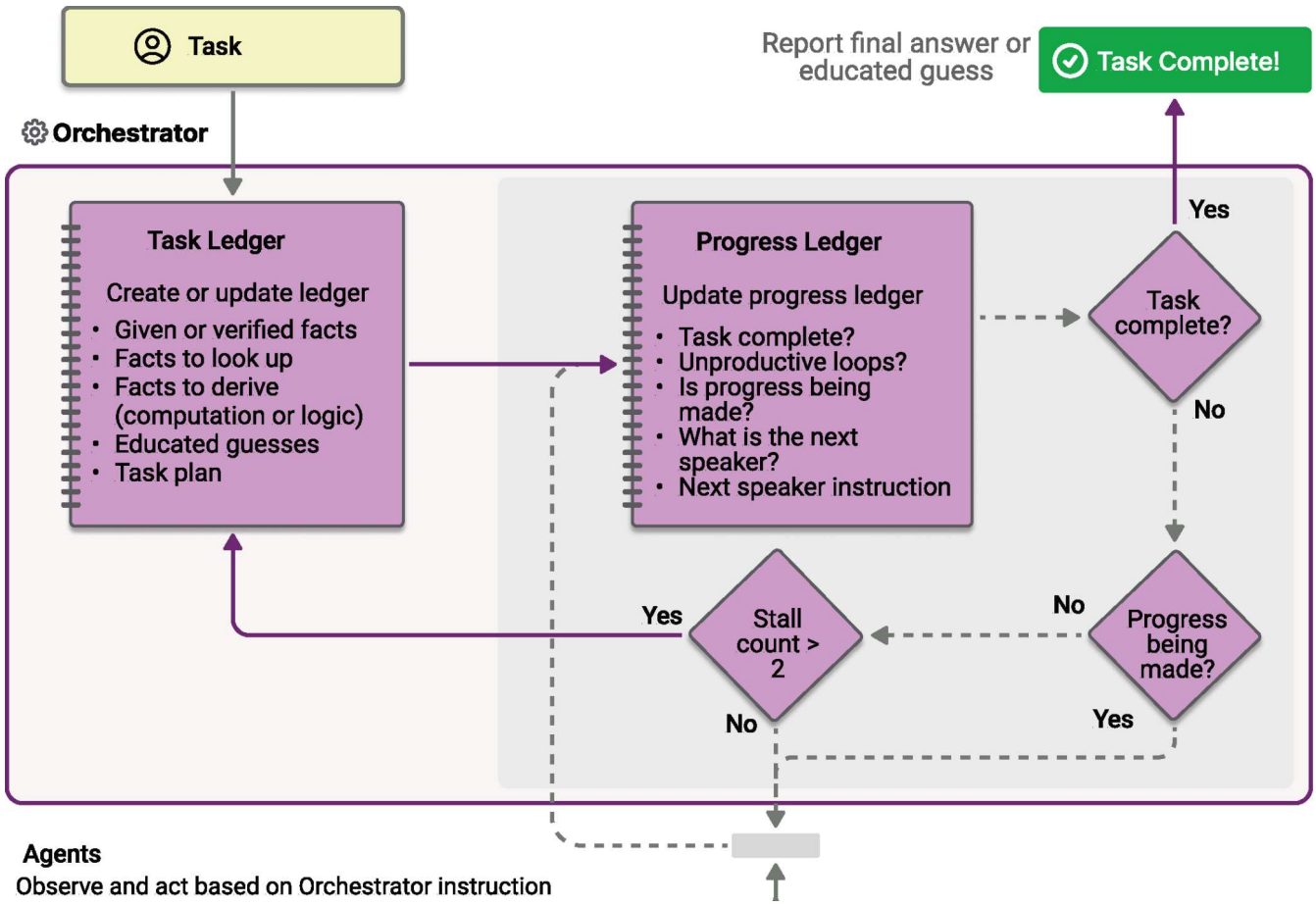
Previous Next

100 XP

Use Magentic orchestration

5 minutes

Magentic orchestration is a flexible, general-purpose multi-agent pattern designed for complex, open-ended tasks that require dynamic collaboration. This pattern uses a dedicated Magentic manager to coordinate a team of specialized agents. The manager decides which agent should act next based on the evolving context, task progress, and agent capabilities.



The Magentic manager maintains a shared context, tracks progress, and adapts the workflow in real time. This approach allows the system to break down complex problems, assign subtasks, and iteratively refine solutions. The process focuses as much on building and documenting the approach as it does on delivering the final solution. A dynamic task ledger is built and refined as the workflow progresses, recording goals, subgoals, and execution plans.

When to use Magentic orchestration

Consider using the Magentic orchestration pattern in these scenarios:

- The problem is complex or open-ended with no predetermined solution path.
- Input and feedback from multiple specialized agents are needed to shape a valid solution.
- The system must generate a documented plan of approach for human review.
- Agents have tools that can directly interact with external systems and resources.
- A step-by-step, dynamically built execution plan adds value before running the tasks.

When to avoid Magentic orchestration

You may want to avoid this pattern when:

- The solution path is fixed or deterministic.
- There's no need to produce a ledger or plan of approach.
- The task is simple enough for a more lightweight orchestration pattern.
- Speed is the priority, as this method emphasizes planning over fast execution.
- You expect frequent stalls or loops without a clear resolution path.

Implementing Magentic orchestration

Implement the Magentic orchestration pattern with the Microsoft Agent Framework:

1. **Define specialized agents**
   Create agent instances (for example, `ChatAgent`) with specific instructions and chat clients. Each agent should have a specialized role and capabilities suited for different aspects of the complex task.

2. **Set up event handling callback**
   Define an async callback function to handle different types of events during orchestration, including orchestrator messages, agent streaming updates, agent messages, and final results.

3. **Build the Magentic workflow**
   Use the `MagenticBuilder` class to create the orchestration. Add your agent instances as participants, configure the event callback with streaming mode, and set up the standard manager with appropriate parameters like max round count and stall limits.

4. **Configure the standard manager**
   The standard manager coordinates agent collaboration using a chat client for planning and

progress tracking. Configure parameters like maximum round count, stall count, and reset count to control the orchestration behavior.

5. **Run the workflow**
   Call the workflow's `run_stream` method with your complex task. The workflow dynamically plans, delegates work to appropriate agents, and coordinates their collaboration to solve the problem.

6. **Process workflow events**
   Iterate through the workflow events using an async loop. Handle different event types including `WorkflowOutputEvent`, which contains the final results from the orchestration.

7. **Extract the final result**
   Collect the final output from the workflow events. The result contains the complete solution developed through the collaborative effort of all participating agents.

Magentic orchestration excels at solving complex, evolving problems that require real-time coordination between specialized agents. It's ideal for tasks where the plan can't be defined in advance and must adapt as new information emerges. Using the Microsoft Agent Framework, you can build systems that dynamically design, refine, and execute solution paths through intelligent agent collaboration.

Next unit: Exercise - Develop a multi-agent solution

Previous Next

100 XP

Exercise - Develop a multi-agent solution

30 minutes

Now it's your opportunity to build a multi agent solution with the Microsoft Agent Framework. In this exercise, you create an application that automatically triages and resolves issues presented in log files of a system. Using Azure AI Agents, you create an incident manager agent and a devops agent that collaborates to fix the issues.

Launch the exercise and follow the instructions.

Launch Exercise

Tip

After completing the exercise, if you're finished exploring multi-agents with Microsoft Agent Framework, delete the Azure resources that you created during the exercise.

Next unit: Knowledge check

Previous Next

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!

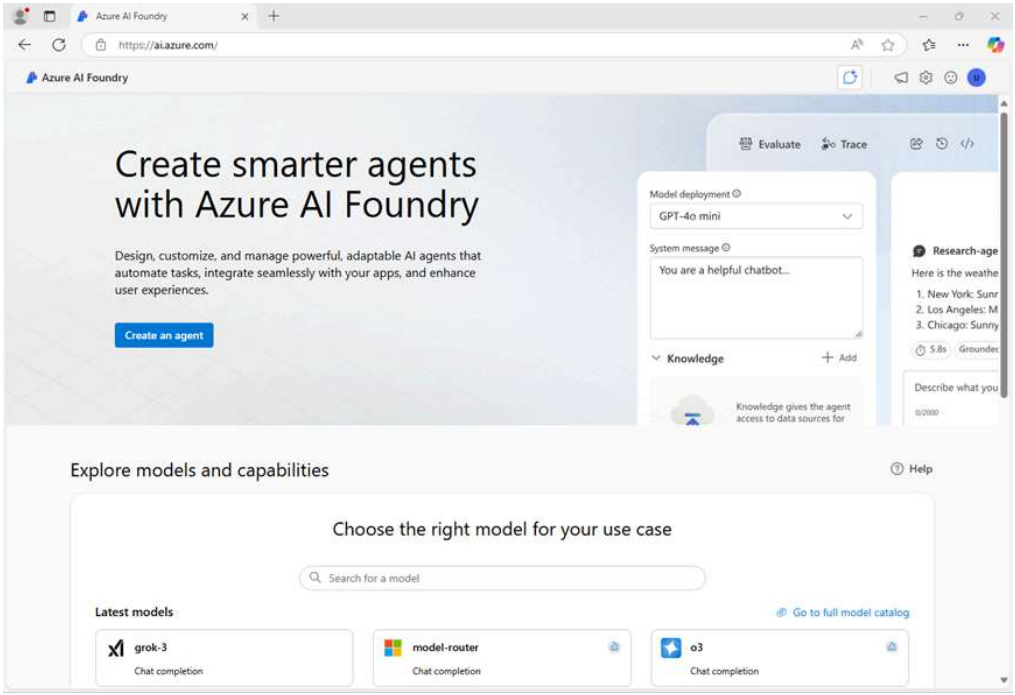
This exercise should take approximately **30** minutes to complete.

> **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 a Microsoft Foundry project

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

1. In a web browser, open the 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 **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):



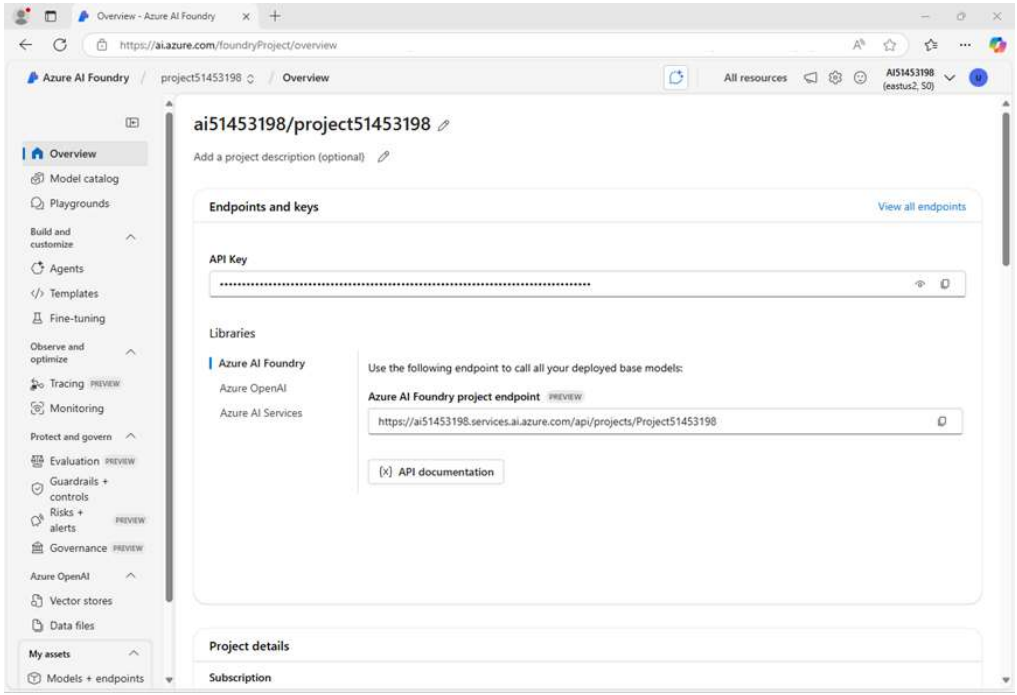
> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

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:
    - **Foundry resource**: *A valid name for your 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 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 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.

```
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:

```
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 .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 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 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:

```
# 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**:

```
# 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 Foundry project settings from the .env configuration.

5. Add the following code under the comment **Create agents**:

(Be sure to maintain the indentation level)

```
```

```
# 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 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.

200 XP

Knowledge check

3 minutes

1. What's the first step in the Microsoft Agent Framework's unified orchestration workflow?

- Select and create an orchestration pattern
- Define your agents and describe their capabilities
- Start a runtime to manage execution

2. For brainstorming and collaborative problem solving among multiple agents, which orchestration pattern is most suitable?

- Group Chat
- Magentic
- Sequential

3. Which pattern dynamically transfers control between agents based on context or rules?

- Handoff
- Concurrent
- Sequential

Submit answers

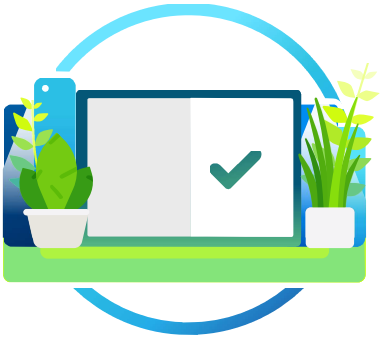
Next unit: Summary

Previous Next

200 XP

Knowledge check

3 minutes



Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What's the first step in the Microsoft Agent Framework's unified orchestration workflow?

- Select and create an orchestration pattern
- Define your agents and describe their capabilities ✓ Correct
- Start a runtime to manage execution

2. For brainstorming and collaborative problem solving among multiple agents, which orchestration pattern is most suitable?

- Group Chat ✓ Correct
- Magentic
- Sequential

3. Which pattern dynamically transfers control between agents based on context or rules?

- Handoff ✓ Correct
- Concurrent
- Sequential

Next unit: Summary

Previous Next

Summary

2 minutes

In this module, you explored how to design and manage multi-agent orchestration workflows using the Microsoft Agent Framework SDK. You learned how multi-agent systems provide advantages over single-agent approaches, including improved scalability, specialization, and collaborative problem solving. You also learned several different orchestration patterns—concurrent, sequential, handoff, group chat, and magentic—and reviewed guidance on when and how to use each. You also saw how the SDK provides a unified interface for defining agents, running orchestrations, handling structured data, and retrieving results asynchronously, enabling you to build flexible, reliable, and maintainable multi-agent workflows.

Tip

For more information about Microsoft Agent Framework SDK, see Microsoft Agent Framework SDK documentation.

All units complete:

Previous

Complete module

Introduction

2 minutes

AI agents are powerful on their own, but many real-world tasks require collaboration across multiple agents. Coordinating these interactions manually can be complex, especially when agents are remote or distributed.

The Agent-to-Agent (A2A) protocol addresses this challenge by providing a standardized framework for agent discovery, communication, and coordinated task execution. By implementing A2A, you can easily manage connections to remote agents, delegate requests to the appropriate agent, and enables seamless communication between agents in a standardized, secure way.

For example, imagine a technical writer who wants to create compelling blog content. One agent generates compelling article headlines, while another creates detailed outlines. Using A2A, a routing agent coordinates the workflow: it sends the user's request to the title agent, passes the generated title to the outline agent, and returns the final outline to the user—all automatically.

In this module, you learn how to implement the A2A protocol with Azure AI Agents. You also practice configuring a routing agent, registering remote agents, and building a coordinated workflow that allows multiple agents to collaborate effectively.

Next unit: Define an A2A agent

Next

Define an A2A agent

5 minutes

The **Agent-to-Agent (A2A) protocol** is a standardized way for AI agents to communicate and collaborate with each other. It defines how agents can share context, invoke each other's capabilities, and exchange information securely. By adhering to the A2A protocol, agents from different vendors or platforms can work together seamlessly, enabling more complex and integrated AI solutions.

Before an A2A agent can participate in multi-agent workflows, it needs to explain what it can do. **Agent Skills** and how other agents or clients can discover those capabilities are exposed through an **Agent Card**.

Advantages of the Agent-to-Agent (A2A) protocol

The **Agent-to-Agent (A2A) protocol** offers several advantages for AI agent interactions:

- **Enhanced Collaboration:**
  A2A enables agents from different vendors and platforms to **share context and work together**, allowing seamless automation across systems that are traditionally disconnected.
- **Flexible Model Selection:**
  Each A2A agent can choose which large language model (LLM) to use for handling requests, enabling **optimized or fine-tuned models per agent**, unlike some MCP scenarios that rely on a single LLM connection.
- **Integrated Authentication:**
  Authentication is built into the A2A protocol, providing a **robust security framework** for secure agent-to-agent communication.

Agent Skills

An Agent Skill describes a specific capability or function that the agent can perform. Think of it as a building block that communicates to clients or other agents what tasks the agent is designed to handle.

Key elements of an Agent Skill include:

- ID: A unique identifier for the skill.
- Name: A human-readable name describing the skill.
- Description: A detailed explanation of what the skill does.
- Tags: Keywords for categorization and easier discovery.
- Examples: Sample prompts or use cases to illustrate the skill in action.
- Input/Output Modes: Supported data formats or media types (for example, text, JSON).

When defining a skill for your agent, consider the tasks it should perform, how to describe them clearly, and how other agents or clients might use them. For example, a simple "Hello World" skill could return a basic greeting in text format, whereas a blog-writing skill might accept a topic and return a suggested title or outline.

Agent Card

The Agent Card is like a digital business card for your agent. It's a structured document that a routing agent or client can retrieve to discover your agent's capabilities and how to interact with it.

Key elements of an Agent Card include:

- Identity Information: Name, description, and version of the agent.
- Endpoint URL: Where the agent's A2A service can be accessed.
- Capabilities: Supported A2A features such as streaming or push notifications.
- Default Input/Output Modes: The primary media types the agent can handle.
- Skills: A list of the agent's skills that other agents can invoke.
- Authentication Support: Indicates if the agent requires credentials for access.

When creating an Agent Card, ensure it accurately represents your agent's skills and endpoints. This allows clients or routing agents to discover the agent, understand what it can do, and interact with it appropriately.

Putting it together

Once an agent defines its skills and publishes an Agent Card:

- Other agents or clients can discover the agent automatically.
- Requests can be routed to the agent's appropriate skill.

- Responses are returned in supported formats, enabling smooth collaboration across multiple agents.

For example, in a technical writer workflow, one agent could define skills for generating article titles, and another for creating outlines. The routing agent retrieves each agent's card to discover these capabilities and orchestrates a workflow where a title generated by one agent feeds into the outline agent, producing a cohesive final response.

Next unit: Implement an agent executor

Previous Next

Implement an agent executor

5 minutes

The **Agent Executor** is a core component of an A2A agent. It defines how your agent processes incoming requests, generates responses, and communicates with clients or other agents. Think of it as the bridge between the A2A protocol and your agent's specific business logic.

Understand the Agent Executor

The `AgentExecutor` interface handles all incoming requests sent to your agent. It receives information about the request, processes it according to the agent's capabilities, and sends responses or events back through a communication channel.

Key responsibilities:

- Execute tasks requested by users or other agents.
- Stream responses or send individual messages back to the client.
- Handle task cancellation if supported.

Implement the interface

An Agent Executor typically defines two primary operations:

Execute

- Processes incoming requests and generates responses.
- Accesses request details (for example, user input, task context).
- Sends results back via an event queue, which may include messages, task updates, or artifacts.

Cancel

- Handles requests to cancel an ongoing task.
- May not be supported for simple agents.

The executor uses the **RequestContext** to understand the incoming request and an **EventQueue** to communicate results or events back to the client.

Request handling flow

Consider a "Hello World" agent workflow:

1. The agent has a small helper class that implements its core logic (for example, returning a string).
2. The executor receives a request and calls the agent's logic.
3. The executor wraps the result as an event and places it on the event queue.
4. The routing mechanism sends the event back to the requester.

For cancellation, a basic agent might only indicate that cancellation isn't supported.

The Agent Executor is central to making your A2A agent functional. It defines how the agent executes tasks and communicates results, providing a standardized interface for clients and other agents. Properly implemented executors enable seamless integration and collaboration in multi-agent workflows.

Next unit: Host an A2A server

Previous Next

Host an A2A server

5 minutes

Once your agent defines its skills and Agent Card, the next step is to host it on a server. Hosting makes your agent accessible to clients and other agents over HTTP, enabling real-time interactions and multi-agent workflows.

Hosting an agent allows it to:

- Expose its capabilities through its **Agent Card**, which clients and other agents can discover.
- Receive incoming A2A requests and forward them to your **Agent Executor** for processing.
- Manage task lifecycles, including streaming responses and stateful interactions.

Effectively, the server acts as a bridge between your agent's logic and the external world, ensuring it can participate in coordinated workflows.

Core components of the agent server

To host an agent, you need three essential components working together:

Agent Card

- Describes the agent's capabilities, skills, and input/output modes.
- Exposed at a standard endpoint (typically `/.well-known/agent-card.json`) so clients and other agents can discover your agent.
- Can include multiple versions or an "extended" card for authenticated users.

Request Handler

- Routes incoming requests to the appropriate methods on your **Agent Executor** (for example, `execute` or `cancel`).
- Manages the task lifecycle using a **Task Store**, which tracks tasks, streaming data, and resubscriptions.
- Even simple agents require a task store to handle interactions reliably.

Server Application

- Built using a web framework (Starlette in Python) to handle HTTP requests.
- Combined with an ASGI server (like Uvicorn) to start listening on a network interface and port.
- Exposes the agent card and request handler endpoints, enabling clients to interact with your agent.

Set up the A2A agent server

1. Define your agent's skills and Agent Card.
2. Initialize a request handler that links your **Agent Executor** with a **Task Store**.
3. Set up the server application, providing the Agent Card and request handler.
4. Start the server using an ASGI server (Uvicorn) to make it accessible on the network.
5. Once running, the agent listens for incoming requests and responds according to its defined skills.

A "Hello World" agent may expose a basic greeting skill. Once hosted, it can respond to any requests sent to its endpoint. A more complex agent can serve multiple skills or an extended Agent Card for authenticated users.

Hosting an A2A agent combines the Agent Card, request handler, and agent executor to make it available for client and agent interactions. This setup ensures tasks are managed correctly and responses are delivered reliably, enabling your agent to participate in multi-agent workflows.

Next unit: Connect to your A2A agent

< Previous Next >

Connect to your A2A agent

5 minutes

Once your A2A agent server is running, the next step is understanding how a client can interact with it. A client acts as the bridge between your application and the agent server.

The client responsibilities include:

- Discovering the Agent Card, which contains metadata about the agent and its endpoints.
- Sending requests to the agent for processing.
- Receiving and interpreting the agent's responses, which can be either direct messages or task-based results.

Connect to your agent server

- The client must know the **base URL** of the server.
- The client typically retrieves the Agent Card from a well-known endpoint on the server.
- Once the Agent Card is obtained, the client can be initialized with it, establishing a connection ready to send messages.

Send requests to the agent

There are two main types of requests a client can make:

- **Non-Streaming Requests:** The client sends a message and waits for a complete response. This type of request is suitable for simple interactions or when a single response is expected.
- **Streaming Requests:** The client sends a message and receives responses incrementally as the agent processes the request. This type of request is useful for long-running tasks or when you want to update the user in real-time.

In both cases, requests usually include a `role` (fo example, user) and the message content. More complex agents may return **task objects** instead of immediate messages, allowing for task tracking or cancellation.

Handle the agent response

Agent responses may include:

- **Direct messages:** Immediate outputs from the agent, such as text or structured content.
- **Task-based responses:** Objects representing ongoing tasks, which may require follow-up calls to check status or retrieve results.

Clients should be prepared to handle both response types and interpret the returned data appropriately.

Interacting with the agent

- Each request should be uniquely identifiable, often using a generated ID.
- Streaming responses are asynchronous and may provide partial results before the final output.
- Simple agents may return messages directly, while more advanced agents may manage multiple tasks simultaneously.

Connecting a client to your agent server involves fetching the Agent Card, establishing a connection, sending requests, and handling responses. By grasping these core concepts, you can confidently interact with your remote agent, whether you're sending simple messages or managing complex tasks.

Next unit: Exercise - Connect to remote Azure AI Agents with the A2A protocol

Previous Next

100 XP

Exercise - Connect to remote Azure AI Agents with the A2A protocol

30 minutes

If you have an Azure subscription, you can complete this exercise to develop an A2A client-server application that interacts with remote agents.

> **Note**
>
> If you don't have an Azure subscription, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Module assessment

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 Microsoft Foundry SDK for Python. You can develop similar solutions using the SDKs for Microsoft .NET, JavaScript, and Java. Refer to Microsoft Foundry SDK client libraries for details.

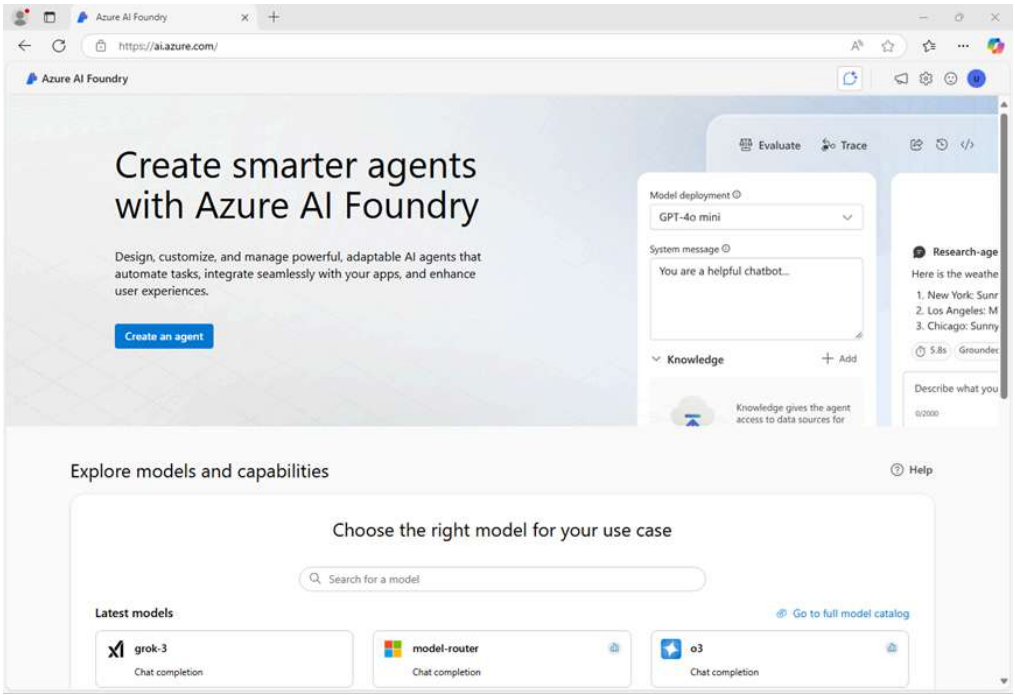
This exercise should take approximately **30** minutes to complete.

> **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 a Foundry project

Let's start by creating a Foundry project.

1. In a web browser, open the 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 **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):



> **Important**: Make sure the **New Foundry** toggle is *Off* for this lab.

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:
    - **Foundry resource**: *A valid name for your Foundry resource*
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Create or select a resource group*

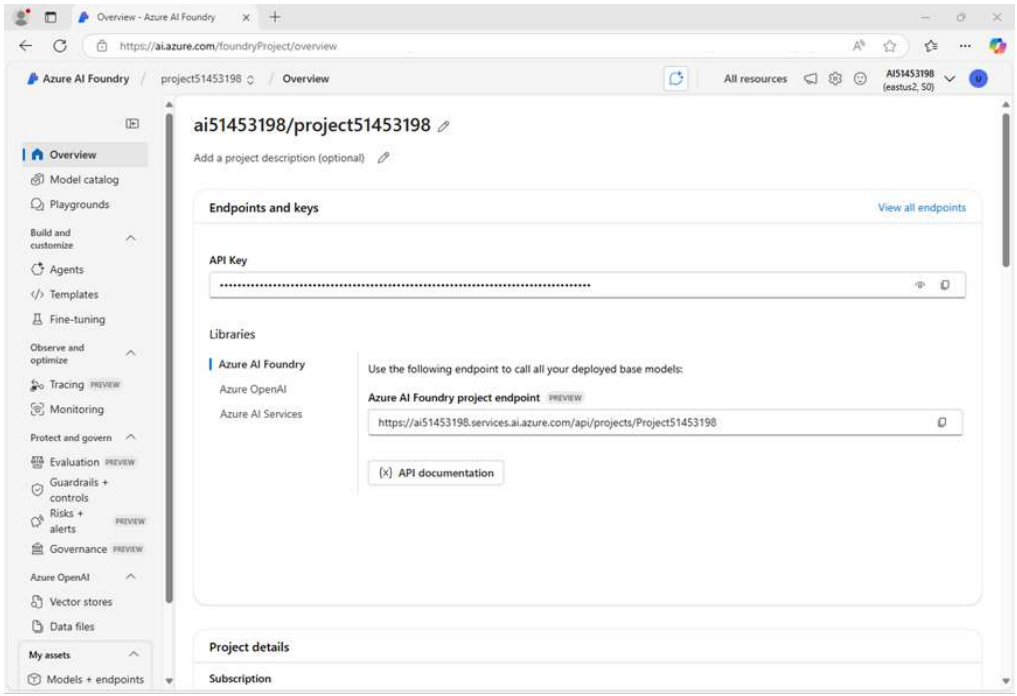
- **Region**: *Select any **AI Foundry recommended****

> * 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 **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 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/06-build-remote-agents-with-a2a/python
   ls -a -l
   ```

   The provided files include:

   ```
   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:

   ```
   ```

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-projects azure-ai-agents a2a-sdk
```

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 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:

```
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 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.

```
# 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:

Code Copy

```
# 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:

Code Copy

```
# 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:

Code Copy

```
# 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:

Code Copy

```
# 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 Copy

```
code server.py
```

9. Find the comment **Define agent skills** and add the following code to specify the agent's functionality:

Code Copy

```
# 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 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:

```
cd ../routing_agent
```

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

```
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**:

```
# 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:

```
# 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 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.

200 XP

Module assessment

3 minutes

1. What is the primary role of an A2A server?

- It executes business logic for the agent directly.
- It routes requests between clients and connected agents.
- It stores static agent responses for reuse.

2. What does the Agent Executor do in an A2A agent?

- Manages network connections between clients and servers.
- Processes incoming requests and generates responses or events.
- Provides a GUI for monitoring agent activity.

3. What is an agent card used for in A2A?

- It stores the agent's API key for authentication.
- It provides metadata about the agent, such as its capabilities and available functions.
- It visualizes the agent's workflow in a GUI dashboard.

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. What is the primary role of an A2A server?

- It executes business logic for the agent directly.
- It routes requests between clients and connected agents. ✓ Correct
- It stores static agent responses for reuse.

2. What does the Agent Executor do in an A2A agent?

- Manages network connections between clients and servers.
- Processes incoming requests and generates responses or events. ✓ Correct
- Provides a GUI for monitoring agent activity.

3. What is an agent card used for in A2A?

- It stores the agent's API key for authentication.
- It provides metadata about the agent, such as its capabilities and available functions. ✓ Correct
- It visualizes the agent's workflow in a GUI dashboard.

Next unit: Summary

Previous Next

Summary

1 minute

In this module, you learned how to connect Python clients to Azure AI Agents using the Agent-to-Agent (A2A) protocol.

By running an A2A server and connecting your client, you explored how agents are discovered and communicated with dynamically using the Agent Card. You learned about executors, which handle agent requests, and how messages—both streaming and non-streaming—flow between clients and agents. Understanding these concepts allows you to build flexible, discoverable agent networks that can delegate tasks and respond to requests across distributed environments.

All units complete:

Previous

Complete module

Introduction

1 minute

As generative AI models become more powerful and accessible, developers are moving beyond simple chat applications to build intelligent agents that can automate complex tasks. These AI agents combine large language models with specialized tools to access data, perform actions, and complete entire workflows with minimal human intervention.

Visual Studio Code is a powerful platform for AI agent development, especially with the Microsoft Foundry extension. This extension brings enterprise-grade AI capabilities directly into your development environment. With the Microsoft Foundry for Visual Studio Code extension, you can discover and deploy models, create and configure agents, and test them in interactive playgrounds—all without leaving your code editor.

This module introduces the core concepts of AI agent development and shows you how to use the Microsoft Foundry extension for Visual Studio Code to build, test, and deploy intelligent agents. You'll learn how to use Microsoft Foundry's Agent Service to create agents that can handle complex tasks, use various tools, and integrate with your existing applications.

Next unit: Get started with the Microsoft Foundry extension

Next

100 XP

Get started with the Microsoft Foundry extension

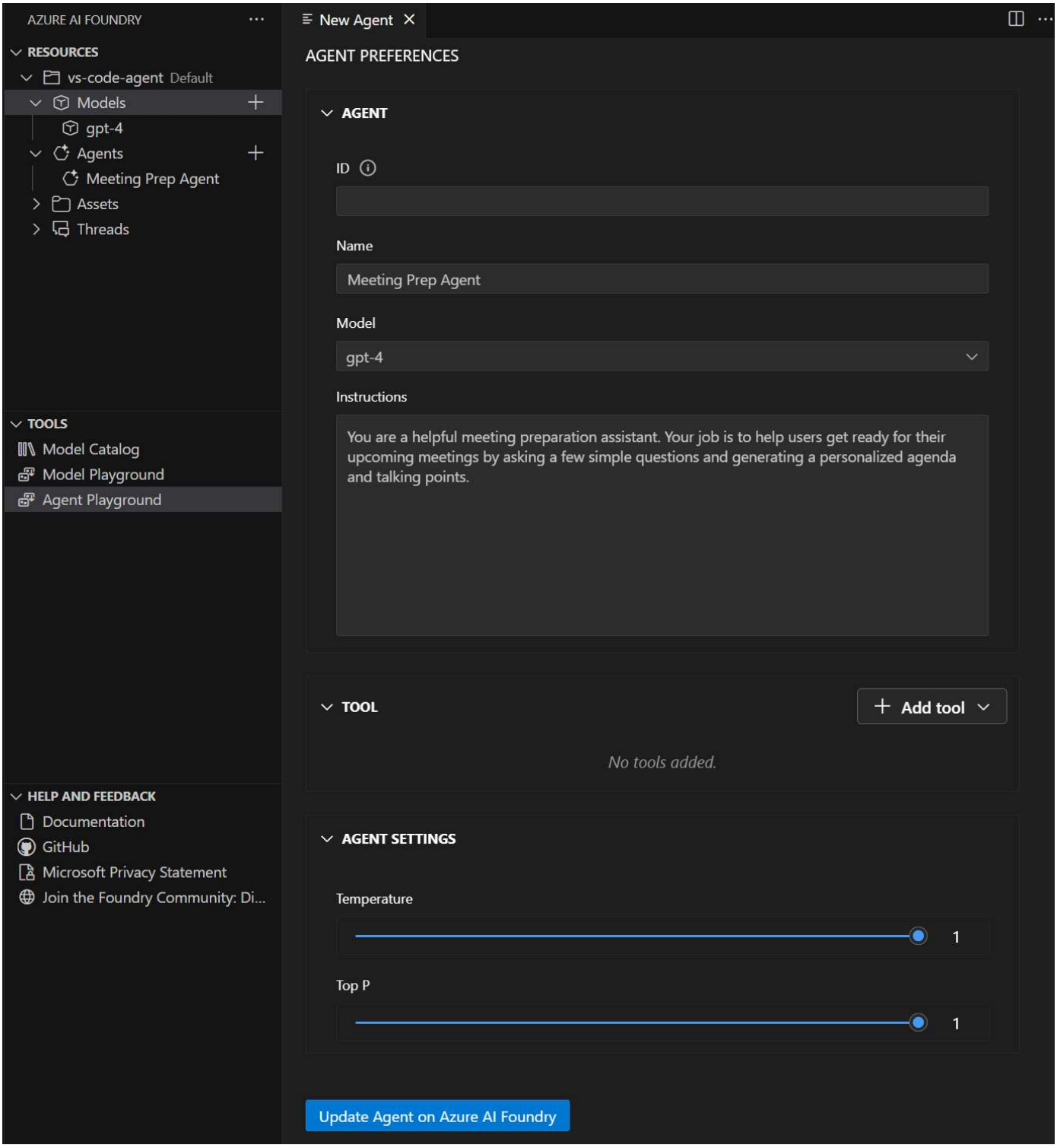
3 minutes

The Microsoft Foundry for Visual Studio Code extension transforms your development environment into a comprehensive platform for building, testing, and deploying AI agents. This extension provides direct access to the capabilities of Microsoft Foundry's Agent Service without leaving your code editor, streamlining the entire agent development workflow.

What is the Microsoft Foundry for Visual Studio Code extension?

The Microsoft Foundry for Visual Studio Code extension is a powerful tool that brings enterprise-grade AI agent development capabilities directly into Visual Studio Code. It provides an integrated experience for:

- **Agent Discovery and Management** - Browse, create, and manage AI agents within your Microsoft Foundry projects
- **Visual Agent Designer** - Use an intuitive interface to configure agent instructions, tools, and capabilities
- **Integrated Testing** - Test agents in real-time using the built-in playground without switching contexts
- **Code Generation** - Generate sample integration code to connect agents with your applications
- **Deployment Pipeline** - Deploy agents directly to Microsoft Foundry for production use



Key features and capabilities

The extension offers several key features that accelerate AI agent development:

Agent Designer Interface

A visual designer that simplifies agent creation and configuration. You can define agent instructions, select appropriate models, and configure tools through an intuitive graphical interface.

Built-in Playground

An integrated testing environment where you can interact with your agents in real-time, test different scenarios, and refine agent behavior before deployment.

Tool Integration

Seamless integration with various tools including:

- **RAG (Retrieval-Augmented Generation)** for knowledge-based responses
- **Search capabilities** for information retrieval
- **Custom actions** for specific business logic
- **Model Context Protocol (MCP) servers** for extended functionality

Project Integration

Direct connection to your Microsoft Foundry projects, allowing you to work with existing resources and deploy new agents to your established infrastructure.

Installing the extension

To get started with the Microsoft Foundry for Visual Studio Code extension, first you need to have Visual Studio Code installed on your machine. You can download it from the Visual Studio Code website .

You can install the Microsoft Foundry extension directly from the Visual Studio Code Marketplace:

1. Open Visual Studio Code.
2. Select Extensions from the left pane, or press `Ctrl` + `Shift` + `X`.
3. Search for and select Microsoft Foundry.
4. Select Install.
5. Verify the extension is installed successfully from the status messages.

Getting started workflow

The typical workflow for using the Microsoft Foundry extension follows these steps:

1. **Install and configure** the extension in Visual Studio Code
2. **Connect** to your Microsoft Foundry project
3. **Create or import** an AI agent using the designer
4. **Configure** agent instructions and add necessary tools
5. **Test** the agent using the integrated playground
6. **Iterate** on the design based on test results
7. **Generate code** for application integration

This streamlined workflow enables rapid prototyping and deployment of AI agents, making it easier for developers to build intelligent automation solutions that can handle complex real-world tasks.

The Microsoft Foundry for Visual Studio Code extension represents a significant step forward in making AI agent development more accessible and efficient, providing developers with enterprise-grade tools in a familiar development environment.

Next unit: Develop AI agents in Visual Studio Code

Previous Next

100 XP

Develop AI agents in Visual Studio Code

5 minutes

Creating and configuring AI agents in Visual Studio Code using the Microsoft Foundry extension provides a streamlined development experience that combines the power of Microsoft Foundry Agent Service with the familiar Visual Studio Code environment. This approach enables you to design, configure, and test agents without leaving your development environment.

Understanding Microsoft Foundry Agent Service

Microsoft Foundry Agent Service is a managed service in Azure designed to provide a comprehensive framework for creating, managing, and deploying AI agents. The service builds on the OpenAI Assistants API foundation while offering enhanced capabilities including:

- **Expanded model choice** - Support for multiple AI models beyond OpenAI
- **Enterprise security** - Built-in security features for production environments
- **Advanced data integration** - Seamless connection to Azure data services
- **Tooling ecosystem** - Access to a various built-in and custom tools

The Visual Studio Code extension provides direct access to these capabilities through an intuitive interface that simplifies the agent development process.

Creating agents with the extension

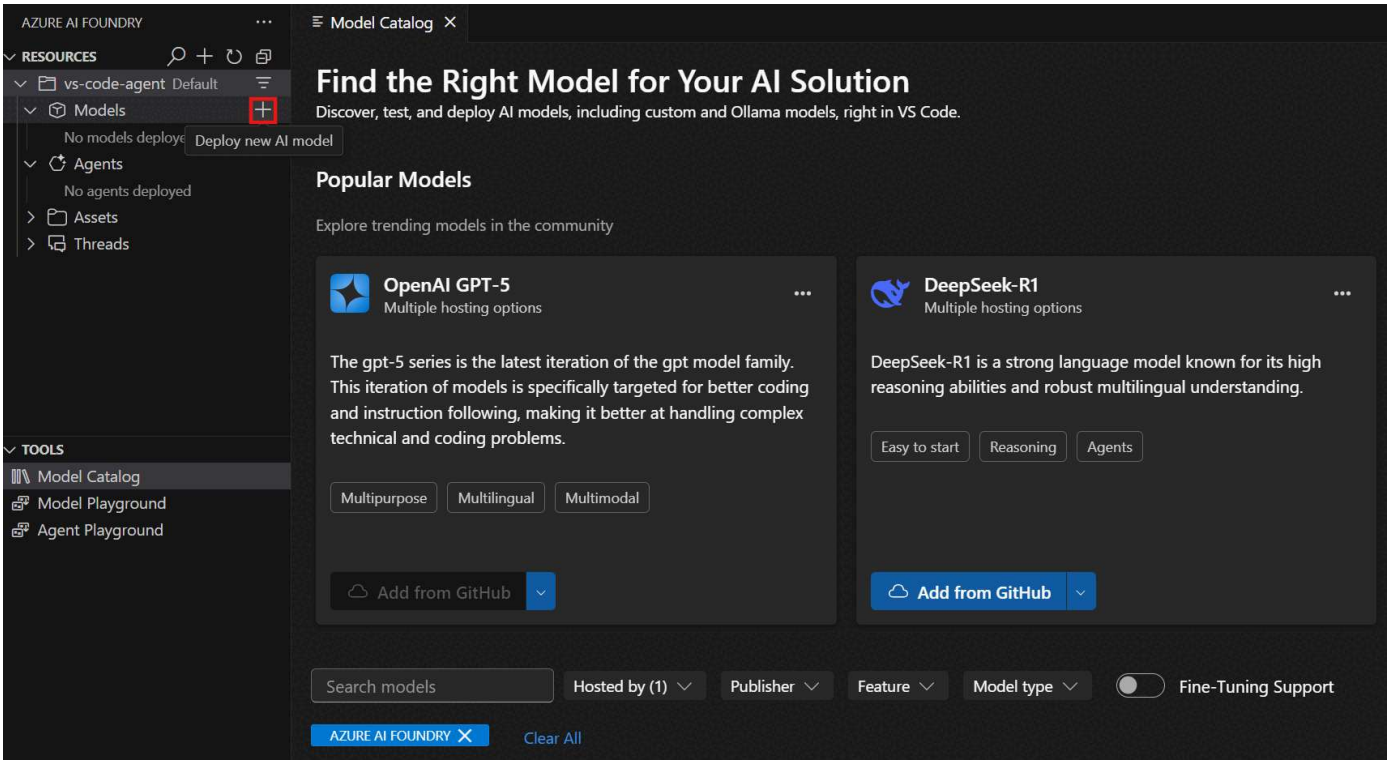
The Microsoft Foundry extension provides multiple ways to create AI agents, whether you're starting from scratch or building on existing work. The flexible approach accommodates different development preferences and scenarios.

Prerequisites for agent creation

Before creating an agent, complete the following steps:

1. Complete the extension setup and sign in to your Azure account
2. Create a default Microsoft Foundry project, or select an existing one

3. Select and deploy the model for your agent to use, or use an existing deployment

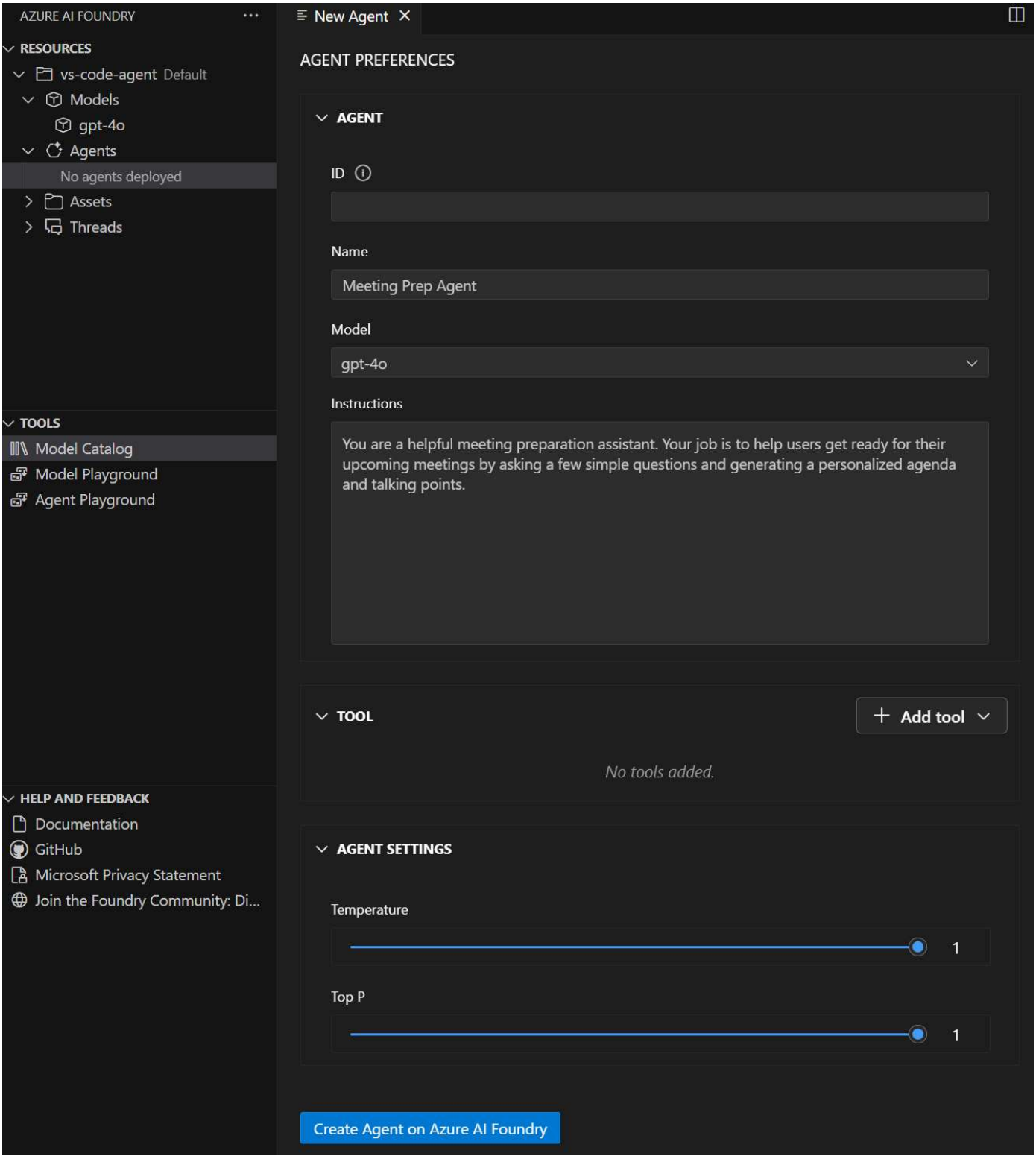


Creating a new agent

To create a new AI agent, follow these steps:

1. Open the Microsoft Foundry Extension view in Visual Studio Code
2. Navigate to the Resources section
3. Select the + (plus) icon next to the Agents subsection to create a new AI agent
4. Configure the agent properties in the Agent Designer view that opens

When you create an agent, the extension opens both the agent .yaml file and the Designer view, providing you with both a visual interface and direct access to the configuration file.



Configuring agent properties

Once you create an agent, the extension provides comprehensive configuration options to define how your agent behaves and interacts with users. The Agent Designer provides an intuitive interface for setting up these properties.

Basic Configuration

In the Agent Designer, configure the following essential properties:

- **Agent name** - Enter a descriptive name for your agent in the prompt
- **Model selection** - Choose your model deployment from the dropdown (this is the deployment name you chose when deploying a model)
- **Description** - Add a clear description of what your agent does
- **System instructions** - Define the agent's behavior, personality, and response style
- **Agent ID** - Automatically generated by the extension

Understanding the agent YAML file

Your AI agent is defined in a YAML file that contains all configuration details. Here's an example structure:

YAML

```yaml
# yaml-language-server: $schema=https://aka.ms/ai-foundry-vsc/agent/1.0.0
version: 1.0.0
name: my-agent
description: Description of the agent
id: ''
metadata:
  authors:
    - author1
    - author2
  tags:
    - tag1
    - tag2
model:
  id: 'gpt-4o-1'
  options:
    temperature: 1
    top_p: 1
instructions: Instructions for the agent
tools: []
```

This YAML file is opened automatically alongside the Designer view, allowing you to work with either the visual interface or edit the configuration directly.

Agent instruction design

Well-crafted instructions are the foundation of effective AI agents. They define how your agent understands its role, responds to users, and handles various scenarios.

Best practices for instructions

When writing system instructions for your agent:

- **Be specific and clear -** Define exactly what the agent should do and how it should behave
- **Provide context -** Explain the agent's role and the environment it operates in
- **Set boundaries -** Clearly define what the agent should and shouldn't do
- **Include examples -** Show the agent examples of desired interactions when helpful
- **Define personality -** Establish the tone and style of responses

Instruction examples

For a customer service agent, effective instructions might include:

- The agent's role and purpose
- Guidelines for handling different types of customer inquiries
- Escalation procedures for complex issues
- Tone and communication style preferences

Deploying agents

Once you configure your agent, you can deploy it to Microsoft Foundry.

Deployment process

To deploy your agent:

1. **Select the "Create on Microsoft Foundry" button** in the bottom-left of the Designer
2. **Wait for deployment completion -** The extension handles the deployment process
3. **Refresh the Azure Resources view** in the Visual Studio Code navbar
4. **Verify deployment -** The deployed agent appears under the Agents subsection

Managing deployed agents

After deployment, you can:

- **View agent details -** Select the deployed agent to see the Agent Preferences page
- **Edit the agent -** Select "Edit Agent" to modify configuration and redeploy with the **Update on Microsoft Foundry** button
- **Generate integration code -** Select "Open Code File" to create sample code for using the agent
- **Test in playground -** Select "Open Playground" to interact with the deployed agent

Testing and iteration

The integrated playground enables immediate testing of your agent configuration, allowing you to validate behavior and make adjustments in real-time.

Using the playground

After configuring your agent, you can test it using the built-in playground:

- **Real-time conversations -** Chat with your agent to test responses
- **Instruction validation -** Verify the agent follows its configured instructions
- **Behavior testing -** Test how the agent handles different types of requests
- **Iterative refinement -** Make adjustments based on testing results

Working with agent threads

When you interact with deployed agents, the system creates threads to manage conversation sessions:

- **Threads -** Conversation sessions between an agent and user that store messages and handle context management
- **Messages** - Individual interactions that can include text, images, and files
- **Runs -** Single executions of an agent that use the agent's configuration and thread messages

You can view and manage these threads through the Azure Resources view in the extension.

Creating and configuring AI agents with the Microsoft Foundry Visual Studio Code extension provides a powerful yet accessible approach to agent development. The extension provides visual design tools, direct YAML editing, comprehensive configuration options, and integrated testing capabilities. These features enable developers to rapidly prototype and deploy sophisticated AI agents that can handle complex real-world scenarios.

Next unit: Extend AI agent capabilities with tools

Previous Next

Extend AI agent capabilities with tools

5 minutes

One of the most powerful features of AI agents is their ability to use tools that extend their capabilities beyond simple text generation. The Microsoft Foundry for Visual Studio Code extension makes it easy to add and configure tools for your agents. These tools enable agents to perform actions, access data, and integrate with external systems.

Understanding agent tools

Tools are programmatic functions that enable agents to automate actions and access information beyond their training data. When an agent determines that a tool is needed to respond to a user request, it can automatically invoke the appropriate tool, process the results, and incorporate them into its response. This capability transforms agents from simple text generators into powerful automation systems that can interact with real-world data and services.

Built-in tools

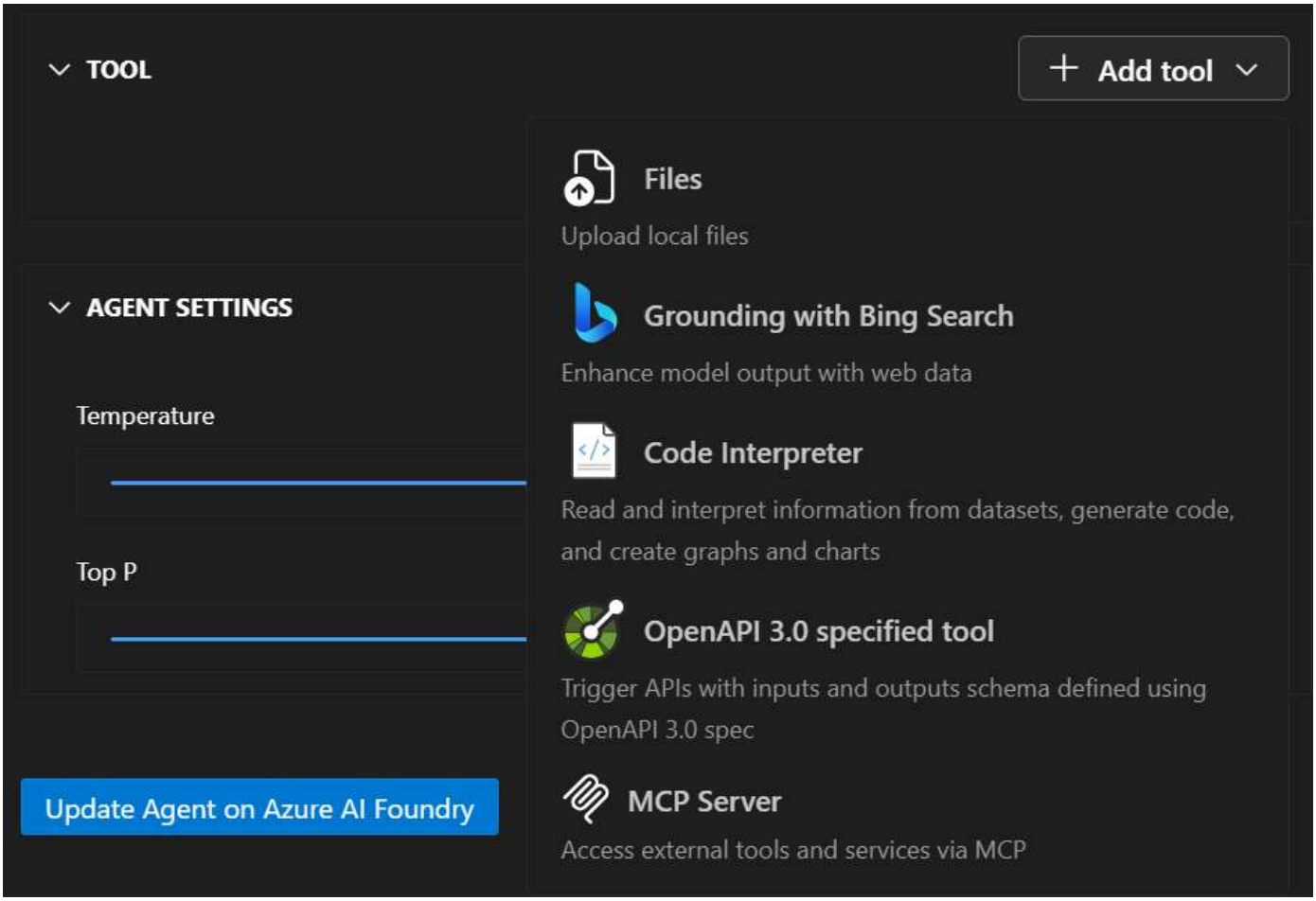
Microsoft Foundry provides several built-in tools that you can easily add to your agents without any additional configuration or setup. These tools are production-ready and handle common use cases that many agents require.

- **Code Interpreter** - Enables agents to write and execute Python code for mathematical calculations, data analysis, chart generation, file processing, and complex problem-solving
- **File Search** - Provides retrieval-augmented generation by uploading and indexing documents, searching knowledge bases, and supporting various file formats (PDF, Word, text files)
- **Grounding with Bing Search** - Allows agents to search the internet for real-time data, current events, and trending topics while providing citations and sources
- **OpenAPI Specified Tools** - Connects agents to external APIs and services through OpenAPI 3.0 specifications
- **Model Context Protocol (MCP)** - Standardized tool interfaces for extended functionality and community-driven tools

Adding tools in Visual Studio Code

The Microsoft Foundry extension provides an intuitive interface for adding tools to your agents through a streamlined process. The visual interface makes it easy to browse, configure, and test tools without writing any code:

1. **Select your agent** in the extension
2. **Navigate to the Tools section** in the configuration panel
3. **Browse available tools** from the tool library
4. **Configure tool settings** as needed
5. **Test tool integration** using the playground



When you add a tool, you can also add any new assets it needs. For example, if you add a File Search tool, you can use an existing vector store asset or make a new asset for your vector store to host your uploaded files.

Model Context Protocol (MCP) servers

MCP servers provide a standardized way to add tools to your agents using an open protocol. This approach enables you to use community-built tools and create reusable components that work across different agent implementations.

Key benefits include:

- **Standardized protocol** for consistent tool communication
- **Reusable components** that work across different agents
- **Community-driven tools** available through MCP registries
- **Simplified integration** with consistent interfaces

The extension supports adding MCP servers through registry browsing, custom server addition, configuration management, and testing and validation.

Tool management and best practices

Effective tool management ensures your agents perform reliably and efficiently in production environments. Following best practices helps you avoid common pitfalls and optimize agent performance:

Tool Selection Guidelines

- Identify what capabilities your agent requires
- Start with built-in tools before adding custom solutions
- Test thoroughly to validate tool behavior in various scenarios
- Monitor performance to track tool usage and effectiveness

Adding tools and extending agent capabilities through the Microsoft Foundry Visual Studio Code extension enables you to create sophisticated AI agents that can handle complex real-world tasks. By combining built-in tools with custom functions and MCP servers, you can build agents that seamlessly integrate with your existing systems and business processes while maintaining enterprise-grade security and performance.

Next unit: Exercise - Build an AI agent using the Microsoft Foundry extension

Previous Next

Exercise - Build an AI agent using the Microsoft Foundry extension

30 minutes

If you have an Azure subscription, you can explore the Microsoft Foundry Extension for Visual Studio Code by completing this hands-on exercise. This exercise guides you through creating and deploying an AI agent using the extension.

> **Note**
>
> If you don't have an Azure subscription, and you want to explore Microsoft Foundry, you can sign up for an account , which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

Next unit: Module assessment

Develop an AI agent with VS Code extension

In this exercise, you'll use the Microsoft 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 Microsoft 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 **Microsoft Foundry** and press Enter.
4. Select the **Microsoft 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 **Microsoft Foundry** extension icon.
2. In the Resources view, select **Sign in to Azure...** and follow the authentication prompts.

   > **Note**: You won't see this option if you're already signed in.

3. Create a new Foundry project by selecting the **+** (plus) icon next to **Resources** in the Foundry Extension view.
4. Select your Azure subscription from the dropdown.
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 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."

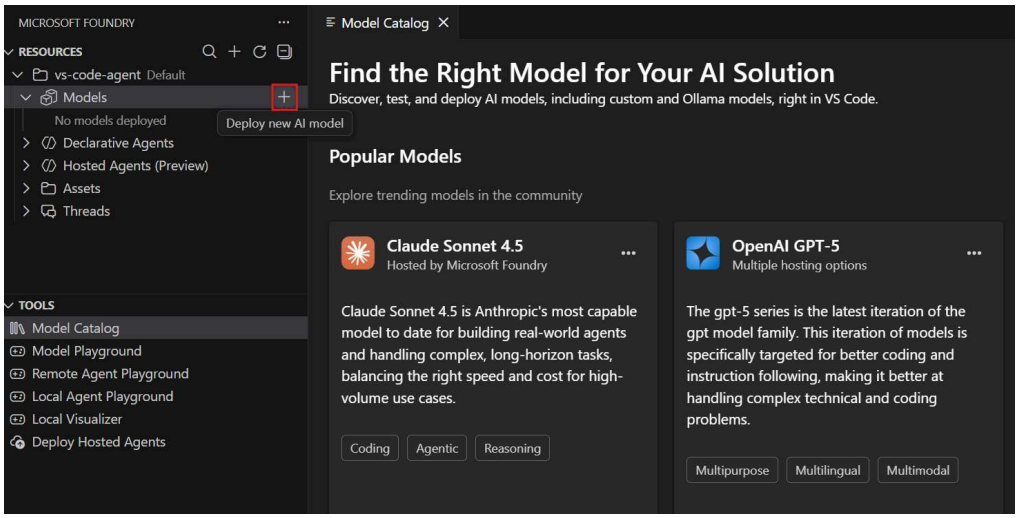
Deploy a model

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 **Microsoft Foundry: Open Model Catalog**.

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

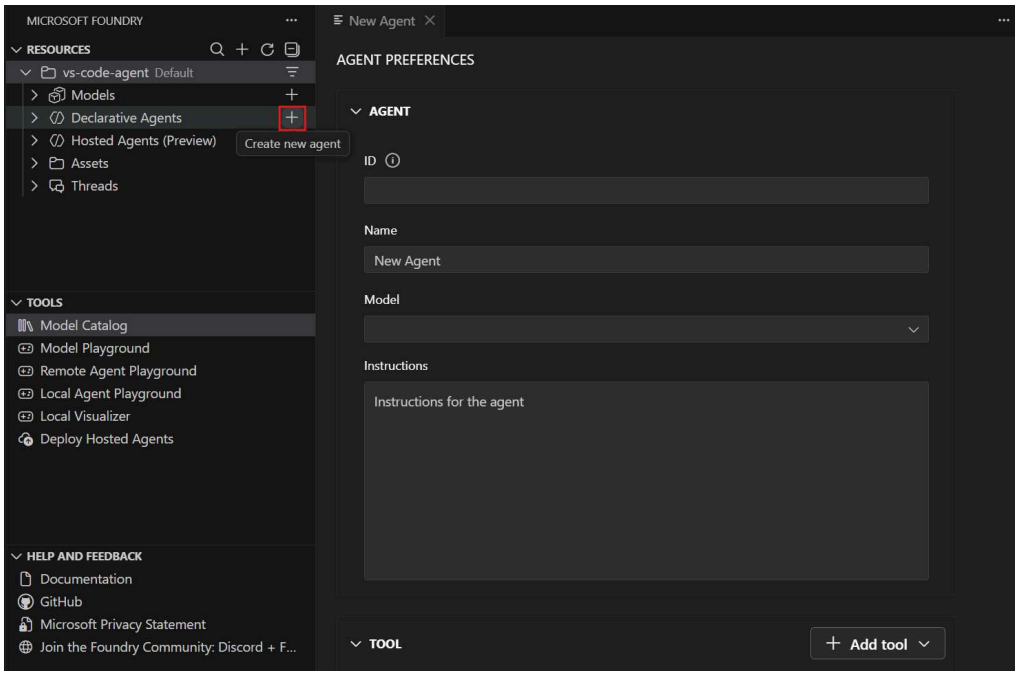


3. Select **Deploy** next to the gpt-4 model.
4. Configure the deployment settings:
    - **Deployment name**: Enter a name like "gpt-4-deployment"
    - **Deployment type**: Select **Global Standard** (or **Standard** if Global Standard is not available)
    - **Model version**: Leave as default
    - **Tokens per minute**: Leave as default
5. Select **Deploy in Microsoft 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. Rather than writing code, you'll configure the agent's instructions, settings, and tools through the user interface.

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



3. Choose a location to save your agent files when prompted.
4. A **New Agent** tab will open to an "Agent Preferences" editor, along with a `.yaml` configuration file.

Configure your agent in the designer

1. In the agent preferences, configure the following fields:
    - **Name**: Enter a descriptive name for your agent (e.g., "data-research-agent")
    - **Model**: Select your GPT-4o deployment from the dropdown
    - **Instructions**: Enter system instructions such as:

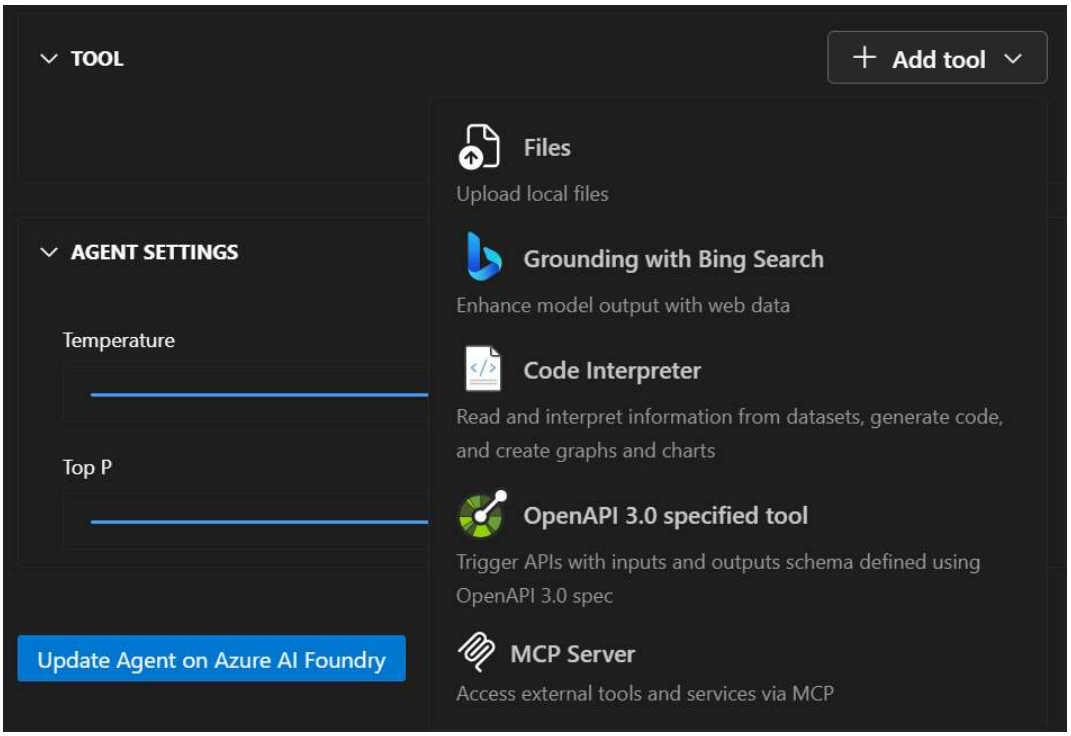
```
You are an AI agent that helps users research information from various sources. Use the available tools to access up-to-date information and provide comprehensive responses based on external data sources.
```

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 Microsoft Foundry

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

Test your agent in the playground

1. Right-click on your deployed agent in the **Declarative 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 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 Foundry VS Code extension, you should clean up the resources to avoid incurring unnecessary Azure costs.

Delete your agents

1. In the 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.

200 XP

Module assessment

3 minutes

1. When you create a new agent in the Microsoft Foundry extension, what two views are automatically opened?

- The YAML file and the Playground view
- The YAML file and the Designer view
- The Designer view and the Code Generation view

2. What is a key benefit of using Model Context Protocol (MCP) servers for agent tools?

- They only work with Microsoft-built tools
- They provide reusable components that work across different agents
- They replace the need for all built-in tools

3. How does the Microsoft Foundry Agent Service manage conversation sessions when users interact with deployed agents?

- The system creates individual threads for each conversation to manage context and message history
- Each message is processed independently with no conversation history
- All user conversations are combined into one shared global session

Submit answers

Next unit: Summary

Previous Next

Module assessment

3 minutes

Module assessment passed

Great job! You passed the module assessment.

Score: 100%

1. When you create a new agent in the Microsoft Foundry extension, what two views are automatically opened?

- The YAML file and the Playground view
- The YAML file and the Designer view ✓ Correct
- The Designer view and the Code Generation view

2. What is a key benefit of using Model Context Protocol (MCP) servers for agent tools?

- They only work with Microsoft-built tools
- They provide reusable components that work across different agents ✓ Correct
- They replace the need for all built-in tools

3. How does the Microsoft Foundry Agent Service manage conversation sessions when users interact with deployed agents?

- The system creates individual threads for each conversation to manage context and message history ✓ Correct

- Each message is processed independently with no conversation history
- All user conversations are combined into one shared global session

Next unit: Summary

Previous Next

Summary

1 minute

In this module, you learned how to develop AI agents using the Microsoft Foundry for Visual Studio Code extension. You explored how to create, configure, and test agents directly within your development environment. You also learned to extend their capabilities with various tools and deploy them to production environments.

Tip

For more information, see Work with the Microsoft Foundry for Visual Studio Code extension.

All units complete:

Previous

Complete module