Azure AI Agent Service documentation

Azure AI Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality, and extensible AI agents.

Azure AI Agent Service

OVERVIEW

What is Azure AI Agent?

Models and region support

Quotas and limits

WHAT'S NEW

What's new in Azure AI Agent Service?

QUICKSTART

Create an agent

Azure AI Agent Service concepts

CONCEPT

What are agents?

Tracing with Application Insights

Azure AI Agent Service tools

HOW-TO GUIDE

Tools overview

Azure AI Search

Grounding with Bing Search

File search

Function calling

Code interpreter

What is Azure AI Agent Service?

Article • 01/10/2025

Azure AI Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality, and extensible AI agents without needing to manage the underlying compute and storage resources. What originally took hundreds of lines of code to support client side function calling can now be done in just a few lines of code with Azure AI Agent Service.

What is an AI agent?

Within Azure AI Foundry, an AI Agent acts as a "smart" microservice that can be used to answer questions (RAG), perform actions, or completely automate workflows. It achieves this by combining the power of generative AI models with tools that allow it to access and interact with real-world data sources.

Because Azure AI Agent Service uses the same wire protocol as Azure OpenAI Assistants, you can use either OpenAI SDKs or Azure AI Foundry SDKs to create and run an agent in just a few lines of code. For example, to create an AI Agent with Azure AI Foundry SDK, you can simply define which model the AI uses, the instructions for how it should complete tasks, and the tools it can use to access and interact with other services.

Python

```
agent = project_client.agents.create_agent(
    model="gpt-4o-mini",
    name="my-agent",
    instructions="You are helpful agent",
    tools=code_interpreter.definitions,
    tool_resources=code_interpreter.resources,
)
```

After defining an agent, you can start asking it to perform work by invoking a run on top of an activity thread, which is simply a conversation between multiple agents and users.

Python

```
# Create a thread with messages
thread = project_client.agents.create_thread()
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content="Could you please create a bar chart for the operating profit
using the following data and provide the file to me? Company A: $1.2
```

```
million, Company B: $2.5 million, Company C: $3.0 million, Company D: $1.8
million",
)

# Ask the agent to perform work on the thread
run = project_client.agents.create_and_process_run(thread_id=thread.id,
agent_id=agent.id)

# Fetch and log all messages to see the agent's response
messages = project_client.agents.list_messages(thread_id=thread.id)
print(f"Messages: {messages}")
```

Whenever the run operation is invoked, Azure AI Agent Service will complete the entire tool calling lifecycle for you by 1) running the model with the provided instructions, 2) invoking the tools as the agent calls them, and 3) returning the results back to you.

Once you've gotten the basics, you can start using multiple agents together to automate even more complex workflows with AutoGen and Semantic Kernel. Because Azure AI Agent Service is a fully managed service, you can focus on building workflows and the agents that power them without needing to worry about scaling, security, or management of the underlying infrastructure for individual agents.

Why use Azure AI Agent Service?

When compared to developing with the Inference API directly, Azure AI Agent Service provides a more streamlined and secure way to build and deploy AI agents. This includes:

- **Automatic tool calling** – no need to parse a tool call, invoke the tool, and handle the response; all of this is now done server-side
- **Securely managed data** – instead of managing your own conversation state, you can rely on threads to store all the information you need
- **Out-of-the-box tools** – In addition to the file retrieval and code interpreter tools provided by Azure OpenAI Assistants, Azure AI Agent Service also comes with a set of tools that you can use to interact with your data sources, such as Bing, Azure AI Search, and Azure Functions.

What originally took hundreds of lines of code can now be done in just a few with Azure AI Agent Service.

Comparing Azure agents and Azure OpenAI assistants

Both services enable you to build agents using the same API and SDKs, but if you have additional enterprise requirements, you might want to consider using Azure AI Agent

Service. Azure AI Agent Service provides all the capabilities of assistants in addition to:

Flexible model selection - Create agents that use Azure OpenAI models, or others such as Llama 3, Mistral and Cohere. Choose the most suitable model to meet your business needs.

Extensive data integrations - Ground your AI agents with relevant, secure enterprise knowledge from various data sources, such as Microsoft Bing, Azure AI Search, and other APIs.

Enterprise grade security - Ensure data privacy and compliance with secure data handling, keyless authentication, and no public egress.

Choose your storage solution - Either bring your own Azure Blob storage for full visibility and control of your storage resources, or use platform-managed storage for secure ease-of-use.

Responsible AI

At Microsoft, we're committed to the advancement of AI driven by principles that put people first. Generative models such as the ones available in Azure OpenAI have significant potential benefits, but without careful design and thoughtful mitigations, such models have the potential to generate incorrect or even harmful content. Microsoft has made significant investments to help guard against abuse and unintended harm, which includes incorporating Microsoft's principles for responsible AI use, adopting a Code of Conduct for use of the service, building content filters to support customers, and providing responsible AI information and guidance that customers should consider when using Azure AI Agent Service.

Get started with Azure AI Agent Service

To get started with Azure AI Agent Service, you need to create an Azure AI Foundry hub and an Agent project in your Azure subscription.

Start with the quickstart guide if it's your first time using the service.

1. You can create a AI hub and project with the required resources.
2. After you create a project, you can deploy a compatible model such as GPT-4o.
3. When you have a deployed model, you can also start making API calls to the service using the SDKs.

Next steps

Learn more about the models that power agents.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

What's new in Azure AI Agent Service

Article • 04/23/2025

This article provides a summary of the latest releases and major documentation updates for Azure AI Agent Service.

April 2025

Azure monitor integration

You can now see metrics related to Agents in Azure monitor

- The number of files indexed for file search.
- The number of runs in a given timeframe.

See the Azure monitor and metrics reference articles for more information.

BYO thread storage

The Standard Agent Setup now supports **Bring Your Own (BYO) thread storage using an Azure Cosmos DB for NoSQL account**. This feature ensures all thread messages and conversation history are stored in your own resources. See the Quickstart for more information on how to deploy a Standard agent project.

March 2025

Microsoft Fabric tool

The Microsoft Fabric tool is now available for the Azure AI Agent Service, allowing users to interact with data you have in Microsoft Fabric through chat and uncover data-driven and actionable insights. See the how-to article for more information.

February 2025

Use Azure AI Agent Service in the Azure AI Foundry portal

You can now use the Azure AI Agent Service in the Azure AI Foundry. Create, debug and modify agents, view threads, add tools and chat with agents without writing code. See the quickstart for steps on getting started.

December 2024

Azure AI Service public preview

Azure AI Service is now available in preview. The service builds off of the Assistants API in Azure OpenAI, and offers several additional features, such as:

- Several additional tools to enhance your AI agents' functionality, such as the ability to use Bing and as a knowledge source and call functions.
- The ability to use non Azure OpenAI models:
  - Llama 3.1-70B-instruct
  - Mistral-large-2407
  - Cohere command R+
- Enterprise ready security with secure data handling, keyless authentication, and no public egress.
- The ability to either use Microsoft-managed storage, or bring your own.
- SDK support for:
  - .NET
  - The Azure Python SDK
  - The OpenAI Python SDK
- Debugging support using tracing with Application Insights

Next steps

Use the quickstart article to get started creating a new AI Agent.

Azure AI Agent Service frequently asked questions

FAQ

If you can't find answers to your questions in this document, and still need help check the Azure AI services support options guide. Azure AI Agent Service is part of Azure AI services.

Do you store any data used in the AI Agent Service API?

Yes. Unlike Chat Completions API, Azure AI Agent Service is a stateful API, meaning it retains data. There are two types of data stored in the AI Agent Service API:

- Stateful entities: Threads, messages, and runs created during AI Agent Service use.
- Files: Uploaded during AI Agent Service setup or as part of a message.

Where is this data stored?

Data is stored in a secure, Microsoft-managed storage account that is logically separated.

How long is this data stored?

All used data persists in this system unless you explicitly delete this data. Use the delete function with the thread ID of the thread you want to delete. Clearing the Run in the AI Agent Service Playground doesn't delete threads, however deleting them using delete function won't list them in the thread page.

Does AI Agent Service support customer-managed key encryption (CMK)?

Today we support CMK for Threads and Files in AI Agent Service.

Is my data used by Microsoft for training models?

No. Data is not used for Microsoft not used for training models. See the Responsible AI documentation for more information.

Where is data stored geographically?

Azure AI Agent Service endpoints are regional, and data is stored in the same region as the endpoint. For more information, see the Azure data residency documentation.

How am I charged for AI Agent Service?

- Inference cost (input and output) of the base model you're using for each Agent (for example gpt-4-0125). If you've created multiple agents, you'll be charged for the base model attached to each Agent.
- If you've enabled the Code Interpreter tool - for example your agent calls Code Interpreter simultaneously in two different threads, this would create two Code Interpreter sessions, each of which would be charged. Each session is active by default for one hour, which means that you would only pay this fee once if your user keeps giving instructions to Code Interpreter in the same thread for up to one hour.
- File search is billed based on the vector storage used.

For more information, see the pricing page.

Is there any additional pricing or quota for using AI Agent Service?

No. All quotas apply to using models with AI Agent Service.

Does the AI Agent Service API support non-Azure OpenAI models?

Yes, the AI Agent Service API supports non-Azure OpenAI models. See the models page for more information.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quickstart: Create a new agent (Preview)

Article • 02/19/2025

Azure AI Agent Service allows you to create AI agents tailored to your needs through custom instructions and augmented by advanced tools like code interpreter, and custom functions.

Prerequisites

- An Azure subscription - Create one for free.
- Make sure you have the **Azure AI Developer** RBAC role assigned.

Basic agent setup support

Before getting started, determine if you want to perform a basic agent setup or a standard agent setup. Azure AI Foundry only supports basic agent setup.

Basic Setup: Agents use multitenant search and storage resources fully managed by Microsoft. You don't have visibility or control over these underlying Azure resources. A basic setup can be created using the Azure AI Foundry portal or an automated bicep template.

Standard Setup: Agents use customer-owned, single-tenant search and storage resources. With this setup, you have full control and visibility over these resources, but you incur costs based on your usage. Standard setup can only be performed using an automated bicep template.

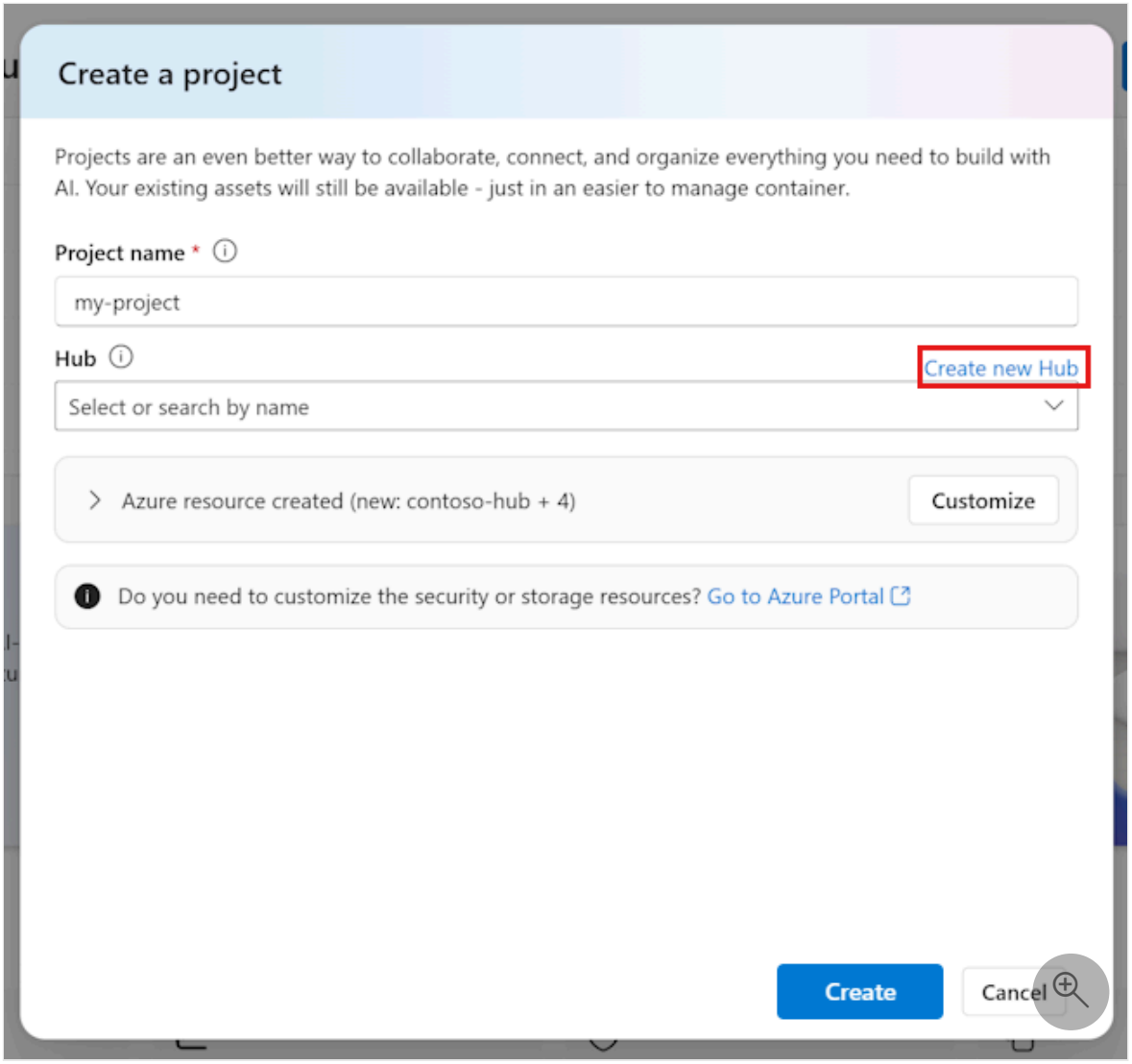
> **Important**
>
> The Azure AI Foundry portal only supports basic setup at this time. If you want to perform a standard agent setup, use the other tabs at the top of the article to learn about standard agent configuration.

Create a hub and project in Azure AI Foundry portal

To create a new hub and project, you need either the Owner or Contributor role on the resource group or on an existing hub. If you're unable to create a hub due to permissions, reach out to your administrator.

To create a project in Azure AI Foundry, follow these steps:

1. Go to Azure AI Foundry. If you are in a project, select Azure AI Foundry at the top left of the page to go to the Home page.
2. Select + **Create project**.
3. Enter a name for the project.
4. If you have a hub, you'll see the one you most recently used selected.
5. If you have access to more than one hub, you can select a different hub from the dropdown.
6. If you want to create a new one, select **Create new hub** and supply a name. If you want to customize the default values, see the Azure AI Foundry documentation.



7. Select **Create**.

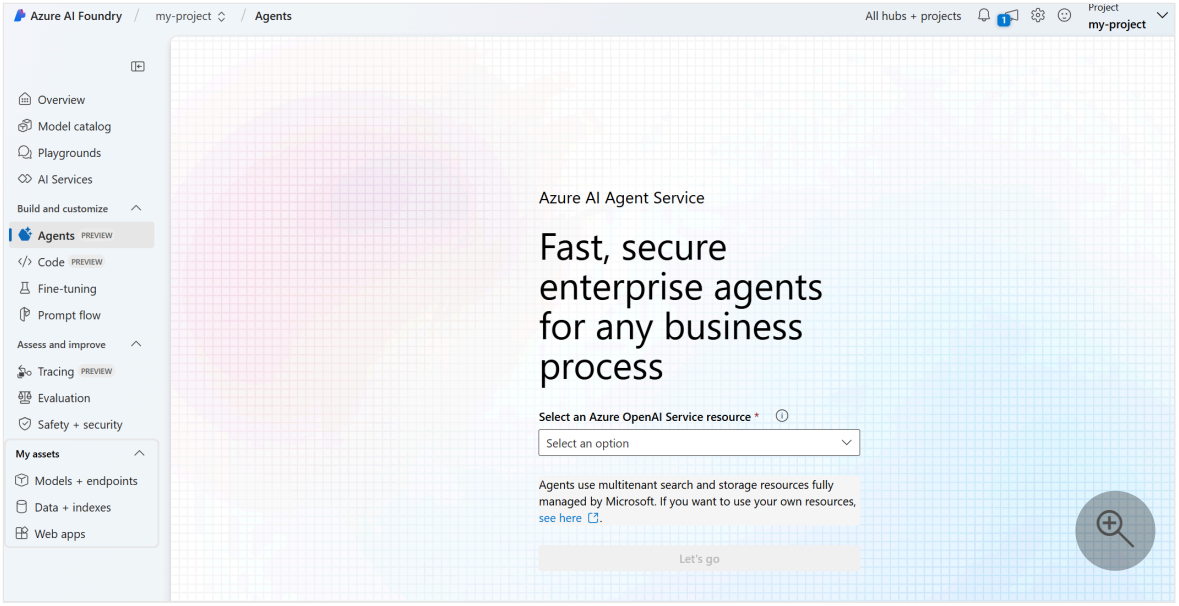
Deploy a model

1. Sign in to Azure AI Foundry.

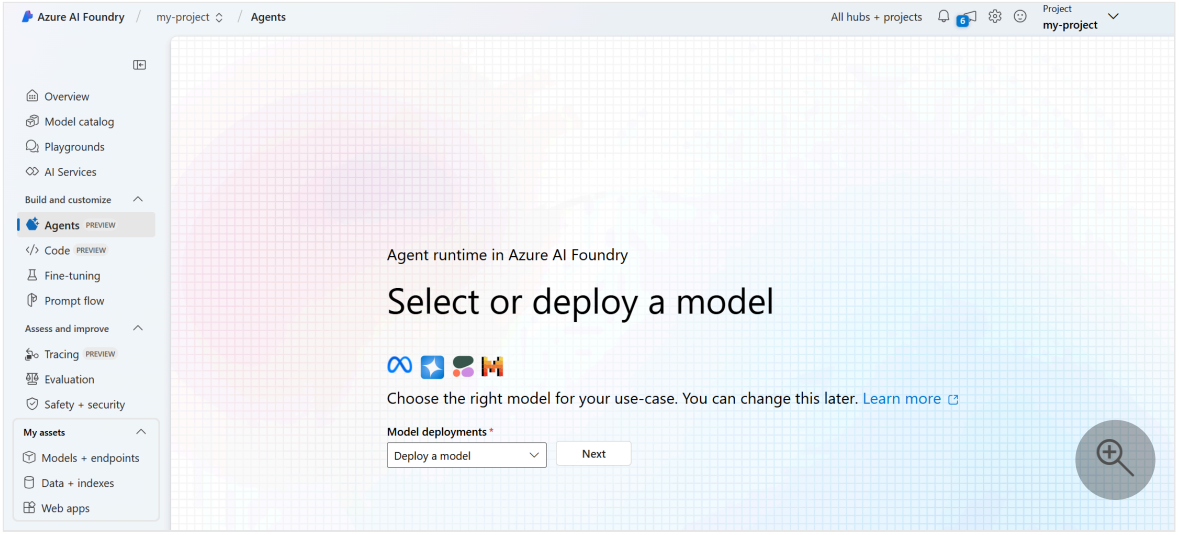
2. Go to your project or create a new project in Azure AI Foundry portal.

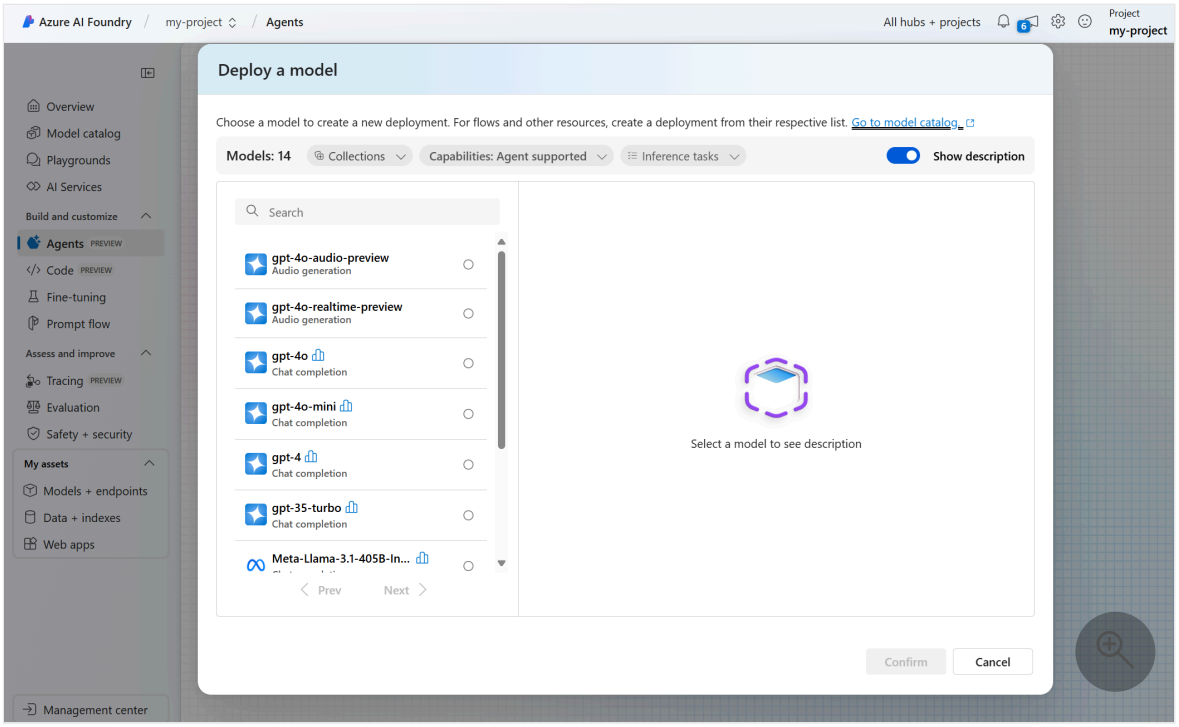
3. From your project overview, select **Agents**, located under **Build and customize**.

4. Select your Azure OpenAI resource.



5. Select a model deployment for the Agent to use. If you don't have one, a screen to deploy a new model will open. Otherwise you can select **Deploy a model**.



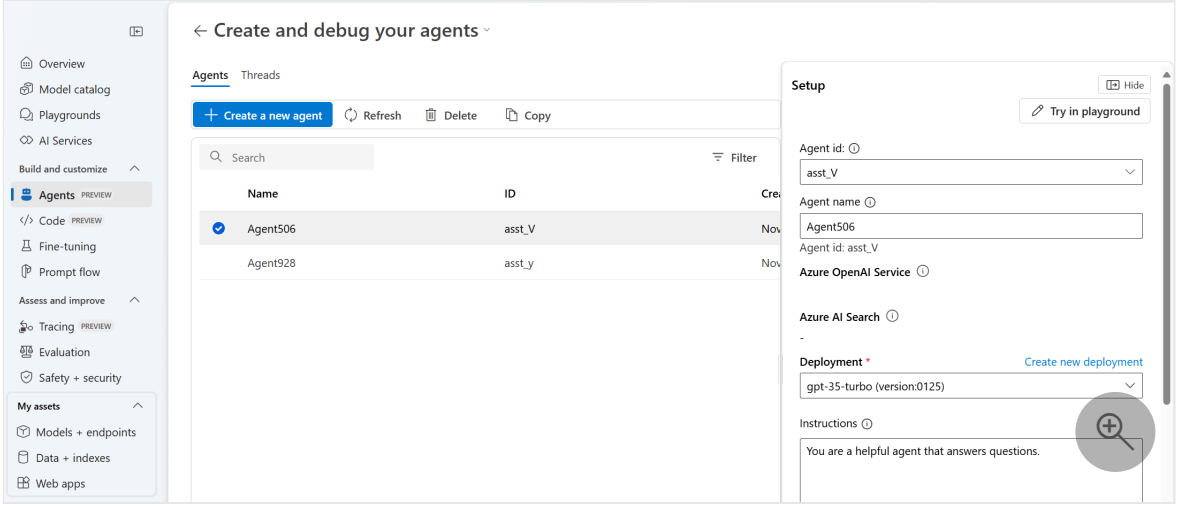


Use the agent playground

The **Agents playground** allows you to explore, prototype, and test agents without needing to run any code. From this page, you can quickly iterate and experiment with new ideas.

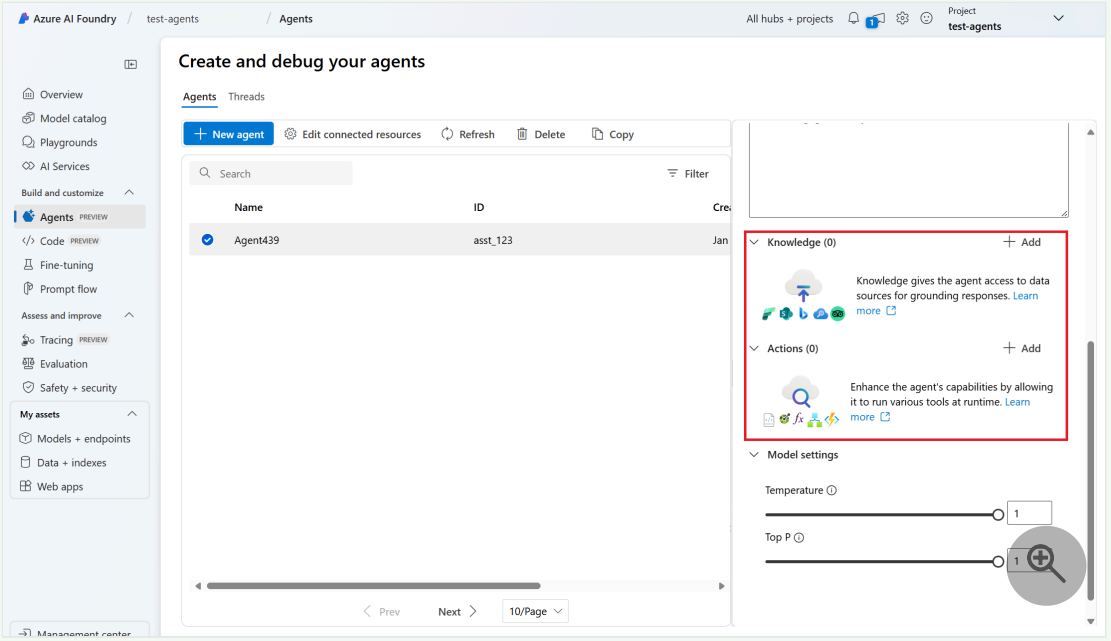
1. In the **Create and debug your agents** screen, select your agent, or create a new one with **New agent**. The **Setup** pane on the right is where you can change its parameters and tools.

   You can optionally give your agent a name other than the one generated for it, and add instructions to help improve its performance. Give your agent clear directions on what to do and how to do it. Include specific tasks, their order, and any special instructions like tone or engagement style.



Tip

Your agent can access multiple tools such as code interpreter that extend its capabilities, such as the ability to search the web with Bing, run code, and more. In the **Setup** pane, scroll down to **knowledge** and **action** and select **Add** to see the tools available for use.



See also

Check out the models that you can use with Agents.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Azure AI Agent Service quotas and limits

Article • 01/08/2025

This article contains a reference and a detailed description of the quotas and limits for Azure AI Agent Service.

Quotas and limits for the Azure AI Agent Service

The following sections provide you with a guide to the default quotas and limits that apply to Azure AI Agent Service:

Expand table

Limit Name	Limit Value
Max files per agent/thread	10,000 when using the API or Azure AI Foundry portal. In Azure OpenAI Studio the limit was 20.
Max file size for agents & fine-tuning	512 MB
Max size for all uploaded files for agents	100 GB
agents token limit	2,000,000 token limit

Quotas and limits for Azure OpenAI models

See the Azure OpenAI for current quotas and limits for the Azure OpenAI models that you can use with Azure AI Agent Service.

Next steps

Learn about the models available for Azure AI Agent Service

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Models supported by Azure AI Agent Service

Article • 04/15/2025

Agents are powered by a diverse set of models with different capabilities and price points. Model availability varies by region and cloud. Certain tools and capabilities require the latest models. The following models are available in the REST API and SDKs.

Azure OpenAI models

Azure OpenAI provides customers with choices on the hosting structure that fits their business and usage patterns. The service offers two main types of deployment:

- **Standard** is offered with a global deployment option, routing traffic globally to provide higher throughput.
- **Provisioned** is also offered with a global deployment option, allowing customers to purchase and deploy provisioned throughput units across Azure global infrastructure.

All deployments can perform the exact same inference operations, however the billing, scale, and performance are substantially different. To learn more about Azure OpenAI deployment types see our deployment types guide.

Azure AI Agent Service supports the following Azure OpenAI models in the listed regions.

> **Note**
>
> The following table is for pay-as-you-go. For information on Provisioned Throughput Unit (PTU) availability, see provisioned throughput in the Azure OpenAI documentation. `GlobalStandard` customers also have access to global standard models.

Expand table

Region	gpt-4o, 2024-11-20	gpt-4o, 2024-05-13	gpt-4o, 2024-08-06	gpt-4o-mini, 2024-07-18	gpt-4, 0613	gpt-4, 1106-Preview	gpt-4, 0125-Preview	gpt-4, turbo-2024-04-09	gpt-4-32k, 0613	gpt-35-turbo, 0613	gpt-35-turbo, 1106	gpt-35-turbo, 0125	gpt-35-turbo-16k, 0613
australiaeast		-	-	-	✅	✅	-	-	✅	✅	✅	✅	✅
eastus		✅	✅	✅	-	-	✅	✅	-	✅	-	✅	✅
eastus2		✅	✅	✅	-	✅	-	✅	-	✅	-	✅	✅
francecentral		-	-	-	✅	✅	-	-	✅	✅	✅	-	✅
japaneast	✅	-	-	-	-	-	-	-	-	✅	-	✅	✅
norwayeast		-	-	-	-	✅	-	-	-	-	-	-	-
polandcentral	-	-	-	-	-	-	-	-	-	-	-	-	-
southindia		-	-	-	-	✅	-	-	-	-	✅	✅	-
swedencentral	✅	✅	✅	✅	✅	✅	-	✅	✅	✅	✅	-	✅
switzerlandnorth	-	-	-	-	✅	-	-	-	✅	-	-	✅	✅

Region	gpt-4o, 2024-11-20	gpt-4o, 2024-05-13	gpt-4o, 2024-08-06	gpt-4o-mini, 2024-07-18	gpt-4, 0613	gpt-4, 1106-Preview	gpt-4, 0125-Preview	gpt-4, turbo-2024-04-09	gpt-4-32k, 0613	gpt-35-turbo, 0613	gpt-35-turbo, 1106	gpt-35-turbo, 0125	gpt-35-turbo-16k, 0613
uaenorth		-	-	-	-	-	-	-	-	-	-	-	-
uksouth		-	-	-	-	✅	✅	-	-	✅	✅	✅	✅
westus		✅	✅	✅	-	✅	-	✅	-	-	✅	✅	-
westus3		✅	✅	✅	-	✅	-	✅	-	-	-	✅	-

Non-Microsoft models

The Azure AI Agent Service also supports the following models from the Azure AI Foundry model catalog.

- Meta-Llama-405B-Instruct
- Cohere-command-r-plus
- Cohere-command-r

To use these models, you can use Azure AI Foundry portal to make a deployment, and then reference the deployment name in your agent. For example:

Python

```
agent = project_client.agents.create_agent( model="llama-3", name="my-agent", instructions="You are a helpful agent" )
```

Next steps

Create a new Agent project

Grounding with Bing Search

Article • 04/02/2025

Grounding with Bing Search allows your Azure AI Agents to incorporate real-time public web data when generating responses. You need to create a Grounding with Bing Search resource, and then connect this resource to your Azure AI Agents. When a user sends a query, Azure AI Agents decide if Grounding with Bing Search should be leveraged or not. If so, it will leverage Bing to search over public web data and return relevant chunks. Lastly, Azure AI Agents will use returned chunks to generate a response.

You can ask questions such as "*what is the top news today*" or "*what is the recent update in the retail industry in the US?*", which require real-time public data.

Developers and end users don't have access to raw content returned from Grounding with Bing Search. The model response, however, includes citations with links to the websites used to generate the response, and a link to the Bing query used for the search. You can retrieve the **model response** by accessing the data in the thread that was created. These two *references* must be retained and displayed in the exact form provided by Microsoft, as per Grounding with Bing Search's Use and Display Requirements. See the how to display Grounding with Bing Search results section for details.

> **Important**
>
> 1. Your usage of Grounding with Bing Search can incur costs. See the **pricing page** for details.
> 2. By creating and using a Grounding with Bing Search resource through code-first experience, such as Azure CLI, or deploying through deployment template, you agree to be bound by and comply with the terms available at https://www.microsoft.com/en-us/bing/apis/grounding-legal, which may be updated from time to time.
> 3. When you use Grounding with Bing Search, your customer data is transferred outside of the Azure compliance boundary to the Grounding with Bing Search service. Grounding with Bing Search is not subject to the same data processing terms (including location of processing) and does not have the same compliance standards and certifications as the Azure AI Agent Service, as described in the Grounding with Bing Search Terms of Use. It is your

responsibility to assess whether use of Grounding with Bing Search in your agent meets your needs and requirements.

How Grounding with Bing Search works

The user query is the message that an end user sends to an agent, such as *"should I take an umbrella with me today? I'm in Seattle."* Instructions are the system message a developer can provide to share context and provide instructions to the AI model on how to use various tools or behave.

When a user sends a query, the customer's AI model deployment first processes it (using the provided instructions) to later perform a Bing search query (which is visible to developers). Grounding with Bing returns relevant search results to the customer's model deployment, which then generates the final output.

> **Note**
>
> When using Grounding with Bing Search, only the Bing search query and your resource key are sent to Bing, and no end user-specific information is included. Your resource key is sent to Bing solely for billing and rate limiting purposes.

The authorization will happen between Grounding with Bing Search service and Azure AI Agent service. Any Bing search query that is generated and sent to Bing for the purposes of grounding is transferred, along with the resource key, outside of the Azure compliance boundary to the Grounding with Bing Search service. Grounding with Bing Search is subject to Bing's terms and do not have the same compliance standards and certifications as the Azure AI Agent Service, as described in the Grounding with Bing Search Terms of Use. It is your responsibility to assess whether the use of Grounding with Bing Search in your agent meets your needs and requirements.

Usage support

Expand table

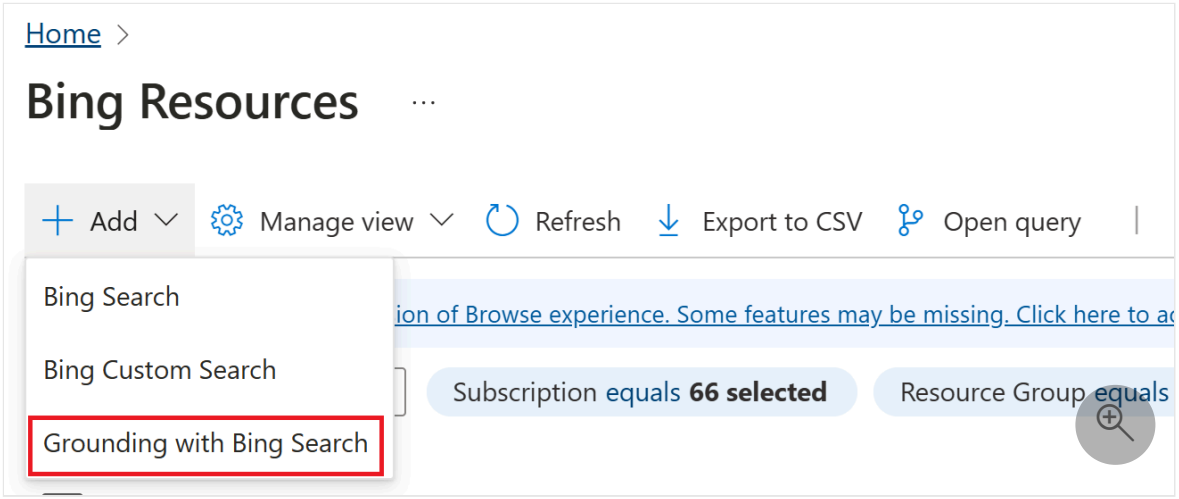
Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	✔	✔

Setup

> **Note**
>
> 1. Grounding with Bing Search works with all Azure OpenAI models that Azure AI Agent Service supports, except `gpt-4o-mini, 2024-07-18`.

1. Create an Azure AI Agent by following the steps in the quickstart.

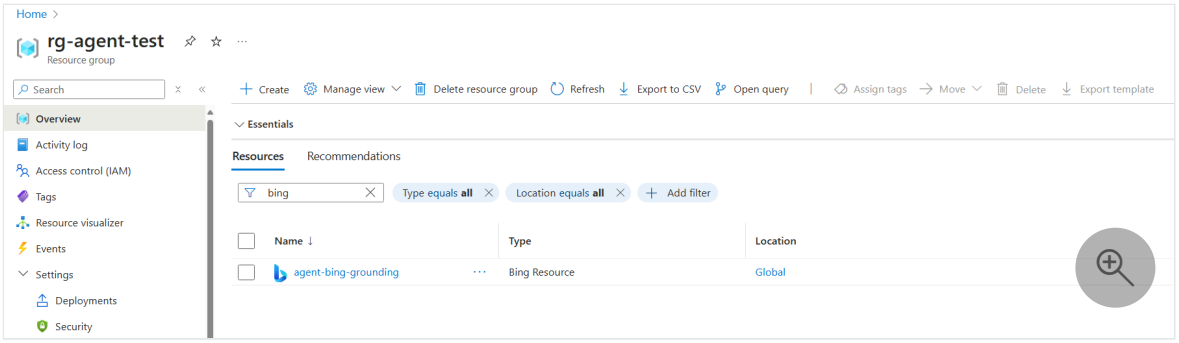
2. Create a Grounding with Bing Search resource. You need to have `owner` or `contributor` role in your subscription or resource group to create it.
   a. You can create one in the Azure portal, and select the different fields in the creation form. Make sure you create this Grounding with Bing Search resource in the same resource group as your Azure AI Agent, AI Project, and other resources.



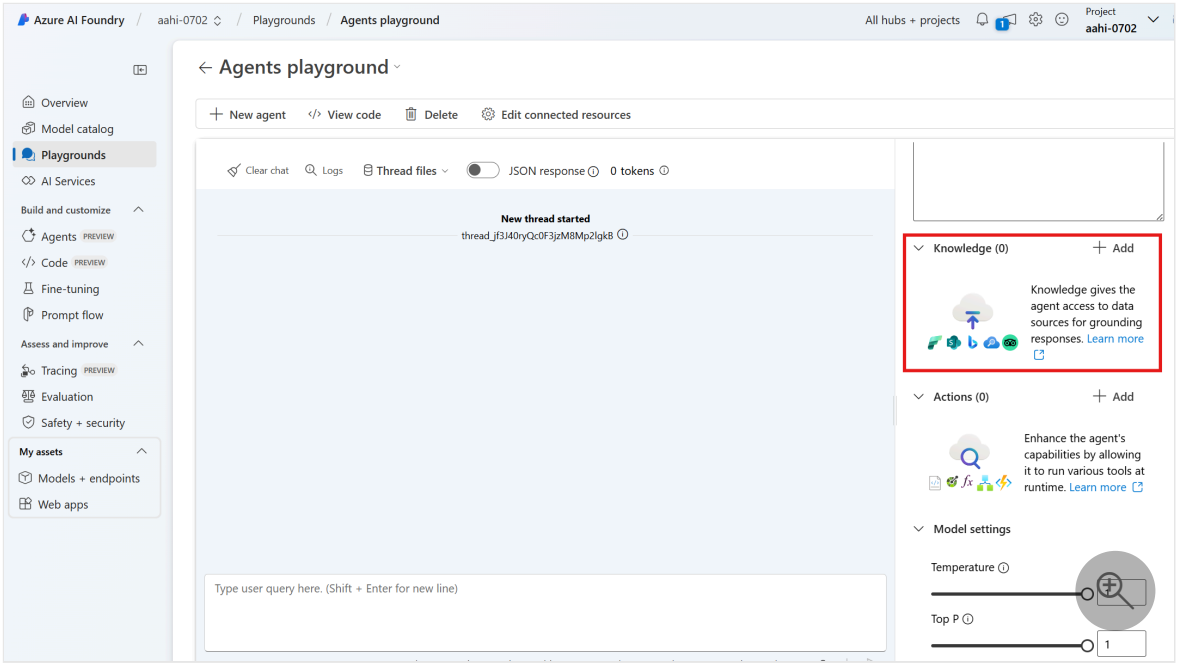
   a. You can also create one through code-first experience. If so, you need to manually register Bing Search as an Azure resource provider. You must have permission to perform the `/register/action` operation for the resource provider. The permission is included in the **Contributor** and **Owner** roles.

```
Console
    az provider register --namespace 'Microsoft.Bing'
```

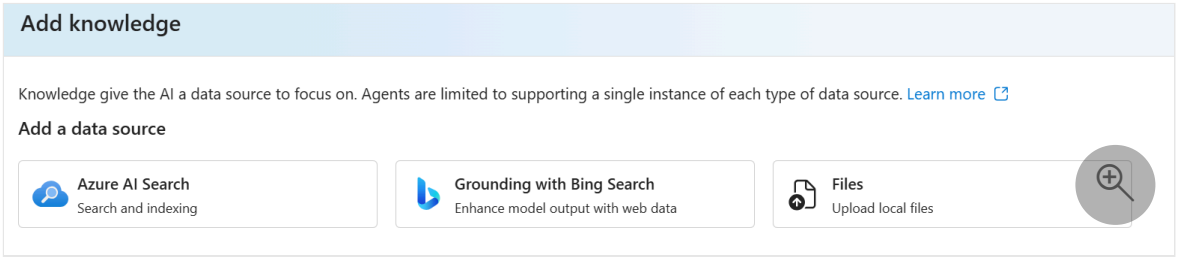
3. After you have created a Grounding with Bing Search resource, you can find it in Azure portal. Navigate to the resource group you've created the resource in, search for the Grounding with Bing Search resource you have created.



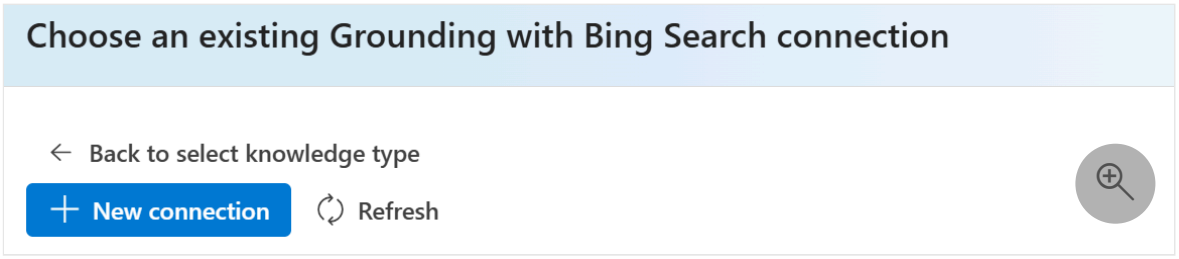
4. You can add the Grounding with Bing Search tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal, in the **Create and debug** screen for your agent, scroll down the **Setup** pane on the right to **knowledge**. Then select **Add**.



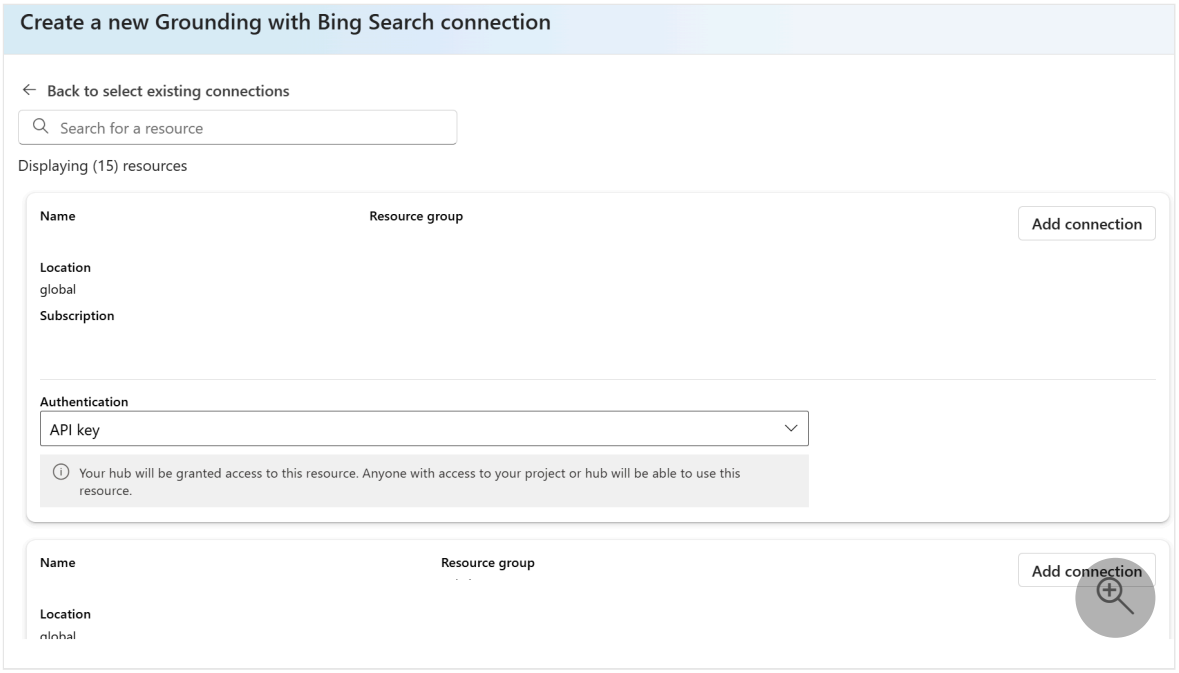
5. Select **Grounding with Bing Search** and follow the prompts to add the tool. Note you can add only one per agent.



6. Click to add new connections. Once you have added a connection, you can directly select from existing list.



7. Select the Grounding with Bing Search resource you want to use and click to add connection.

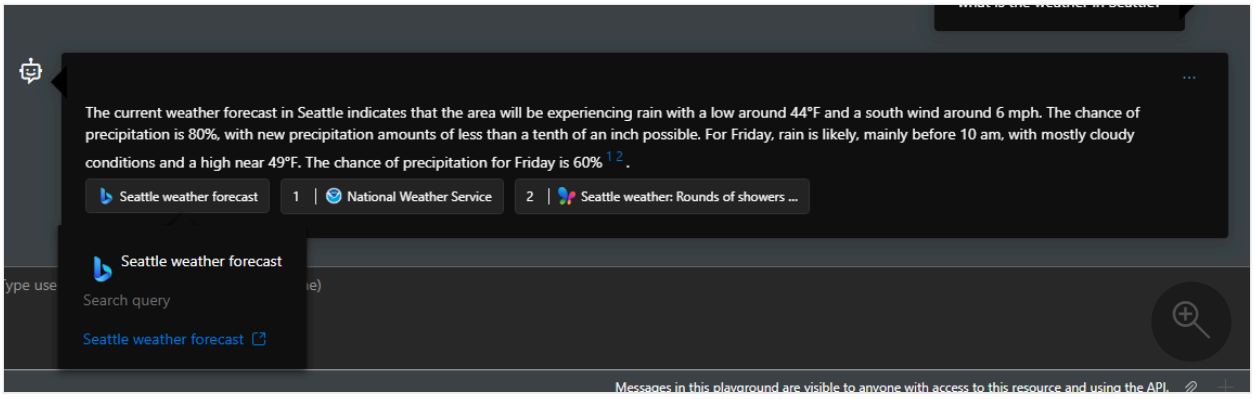


How to display Grounding with Bing Search results

According to Grounding with Bing's terms of use and use and display requirements, you need to display both website URLs and Bing search query URLs in your custom interface. You can find website URLs through `annotations` parameter in API response and Bing search query URLs through `runstep` details. To render the webpage, we recommend you replace the endpoint of Bing search query URLs with `www.bing.com` and your Bing search query URL would look like "https://www.bing.com/search?q={search query}"

Python

```
run_steps = project_client.agents.list_run_steps(run_id=run.id,
thread_id=thread.id)
run_steps_data = run_steps['data']
print(f"Last run step detail: {run_steps_data}")
```



Next steps

See the full sample for Grounding with Bing Search.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Azure AI Agent Service file search tool

Article • 02/06/2025

File search augments agents with knowledge from outside its model, such as proprietary product information or documents provided by your users.

> **Note**
>
> Using the standard agent setup, the improved file search tool ensures your files remain in your own storage, and your Azure AI Search resource is used to ingest them, ensuring you maintain complete control over your data.

File sources

- Upload local files
- Azure Blob Storage

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	File upload only	File upload and using BYO blob storage

Dependency on agent setup

Basic agent setup

The file search tool has the same functionality as Azure OpenAI Assistants. Microsoft managed search and storage resources are used.

- Uploaded files get stored in Microsoft managed storage
- A vector store is created using a Microsoft managed search resource

Standard agent setup

The file search tool uses the Azure AI Search and Azure Blob Storage resources you connected during agent setup.

- Uploaded files get stored in your connected Azure Blob Storage account
- Vector stores get created using your connected Azure AI Search resource

For both agent setups, Azure OpenAI handles the entire ingestion process, which includes:

- Automatically parsing and chunking documents
- Generating and storing embeddings
- Utilizing both vector and keyword searches to retrieve relevant content for user queries.

There is no difference in the code between the two setups; the only variation is in where your files and created vector stores are stored.

How it works

The file search tool implements several retrieval best practices out of the box to help you extract the right data from your files and augment the model's responses. The file search tool:

- Rewrites user queries to optimize them for search.
- Breaks down complex user queries into multiple searches it can run in parallel.
- Runs both keyword and semantic searches across both agent and thread vector stores.
- Reranks search results to pick the most relevant ones before generating the final response.
- By default, the file search tool uses the following settings:
  - Chunk size: 800 tokens
  - Chunk overlap: 400 tokens
  - Embedding model: text-embedding-3-large at 256 dimensions
  - Maximum number of chunks added to context: 20

Vector stores

Vector store objects give the file search tool the ability to search your files. Adding a file to a vector store automatically parses, chunks, embeds, and stores the file in a vector

database that's capable of both keyword and semantic search. Each vector store can hold up to 10,000 files. Vector stores can be attached to both agents and threads. Currently you can attach at most one vector store to an agent and at most one vector store to a thread.

Similarly, these files can be removed from a vector store by either:

- Deleting the vector store file object or,
- By deleting the underlying file object, which removes the file it from all vector_store and code_interpreter configurations across all agents and threads in your organization

The maximum file size is 512 MB. Each file should contain no more than 5,000,000 tokens per file (computed automatically when you attach a file).

Ensuring vector store readiness before creating runs

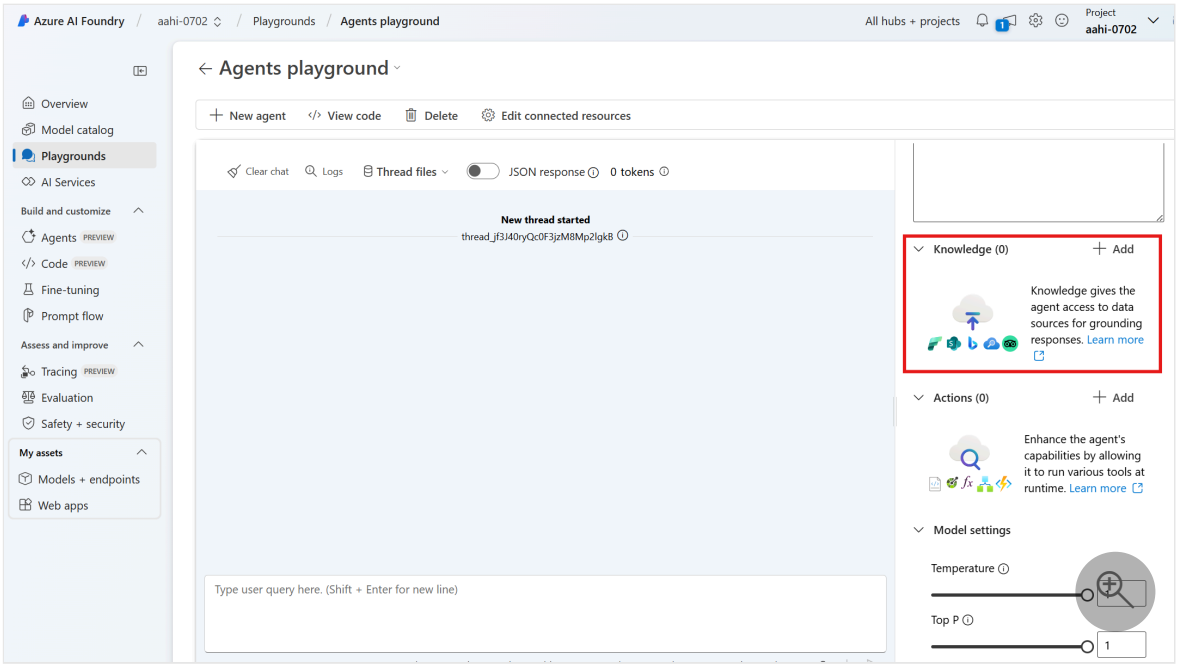
We highly recommend that you ensure all files in a vector_store are fully processed before you create a run. This ensures that all the data in your vector store is searchable. You can check for vector store readiness by using the polling helpers in the SDKs, or by manually polling the vector store object to ensure the status is completed.

As a fallback, there's a 60-second maximum wait in the run object when the thread's vector store contains files that are still being processed. This is to ensure that any files your users upload in a thread are fully searchable before the run proceeds. This fallback wait does not apply to the agent's vector store.

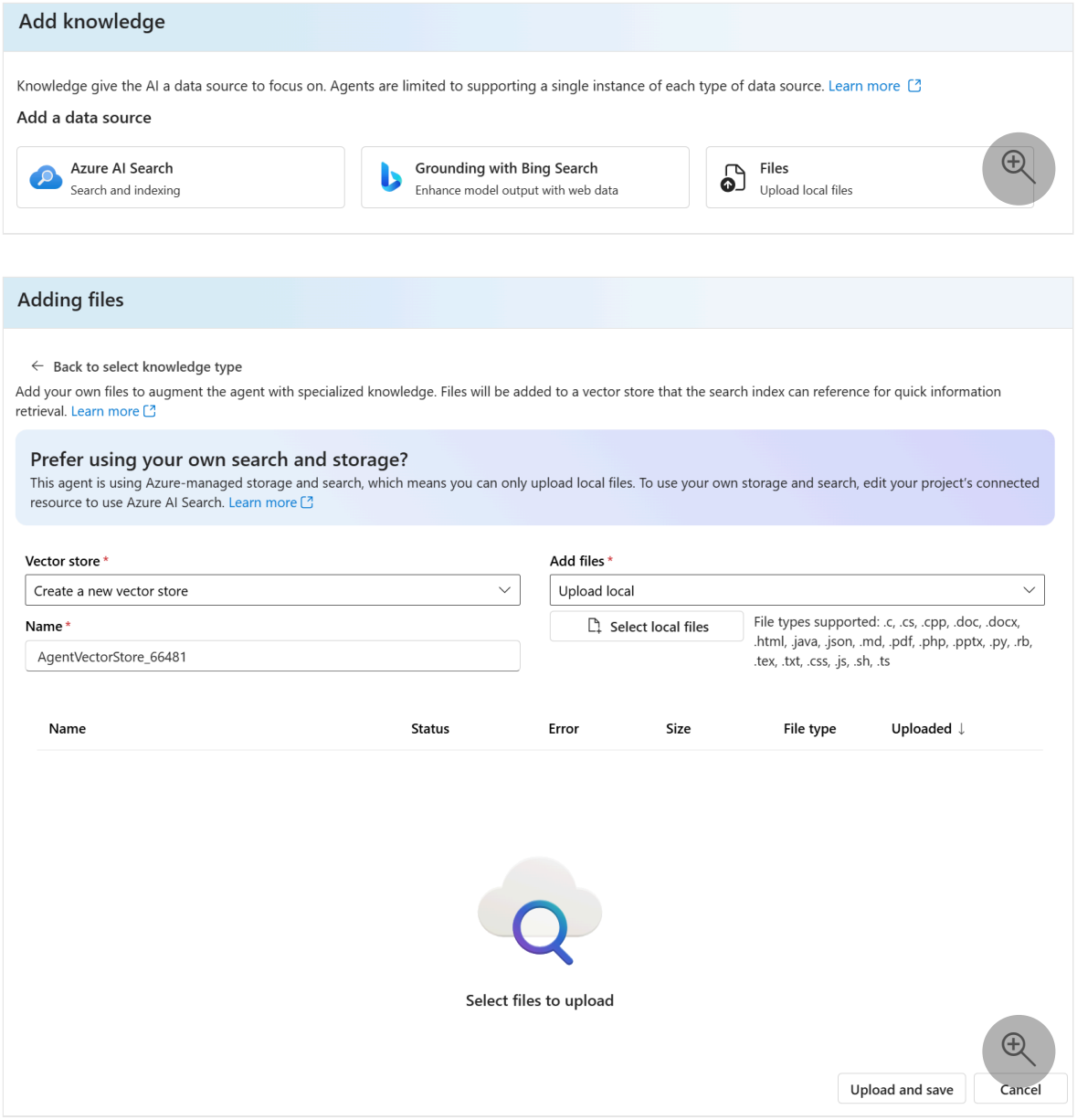
Add file search to an agent using the Azure AI Foundry portal

You can add the Bing Search tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal:

1. In the **Create and debug** screen for your agent, scroll down the **Setup** pane on the right to **knowledge**. Then select **Add**.



2. Select **Files** and follow the prompts to add the tool.



Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Use an existing AI Search index with the Azure AI Search tool

Article • 02/19/2025

Use an existing Azure AI Search index with the agent's Azure AI Search tool.

> **Note**
>
> Azure AI Search indexes must meet the following requirements:
>
> - The index must contain at least one searchable & retrievable text field (type Edm.String)
> - The index must contain at least one searchable vector field (type Collection(Edm.Single))
> - The index must use a vector profile/integrated vectorization

Search types

Index without semantic configuration

- By default, the Azure AI Search tool runs a hybrid search (keyword + vector) on all text fields.

Index with semantic configuration

- By default, the Azure AI Search tool runs hybrid + semantic search on all text fields.

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	✔	✔

Setup: Create an agent that can use an existing Azure AI Search index

Prerequisite: Have an existing Azure AI Search index

A prerequisite of using the Azure AI Search tool is to have an existing Azure AI Search index. If you don't have an existing index, you can create one in the Azure portal using the import and vectorize data wizard.

- Quickstart: Create a vector index with the import and vectorize data wizard in the Azure portal

Create a project connection to the Azure AI Search resource with the index you want to use

Once you have completed the agent setup, you must create a project connection to the Azure AI Search resource that contains the index you want to use.

If you already connected the AI Search resource that contains the index you want to use to your project, skip this step.

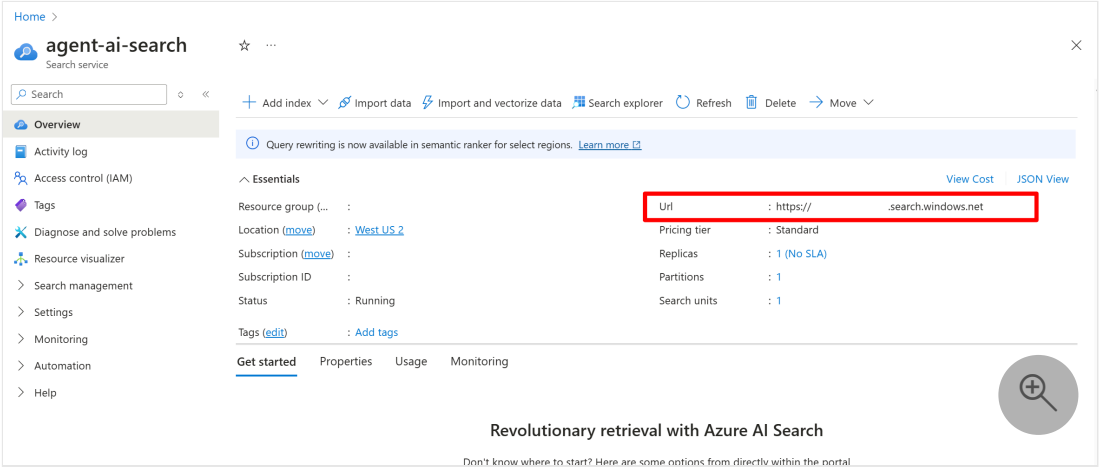
Get your Azure AI Search resource connection key and endpoint

1. Access your Azure AI Search resource.

    - In the Azure portal, navigate to the AI Search resource that contains the index you want to use.

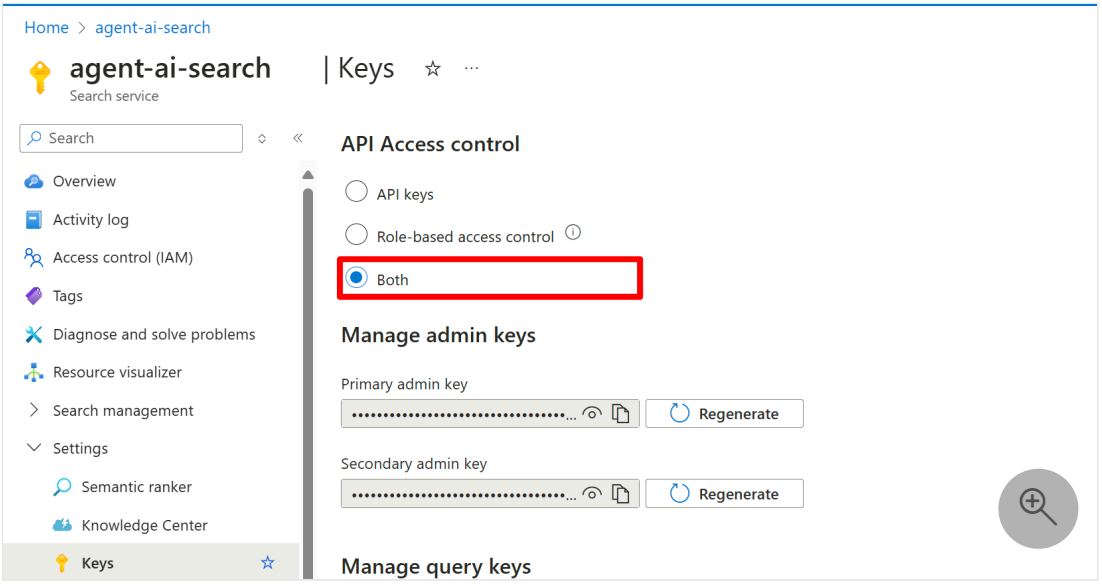
2. Copy the connection endpoint.

    - In the Overview tab, copy the URL of your resource. The URL should be in the format `https://<your-resource-name>.search.windows.net/`.



3. Verify API Access control is set to **Both** and copy one of the keys under **Manage admin keys**.

- From the left-hand navigation bar, scroll down to the Settings section and select **Keys**.
- Under the **API Access Control** section, ensure the option **Both** API key and Role-based access control is selected.
- If you want the connection to use API Keys for authentication, copy one of the keys under **Manage admin keys**.



Create an Azure AI Search project connection

If you use Microsoft Entra ID for the connection authentication type, you need to manually assign the project managed identity the roles Search Index Data Contributor and Search Service Contributor to the Azure AI Search resource.

Azure CLI

Create the following connections.yml file

You can use either an API key or credential-less YAML configuration file. Replace the placeholders for `name`, `endpoint` and `api_key` with your Azure AI Search resource values. For more information on the YAML configuration file, see the Azure AI Search connection YAML schema.

- API Key example:

yml

```yml
name: my_project_acs_connection_keys
type: azure_ai_search
endpoint: https://contoso.search.windows.net/
api_key: XXXXXXXXXXXXXXX
```

- Credential-less

```yml
name: my_project_acs_connection_credentialless
type: azure_ai_search
endpoint: https://contoso.search.windows.net/
```

Then, run the following command:

Replace `my_resource` and `my_project_name` with your resource group and project name created in the agent setup.

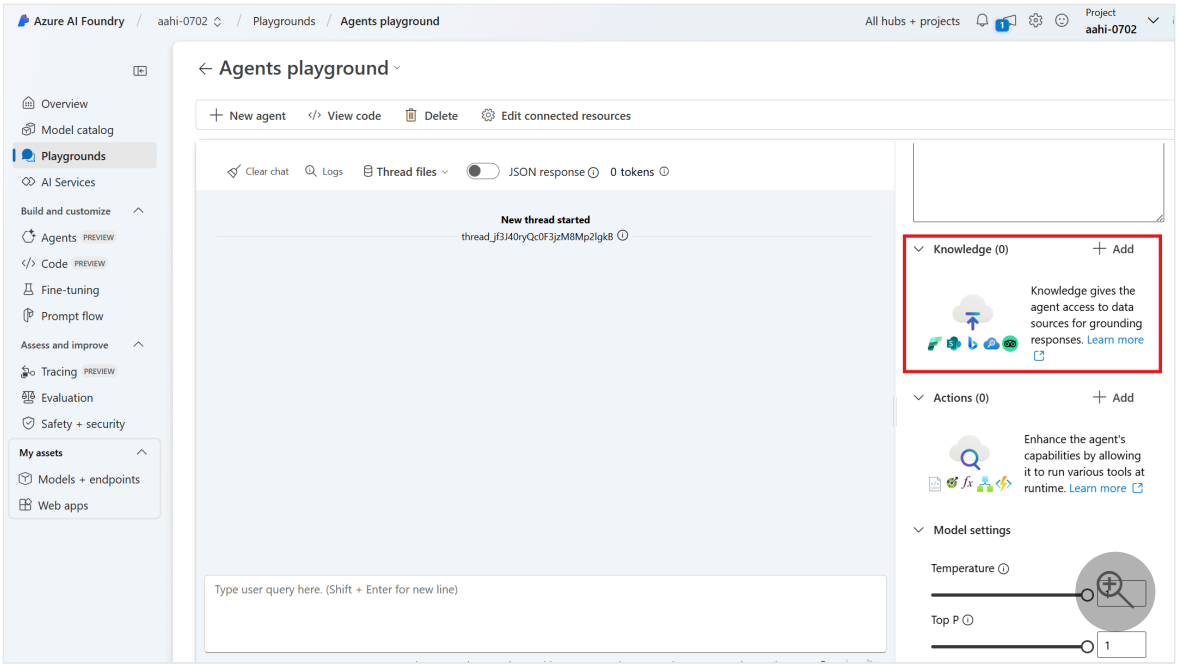
```Azure CLI
az ml connection create --file {connection.yml} --resource-group
{my_resource_group} --workspace-name {my_project_name}
```

Now that you have created a project connection to your Azure AI Search resource, you can configure and start using the Azure AI Search tool with the SDK. See the code examples tab to get started.

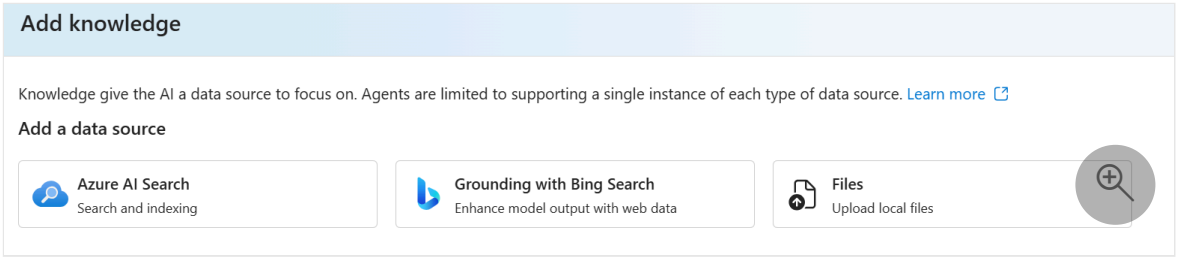
Add the Azure AI Search tool to an agent

you can add the Azure AI Search tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal:

1. In the **Create and debug** screen for your agent, scroll down the **Setup** pane on the right to **knowledge**. Then select **Add**.



2. Select **Azure AI Search** and follow the prompts to add the tool.



Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Use the Microsoft Fabric data agent

Article • 04/07/2025

Integrate your Azure AI Agent with the Microsoft Fabric data agent to unlock powerful data analysis capabilities. The Fabric data agent transforms enterprise data into conversational Q&A systems, allowing users to interact with the data through chat and uncover data-driven and actionable insights.

You need to first build and publish a Fabric data agent and then connect your Fabric data agent with the published endpoint. When a user sends a query, Azure AI Agent will first determine if the Fabric data agent should be leveraged or not. If so, it will use the end user's identity to generate queries over data they have access to. Lastly, Azure AI Agent will generate responses based on queries returned from Fabric data agents. With Identity Passthrough (On-Behalf-Of) authorization, this integration simplifies access to enterprise data in Fabric while maintaining robust security, ensuring proper access control and enterprise-grade protection.

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	✔	✔

Prerequisites

- You have created and published a Fabric data agent endpoint
- Developers and end users have at least `AI Developer` RBAC role.
- Developers and end users have at least `READ` access to the Fabric data agent and the underlying data sources it connects with.
- Your Fabric Data Agent and Azure AI Agent need to be in the same tenant.

Setup

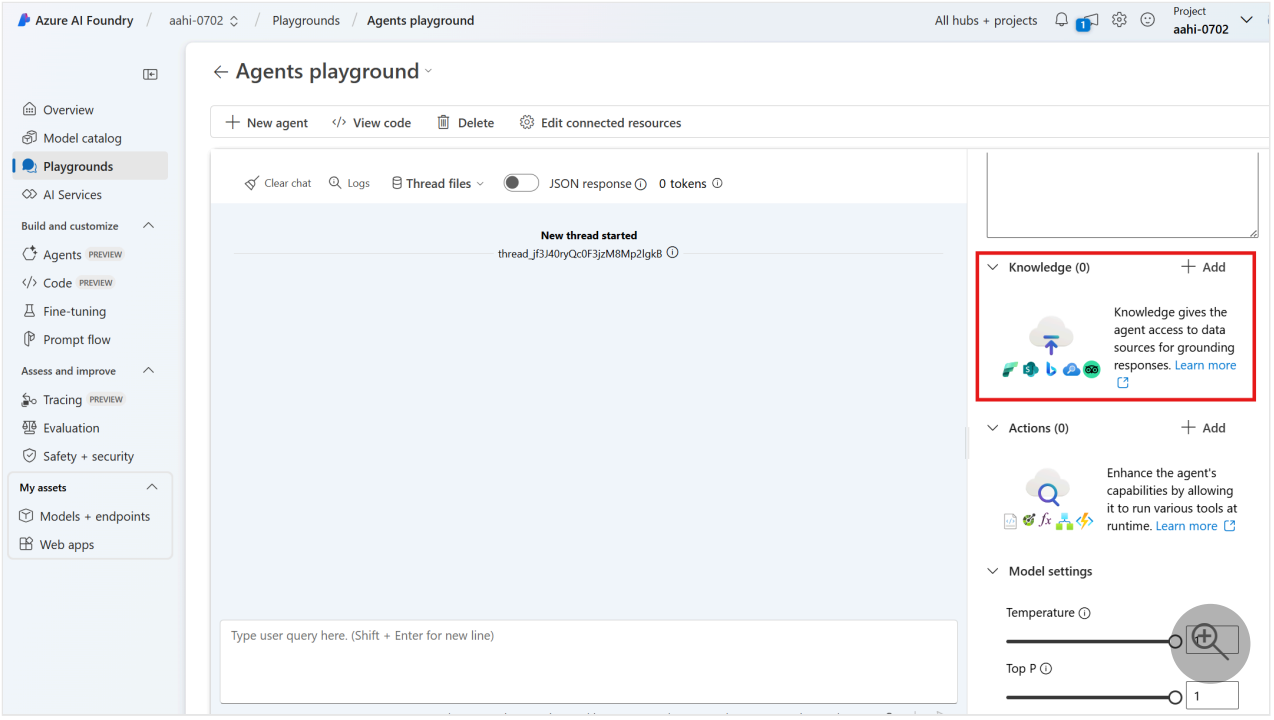
> **Note**

- The model you selected in Azure AI Agent setup is only used for agent orchestration and response generation. It doesn't impact which model Fabric data agent uses for NL2SQL operation.

1. Create an Azure AI Agent by following the steps in the quickstart.

2. Create and publish a Fabric data agent

3. You can add the Microsoft Fabric tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal, in the Create and debug screen for your agent, scroll down the Setup pane on the right to knowledge. Then select Add.

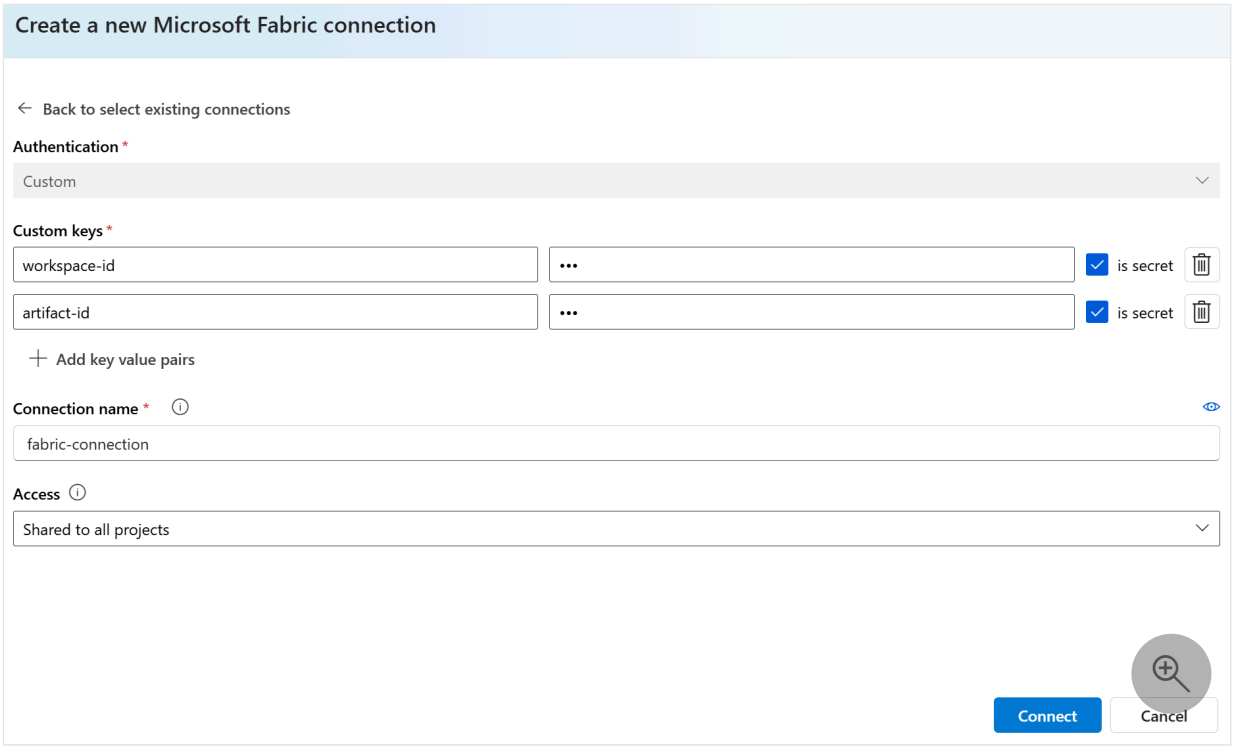


4. Select **Microsoft Fabric** and follow the prompts to add the tool. You can add only one per agent.

5. Click to add new connections. Once you have added a connection, you can directly select from existing list.

   a. To create a new connection, you need to find `workspace-id` and `artifact-id` in your published Fabric data agent endpoint. Your Fabric data agent endpoint would look like `https://<environment>.fabric.microsoft.com/groups/<workspace_id>/aiskills/<artifact-id>`

   b. Then, you can add both to your connection. Make sure you have checked `is secret` for both of them



Next steps

See the full sample for Fabric data agent.

Bring your licensed data

Article • 04/10/2025

Azure AI Agent Service integrates your own licensed data from specialized data providers, such as Tripadvisor. This integration enhances the quality of your agent's responses with high-quality, fresh data, such as travel guidance and reviews. These insights empower your agents to deliver nuanced, informed solutions tailored to specific use cases.

Tripadvisor is the first licensed data provider and you can ground with your licensed Tripadvisor data through the API, SDK, and Azure AI Foundry portal.

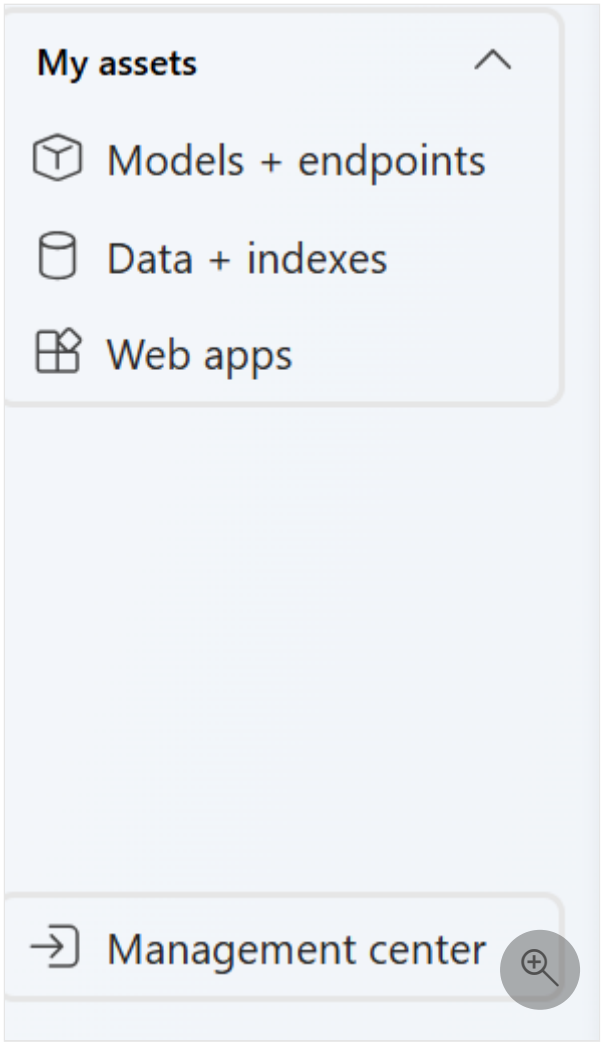
> **Important**
>
> - Your use of connected non-Microsoft services is subject to the terms between you and the service provider. By connecting to a non-Microsoft service, you acknowledge that some of your data, such as prompt content, is passed to the non-Microsoft service, and/or your application might receive data from the non-Microsoft service. You're responsible for your use of non-Microsoft data.
> - Grounding with licensed data incurs usage with licensed data providers, review the pricing plan with your selected licensed data providers.

Prerequisites

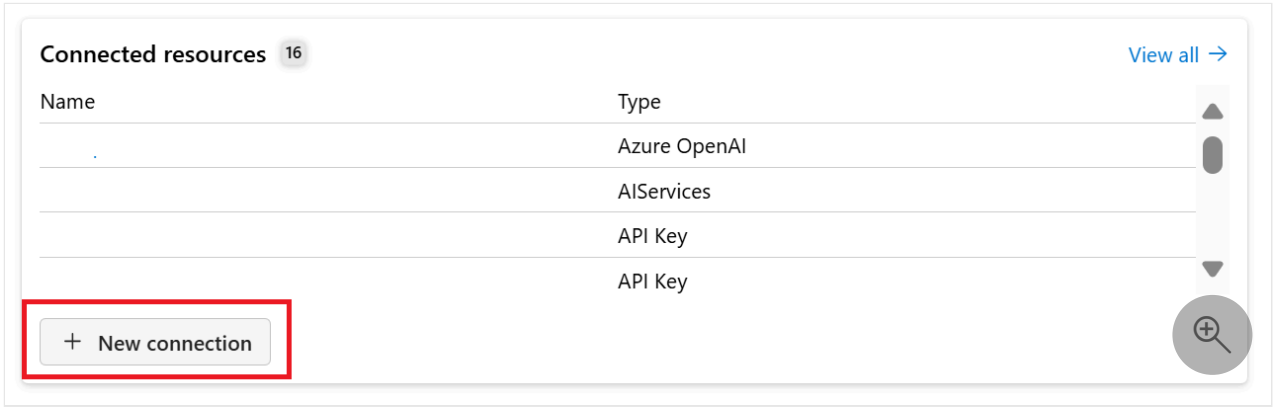
- Obtain an API key for your Tripadvisor developer account.
- Make sure when you put 0.0.0.0/0 for the IP address restriction to allow traffic from Azure AI Agent Service.

Setup

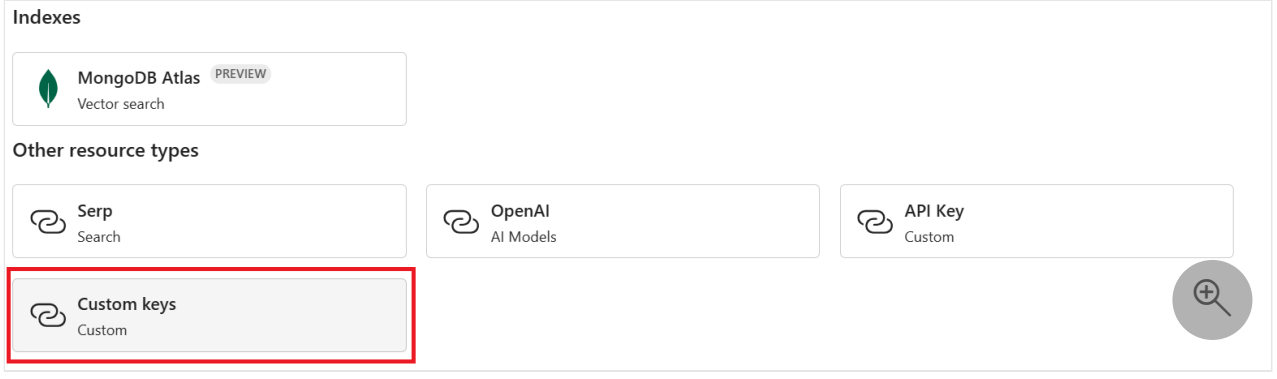
1. Go to Azure AI Foundry portal and select your AI Project. Select **Management Center**.



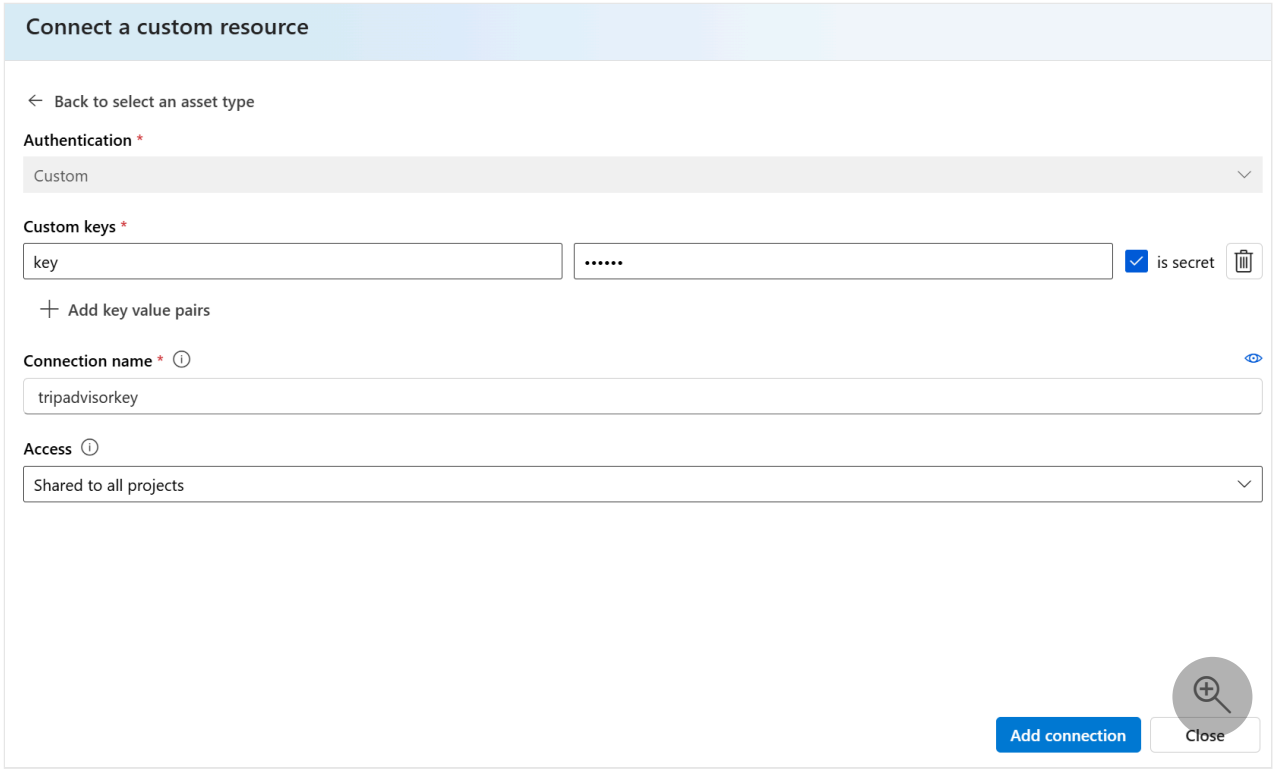
2. Select **+new connection** in the settings page.



3. Select **custom keys** in **other resource types**.

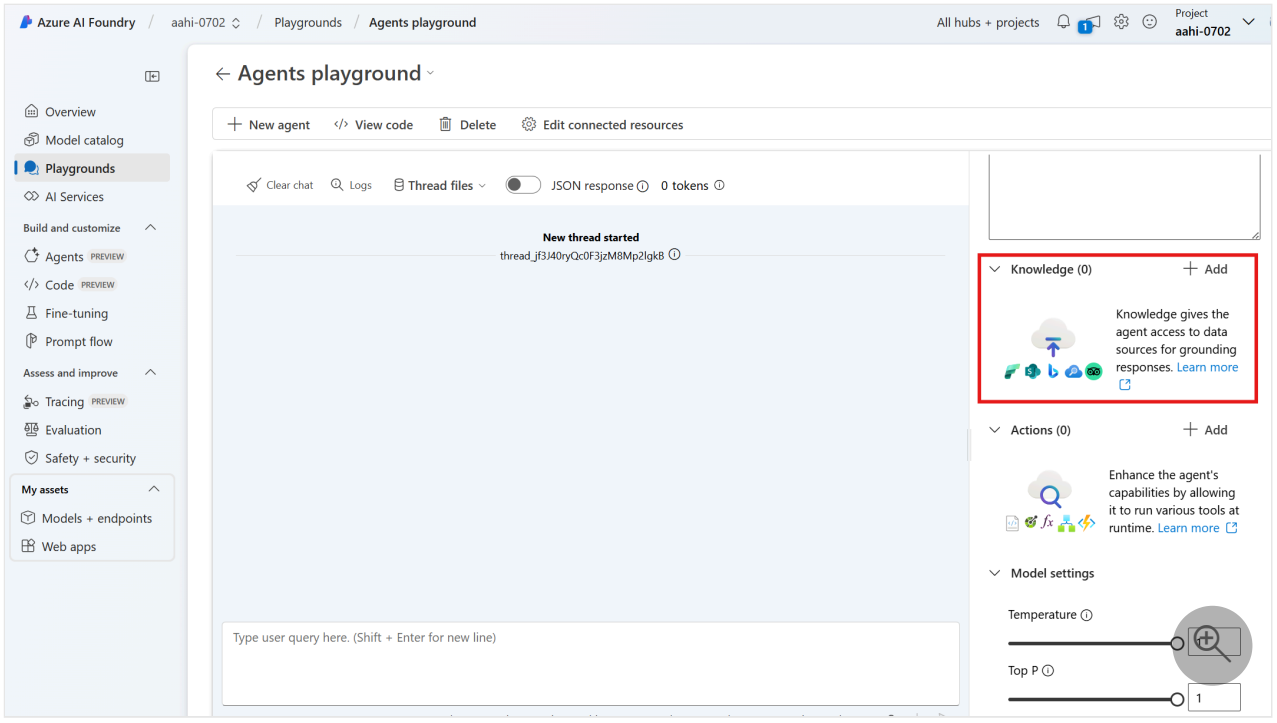


4. Enter the following information to create a connection to store your Tripadvisor key:
   a. Set **Custom keys** to "key", with the value being your Tripadvisor API key.
   b. Make sure **is secret** is checked.
   c. Set the connection name to your connection name. You use this connection name in your sample code or Foundry Portal later.
   d. For the **Access** setting, you can choose either *this project only* or *shared to all projects*. Just make sure in your code, the connection string of the project you entered has access to this connection.

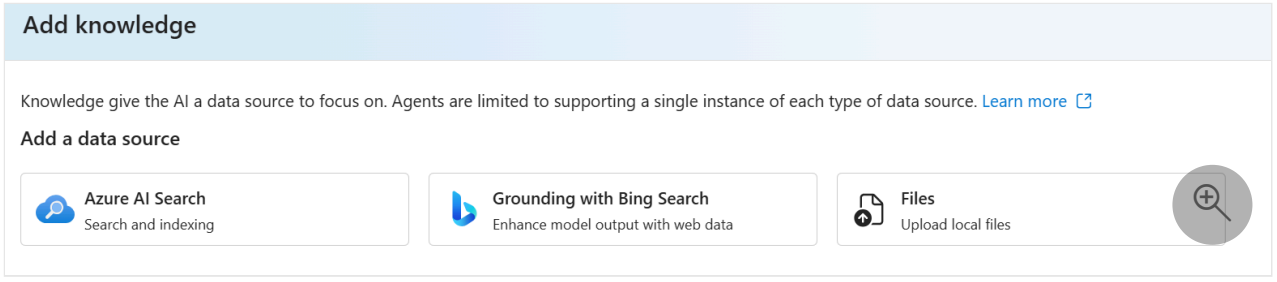


Use Tripadvisor tool through Foundry portal

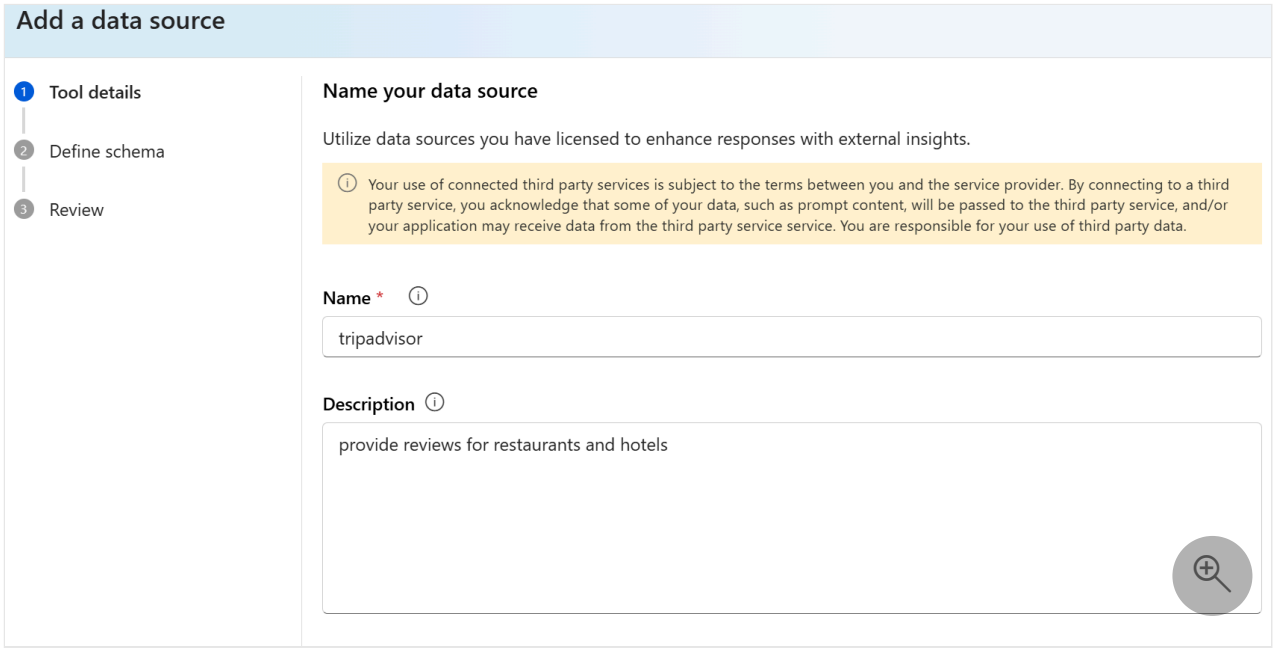
1. To use the Tripadvisor tool in the Azure AI Foundry, in the **Create and debug** screen for your agent, scroll down the **Setup** pane on the right to **action**. Then select **Add**.



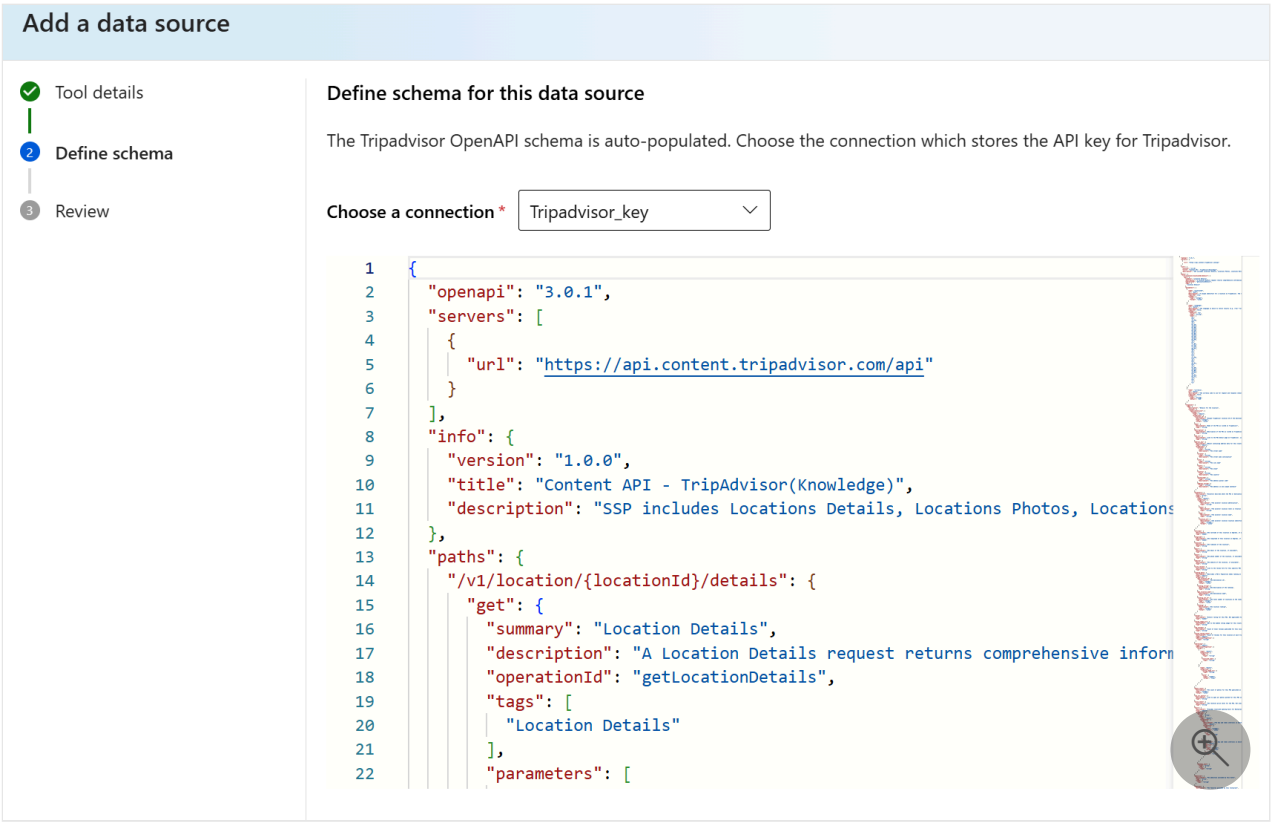
2. Select **Tripadvisor** and follow the prompts to add the tool.



3. Give a name for your Tripadvisor tool and provide an optional description.



4. Select the custom key connection you just created.



5. Finish and start chatting.

Connect Tripadvisor through code-first experience

You can follow the instructions in OpenAPI Spec tool to connect Tripadvisor through OpenAPI spec.

1. Remember to store and import Tripadvisor OpenAPI spec. You can find it through Foundry Portal.

2. Make sure you have updated the authentication method to be `connection` and fill in the connection ID of your custom key connection.

Python

```
auth =
OpenApiConnectionAuthDetails(security_scheme=OpenApiConnectionSecurityScheme(
connection_id="your_connection_id"))
```

Azure AI Agents function calling

Article • 02/19/2025

Azure AI Agents supports function calling, which allows you to describe the structure of functions to an Assistant and then return the functions that need to be called along with their arguments.

> **Note**
>
> Runs expire ten minutes after creation. Be sure to submit your tool outputs before the expiration.

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
	✔	✔	✔	✔	✔	✔

See also

- Lean how to ground agents by using Bing Web Search

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Azure AI Agent Service Code Interpreter

Article • 02/19/2025

Code Interpreter allows the agents to write and run Python code in a sandboxed execution environment. With Code Interpreter enabled, your agent can run code iteratively to solve more challenging code, math, and data analysis problems. When your Agent writes code that fails to run, it can iterate on this code by modifying and running different code until the code execution succeeds.

> **Important**
>
> Code Interpreter has additional charges beyond the token based fees for Azure OpenAI usage. If your Agent calls Code Interpreter simultaneously in two different threads, two code interpreter sessions are created. Each session is active by default for one hour.

Supported models

The models page contains the most up-to-date information on regions/models where agents and code interpreter are supported.

We recommend using Agents with the latest models to take advantage of the new features, larger context windows, and more up-to-date training data.

Usage support

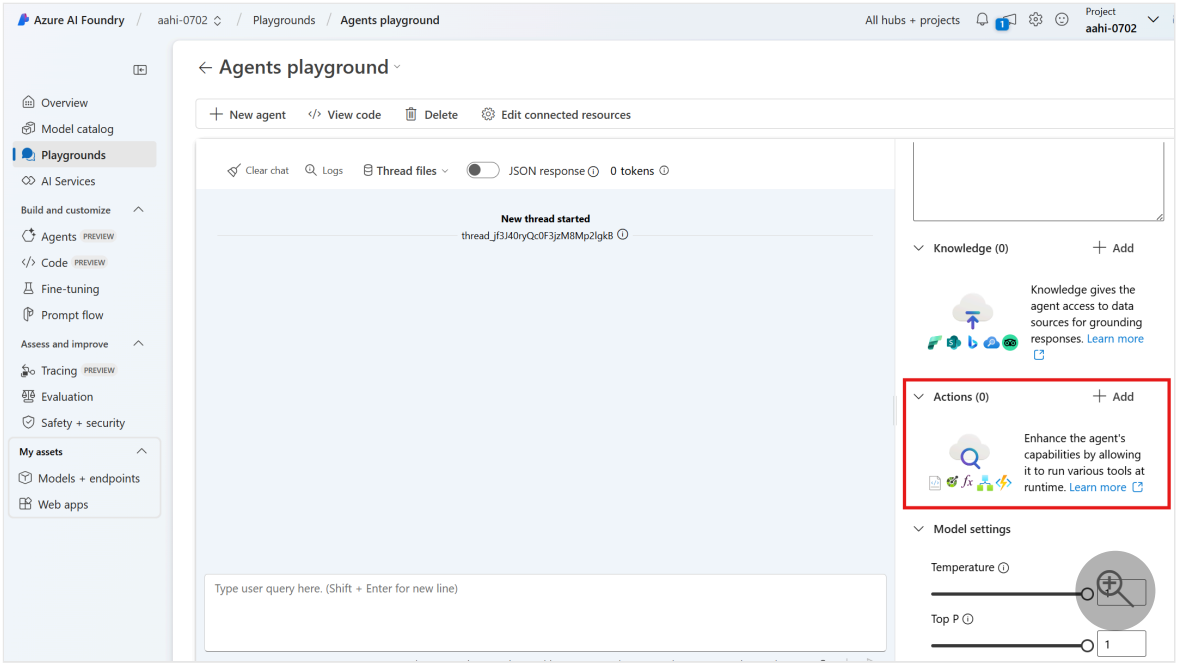
Expand table

Azure AI foundry support	Python SDK	C# SDK	JavaScript SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	✔	✔

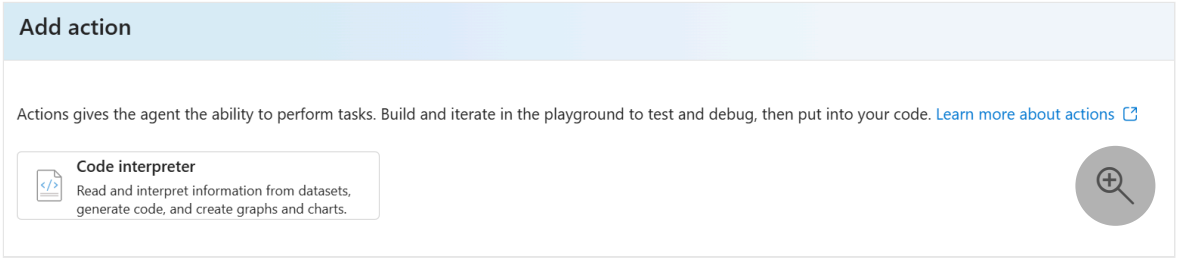
Using the code interpreter tool with an agent

You can add the code interpreter tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal:

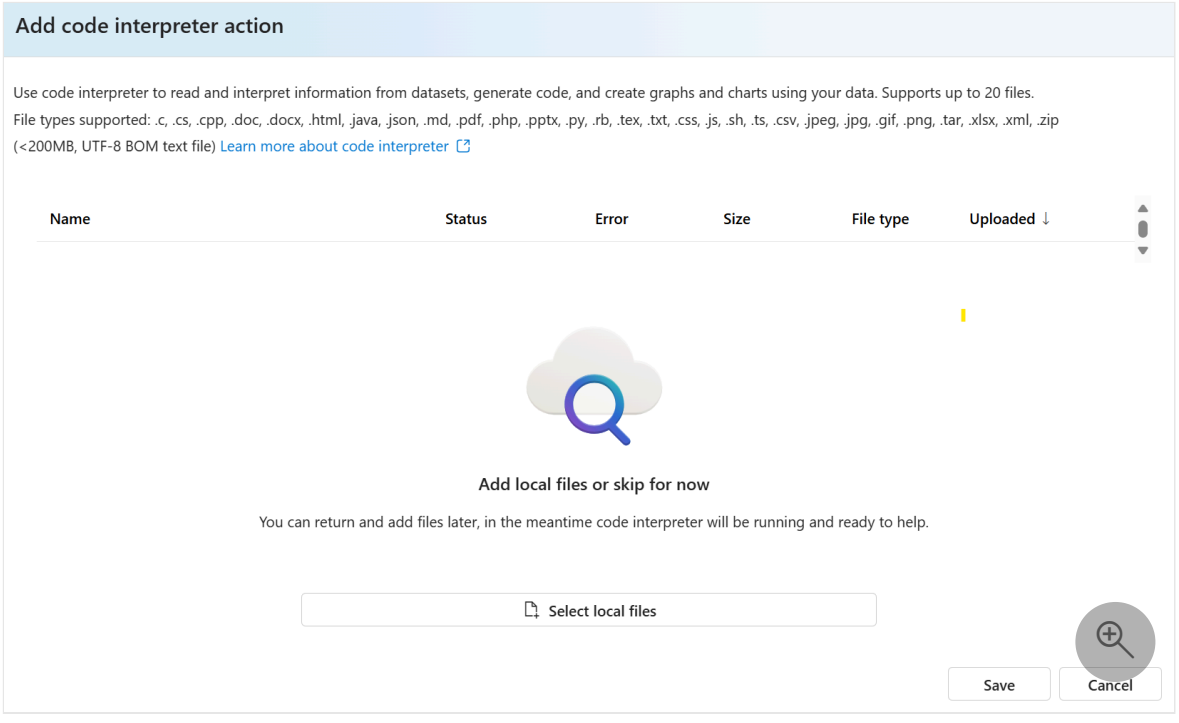
1. In the **Create and debug** screen for your agent, scroll down the **Setup** pane on the right to **action**. Then select **Add**.



2. Select **Code interpreter** and follow the prompts to add the tool.



3. You can optionally upload files for your agent to read and interpret information from datasets, generate code, and create graphs and charts using your data.



See also

- Learn more about agents.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

How to use Azure AI Agent Service with OpenAPI Specified Tools

Article • 03/25/2025

You can now connect your Azure AI Agent to an external API using an OpenAPI 3.0 specified tool, allowing for scalable interoperability with various applications. Enable your custom tools to authenticate access and connections with managed identities (Microsoft Entra ID) for added security, making it ideal for integrating with existing infrastructure or web services.

OpenAPI Specified tool improves your function calling experience by providing standardized, automated, and scalable API integrations that enhance the capabilities and efficiency of your agent. OpenAPI specifications provide a formal standard for describing HTTP APIs. This allows people to understand how an API works, how a sequence of APIs works together, generate client code, create tests, apply design standards, and more. Currently, we support three authentication types with the OpenAPI 3.0 specified tools: `anonymous`, `API key`, `managed identity`.

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	REST API	Basic agent setup	Standard agent setup
✔	✔	✔	✔	✔	✔

Prerequisites

1. Ensure you've completed the prerequisites and setup steps in the quickstart.
2. Check the OpenAPI spec for the following requirements:
   a. Although not required by the OpenAPI spec, `operationId` is required for each function to be used with the OpenAPI tool.
   b. `operationId` should only contain letters, `-` and `_`. You can modify it to meet the requirement. We recommend using descriptive name to help models efficiently decide which function to use.

Authenticating with API Key

With API key authentication, you can authenticate your OpenAPI spec using various methods such as an API key or Bearer token. Only one API key security schema is supported per OpenAPI spec. If you need multiple security schemas, create multiple OpenAPI spec tools.

1. Update your OpenAPI spec security schemas. it has a `securitySchemes` section and one scheme of type `apiKey`. For example:

   JSON

   ```
   "securitySchemes": {
       "apiKeyHeader": {
               "type": "apiKey",
               "name": "x-api-key",
               "in": "header"
           }
   }
   ```

   You usually only need to update the `name` field, which corresponds to the name of `key` in the connection. If the security schemes include multiple schemes, we recommend keeping only one of them.

2. Update your OpenAPI spec to include a `security` section:

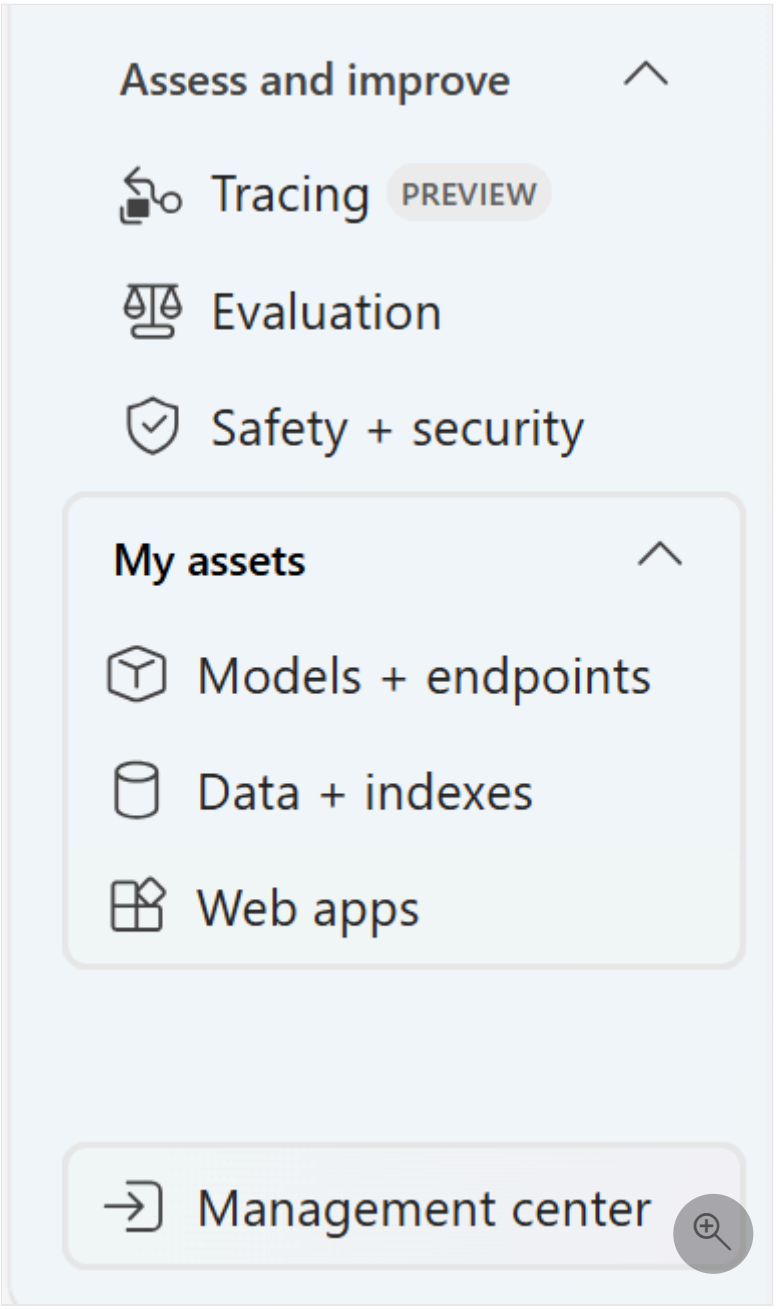
   JSON

   ```
   "security": [
       {
       "apiKeyHeader": []
       }
   ]
   ```

3. Remove any parameter in the OpenAPI spec that needs API key, because API key will be stored and passed through a connection, as described later in this article.

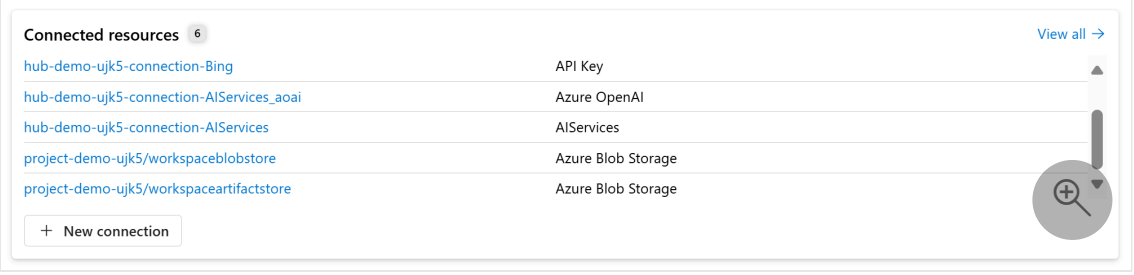
4. Create a `custom keys` connection to store your API key.

a. Go to the Azure AI Foundry portal and select the AI Project. Click **connected resources**.

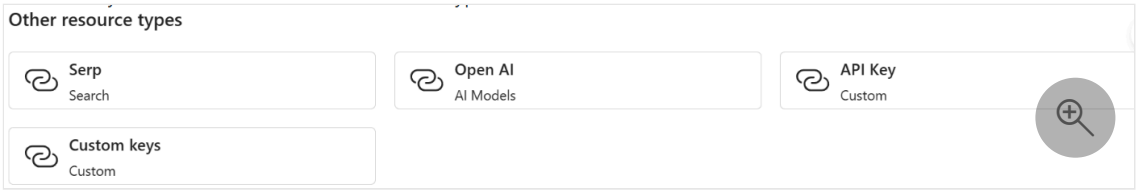


b. Select **+ new connection** in the settings page.

> **Note**
>
> If you regenerate the API key at a later date, you need to update the connection with the new key.



c. Select **custom keys** in **other resource types**.



d. Enter the following information

- key: `name` field of your security scheme. In this example, it should be `x-api-key`

```JSON
        "securitySchemes": {
           "apiKeyHeader": {
                      "type": "apiKey",
                      "name": "x-api-key",
                      "in": "header"
                  }
         }
```

- value: YOUR_API_KEY
- Connection name: YOUR_CONNECTION_NAME (You will use this connection name in the sample code below.)
- Access: you can choose either *this project only* or *shared to all projects*. Just make sure in the sample code below, the project you entered connection string for has access to this connection.

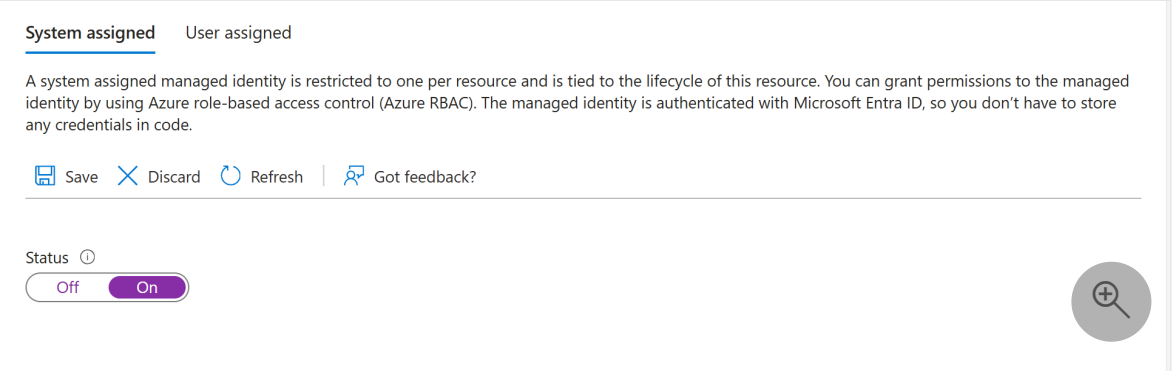
5. Once you have created a connection, you can use it through the SDK or REST API. Use the tabs at the top of this article to see code examples.

Authenticating with managed identity (Microsoft Entra ID)

Microsoft Entra ID is a cloud-based identity and access management service that your employees can use to access external resources. Microsoft Entra ID allows you to authenticate your APIs with additional security without the need to pass in API keys. Once you have set up managed identity authentication, it will authenticate through the Azure AI Service your agent is using.

To set up authenticating with Managed Identity:

1. Enable the Azure AI Service of your agent has `system assigned managed identity` enabled.

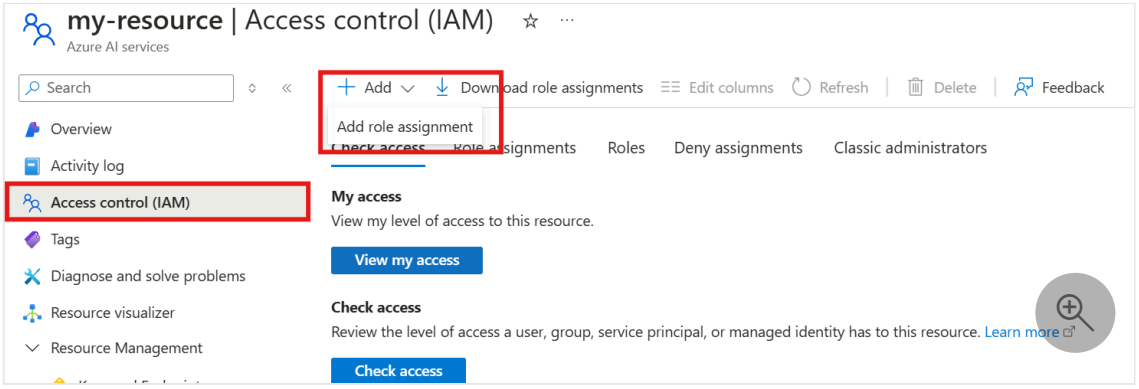


2. Create a resource of the service you want to connect to through OpenAPI spec.

3. Assign proper access to the resource.

   a. Click **Access Control** for your resource

   b. Click **Add** and then **add role assignment** at the top of the screen.



   c. Select the proper role assignment needed, usually it will require at least *READER* role. Then click **Next**.

   d. Select **Managed identity** and then click **select members**.

   e. In the managed identity dropdown menu, search for **Azure AI services** and then select the AI Service of your agent.

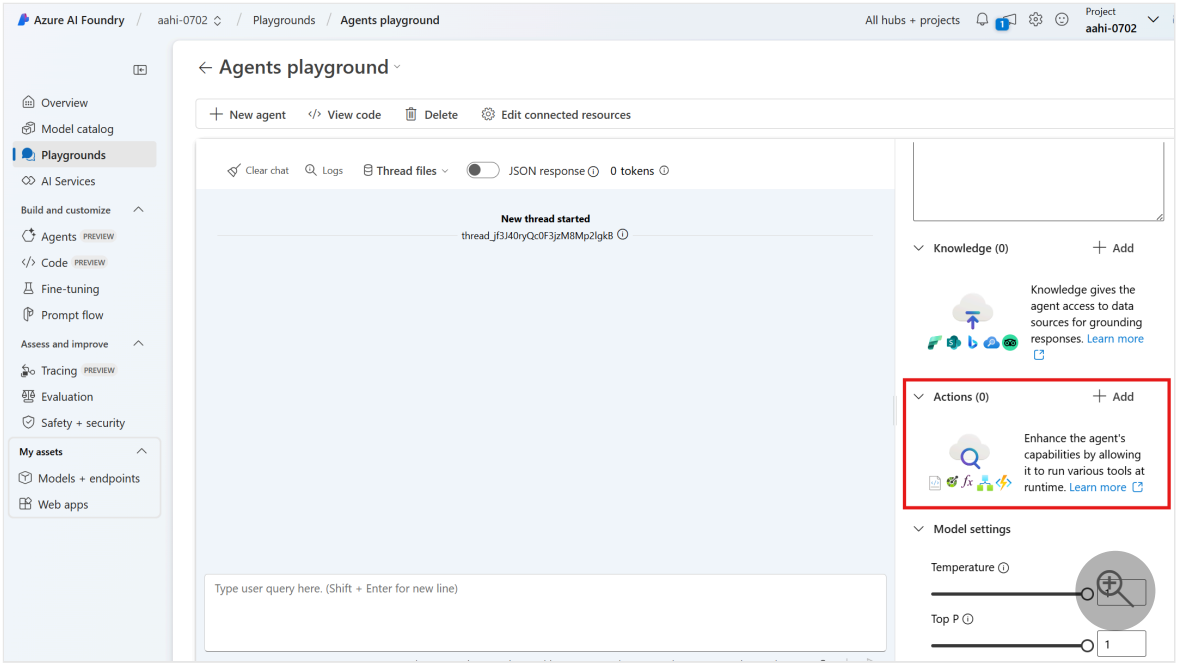
   f. Click **Finish**.

4. Once the setup is done, you can continue by using the tool through the Foundry Portal, SDK, or REST API. Use the tabs at the top of this article to see code samples.

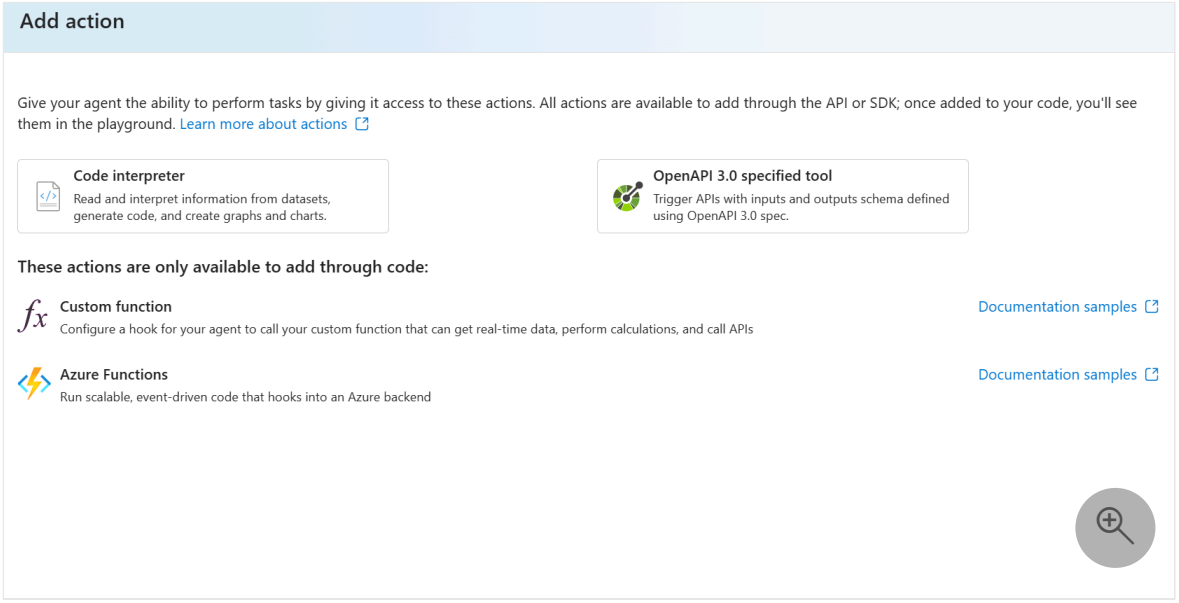
Add OpenAPI spec tool in the Azure AI Foundry portal

You can add the Grounding with Bing Search tool to an agent programatically using the code examples listed at the top of this article, or the Azure AI Foundry portal. If you want to use the portal:

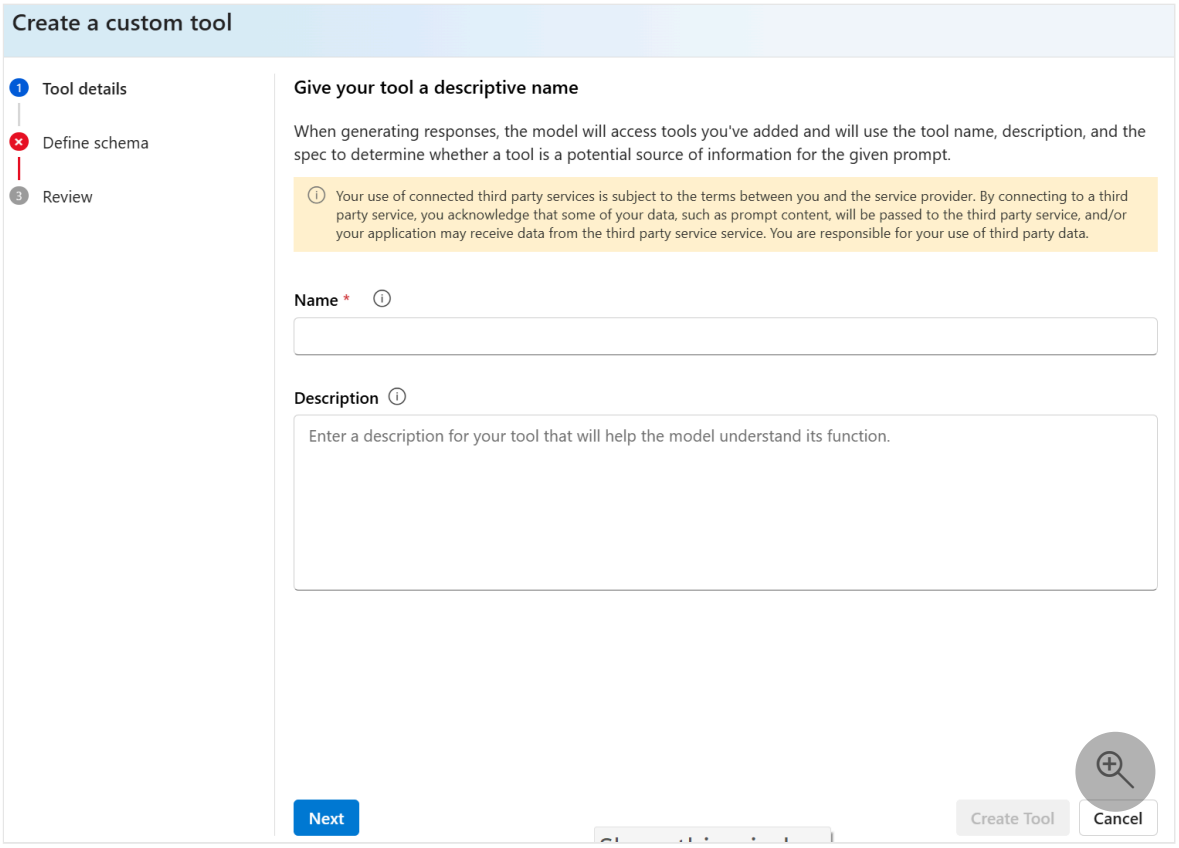
1. in the **Create and debug** screen or **Agent playground**, select your agent.
2. Scroll down the **Setup** pane on the right to **action**. Then select **Add**.



3. Select **OpenAPI 3.0 specified tool**.



4. Give your tool a name (required) and a description (optional). The description will be used by the model to decide when and how to use the tool.



5. Click Next and select your authentication method. Choose `connection` for `API key`.
   a. If you choose `connection`, you need to select the custom keys connection you have created before.
   b. If you choose `managed identity`, you need to input the audience to get your token. An example of an audience would be `https://cognitiveservices.azure.com/` to connect to Azure AI Services. Make sure you have already set up authentication and role assignment (as described in the section above).

6. Copy and paste your OpenAPI specification in the text box.

7. Review and add the tool to your agent.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Use Azure Functions with Azure AI Agent Service

Article • 02/19/2025

The Azure AI Agent Service integrates with Azure Functions, enabling you to create intelligent, event-driven applications with minimal overhead. This combination allows AI-driven workflows to leverage the scalability and flexibility of serverless computing, making it easier to build and deploy solutions that respond to real-time events or complex workflows.

Azure Functions provide support for triggers and bindings, which simplify how your AI Agents interact with external systems and services. Triggers determine when a function executes—such as an HTTP request, message from a queue, or a file upload to Azure Blob Storage and allows agents to act dynamically based on incoming events.

Meanwhile, bindings facilitate streamlined connections to input or output data sources, such as databases or APIs, without requiring extensive boilerplate code. For instance, you can configure a trigger to execute an Azure Function whenever a customer message is received in a chatbot and use output bindings to send a response via the Azure AI Agent.

Prerequisites

- Azure Functions Core Tools v4.x
- Azure AI Agent Service
- Azurite

Prepare your local environment

The following examples highlight how to use the Azure AI Agent Service function calling where function calls are placed on a storage queue by the Agent Service to be processed by an Azure Function listening to that queue.

You can find the template and code used here on GitHub.

Usage support

Expand table

Azure AI foundry support	Python SDK	C# SDK	REST API	Basic agent setup	Standard agent setup
	✔		✔		✔

Create Azure resources for local and cloud dev-test

Once you have your Azure subscription, run the following in a new terminal window to create Azure OpenAI and other resources needed:

Bash

```
azd init --template https://github.com/Azure-Samples/azure-functions-ai-services-agent-python
```

Mac/Linux:

Bash

```
chmod +x ./infra/scripts/*.sh
```

Windows:

Powershell

```
set-executionpolicy remotesigned
```

Provision resources

Run the following command to create the required resources in Azure.

Bash

```
azd provision
```

Create local.settings.json

ⓘ Note

This file should be in the same folder as `host.json`. It is automatically created if you ran `azd provision`.

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "STORAGE_CONNECTION__queueServiceUri":
"https://<storageaccount>.queue.core.windows.net",
    "PROJECT_CONNECTION_STRING": "<project connnection for AI Project>",
    "AzureWebJobsStorage": "UseDevelopmentStorage=true"
    }
}
```

Run your app using Visual Studio Code

1. Open the folder in a new terminal.
2. Run the `code .` code command to open the project in Visual Studio Code.
3. In the command palette (F1), type `Azurite: Start`, which enables debugging with local storage for Azure Functions runtime.
4. Press **Run/Debug (F5)** to run in the debugger. Select **Debug anyway** if prompted about local emulator not running.
5. Send POST `prompt` endpoints respectively using your HTTP test tool. If you have the RestClient extension installed, you can execute requests directly from the test.http project file.

Deploy to Azure

Run this command to provision the function app, with any required Azure resources, and deploy your code:

shell

```
azd up
```

You're prompted to supply these required deployment parameters:

Expand table

Parameter	Description
Environment name	An environment that's used to maintain a unique deployment context for your app. You won't be prompted if you created the local project using `azd init`.
Azure subscription	Subscription in which your resources are created.
Azure location	Azure region in which to create the resource group that contains the new Azure resources. Only regions that currently support the Flex Consumption plan are shown.

After publish completes successfully, `azd` provides you with the URL endpoints of your new functions, but without the function key values required to access the endpoints. To learn how to obtain these same endpoints along with the required function keys, see Invoke the function on Azure in the companion article Quickstart: Create and deploy functions to Azure Functions using the Azure Developer CLI.

Redeploy your code

You can run the `azd up` command as many times as you need to both provision your Azure resources and deploy code updates to your function app.

> **Note**
>
> Deployed code files are always overwritten by the latest deployment package.

Clean up resources

When you're done working with your function app and related resources, you can use this command to delete the function app and its related resources from Azure and avoid incurring any further costs (--purge does not leave a soft delete of AI resource and recovers your quota):

shell

```
azd down --purge
```

Feedback

Use your own resources

Article • 04/14/2025

Use this article if you want to use the Azure Agent Service with resources you already have.

> **Note**
>
> - If you use an existing AI Services / Azure OpenAI Service resource, no model will be deployed. You can deploy a model to the resource after the agent setup is complete.
> - Make sure your Azure OpenAI resource and Azure AI Foundry project are in the same region.

Choose basic or standard agent setup

To use your own resources, you can edit the parameters in the provided deployment templates. To start, determine if you want to edit the basic agent setup template, or the standard agent setup template.

Basic Setup: Agents created in a basic project use multitenant search and storage resources fully managed by Microsoft. You don't have visibility or control over these underlying Azure resources. You can only use your own AI services account with this option.

Standard Setup: Agents created in a standard project use customer-owned, single-tenant search and storage resources. With this setup, you have full control and visibility over these resources, but you incur costs based on your usage. You can use your own AI services account, Azure Storage account, Cosmos DB for NoSQL account and/or Azure AI Search resource with this option.

Basic agent setup: Use an existing AI Services/Azure OpenAI resource

Replace the parameter value for `aiServiceAccountResourceId` with the full arm resource ID of the AI Services or Azure OpenAI resource you want to use.

1. To get the AI Services account resource ID, sign in to the Azure CLI and select the subscription with your AI Services account:

   ```
   az login
   ```

2. Replace `<your-resource-group>` with the resource group containing your resource and `your-ai-service-resource-name` with the name of your AI Service resource, and run:

   `az cognitiveservices account show --resource-group <your-resource-group> --name <your-ai-service-resource-name> --query "id" --output tsv`

   The value returned is the `aiServiceAccountResourceId` you need to use in the template.

3. In the basic agent template file, replace the following placeholders:

   ```
   aiServiceAccountResourceId:/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.CognitiveServices/accounts/{serviceName}

   [Azure OpenAI Only] aiServiceKind: AzureOpenAI
   ```

   If you want to use an existing Azure OpenAI resource, you will need to update the `aiServiceAccountResourceId` and the `aiServiceKind` parameters in the parameter file. The aiServiceKind parameter should be set to AzureOpenAI.

Standard agent setup: Use an existing AI Services/Azure OpenAI, Azure Storage account, Azure Cosmos DB for NoSQL account, and/or Azure AI Search resource

Use an existing AI Services / Azure OpenAI, Azure Storage account, Azure Cosmos DB for NoSQL account and/or Azure AI Search resource by providing the full ARM resource ID in the standard agent template file.

Use an existing AI Services or Azure OpenAI resource

1. Follow the steps in basic agent setup to get the AI Services account resource ID.

2. In the standard agent template file, replace the following placeholders:

   ```
   aiServiceAccountResourceId:/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.CognitiveServices/accounts/{serviceName}

   [Azure OpenAI Only] aiServiceKind: AzureOpenAI
   ```

Use an existing Azure Storage account for file storage

1. To get your storage account resource ID, sign in to the Azure CLI and select the subscription with your storage account:

   `az login`

2. Then run the command:

   `az search service show --resource-group <your-resource-group> --name <your-storage-account> --query "id" --output tsv`

   The output is the `aiStorageAccountResourceID` you need to use in the template.

3. In the standard agent template file, replace the following placeholders:

   ```
   aiStorageAccountResourceId:/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Storage/storageAccounts/{storageAccountName}
   ```

Use an existing Azure Cosmos DB for NoSQL account for thread storage

1. To get your Azure Cosmos DB account resource ID, sign in to the Azure CLI and select the subscription with your account:

   Console

   ```
   az login
   ```

2. Then run the command:

   Console

   ```
   az cosmosdb show --resource-group  <your-resource-group> --name <your-comosdb-account>  --query "id" --output tsv
   ```

   The output is the `cosmosDBResourceId` you need to use in the template.

3. In the standard agent template file, replace the following placeholders:

`cosmosDBResourceId:/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.DocumentDB/databaseAccounts/{cosmosDbAccountName}`

Use an existing Azure AI Search resource

1. To get your Azure AI Search resource ID, sign into Azure CLI and select the subscription with your search resource:

   `az login`

2. Then run the command:

   `az search service show --resource-group <your-resource-group> --name <your-search-service> --query "id" --output tsv`

3. In the standard agent template file, replace the following placeholders:

   ```
   aiSearchServiceResourceId:/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.Search/searchServices/{searchServiceName}
   ```

See also

- Learn about the different tools agents can use.

Tracing using Application Insights

Article • 12/13/2024

Determining the reasoning behind your agent's executions is important for troubleshooting and debugging. However, it can be difficult for complex agents for a number of reasons:

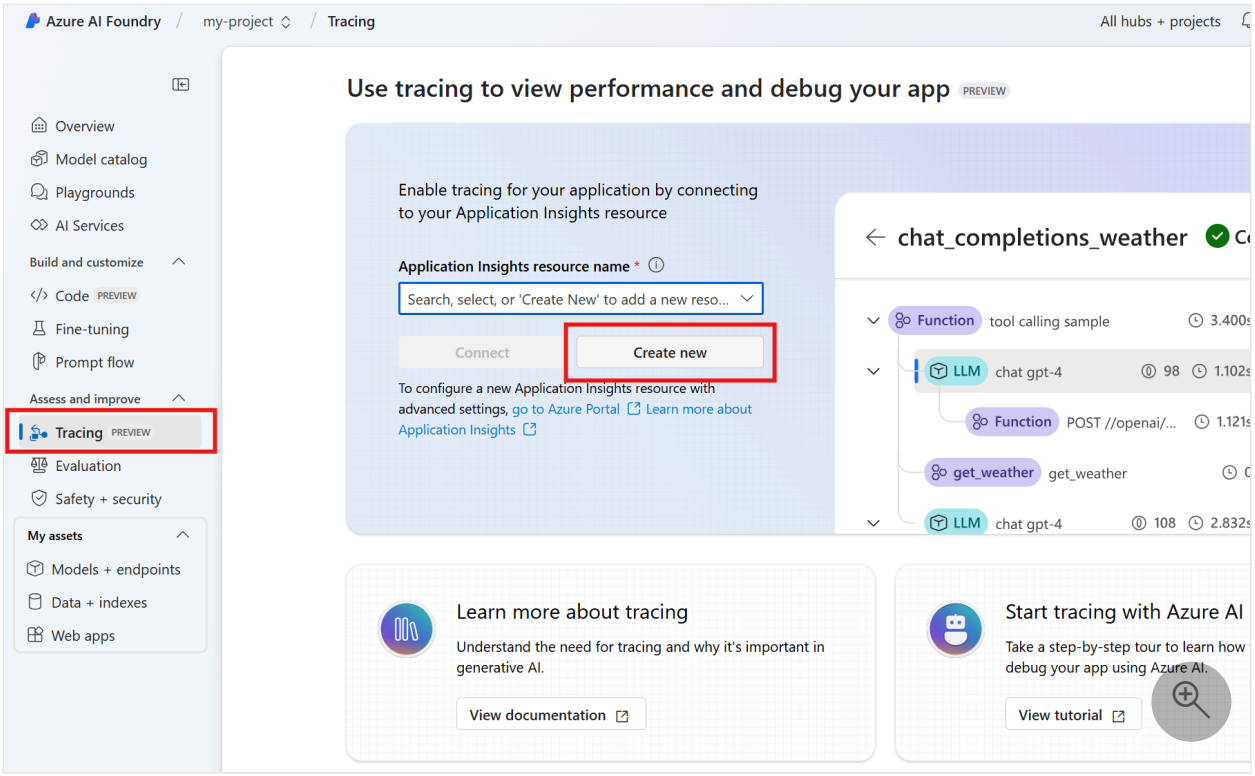
- There could be a high number of steps involved in generating a response, making it hard to keep track of all of them.
- The sequence of steps might vary based on user input.
- The inputs/outputs at each stage might be long and deserve more detailed inspection.
- Each step of an agent's runtime might also involve nesting. For example, an agent might invoke a tool, which uses another process, which then invokes another tool. If you notice strange or incorrect output from a top-level agent run, it might be difficult to determine exactly where in the execution the issue was introduced.

Tracing solves this by allowing you to clearly see the inputs and outputs of each primitive involved in a particular agent run, in the order in which they were invoked.

Creating an Application Insights resource

Tracing lets you analyze your agent's performance and behavior by using OpenTelemetry and adding an Application Insights resource to your Azure AI Foundry project.

To add an Application Insights resource, navigate to the **Tracing** tab in the AI Foundry portal, and create a new resource if you don't already have one.



Once created, you can get an Application Insights connection string, configure your agents, and observe the full execution path of your agent through Azure Monitor. Typically you want to enable tracing before you create an agent.

Trace an agent

First, use `pip install` to install OpenTelemetry and the Azure SDK tracing plugin.

Bash

```
pip install opentelemetry
pip install azure-core-tracing-opentelemetry
```

You will also need an exporter to send results to your observability backend. You can print traces to the console or use a local viewer such as Aspire Dashboard. To connect to Aspire Dashboard or another OpenTelemetry compatible backend, install the OpenTelemetry Protocol (OTLP) exporter.

Bash

```
pip install opentelemetry-exporter-otlp
```

Once you have the packages installed, you can use one the following Python samples to implement tracing with your agents. Samples that use console tracing display the results locally in the console. Samples that use Azure Monitor send the traces to the Azure

Monitor in the AI Foundry portal, in the **Tracing** tab in the left navigation menu for the portal.

> **Note**
>
> There is a known bug in the agents tracing functionality. The bug will cause the agent's function tool to call related info (function names and parameter values, which could contain sensitive information) to be included in the traces even when content recording is not enabled.

Using Azure Monitor

- Basic agent example
- Agent example with function calling
- Example with a stream event handler

Using console tracing

- Basic agent example
- Agent example with function calling
- Example with a stream event handler

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

How to configure content filters

Article • 03/05/2025

The content filtering system integrated into Azure AI Foundry runs alongside the core models, including DALL-E image generation models. It uses an ensemble of multi-class classification models to detect four categories of harmful content (violence, hate, sexual, and self-harm) at four severity levels respectively (safe, low, medium, and high), and optional binary classifiers for detecting jailbreak risk, existing text, and code in public repositories.

The default content filtering configuration is set to filter at the medium severity threshold for all four content harms categories for both prompts and completions. That means that content that is detected at severity level medium or high is filtered, while content detected at severity level low or safe is not filtered by the content filters. Learn more about content categories, severity levels, and the behavior of the content filtering system here.

Jailbreak risk detection and protected text and code models are optional and on by default. For jailbreak and protected material text and code models, the configurability feature allows all customers to turn the models on and off. The models are by default on and can be turned off per your scenario. Some models are required to be on for certain scenarios to retain coverage under the Customer Copyright Commitment.

> **Note**
>
> All customers have the ability to modify the content filters and configure the severity thresholds (low, medium, high). Approval is required for turning the content filters partially or fully off. Managed customers only may apply for full content filtering control via this form: Azure OpenAI Limited Access Review: Modified Content Filters. At this time, it is not possible to become a managed customer.

Content filters can be configured at the resource level. Once a new configuration is created, it can be associated with one or more deployments. For more information about model deployment, see the resource deployment guide.

Prerequisites

- You must have an Azure OpenAI resource and a large language model (LLM) deployment to configure content filters. Follow a quickstart to get started.

Understand content filter configurability

Azure OpenAI Service includes default safety settings applied to all models, excluding Azure OpenAI Whisper. These configurations provide you with a responsible experience by default, including content filtering models, blocklists, prompt transformation, content credentials, and others. Read more about it here.

All customers can also configure content filters and create custom safety policies that are tailored to their use case requirements. The configurability feature allows customers to adjust the settings, separately for prompts and completions, to filter content for each content category at different severity levels as described in the table below. Content detected at the 'safe' severity level is labeled in annotations but is not subject to filtering and isn't configurable.

Expand table

Severity filtered	Configurable for prompts	Configurable for completions	Descriptions
Low, medium, high	Yes	Yes	Strictest filtering configuration. Content detected at severity levels low, medium, and high is filtered.
Medium, high	Yes	Yes	Content detected at severity level low isn't filtered, content at medium and high is filtered.
High	Yes	Yes	Content detected at severity levels low and medium isn't filtered. Only content at severity level high is filtered.
No filters	If approved[1]	If approved[1]	No content is filtered regardless of severity level detected. Requires approval[1].
Annotate only	If approved[1]	If approved[1]	Disables the filter functionality, so content will not be blocked, but annotations are returned via API response. Requires approval[1].

[1] For Azure OpenAI models, only customers who have been approved for modified content filtering have full content filtering control and can turn off content filters. Apply for modified content filters via this form: Azure OpenAI Limited Access Review: Modified Content Filters. For Azure Government customers, apply for modified content filters via this form: Azure Government - Request Modified Content Filtering for Azure OpenAI Service.

Configurable content filters for inputs (prompts) and outputs (completions) are available for all Azure OpenAI models.

Content filtering configurations are created within a Resource in Azure AI Foundry portal, and can be associated with Deployments. Learn more about configurability here.

Customers are responsible for ensuring that applications integrating Azure OpenAI comply with the Code of Conduct.

Understand other filters

You can configure the following filter categories in addition to the default harm category filters.

Expand table

Filter category	Status	Default setting	Applied to prompt or completion?	Description
Prompt Shields for direct attacks (jailbreak)	GA	On	User prompt	Filters / annotates user prompts that might present a Jailbreak Risk. For more information about annotations, visit Azure AI Foundry content filtering.
Prompt Shields for indirect attacks	GA	Off	User prompt	Filter / annotate Indirect Attacks, also referred to as Indirect Prompt Attacks or Cross-Domain Prompt Injection Attacks, a potential vulnerability where third parties place malicious instructions inside of documents that the generative AI system can access and process. Requires: Document embedding and formatting.
Protected material - code	GA	On	Completion	Filters protected code or gets the example citation and license information in annotations for code snippets that match any public code sources, powered by GitHub Copilot. For more information about consuming annotations, see the content filtering concepts guide
Protected material - text	GA	On	Completion	Identifies and blocks known text content from being displayed in the

Filter category	Status	Default setting	Applied to prompt or completion?	Description
				model output (for example, song lyrics, recipes, and selected web content).
Groundedness	Preview	Off	Completion	Detects whether the text responses of large language models (LLMs) are grounded in the source materials provided by the users. Ungroundedness refers to instances where the LLMs produce information that is non-factual or inaccurate from what was present in the source materials. Requires: Document embedding and formatting.

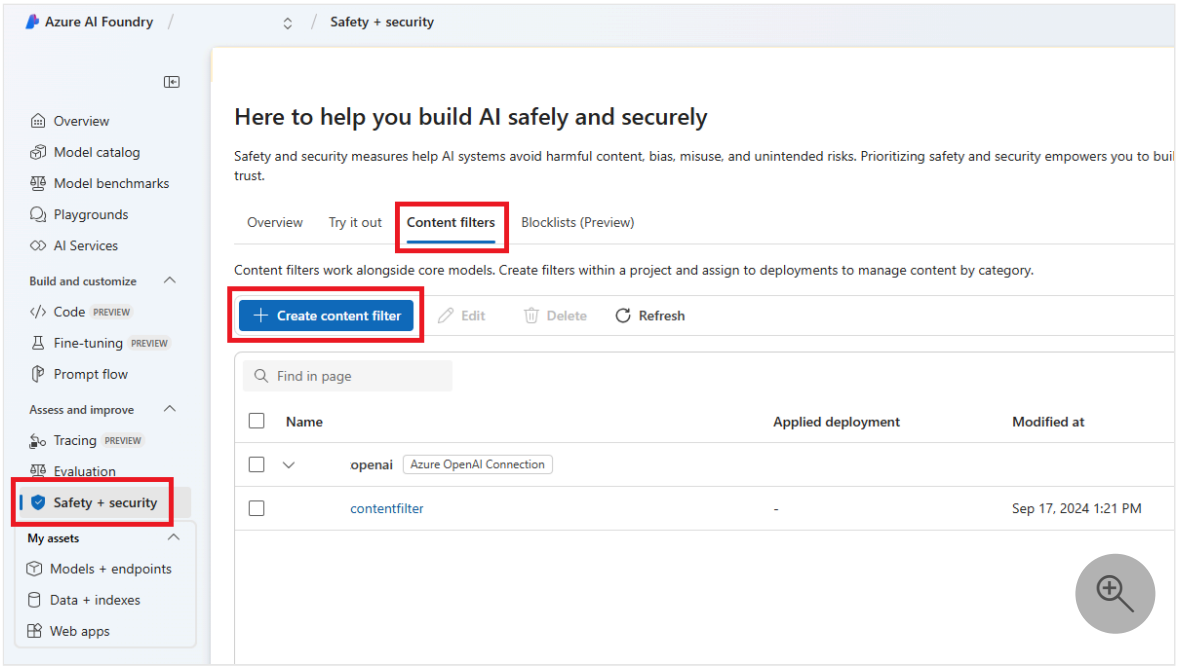
Create a content filter in Azure AI Foundry

For any model deployment in Azure AI Foundry, you can directly use the default content filter, but you might want to have more control. For example, you could make a filter stricter or more lenient, or enable more advanced capabilities like prompt shields and protected material detection.

> **Tip**
>
> For guidance with content filters in your Azure AI Foundry project, you can read more at **Azure AI Foundry content filtering**.

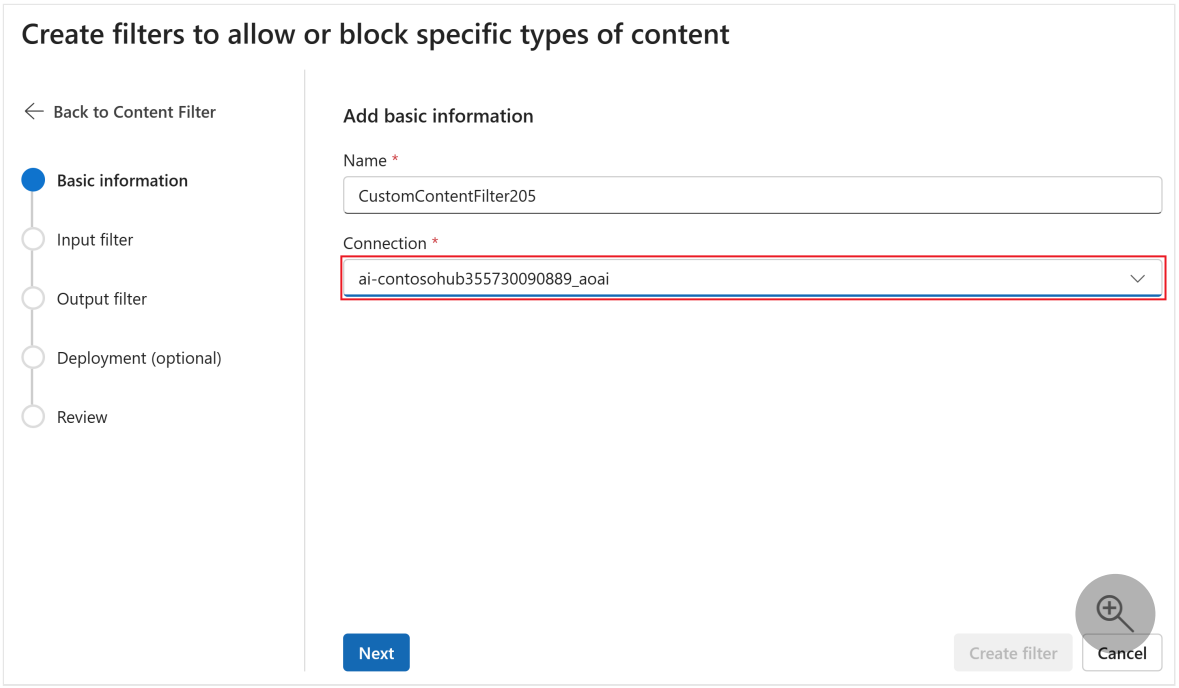
Follow these steps to create a content filter:

1. Go to Azure AI Foundry and navigate to your project. Then select the **Safety + security** page from the left menu and select the **Content filters** tab.



2. Select **+ Create content filter**.

3. On the **Basic information** page, enter a name for your content filtering configuration. Select a connection to associate with the content filter. Then select **Next**.



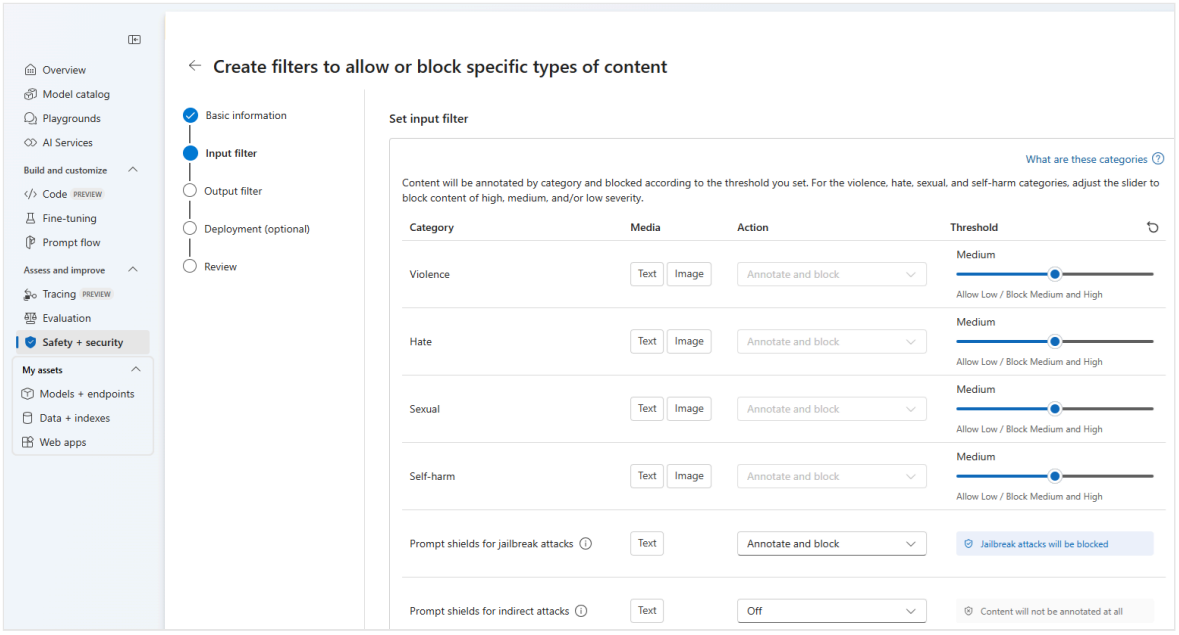
Now you can configure the input filters (for user prompts) and output filters (for model completion).

4. On the **Input filters** page, you can set the filter for the input prompt. For the first four content categories there are three severity levels that are configurable: Low, medium, and high. You can use the sliders to set the severity threshold if you determine that your application or usage scenario requires different filtering than the default values. Some filters, such as Prompt Shields and Protected material

detection, enable you to determine if the model should annotate and/or block content. Selecting **Annotate only** runs the respective model and return annotations via API response, but it will not filter content. In addition to annotate, you can also choose to block content.

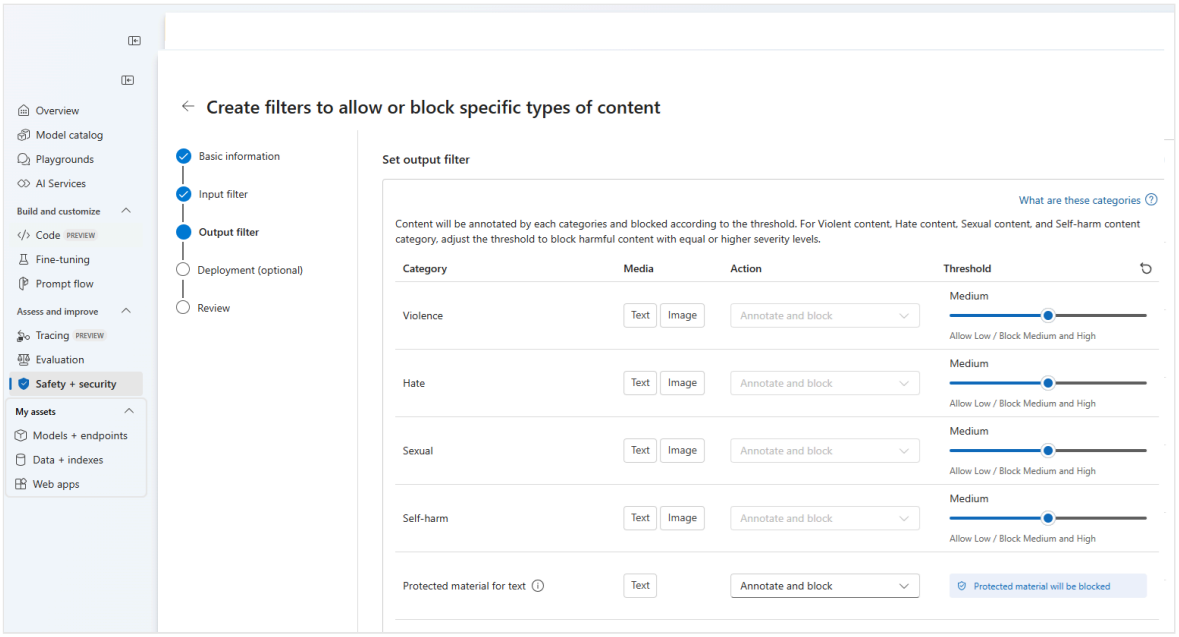
If your use case was approved for modified content filters, you receive full control over content filtering configurations and can choose to turn filtering partially or fully off, or enable annotate only for the content harms categories (violence, hate, sexual and self-harm).

Content will be annotated by category and blocked according to the threshold you set. For the violence, hate, sexual, and self-harm categories, adjust the slider to block content of high, medium, or low severity.

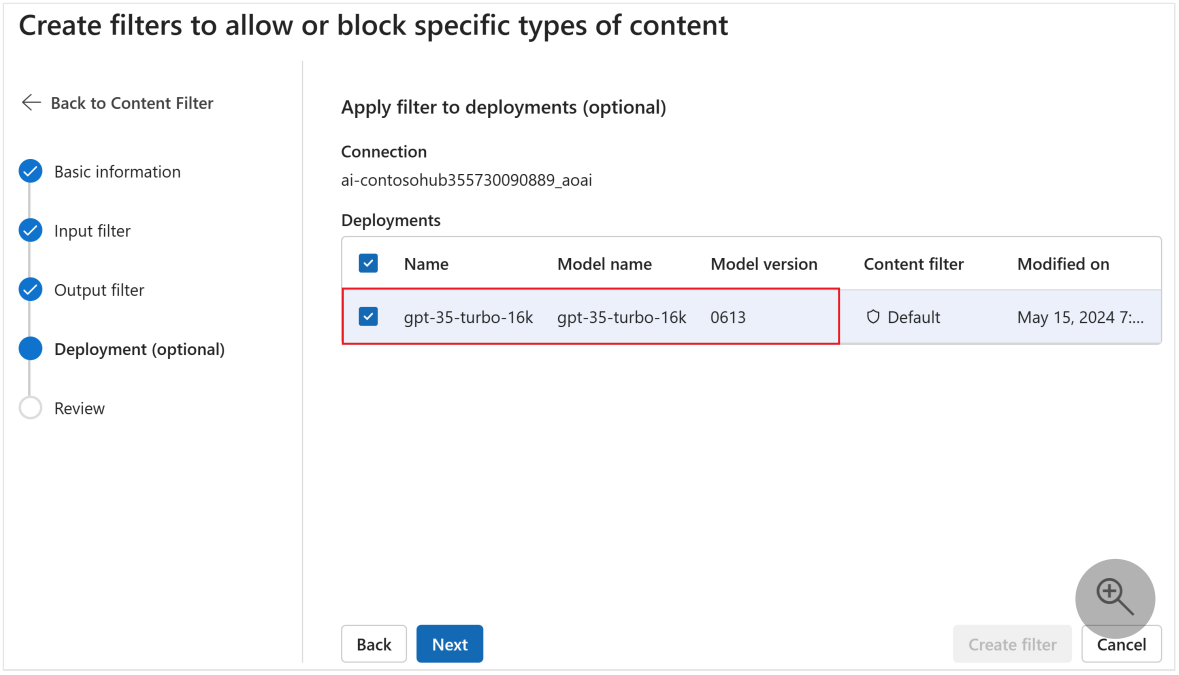


5. On the **Output filters** page, you can configure the output filter, which will be applied to all output content generated by your model. Configure the individual filters as before. This page also provides the Streaming mode option, which lets you filter content in near-real-time as it's generated by the model, reducing latency. When you're finished select **Next**.

   Content will be annotated by each category and blocked according to the threshold. For violent content, hate content, sexual content, and self-harm content category, adjust the threshold to block harmful content with equal or higher severity levels.



6. Optionally, on the **Deployment** page, you can associate the content filter with a deployment. If a selected deployment already has a filter attached, you must confirm that you want to replace it. You can also associate the content filter with a deployment later. Select **Create**.



Content filtering configurations are created at the hub level in the Azure AI Foundry portal. Learn more about configurability in the Azure OpenAI Service documentation.

7. On the **Review** page, review the settings and then select **Create filter**.

Use a blocklist as a filter

You can apply a blocklist as either an input or output filter, or both. Enable the **Blocklist** option on the **Input filter** and/or **Output filter** page. Select one or more blocklists from the dropdown, or use the built-in profanity blocklist. You can combine multiple blocklists into the same filter.

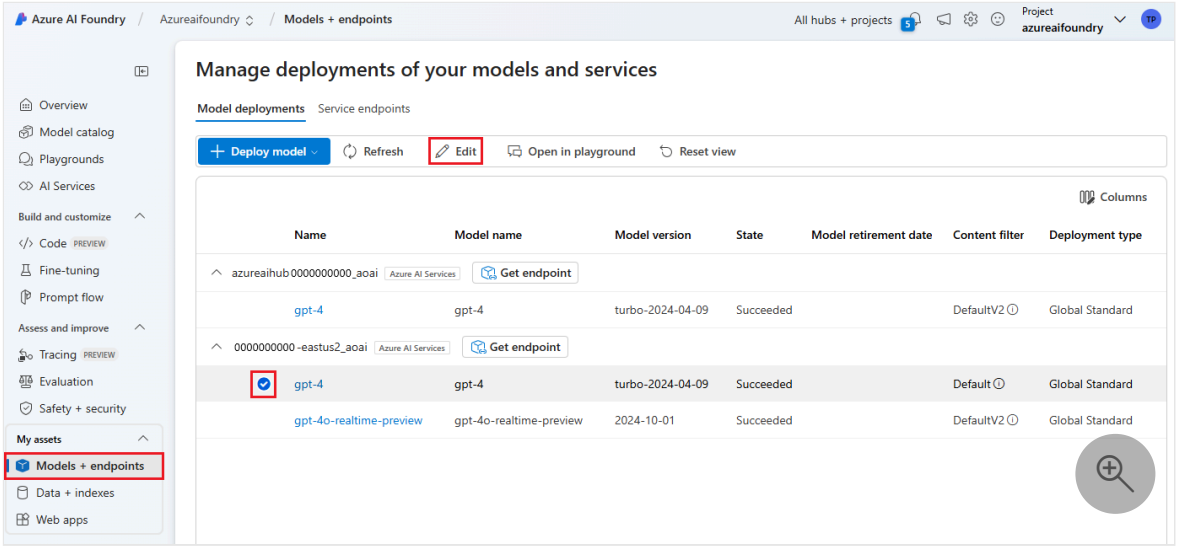
Apply a content filter

The filter creation process gives you the option to apply the filter to the deployments you want. You can also change or remove content filters from your deployments at any time.

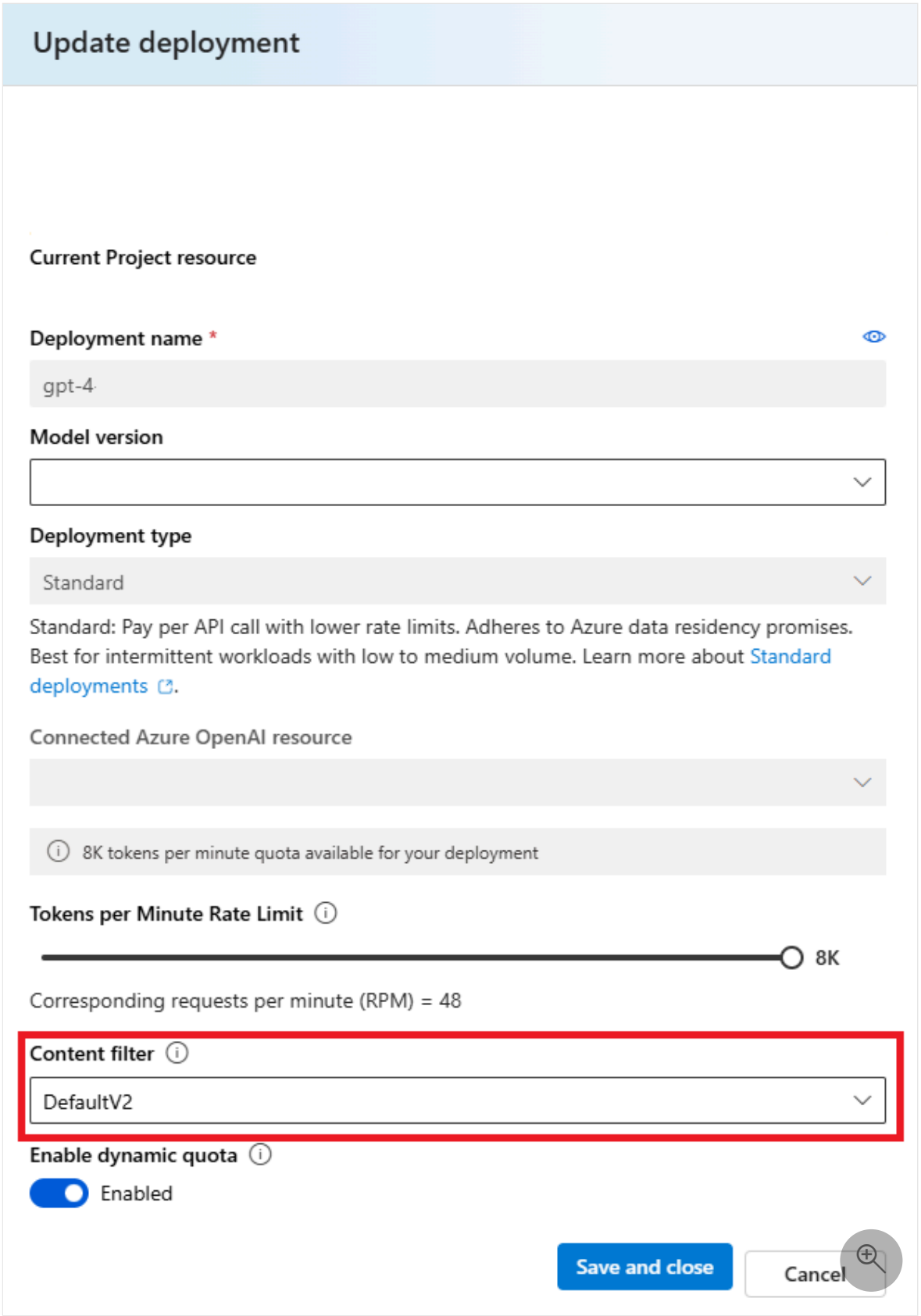
Follow these steps to apply a content filter to a deployment:

1. Go to Azure AI Foundry and select a project.

2. Select **Models + endpoints** on the left pane and choose one of your deployments, then select **Edit**.



3. In the **Update deployment** window, select the content filter you want to apply to the deployment. Then select **Save and close**.



You can also edit and delete a content filter configuration if required. Before you delete a content filtering configuration, you will need to unassign and replace it from any deployment in the **Deployments** tab.

Now, you can go to the playground to test whether the content filter works as expected.

> **Tip**

You can also create and update content filters using the REST APIs. For more information, see the API reference. Content filters can be configured at the resource level. Once a new configuration is created, it can be associated with one or more deployments. For more information about model deployment, see the resource deployment guide.

Specify a content filtering configuration at request time (preview)

In addition to the deployment-level content filtering configuration, we also provide a request header that allows you specify your custom configuration at request time for every API call.

Bash

```
curl --request POST \
    --url 'URL' \
    --header 'Content-Type: application/json' \
    --header 'api-key: API_KEY' \
    --header 'x-policy-id: CUSTOM_CONTENT_FILTER_NAME' \
    --data '{
        "messages": [
            {
                "role": "system",
                "content": "You are a creative assistant."
            },
            {
                "role": "user",
                "content": "Write a poem about the beauty of nature."
            }
        ]
    }'
```

The request-level content filtering configuration will override the deployment-level configuration, for the specific API call. If a configuration is specified that does not exist, the following error message will be returned.

JSON

```
{
    "error":
        {
            "code": "InvalidContentFilterPolicy",
            "message": "Your request contains invalid content filter policy.
Please provide a valid policy."
```

```
        }
}
```

Report content filtering feedback

If you are encountering a content filtering issue, select the **Filters Feedback** button at the top of the playground. This is enabled in the **Images, Chat, and Completions** playground once you submit a prompt.

When the dialog appears, select the appropriate content filtering issue. Include as much detail as possible relating to your content filtering issue, such as the specific prompt and content filtering error you encountered. Do not include any private or sensitive information.

For support, please submit a support ticket.

Follow best practices

We recommend informing your content filtering configuration decisions through an iterative identification (for example, red team testing, stress-testing, and analysis) and measurement process to address the potential harms that are relevant for a specific model, application, and deployment scenario. After you implement mitigations such as content filtering, repeat measurement to test effectiveness. Recommendations and best practices for Responsible AI for Azure OpenAI, grounded in the Microsoft Responsible AI Standard can be found in the Responsible AI Overview for Azure OpenAI.

Related content

- Learn more about Responsible AI practices for Azure OpenAI: Overview of Responsible AI practices for Azure OpenAI models.
- Read more about content filtering categories and severity levels with Azure AI Foundry.
- Learn more about red teaming from our: Introduction to red teaming large language models (LLMs) article.
- Learn how to configure content filters using the API

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Create a new network-secured agent with user-managed identity

Article • 03/11/2025

Azure AI Agent Service offers a standard agent configuration with private networking, allowing you to bring your own (BYO) private virtual network. This setup creates an isolated network environment that lets you securely access data and perform actions while maintaining full control over your network infrastructure. This guide provides a step-by-step walkthrough of the setup process and outlines all necessary requirements.

> **Note**
>
> Standard setup with private networking can only be configured by deploying the Bicep template described in this article. Once deployed, agents must be created using the SDK or REST API. You can't use the Azure AI Foundry portal to create agents in a project with private networking enabled.

Benefits

- **No public egress**: foundational infrastructure ensures the right authentication and security for your agents and tools, without you having to do trusted service bypass.
- **Container injection**: allows the platform network to host APIs and inject a subnet into your network, enabling local communication of your Azure resources within the same virtual network.
- **Private resource access**: If your resources are marked as private and nondiscoverable from the internet, the platform network can still access them, provided the necessary credentials and authorization are in place.

Known limitations

- Azure Blob Storage: Using Azure Blob Storage files with the File Search tool isn't supported.

Prerequisites

1. An Azure subscription - Create one for free.

2. Python 3.8 or later

3. Ensure that the individual deploying the template has the Azure AI Developer role assigned at the resource group level where the template is being deployed.

4. Additionally, to deploy the template, you need to have the preset Role Based Access Administrator role at the subscription level.

    - The **Owner** role at the subscription level satisfies this requirement.
    - The specific admin role that is needed is `Microsoft.Authorization/roleAssignments/write`

5. Ensure that each team member who wants to use the Agent Playground or SDK to create or edit agents has been assigned the built-in **Azure AI Developer** RBAC role for the project.

    - Note: assign these roles after the template has been deployed
    - The minimum set of permissions required is: **agents/*/read**, **agents/*/action**, **agents/*/delete**

6. Install the Azure CLI and the machine learning extension. If you have the CLI already installed, make sure it's updated to the latest version.

7. Register providers. The following providers must be registered:

    - `Microsoft.KeyVault`
    - `Microsoft.CognitiveServices`
    - `Microsoft.Storage`
    - `Microsoft.MachineLearningServices`
    - `Microsoft.Search`
    - `Microsoft.Network`
    - `Microsoft.App`
    - To use Bing Search tool: `Microsoft.Bing`

    Console

    ```
    az provider register --namespace 'Microsoft.KeyVault'
    az provider register --namespace 'Microsoft.CognitiveServices'
    az provider register --namespace 'Microsoft.Storage'
    az provider register --namespace 'Microsoft.MachineLearningServices'
    az provider register --namespace 'Microsoft.Search'
    # only to use Grounding with Bing Search tool
    az provider register --namespace 'Microsoft.Bing'
    ```

Create a new network-secured agent with user-managed identity

Network secured setup: Agents use customer-owned, single-tenant search and storage resources. With this setup, you have full control and visibility over these resources, but you incur costs based on your usage. The following bicep template provides:

- Resources for the hub, project, storage account, key vault, AI Services, and Azure AI Search are created for you. The AI Services, AI Search, and Azure Blob Storage account are connected to your project/hub, and a gpt-4o-mini model is deployed in the westus2 region.
- Customer-owned resources are secured with a provisioned managed network and authenticated with a user-managed identity with the necessary RBAC (Role-Based Access Control) permissions. Private links and DNS (Domain Name System) zones are created on behalf of the customer to ensure network connectivity.

▼ **Bicep Technical Details**

The Bicep template automates the following configurations and resource provisions:

- Creates a User Assigned Identity.
  - The User Assigned Managed Identity requires the following Role-Based Access Roles:
    - KeyVault Secret Officer
    - KeyVault Contributor
    - Storage Blob Data Owner
    - Storage Queue Data Contributor
    - Cognitive Services Contributor
    - Cognitive Services OpenAI User
    - Search Index Data Contributor
    - Search Service Contributor

- Configures a managed virtual network with two subnet resources:
  - Azure resources subnet
    - Enables service endpoints for:
      - `Microsoft.KeyVault`
      - `Microsoft.Storage`
      - `Microsoft.CognitiveServices`
  - Agent resources subnet
    - Configured with subnet delegations for:
      - `Microsoft.app/environments`

- Provisions dependent resources
  - Azure Key Vault, Azure Storage, Azure OpenAI/AI Services, and Azure AI Search resources are created.
  - All resources are configured with:
    - Disabled public network access
    - Private endpoints in the Azure Resource subnet
    - Private DNS integration enabled
    - User assigned identity for authentication
- Creates a hub and project using the resources provisioned and configures them to use the Agent Resource Subnet.
  - Accomplished by configuring the `capabilityHost` (a subresource of hub/project) to use the Agent Resource Subnet for network isolation and secure communication.

Option 1: autodeploy the bicep template

Expand table

Template	Description	Autodeploy
`network-secured-agent.bicep`	Deploy a network secured agent setup that uses user-managed identity authentication on the Agent connections.	Deploy to Azure

Option 2: manually deploy the bicep template

1. To manually run the bicep templates, download the template from GitHub. Download the following from the `network-secured-agent` folder:
   a. `main.bicep`
   b. `azuredeploy.parameters.json`
   c. `modules-network-secured folder`
2. To authenticate to your Azure subscription from the Azure CLI, use the following command:

```
Console
az login
```

3. Create a resource group:

```
Console

az group create --name {my_resource_group} --location eastus
```

Make sure you have the Azure AI Developer role for the resource group you created.

4. Using the resource group you created in the previous step and one of the template files (`network-secured-agent`), run one of the following commands:

   a. To use default resource names, run:

   ```
   Console

   az deployment group create --resource-group {my_resource_group} --
   template-file main.bicep
   ```

   b. To specify custom names for the hub, project, storage account, and/or Azure AI service resources run the following command. A randomly generated suffix is added to prevent accidental duplication.

   ```
   Console

   az deployment group create --resource-group {my_resource_group} --
   template-file main.bicep --parameters aiHubName='your-hub-name'
   aiProjectName='your-project-name' storageName='your-storage-name'
   aiServicesName='your-ai-services-name'
   ```

   c. To customize other parameters, including the OpenAI model deployment, download, and edit the `azuredeploy.parameters.json` file, then run:

   ```
   Console

   az deployment group create --resource-group {my_resource_group} --
   template-file main.bicep --parameters @azuredeploy.parameters.json
   ```

Configure and run an agent

Expand table

Component	Description
Agent	Custom AI that uses AI models with tools.
Tool	Tools help extend an agent's ability to reliably and accurately respond during conversation. Such as connecting to user-defined knowledge bases to ground the model, or enabling web search to provide current information.
Thread	A conversation session between an agent and a user. Threads store Messages and automatically handle truncation to fit content into a model's context.
Message	A message created by an agent or a user. Messages can include text, images, and other files. Messages are stored as a list on the Thread.
Run	Activation of an agent to begin running based on the contents of Thread. The agent uses its configuration and Thread's Messages to perform tasks by calling models and tools. As part of a Run, the agent appends Messages to the Thread.
Run Step	A detailed list of steps the agent took as part of a Run. An agent can call tools or create Messages during its run. Examining Run Steps allows you to understand how the agent is getting to its results.

> **Tip**
>
> The following code shows how to create and run an agent using the Python Azure SDK. For additional languages, see the **quickstart**.

Run the following commands to install the python packages.

Console

```
pip install azure-ai-projects
pip install azure-identity
```

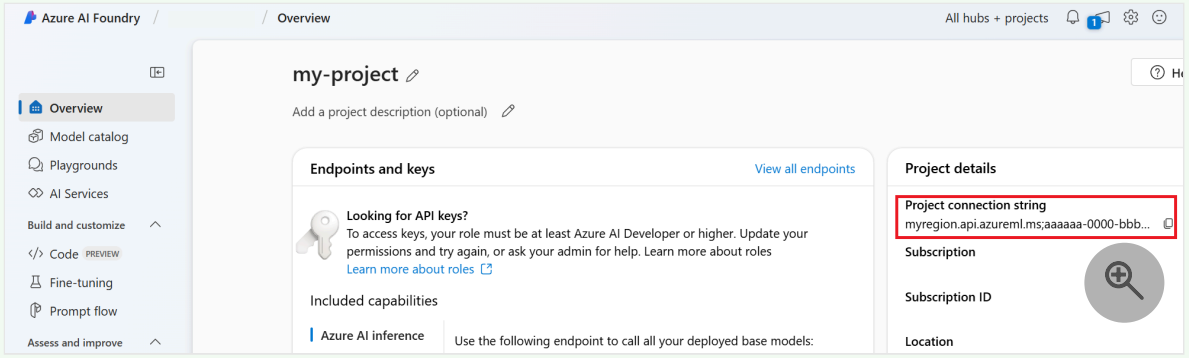
Next, to authenticate your API requests and run the program, use the az login command to sign into your Azure subscription.

Azure CLI

```
az login
```

Use the following code to create and run an agent. To run this code, you need to create a connection string using information from your project. This string is in the format:

`<HostName>;<AzureSubscriptionId>;<ResourceGroup>;<ProjectName>`

> **Tip**
>
> You can also find your connection string in the **overview** for your project in the Azure AI Foundry portal, under **Project details** > **Project connection string**.
>
> 
>

`HostName` can be found by navigating to your `discovery_url` and removing the leading `https://` and trailing `/discovery`. To find your `discovery_url`, run this CLI command:

Azure CLI

```
az ml workspace show -n {project_name} --resource-group
{resource_group_name} --query discovery_url
```

For example, your connection string might look something like:

`eastus.api.azureml.ms;12345678-abcd-1234-9fc6-62780b3d3e05;my-resource-group;my-project-name`

Set this connection string as an environment variable named `PROJECT_CONNECTION_STRING`.

> **Note**
>
> The following code sample shows creating an agent using Python. See the quickstart for examples in other programming languages.

Python

```
import os
from azure.ai.projects import AIProjectClient
from azure.ai.projects.models import CodeInterpreterTool
from azure.identity import DefaultAzureCredential
from typing import Any
from pathlib import Path

# Create an Azure AI Client from a connection string, copied from your Azure
```

```
AI Foundry project.
# It should be in the format "<HostName>;<AzureSubscriptionId>;
<ResourceGroup>;<ProjectName>"
# HostName can be found by navigating to your discovery_url and removing the
leading "https://" and trailing "/discovery"
# To find your discovery_url, run the CLI command: az ml workspace show -n
{project_name} --resource-group {resource_group_name} --query discovery_url
# Project Connection example: eastus.api.azureml.ms;12345678-abcd-1234-9fc6-
62780b3d3e05;my-resource-group;my-project-name
# You will need to login to your Azure subscription using the Azure CLI "az
login" command, and set the environment variables.

project_client = AIProjectClient.from_connection_string(
    credential=DefaultAzureCredential(),
conn_str=os.environ["PROJECT_CONNECTION_STRING"]
)

with project_client:
    # Create an instance of the CodeInterpreterTool
    code_interpreter = CodeInterpreterTool()

    # The CodeInterpreterTool needs to be included in creation of the agent
    agent = project_client.agents.create_agent(
        model="gpt-4o-mini",
        name="my-agent",
        instructions="You are helpful agent",
        tools=code_interpreter.definitions,
        tool_resources=code_interpreter.resources,
    )
    print(f"Created agent, agent ID: {agent.id}")

    # Create a thread
    thread = project_client.agents.create_thread()
    print(f"Created thread, thread ID: {thread.id}")

    # Create a message
    message = project_client.agents.create_message(
        thread_id=thread.id,
        role="user",
        content="Could you please create a bar chart for the operating
profit using the following data and provide the file to me? Company A: $1.2
million, Company B: $2.5 million, Company C: $3.0 million, Company D: $1.8
million",
    )
    print(f"Created message, message ID: {message.id}")

    # Run the agent
    run = project_client.agents.create_and_process_run(thread_id=thread.id,
agent_id=agent.id)
    print(f"Run finished with status: {run.status}")

    if run.status == "failed":
        # Check if you got "Rate limit is exceeded.", then you want to get
more quota
        print(f"Run failed: {run.last_error}")
```

```
    # Get messages from the thread
    messages = project_client.agents.list_messages(thread_id=thread.id)
    print(f"Messages: {messages}")

    # Get the last message from the sender
    last_msg = messages.get_last_text_message_by_role("assistant")
    if last_msg:
        print(f"Last Message: {last_msg.text.value}")

    # Generate an image file for the bar chart
    for image_content in messages.image_contents:
        print(f"Image File ID: {image_content.image_file.file_id}")
        file_name = f"{image_content.image_file.file_id}_image_file.png"

project_client.agents.save_file(file_id=image_content.image_file.file_id,
file_name=file_name)
        print(f"Saved image file to: {Path.cwd() / file_name}")

    # Print the file path(s) from the messages
    for file_path_annotation in messages.file_path_annotations:
        print(f"File Paths:")
        print(f"Type: {file_path_annotation.type}")
        print(f"Text: {file_path_annotation.text}")
        print(f"File ID: {file_path_annotation.file_path.file_id}")
        print(f"Start Index: {file_path_annotation.start_index}")
        print(f"End Index: {file_path_annotation.end_index}")

project_client.agents.save_file(file_id=file_path_annotation.file_path.file_
id, file_name=Path(file_path_annotation.text).name)

    # Delete the agent once done
    project_client.agents.delete_agent(agent.id)
    print("Deleted agent")
```

Next steps

Once you've provisioned your agent, you can add tools such as Grounding with Bing Search to enhance their capabilities.

Feedback

Monitor Azure AI Agent Service

Article • 04/23/2025

This article describes:

- The types of monitoring data you can collect for this service.
- Ways to analyze that data.

> **Note**
>
> If you're already familiar with this service and/or Azure Monitor and just want to know how to analyze monitoring data, see the **Analyze** section near the end of this article.

When you have critical applications and business processes that rely on Azure resources, you need to monitor and get alerts for your system. The Azure Monitor service collects and aggregates metrics and logs from every component of your system. Azure Monitor provides you with a view of availability, performance, and resilience, and notifies you of issues. You can use the Azure portal, PowerShell, Azure CLI, REST API, or client libraries to set up and view monitoring data.

- For more information on Azure Monitor, see the Azure Monitor overview.
- For more information on how to monitor Azure resources in general, see Monitor Azure resources with Azure Monitor.

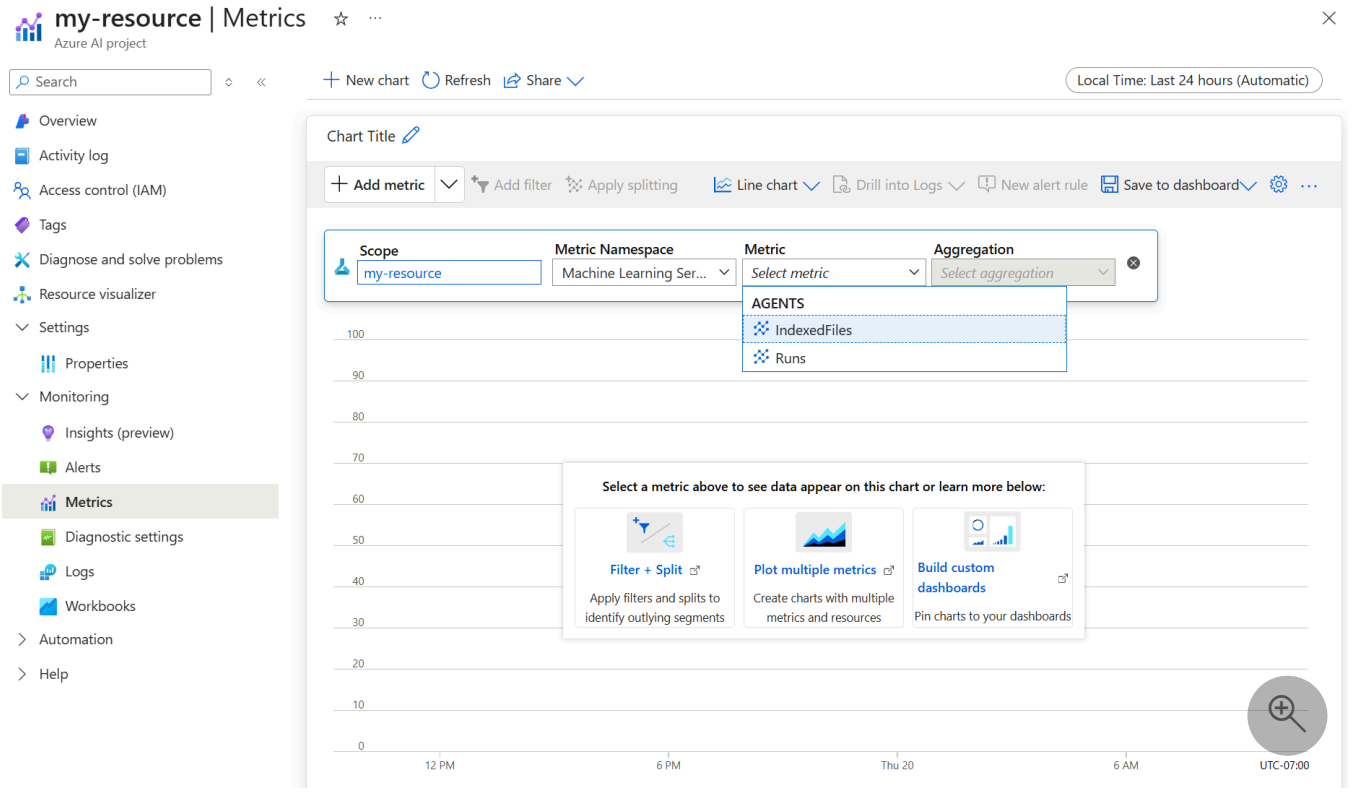
Monitoring is available for agents in a standard agent setup.

Dashboards

Azure AI Agent Service provides out-of-box dashboards. There are two key dashboards to monitor your resource:

- The metrics dashboard in the AI Foundry resource view
- The dashboard in the overview pane within the Azure portal

To access the monitoring dashboards, sign in to the Azure portal and then select **Monitoring** in the left navigation menu, then click **Metrics**.



Azure monitor platform metrics

Azure Monitor provides platform metrics for most services. These metrics are:

- Individually defined for each namespace.
- Stored in the Azure Monitor time-series metrics database.
- Lightweight and capable of supporting near real-time alerting.
- Used to track the performance of a resource over time.
- Collection: Azure Monitor collects platform metrics automatically. No configuration is required.

For a list of all metrics it's possible to gather for all resources in Azure Monitor, see Supported metrics in Azure Monitor.

Azure AI Agent Service has commonality with a subset of Azure AI services. For a list of available metrics for Azure AI Agent Service, see the monitoring data reference.

Analyze monitoring data

There are many tools for analyzing monitoring data.

Azure Monitor tools

Azure Monitor supports the metrics explorer, a tool in the Azure portal that allows you to view and analyze metrics for Azure resources. For more information, see Analyze metrics with Azure Monitor metrics explorer.

Azure Monitor export tools

You can get data out of Azure Monitor into other tools by using the REST API for metrics to extract metric data from the Azure Monitor metrics database. The API supports filter expressions to refine the data retrieved. For more information, see Azure Monitor REST API reference.

To get started with the REST API for Azure Monitor, see Azure monitoring REST API walkthrough.

Alerts

Azure Monitor alerts proactively notify you when specific conditions are found in your monitoring data. Alerts allow you to identify and address issues in your system before your customers notice them. For more information, see Azure Monitor alerts.

There are many sources of common alerts for Azure resources. The Azure Monitor Baseline Alerts (AMBA) site provides a semi-automated method of implementing important platform metric alerts, dashboards, and guidelines. The site applies to a continually expanding subset of Azure services, including all services that are part of the Azure Landing Zone (ALZ).

The common alert schema standardizes the consumption of Azure Monitor alert notifications. For more information, see Common alert schema.

Metric alerts evaluate resource metrics at regular intervals. Metric alerts can also apply multiple conditions and dynamic thresholds.

Every organization's alerting needs vary and can change over time. Generally, all alerts should be actionable and have a specific intended response if the alert occurs. If an alert doesn't require an immediate response, the condition can be captured in a report rather than an alert. Some use cases might require alerting anytime certain error conditions exist. In other cases, you might need alerts for errors that exceed a certain threshold for a designated time period.

Depending on what type of application you're developing with your use of Azure AI Agent Service, Azure Monitor Application Insights might offer more monitoring benefits at the application layer.

Azure AI Agent service alert rules

You can set alerts for any metric listed in the monitoring data reference.

Advisor recommendations

For some services, if critical conditions or imminent changes occur during resource operations, an alert displays on the service **Overview** page in the portal. You can find more information and recommended fixes for the alert in **Advisor recommendations** under **Monitoring** in the left menu. During normal operations, no advisor recommendations display.

For more information on Azure Advisor, see Azure Advisor overview.

Related content

- See Monitoring data reference for a reference of the metrics and other important values created for Azure AI Agent Service.
- See Monitoring Azure resources with Azure Monitor for general details on monitoring Azure resources.

Work with Azure AI Agent Service in Visual Studio Code (Preview)

Article • 04/30/2025

After you get started with the AI Foundry the VS Code extension, you can work with Azure AI Agent Service. Agents are "smart" microservices that:

- Answer questions using their training data or search other sources with Retrieval Augmented Generation (RAG)
- Perform specific actions
- Automate complete workflows

Agents combine AI models with tools to access and interact with your data.

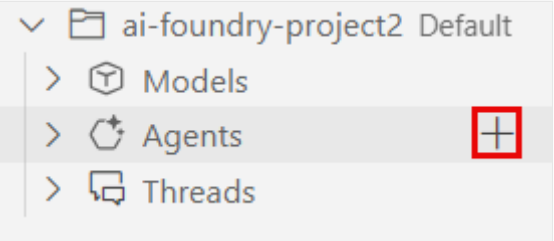
Azure AI Foundry developers can stay productive by developing, testing, and deploying agents in the familiar and powerful environment of VS Code.

> **Important**
>
> Items marked (preview) in this article are currently in public preview. This preview is provided without a service-level agreement, and we don't recommend it for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see Supplemental Terms of Use for Microsoft Azure Previews.

Create and edit Azure AI Agents within the designer view

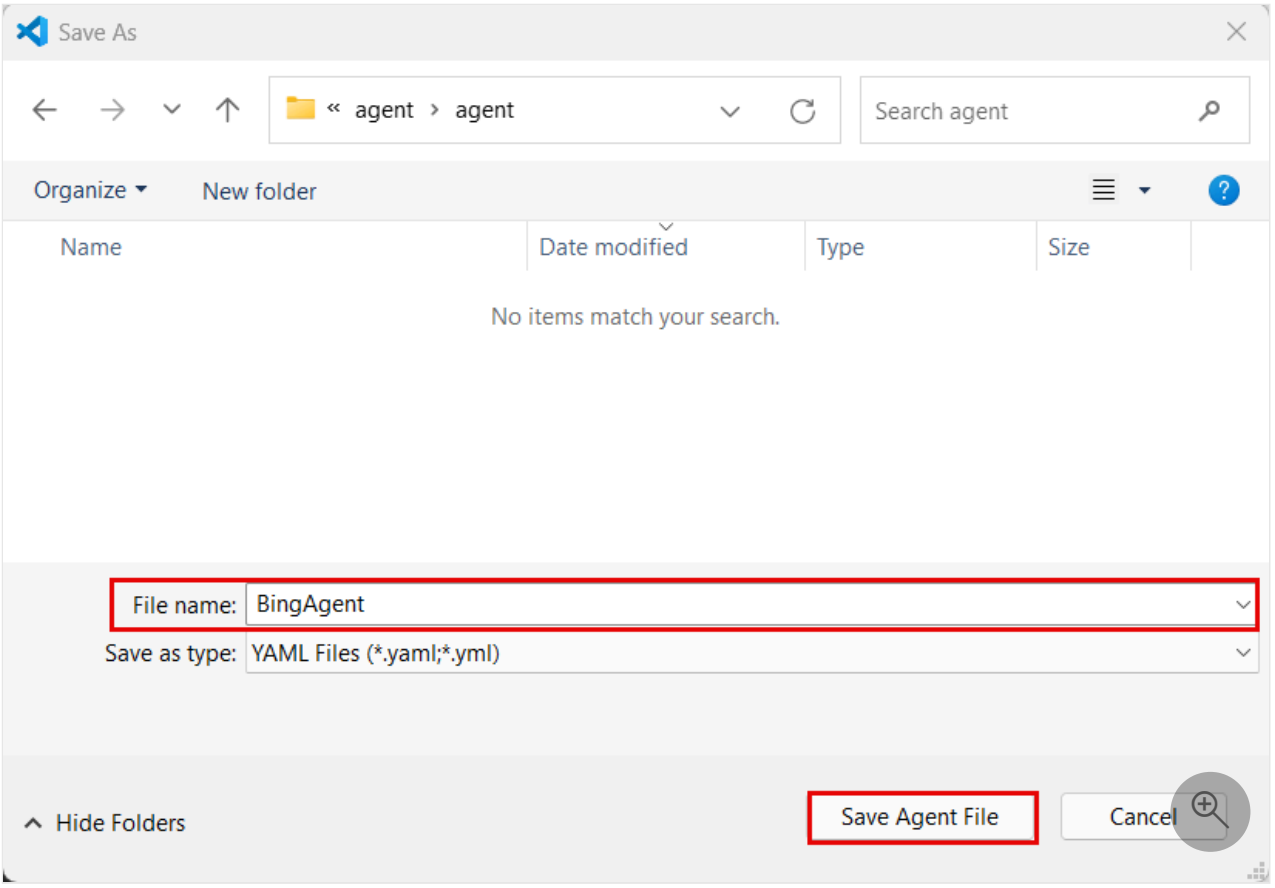
Follow these steps to create an Azure AI Agent:

1. First, finish the Get Started section to sign in to your Azure resources and set your default project.
2. Deploy a model to use with your agent.
3. In the Azure AI Foundry Extension view, find the **Resources** section.
4. Select the + (plus) icon next to the **Agents** subsection to create a new AI Agent.



5. In the **Save As** dialog box, select a directory and enter a name for your new AI Agent .yaml file.

6. Select the **Save Agent File** button to save your AI Agent file.



Interact with your agent in the designer

After you choose your save location, both the agent .yaml file and the Designer view will open to edit your AI Agent.

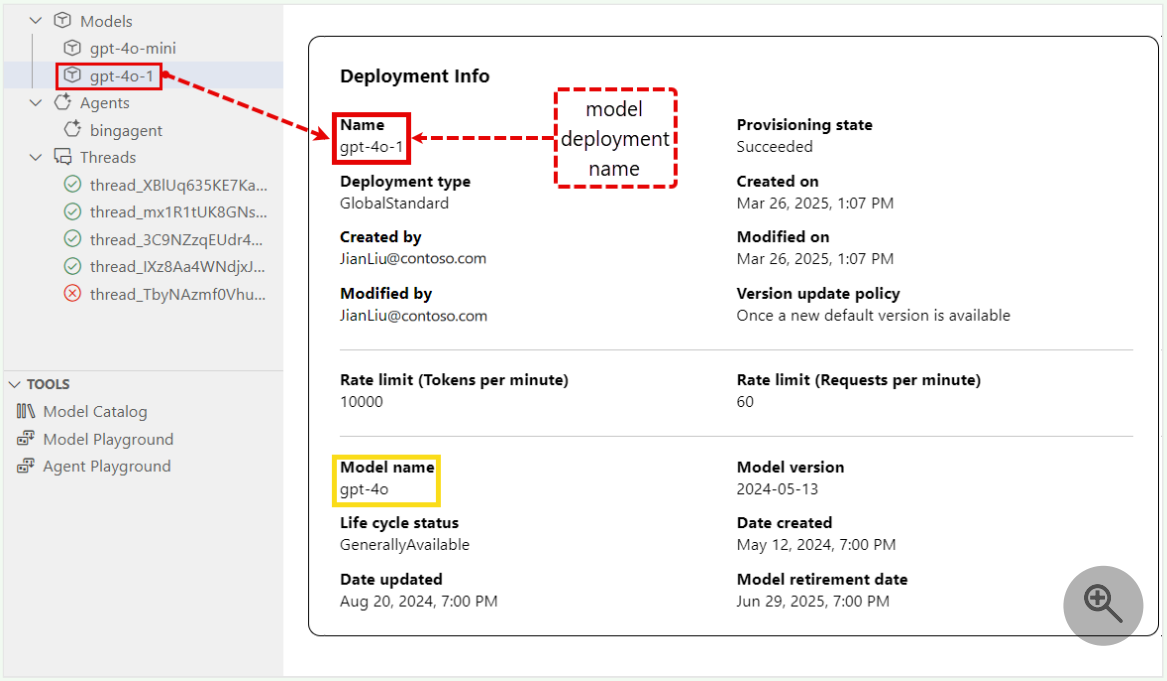
1. Perform the following tasks in the agent designer:

   a. Enter a name for your agent in the prompt.

   b. Enter your model deployment name. The deployment name you chose when you deployed an existing model.

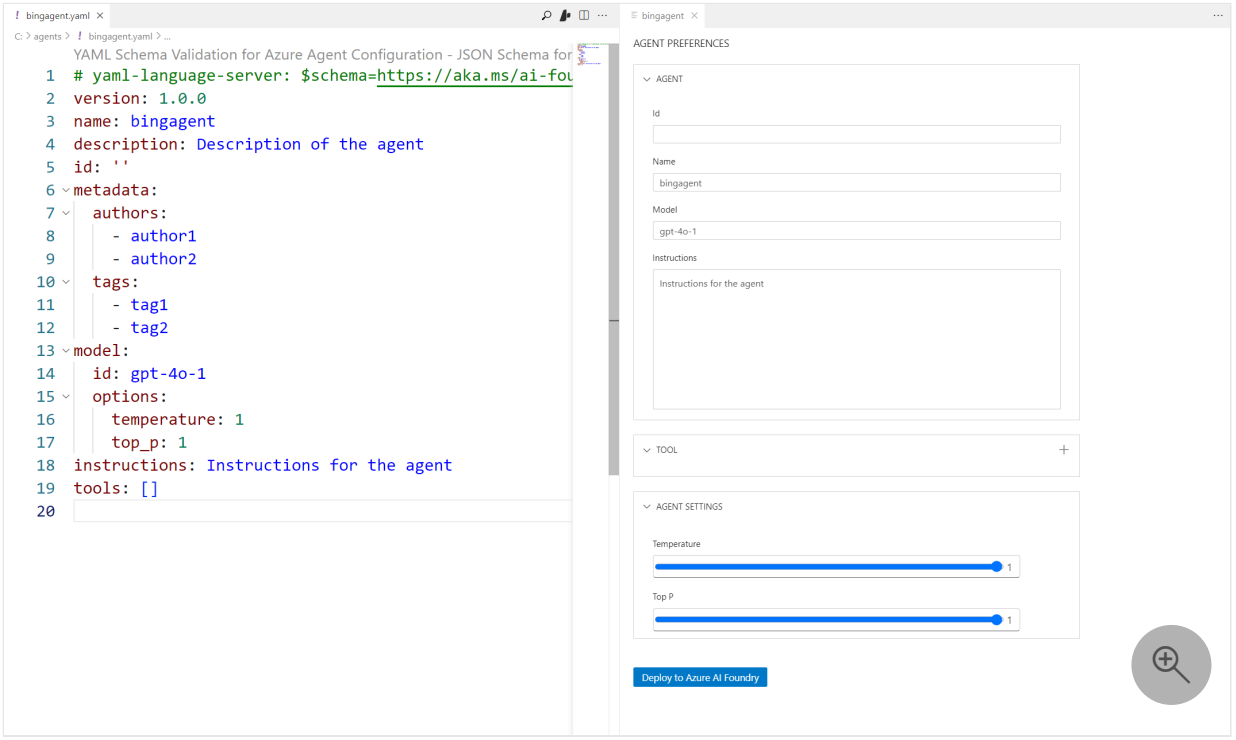
   > **Tip**

The model deployment name must be the exact name you chose for the model you deployed in your Azure AI Foundry project. In the following image, `gpt-4o-1` is the model deployment name you chose at deployment. `gpt-4o` is the model name.



c. Configure the following fields. The **ID** is generated by the extension:

- Add a description for your agent
- Set system instructions
- Configure tools for agent use



d. To save the .yaml file, select **File** > **Save** in the VS Code menu bar.

Explore the Azure AI Agent YAML definition

Your AI Agent .yaml file was opened at the same time the designer was. This file contains the details and setup information for your agent, similar to the following .yaml file example:

yml

```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: []
```

Add tools to the Azure AI Agent

Azure AI Agent Service has a set of knowledge and action tools that you can use to interact with your data sources, such as:

- Grounding with Bing search
- Azure AI Search
- Azure Functions
- File retrieval
- Code interpreter
- OpenAPI Specified tools

Configure the tools YAML file

The Agent Designer adds tools to an AI Agent via .yaml files.

Create a tool configuration .yaml file using the following steps:

1. Perform any setup steps that might be required. See the article for the tool you're interested in using. For example, Grounding with Bing search.

2. Once you complete the setup, create a yaml code file that specifies the tool's configuration. For example, this format for Grounding with Bing Search:

   ```yml
   type: bing_grounding
   name: bing_search
   configuration:
     tool_connections:
       - >-
         /subscriptions/<Azure Subscription ID>/resourceGroups/<Azure Resource Group name>/providers/Microsoft.MachineLearningServices/workspaces/<Azure AI Foundry Project name>/connections/<Bing connection name>
   ```

3. Replace the placeholders in the connection string under the `tool_connections` section with your information:

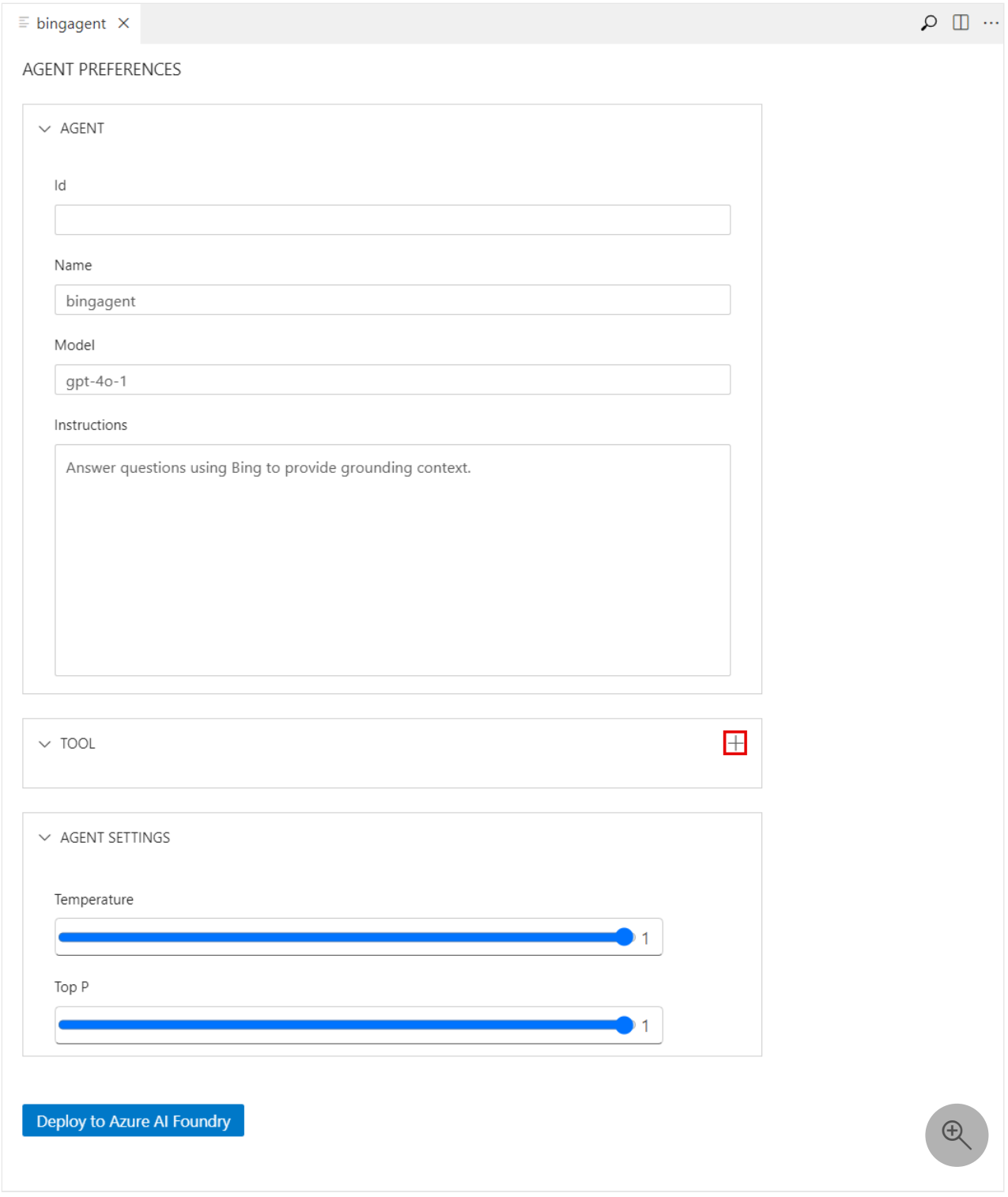
   - Azure Subscription ID
   - Azure Resource Group name
   - Azure AI Foundry Project name
   - Bing connection name

4. To save the .yaml file, select **File** > **Save** in the VS Code menu bar.

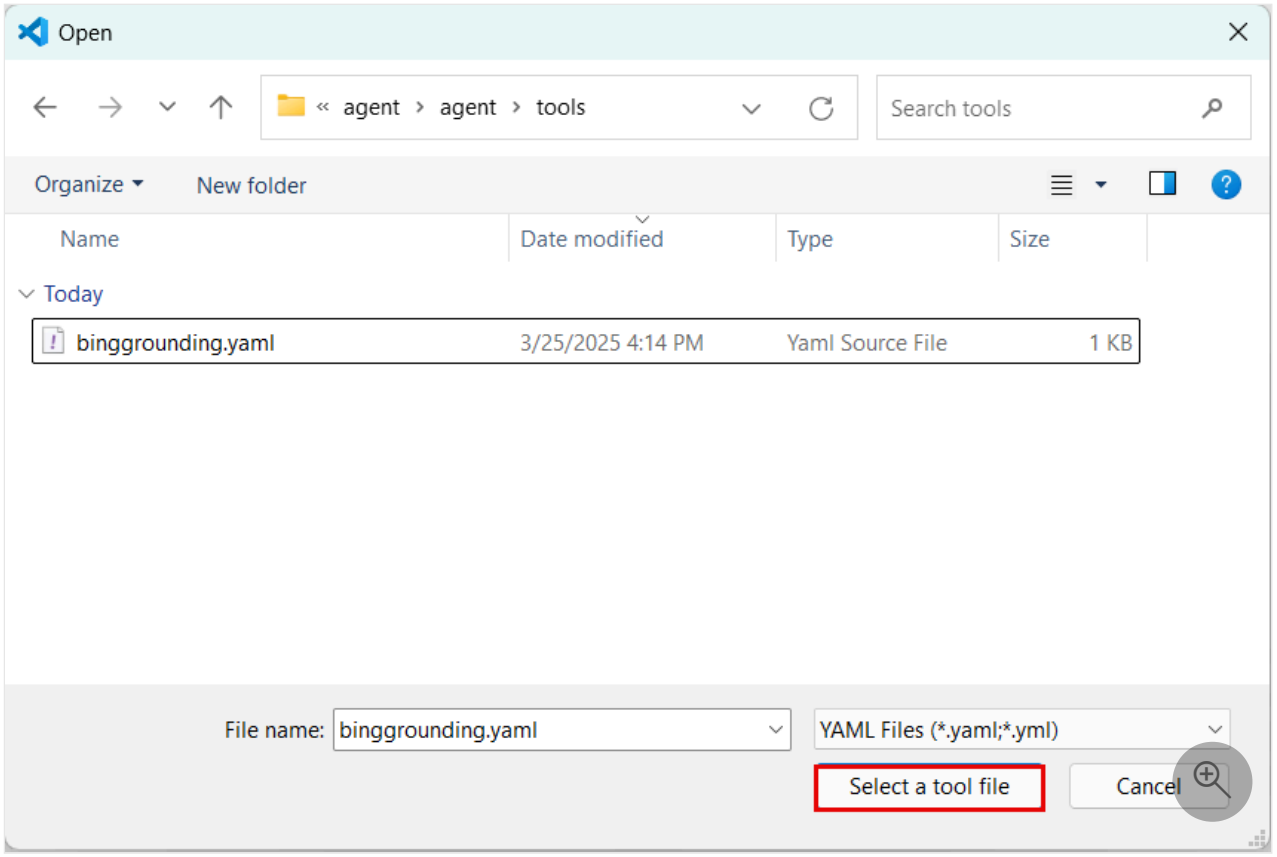
Connect the tools file to the AI Agent

Add a tool to the AI Agent with the following steps:

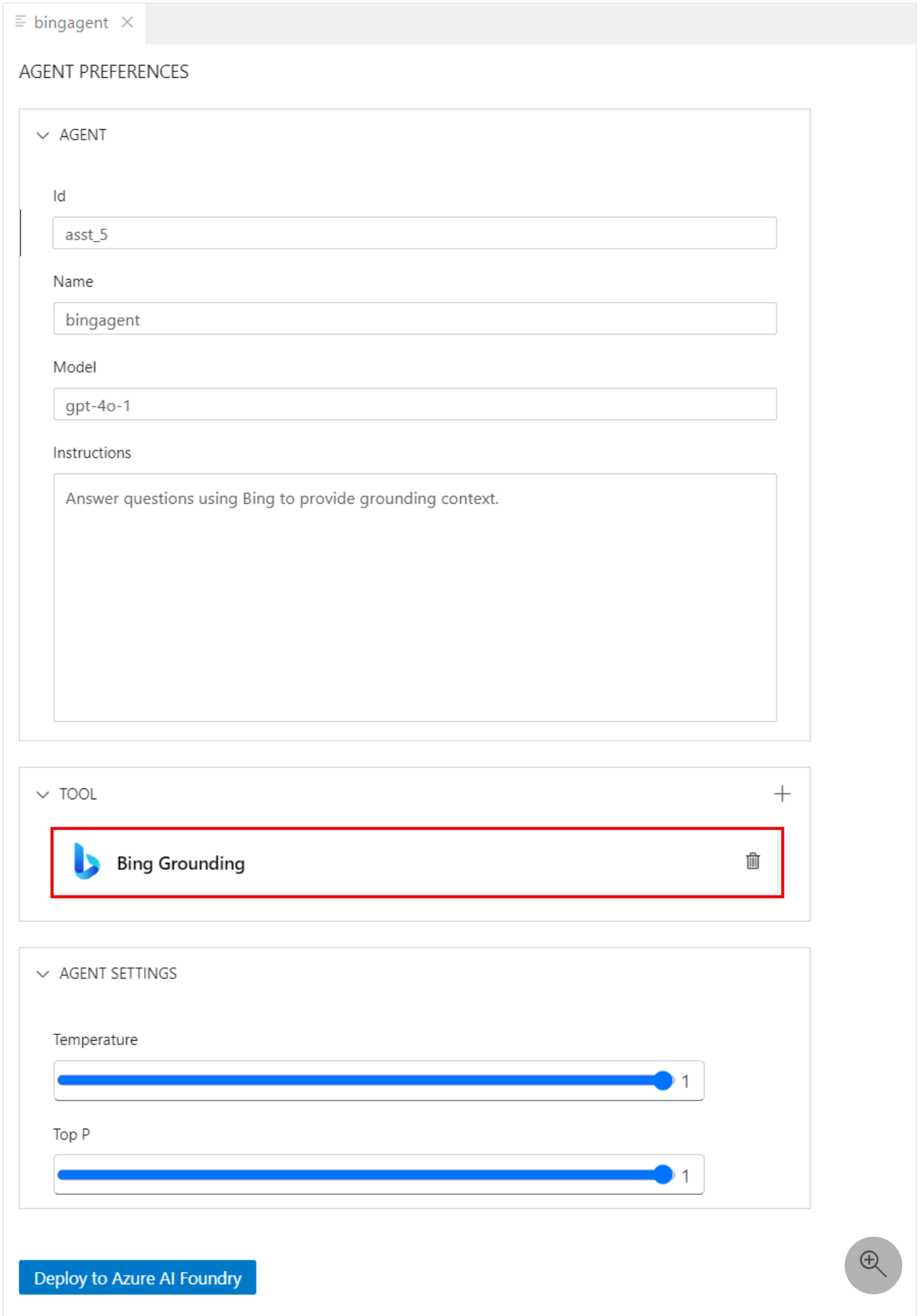
1. Select the + (plus) icon next to the **TOOL** section in the designer.



2. In the file explorer that appears, select the .yaml tool file to use. Select the **Select a tool file** button to add the tool to the agent.



3. The tool is displayed in the **TOOL** section.

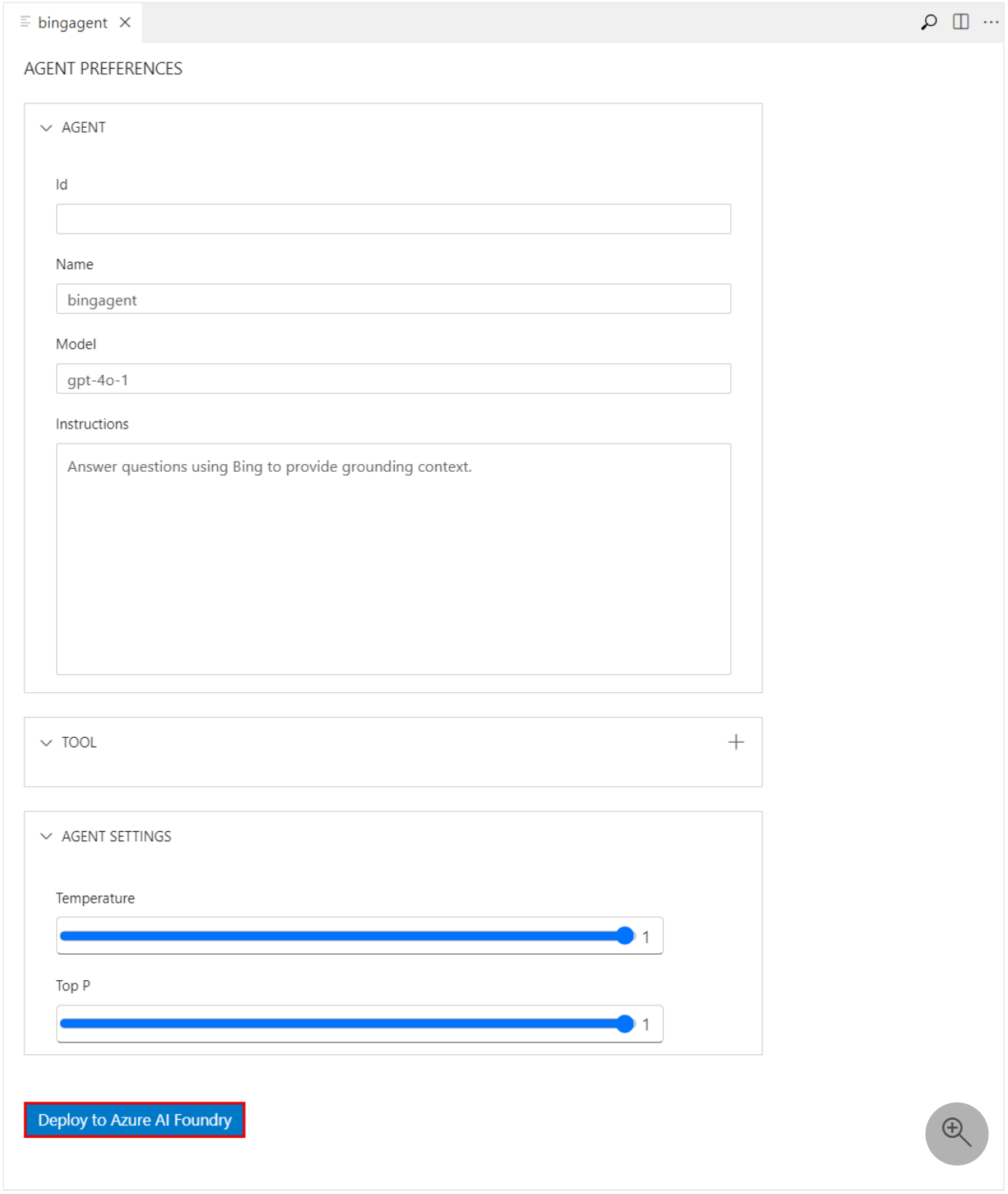


4. To save the .yaml file, select **File** > **Save** in the VS Code menu bar.

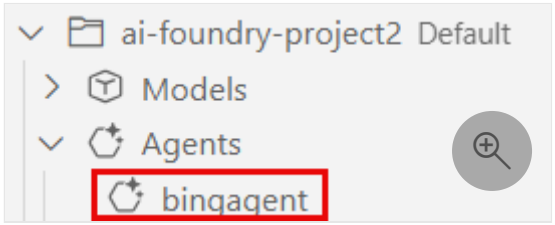
Deploy Azure AI Agents to the Azure AI Foundry Studio

Deploy your agent directly to Azure AI Foundry with the following steps:

1. Select the **Deploy to Azure AI Foundry** button in the bottom-left of the designer.



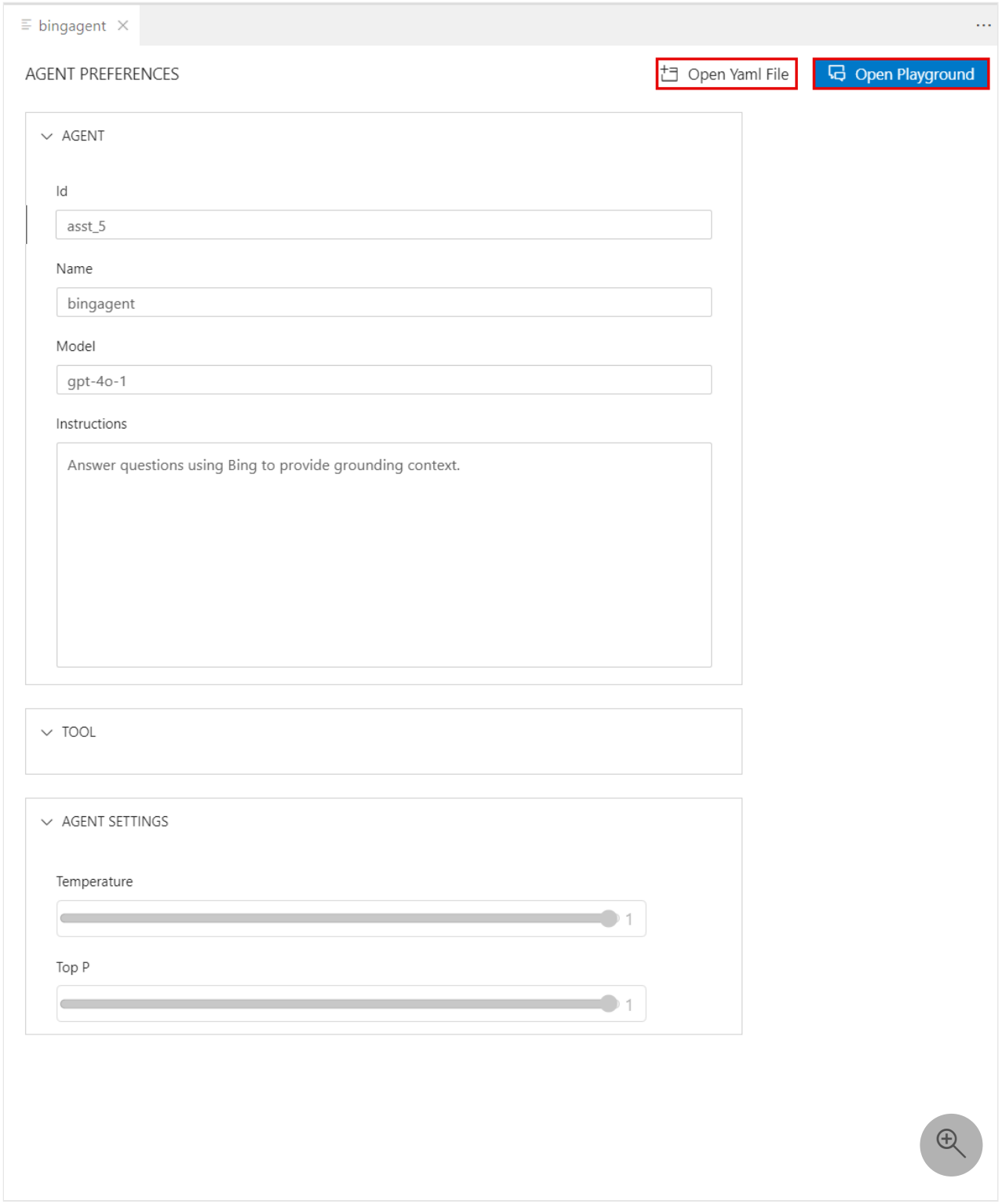
2. In the VS Code navbar, refresh the **Azure Resources** view. The deployed agent is displayed under the **Agents** subsection.



View the deployed AI Agent details

Selecting the deployed agent opens the **Agent Preferences** page in a view only mode.

- Select the **Open Yaml File** to view the yaml definition of the agent.
- Select the **Open Playground** button to open the **Agent Playground**.



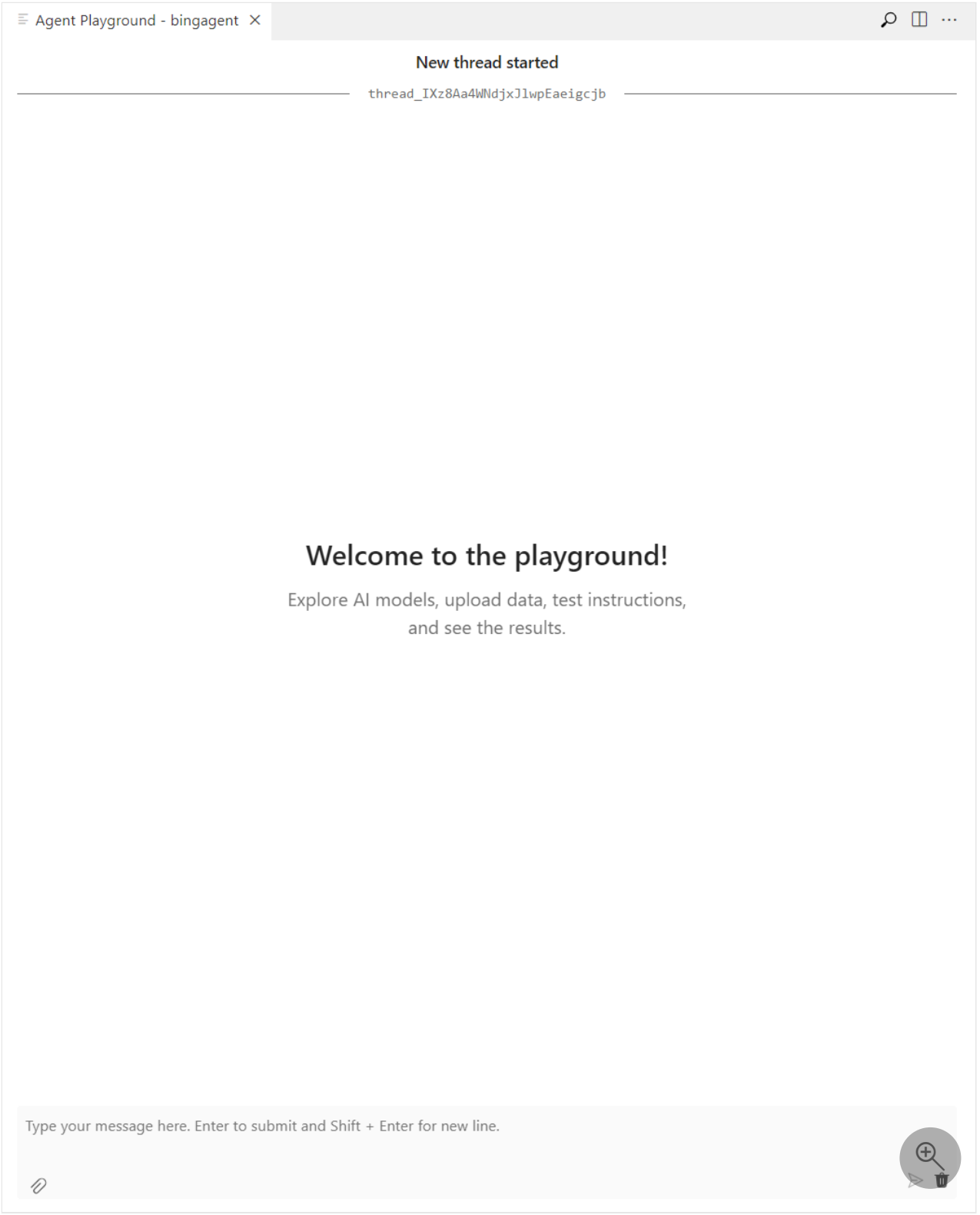
Interact with Agents using agents playground

Open the **Agents Playground** using the following steps:

1. Right-click on your deployed agent and select the **Open Playground** option This action starts a thread with your agent and let you send messages.

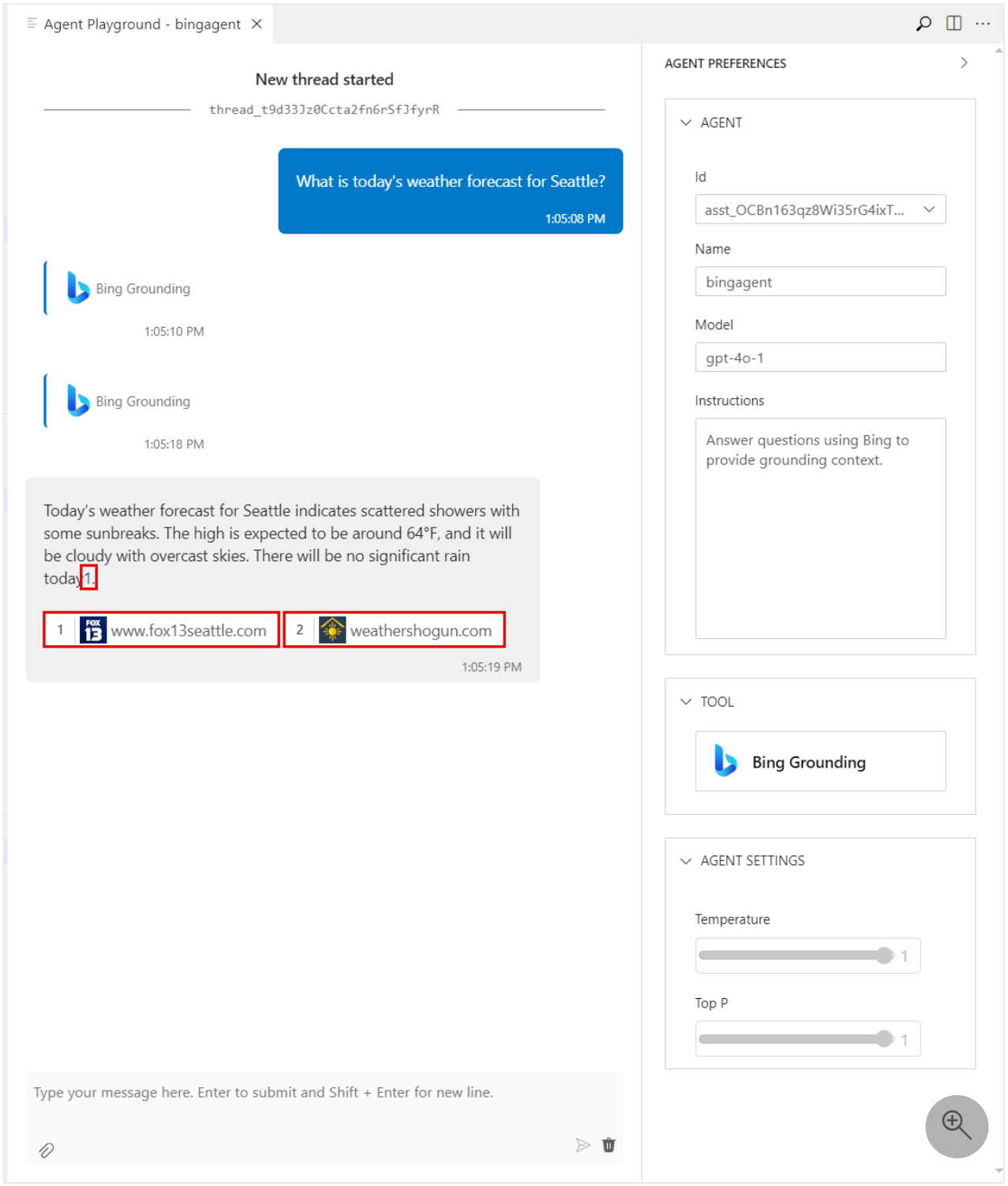
2. Alternatively, select the **Agent Playground** link in the **Tools** subsection, and select your agent from the top-center list.

3. The **Playground** page is displayed.



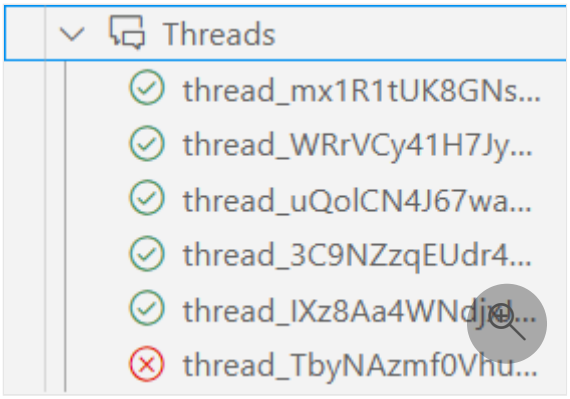
4. Type your prompt and see the outputs. The **Grounding with Bing search** tool is used to search the web for information. The agent uses the model and tools you configured in the

agent designer. The source of the information is displayed in the **Agent Annotations** section, highlighted in the following image.



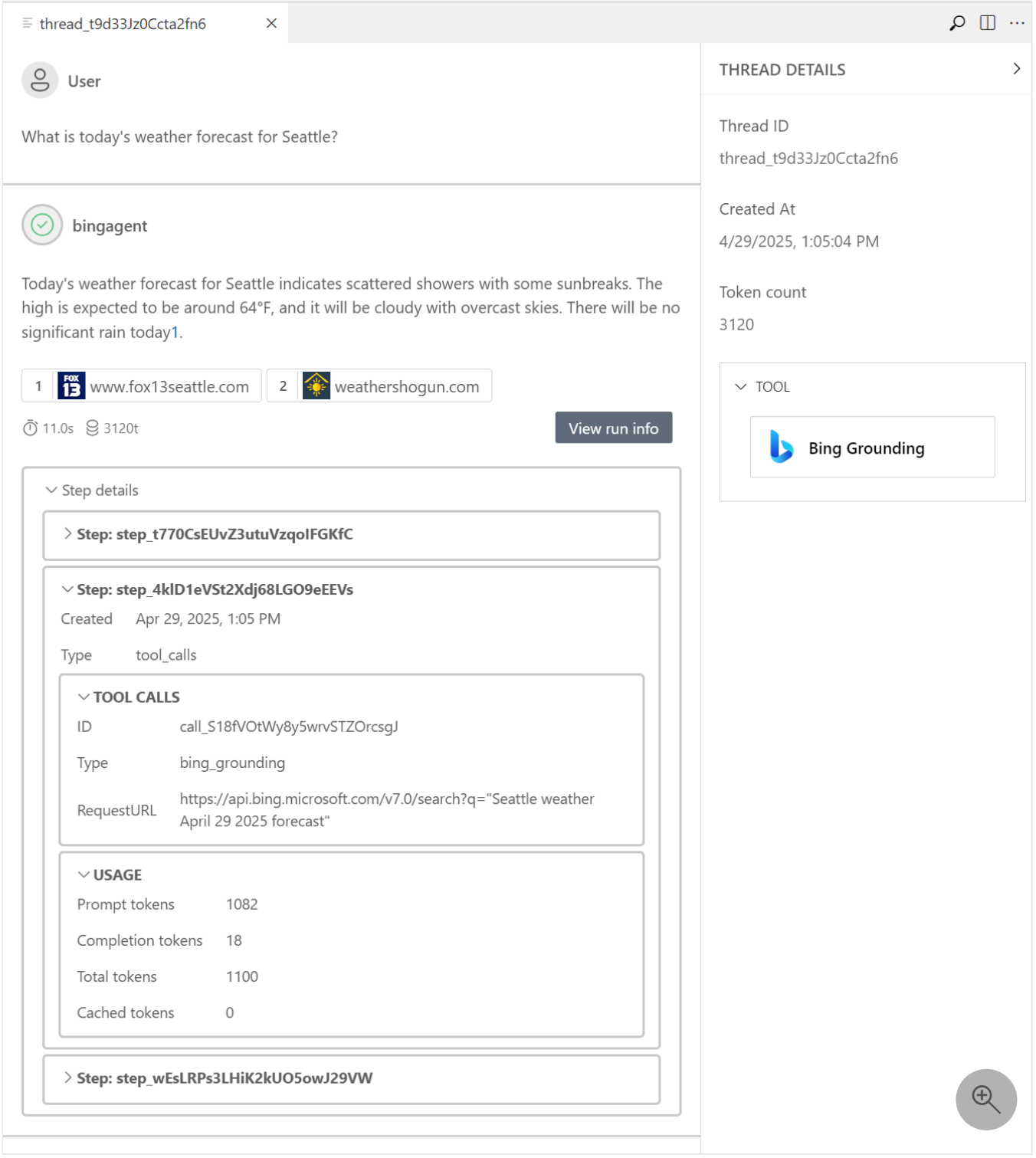
Explore Threads

The **Threads** subsection displays the threads created during a run with your agent. In the Azure Resources Extension view, select the **caret** icon in front of the **Threads** subsection to view the list of threads.



View thread details

Select a thread to see the **Thread Details** page.



- A **Thread** is a conversation session between an agent and a user. Threads store **Messages** and automatically handle truncation to fit content into a model's context.
- A **Message** is a single interaction between the agent and the user. Messages can include text, images, and other files. Messages are stored as a list on the Thread.
- A **Run** is a single execution of an agent. Each run can have multiple threads, and each thread can have multiple messages. The agent uses its configuration and Thread's Messages to perform tasks by calling models and tools. As part of a Run, the agent appends Messages to the Thread.

View run details

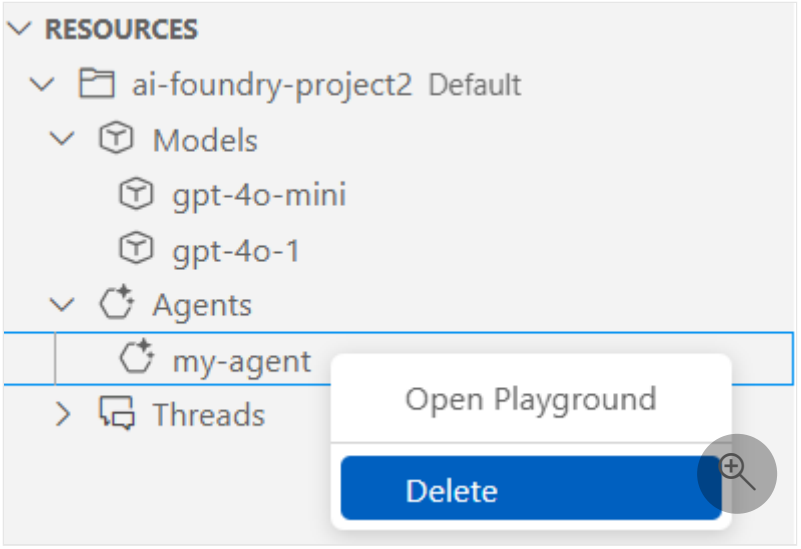
Select the **View run info** button in the **Thread Details** page to see the run information in a JSON file.

Cleanup resources

The Azure resources that you created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Delete your agents

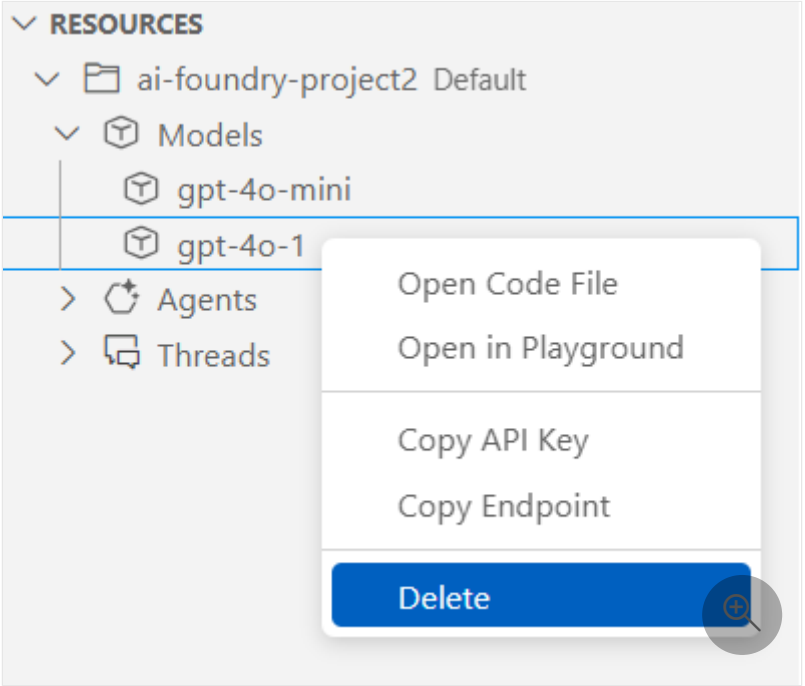
Delete the deployed agent in the online AI Foundry portal. Select **Agents** from the navigation menu on the left, select your agent, then select the **Delete** button.



Delete your models

1. In the VS Code navbar, refresh the **Azure Resources** view. Expand the **Models** subsection to display the list of deployed models.

2. Right-click on your deployed model to delete and select the **Delete** option.



Delete your tools

Delete the connected tool with the following steps:

1. Open the Azure portal
2. Select the Azure Resource Group containing the tool.
3. Select the **Delete** button.

Next steps

- Learn about the tools you can use with Azure AI Agents, such as file search, or code interpreter.

Transparency Note for Azure Agent Service

Article • 01/16/2025

What is a Transparency Note?

An AI system includes not only the technology, but also the people who will use it, the people who will be affected by it, and the environment in which it is deployed. Creating a system that is fit for its intended purpose requires an understanding of how the technology works, what its capabilities and limitations are, and how to achieve the best performance. Microsoft's Transparency Notes are intended to help you understand how our AI technology works, the choices system owners can make that influence system performance and behavior, and the importance of thinking about the whole system, including the technology, the people, and the environment. You can use Transparency Notes when developing or deploying your own system or share them with the people who will use or be affected by your system.

Microsoft's Transparency Notes are part of a broader effort at Microsoft to put our AI Principles into practice. To find out more, see the Microsoft AI principles.

The basics of Azure AI Agent Service

Introduction

Azure AI Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality and extensible AI agents without needing to manage the underlying compute and storage resources. Azure AI Agent Service provides integrated access to **models, tools, and technology** and enables you to extend the functionality of agents with knowledge from connected sources (such as Bing Search, SharePoint, Fabric, Azure Blob storage, and licensed data) and with actions using tools such as Azure Logic Apps, Azure Functions, OpenAPI 3.0 specified tools, and Code Interpreter. Learn more.

Key terms

The following are key components of the Azure AI Agent Service SDK (and the Azure AI Foundry portal experience powered by it):

Expand table

Term	Definition
Developer	A customer of Azure AI Agent Service who builds an Agent.
User	A person who uses and/or operates an Agent that is created by a Developer.
Agent	An application or a system that uses generative AI models with tools to access and interact with real-world data sources, APIs and systems to achieve user-specified goals such as answer questions, perform actions, or completely automate workflows, with or without human supervision.
Tool	A built-in or custom-defined functionality that enables an Agent to perform simple or complex tasks or interact with information sources, applications, and/or services via the Agent Service SDK or Azure AI Foundry portal.
Knowledge Tool	A Tool that enables an Agent to access and process data from internal and external sources, including information beyond its model training cut-off date, to improve the accuracy and relevance of responses to user queries.
Action Tool	A Tool that enables an Agent to perform tasks and take actions on behalf of Users by integrating with external systems, APIs, and services.
Thread	A conversation session between an Agent and a User. Threads store Messages and automatically handle truncation to fit content into a model's context.
Message	A message created by an Agent or a User. Messages can include text, images, and other files. Messages are stored as a list on the Thread.
Run	The activation of an Agent to begin running based on the contents of the Thread. The Agent uses its configuration and the Thread's Messages to perform tasks by calling models and Tools. As part of a Run, the Agent appends Messages to the Thread.
Run Steps	A detailed list of steps the Agent took as part of a Run. An Agent can call Tools or create Messages during its Run. Examining Run Steps allows you to understand how the Agent is getting to its final results.

Relevant capability concepts

Expand table

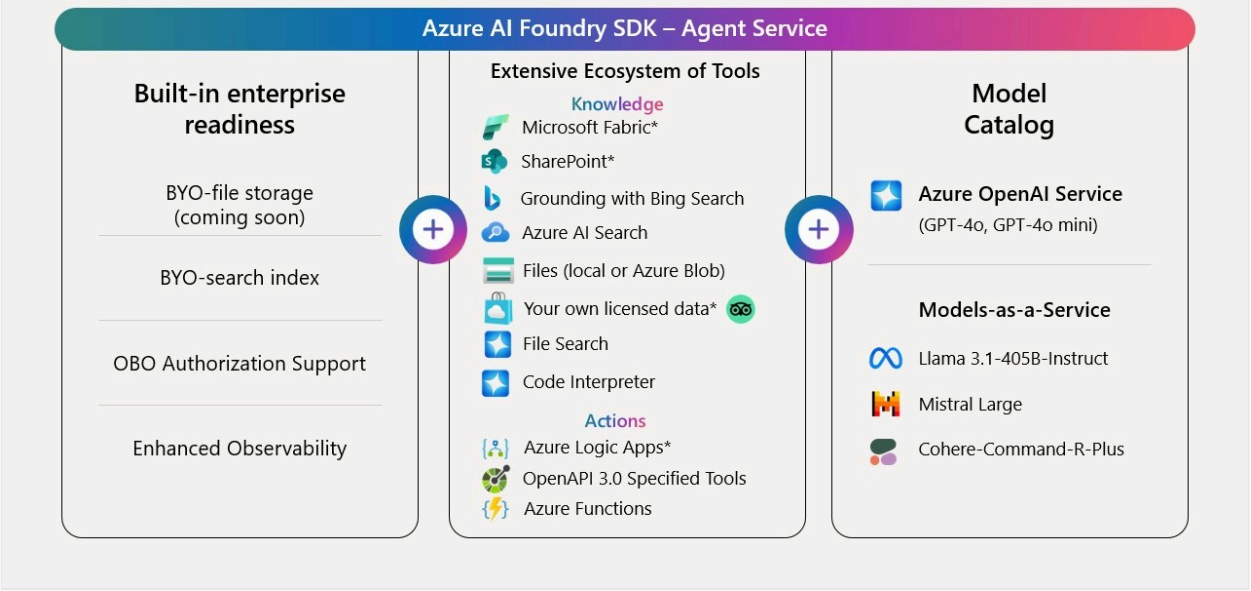
Term	Definition
Agentic AI system	An umbrella term that includes the following common capabilities that developers may enable in their Agents when they use the Azure AI Agent Service.
Autonomy	The ability to independently execute actions and exercise control over system

Term	Definition
	behavior with varying degrees of human supervision.
Reasoning	The ability to process information, understanding context and outcomes of various potential courses of actions, tasks or engagements with third-party users.
Planning	The ability to break down complex, user-specified goals and actions into tasks and sub-tasks for execution. Planned tasks are created by one or more agents.
Memory	The ability to store or retain information or context from previous observations, interactions or system behaviors.
Adaptability	The ability to change or adjust behavior and improve performance based on information gathered from the environment or prior experience.
Extensibility	The ability to call resources (e.g., such as external knowledge sources) and execute functions (e.g., sending an email) from connected systems, software, or platforms, including using Tools.

Capabilities

System behavior

Azure AI Agent Service provides integration with securely managed data, out-of-the box Tools and automatic Tool calling that enable developers to build Agents that can have the ability to reason, plan, and execute tasks from a high-level goal specified by a user. Azure AI Agent Service enables rapid Agent development with built-in memory management and a sophisticated interface to seamlessly integrate with popular compute platforms, bridging LLM capabilities with general purpose, programmatic actions.



Key features of Azure AI Agent Service include:

1. **Rapidly develop and automate processes:** Agents need to seamlessly integrate with the right tools, systems and APIs to perform deterministic or non-deterministic actions.
2. **Integrate with extensive memory and knowledge connectors:** Agents need to manage conversation state and connect with internal and external knowledge sources to have the right context to complete a process.
3. **Flexible model choice:** Agents built with the appropriate model for their tasks can enable better integration of information from multiple data types, yield better results for task-specific scenarios, and improve cost efficiencies in scaled deployments.
4. **Built-in enterprise readiness:** Agents need to be able to support an organization's unique data privacy and compliance needs, scale with an organization's needs, and complete tasks reliably and with high quality.

Extensibility capabilities

Extensibility capabilities of Azure AI Agent Service enable Agents to interact with knowledge sources, systems, and platforms to ground and enhance Agent functionality. Specifically:

Secure grounding of Agent outputs with a rich ecosystem of knowledge sources

Developers can configure a rich ecosystem of knowledge sources to enable an Agent to access and process data from multiple sources, improving accuracy of responses and outputs. Connectors to these data sources operate within your designated network parameters. Knowledge Tools built into Azure AI Agent Service include:

- **File Search** (a built-in retrieval-augmented generation (RAG) tool to process and search through private data in Azure AI Search, Azure Blob Storage, and local files)
- **Grounding with Bing Search** (a web search tool that uses Bing Search to extract information from the web)
- **SharePoint** (built-in tools that connect an organization’s internal documents in SharePoint for grounded responses)
- **Fabric AI Skills** (a built-in tool to chat with structured data on Microsoft Fabric using generative AI)
- **Bring your licensed data** (a tool that enables grounding using proprietary data accessed using a licensed API key obtained by the Developer from the data provider, e.g., TripAdvisor)

Agents simplify secure data access to SharePoint and Fabric AI Skills through on-behalf-of (OBO) authentication, allowing the Agent to access only the SharePoint or Fabric files for which the User has permissions.

Enabling autonomous actions with or without human input through Action Tools

Developers can connect an Agent to external systems, APIs, and services through Action Tools, enabling the Agent to perform tasks and take actions on behalf of Users. Action Tools built into Azure AI Agent Service include:

- **Code Interpreter** (a tool that can write and run Python code in a secure environment, handle diverse data formats and generate files with data and visuals)
- **Azure Logic Apps** (a cloud-based PaaS tool that enables automated workflows using 1,400+ built-in connectors)
- **Azure Functions** (a tool that enables an Agent to execute serverless code for synchronous, asynchronous, long-running, and event-driven actions)
- **OpenAPI 3.0 specified tools** (a custom function defined with OpenAPI 3.0 specification to connect an Agent to external OpenAPI-based APIs securely)

Orchestrating multi-agent systems

Multi-agent systems using Azure AI Agent Service can be designed to achieve performant autonomous workflows for specific scenarios. In multi-agent systems, multiple context-aware autonomous agents, whether humans or AI systems, interact or work together to achieve individual or collective goals specified by the user. Azure AI Agent Service works out-of-the-box with multi-agent orchestration frameworks that are wireline compatible[1] with the Assistants API, such as **AutoGen**, a state-of-the-art research SDK for Python created by Microsoft Research, and **Semantic Kernel**, an enterprise AI SDK for Python, .NET, and Java.

When building a new multi-agent solution, start with building singleton agents with Azure AI Agent Service to get the most reliable, scalable, and secure agents. You can then orchestrate these agents together, using supported orchestration frameworks. AutoGen is constantly evolving to find the best collaboration patterns for agents (and humans) to work together. Features that show production value with AutoGen can then be moved into Semantic Kernel if you're looking for production support and non-breaking changes.

[1]*Wireline compatible* means that an API can communicate and exchange data in a way that is fully compatible with an existing protocol, existing data formats and

communication standards, in this case the Assistants API protocol. It means that two systems can work together seamlessly without needing changes to their core implementation.

Use cases

Intended uses

Azure AI Agent Service is **flexible and use-case agnostic.** This presents multiple possibilities to automate routine tasks and unlock new possibilities for knowledge work - whether it is personal productivity agents that send emails and schedule meetings, research agents that continuously monitor market trends and automate report creation, sales agents that can research leads and automatically qualify them, customer service agents that proactively follow up with personalized messages, or developer agents that can upgrade your code base or evolve a code repository interactively. Here are examples of intended uses of agents developed using Azure AI Agent Service:

- **Healthcare: Streamlined Staff Orientation and Basic Administrative Support:** A hospital's administrative assistant deploys an agent to collate standard operational procedures, staff directories, and shift policies into concise orientations for new nurses; final materials are reviewed and approved by HR, reducing repetitive work without compromising content quality.
- **Retail: Personalized Shopping Guidance:** A local boutique owner can deploy an agent that recommends gift options based on a customer's stated needs and past purchases, guiding shoppers responsibly through complex product catalogs without pushing biased or misleading information.
- **Government: Citizen Request Triage and Community Event Coordination:** A city clerk uses an agent to categorize incoming service requests (e.g., pothole repairs), assign them to the right departments, and compile simple status updates; officials review and finalize communications to maintain transparency and accuracy.
- **Education: Assisting with Research and Reference Gathering:** A teacher relies on an agent to gather age-appropriate articles and resources from reputable sources for a planetary science lesson; the teacher verifies the materials for factual accuracy and adjusts them to fit the curriculum, ensuring students receive trustworthy content.
- **Manufacturing: Inventory Oversight and Task Scheduling:** A factory supervisor deploys an agent to monitor inventory levels, schedule restocking when supplies run low, and optimize shift rosters; management confirms the agent's suggestions and retains final decision-making authority.

Considerations when choosing a use case

We encourage customers to leverage Azure AI Agent Service in their innovative solutions or applications. However, here are some things to consider when choosing a use case:

- **Avoid scenarios where use or misuse of the system could result in significant physical or psychological injury to an individual.** For example, scenarios that diagnose patients or prescribe medications have the potential to cause significant harm.
- **Avoid scenarios where use or misuse of the system could have a consequential impact on life opportunities or legal status**. Examples include scenarios where the AI system or agent could affect an individual's legal status, legal rights, or their access to credit, education, employment, healthcare, housing, insurance, social welfare benefits, services, opportunities, or the terms on which they're provided.
- **Avoid high-stakes scenarios that could lead to harm.** The model used in an agent may reflect certain societal views, biases, and other undesirable content present in the training data or the examples provided in the prompt. As a result, we caution against using agents in high-stakes scenarios where unfair, unreliable, or offensive behavior might be extremely costly or lead to harm.
- **Carefully consider use cases in high stakes domains or industry.** Examples include but are not limited to healthcare, medicine, finance, or legal domains.
- **Legal and regulatory considerations.** Organizations need to evaluate potential specific legal and regulatory obligations when using any AI services and solutions, including agents, which may not be appropriate for use in every industry or scenario. Additionally, agents may not be used in ways prohibited in applicable regulations, terms of service, and relevant codes of conduct.

Limitations

Technical limitations, operational factors and ranges

- **Generative AI model limitations:** Because Azure AI Agent Service works with a variety of models, the overall system inherits the limitations specific to those models. Before selecting a model to incorporate into your agent, carefully evaluate the model to understand its limitations. Consider reviewing the Azure OpenAI Transparency Note for additional information about generative AI limitations that are also likely to be relevant to the system and review other best practices for incorporating generative AI into your agent application.

- **Tool orchestration complexities:** AI Agents depend on multiple integrated tools and data connectors (such as Bing Search, SharePoint, and Azure Logic Apps). If any of these tools are misconfigured, unavailable, or return inconsistent results, or a high number of tools are configured on a single agent, the agent's guidance may become fragmented, outdated, or misleading.
- **Unequal representation and support:** When serving diverse user groups, AI Agents can show uneven performance if language varieties, regional data, or specialized knowledge domains are underrepresented. A retail agent, for example, might offer less reliable product recommendations to customers who speak under-represented languages.
- **Opaque decision-making processes:** As agents combine large language models with external systems, tracing the "why" behind their decisions can become challenging. A user using such an agent may find it difficult to understand why certain tools or combination of tools were chosen to answer a query, complicating trust and verification of the agent's outputs or actions.
- **Evolving best practices and standards:** Agents are an emerging technology, and guidance on safe integration, transparent tool usage, and responsible deployment continues to evolve. Keeping up with the latest best practices and auditing procedures is crucial, as even well-intentioned uses can become risky without ongoing review and refinement.

System performance

Best practices for improving system performance

- **Provide trusted data:** Retrieving or uploading untrusted data into your systems could compromise the security of your systems or applications. To mitigate these risks in your applications using the Azure AI Agent Service, we recommend logging and monitoring LLM interactions (inputs/outputs) to detect and analyze potential prompt injections, clearly delineating user input to minimize risk of prompt injection, restricting the LLM's access to sensitive resources, limiting its capabilities to the minimum required, and isolating it from critical systems and resources. Learn about additional mitigation approaches in Security guidance for Large Language Models.
- **Choose and integrate tools thoughtfully:** Select tools that are stable, well-documented, and suited to the agent's intended uses and objectives. For instance, use a reliable database connector for factual lookups or a well-tested API for executing specific actions. Limit the number of tools to those that genuinely enhance functionality and specify how and when the agent should use them.

- **Provide user proactive controls for system boundaries:** Consider creating user controls to give users operating the AI agent the ability to proactively set boundaries on what actions or tools are permitted, and what domains the agent can operate in.
- **Establish real-time oversight and human-in-the-loop processes:** Consider providing users with adequate real-time controls to authorize, verify, review, and approve agentic system behavior, including actions, planned tasks, operating environments or domain boundaries, and knowledge or action tool access. Particularly for critical or high-stakes tasks, consider incorporating mandatory human review and approval steps by the user. Ensure that a user or human operator can easily intervene, correct, or override the agent's decisions, especially when those decisions have safety or legal implications.
- **Ensure intelligibility and traceability for human decision-making**: Provide users with information before, during, and after actions are taken to help them understand justifications for decisions, identify where to intervene, and troubleshoot issues. Incorporate instrumentation or logging within the system, such as OpenTelemetry traces from Azure AI Agent Service, to trace outputs, including prompts, model steps, and tool calls. This enables reconstruction of the agent's reasoning process, isolation of issues, tuning of prompts, refinement of tool integration, and verification of guideline adherence.
- **Layer agent instructions and guidance:** Break down complex tasks into steps or sub-instructions within the system prompt. This can help the agent tackle multi-step reasoning more effectively, reducing errors and improving the clarity of the final output.
- **Recognize complexity thresholds for scaling:** When a single agent's system message consistently struggles to handle the complexity, breadth, or depth of a task—such as frequently producing incomplete results, hitting reasoning bottlenecks, or requiring extensive domain-specific knowledge—the system may benefit from transitioning to a multi-agent architecture. As a best practice, monitor performance indicators like response accuracy, latency, and error frequency. If refinements to the single agent's prompt no longer yield improved outcomes, consider decomposing the workload into specialized sub-tasks, each governed by its own agent. By segmenting complex tasks (e.g., splitting policy research and policy interpretation into separate agents), you can maintain modularity, leverage specialized domain knowledge more effectively, and reduce cognitive overload on any single agent.

Evaluating and integrating Azure AI Agent Service for your use

- **Map Agent risks and impacts.** Before developing or deploying your agentic application, carefully consider the impact of the intended actions and the consequences of actions or tool use not working as intended – such as generating or taking action on inaccurate information or causing biased or unfair outcomes – at different stages.
- **Ensure adequate human oversight and control.** Consider including controls to help users verify, review and/or approve actions in a timely manner, which may include reviewing planned tasks or calls to external data sources, for example, as appropriate for your system. Consider including controls for adequate user remediation of system failures, particularly in high-risk scenarios and use cases.
- **Clearly define actions and associated requirements.** Clearly defining which actions are allowed (action boundaries), prohibited, or need explicit authorization may help your agentic system operate as expected and with the appropriate level of human oversight.
- **Clearly define intended operating environments.** Clearly define the intended operating environments (domain boundaries) where your agent is designed to perform effectively.
- **Ensure appropriate intelligibility in decision making.** Providing information to users before, during, and after actions are taken and/or tools are called may help them understand action justification or why certain actions were taken or the application is behaving a certain way, where to intervene, and how to troubleshoot issues.
- **Provide trusted data.** Retrieving or uploading untrusted data into your systems could compromise the security of your systems or applications. To mitigate these risks in your applications using the Azure AI Agent Service, we recommend logging and monitoring LLM interactions (inputs/outputs) to detect and analyze potential prompt injections, clearly delineating user input to minimize risk of prompt injection, restricting the LLM's access to sensitive resources, limiting its capabilities to the minimum required, and isolating it from critical systems and resources. Learn about additional mitigation approaches in Security guidance for Large Language Models.
- Follow additional generative AI best practices as appropriate for your system, including recommendations in the Azure OpenAI Transparency Note.

Learn more about responsible AI

- Microsoft AI principles
- Microsoft responsible AI resources
- Microsoft Azure Learning courses on responsible AI

Learn more about Azure AI Agent Service

- Overview of Azure AI Agent Service
- Azure AI Agent Service QuickStart

Data, privacy, and security for Azure AI Agent Service

Article • 12/13/2024

Azure AI Agent Service is a fully managed service designed to empower developers to securely build, deploy, and scale high-quality, and extensible AI agents without needing to manage the underlying compute and storage resources. Azure AI Agent Service integrates **models, tools and technology** and enables you to extend agents with knowledge from connected sources (such as Bing Search, SharePoint, Fabric, Azure Blob storage, and licensed data) and actions using tools such as Azure Logic Apps, Azure Functions, OpenAPI 3.0 specified tools and Code Interpreter

NOTE: This article provides details regarding how data provided by you to the Azure AI Agent service is processed, used, and stored. Please also see the Microsoft Products and Services Data Protection Addendum, which governs data processing by the Azure AI Agent Service (but may not necessarily apply to external tools or services with which Azure AI Agent Service interacts, which are subject to their own data processing terms).

> **Important**
>
> Your prompts (inputs) and completions (outputs) and your data:
>
> - are NOT available to other customers.
> - are NOT available to OpenAI, Meta, Cohere, or Mistral.
> - are NOT used to improve OpenAI, Meta, Cohere, or Mistral models.

When you use Azure AI Agent Service with tools that retrieve data from external sources or services (such as the Grounding with Bing Search tool), the terms (including data processing terms) for those services apply to any data processed by those services. For example, the Grounding with Bing Search tool is subject to separate data collection and privacy terms (see Terms of Use for Grounding with Bing Search in Azure AI Agents Service), and the services are Microsoft-as-controller services and thereby excluded from the Microsoft Products and Services Data Protection Addendum.

What data does the Azure AI Agent Service process?

Azure AI Agent Service processes the following types of data:

- **Prompts and generated content**. Prompts are submitted by the user, and content is generated by the service, via the completions, chat completions, images and audio operations of models with which the Azure AI Agent Service interacts.
- **Uploaded data**. You can provide your own data for use with certain tools in Azure AI Agent service (e.g., File Search, Code Interpreter, Azure AI Search) using your own Azure Storage account or a configured data store.
- **External Data**. When you use the Grounding with Bing Search tool or tools that support function calling, the service processes and stores the outputs of these tools.

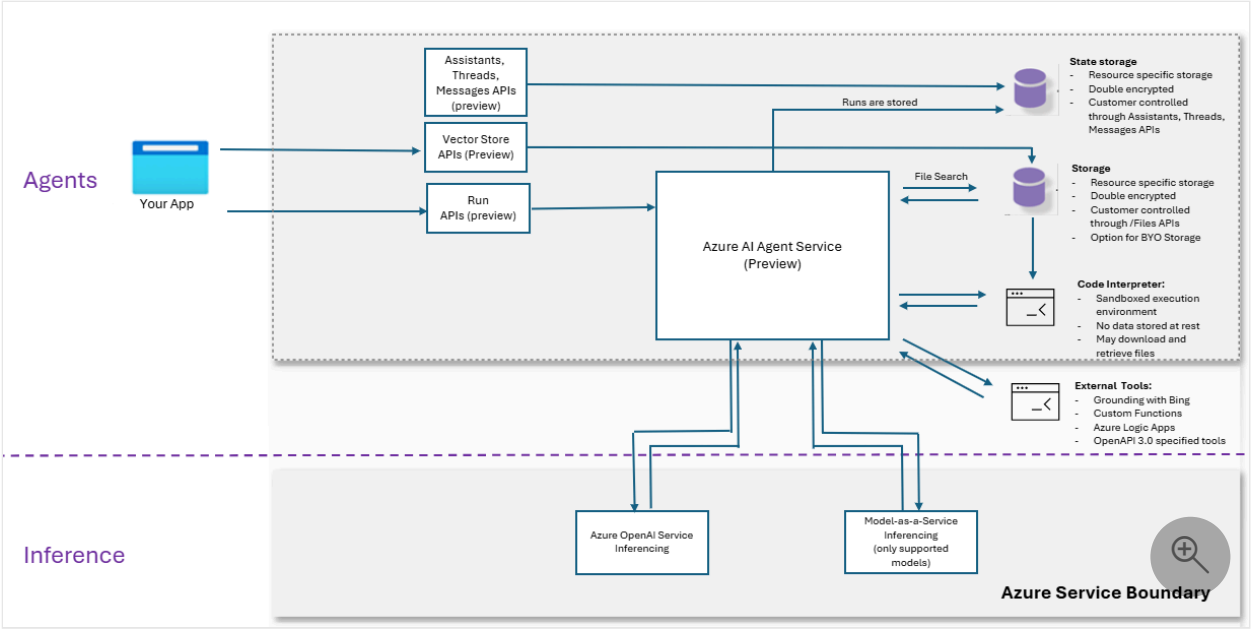
Data for stateful entities. When you use Threads, Messages and Runs, the service will create a data store to persist message history and other content, in accordance with how you configure the feature and in keeping with applicable privacy, security, and compliance commitments.

Augmented data included with or via prompts. When using data associated with stateful entities, the service retrieves relevant data from your configured data store and augments the prompt to produce generations that are grounded with your data. Prompts may also be augmented with data retrieved from a source included in the prompt itself, such as a URL.

How does the Azure AI Agent Service process data?

The diagram below illustrates how your data is processed. This diagram covers several types of processing:

1. How the Azure AI Agent Service processes your prompts via inferencing to generate content (including when additional data from a designated data source is added to a prompt using File Search, Code Interpreter, or other tools).
2. How the Azure AI Agent Service stores data in connection with Messages, Threads, and Runs.
3. How the Azure AI Agent Service processes data ingested into the service from external tools.



Model inferencing

Azure AI Agent Service interacts with the configured model inferencing endpoints you designate. Models (base or fine-tuned) process your input prompts and generate output responses which may be further used or processed by your agent. Data is processed for model inferencing in accordance with the terms that apply to the relevant model. Learn more at Data, privacy, and security for Azure OpenAI Service and Data, privacy, and security for use of models through the model catalog in AI Foundry portal.

Data storage for Azure AI Agent Service features

Some Azure AI Agent Service features store data in the service. This data is either uploaded by the customer or is automatically stored in connection with certain stateful entities such as Messages, Threads, and Runs. Data stored for Azure AI Agent Service:

- Is stored at rest in the Azure OpenAI resource which is created when you configure Azure AI Agent Service in the customer's Azure tenant, within the same geography as the Azure OpenAI resource.
- Can be double encrypted at rest, by default with Microsoft's AES-256 encryption and optionally with a customer managed key (except preview features may not support customer managed keys).
- Can be deleted by the customer at any time.

Location of data processing

To use the Azure AI Agent Service, you must create an Azure OpenAI Service resource in which the Azure AI Agent Service will be hosted. If you use features of the Azure AI Agent Service that store data at rest, that data will be stored in the geography where your Azure OpenAI resource is located. When the Azure AI Agent Service interacts with models and tools, the location of processing of data will depend on the configuration of the model inferencing endpoints and the hosting location of the tools.

Preventing harmful content generation

To reduce the risk of harmful use of the Azure AI Agent Service, the Service includes content filtering support. The outputs processed by the Azure AI Agent Service will be filtered in accordance with any content filtering that has been applied to the model deployment used by your agent.

See also

- Terms of Use for Grounding with Bing Search in Azure AI Agent Service
- Data Residency in Azure

Azure AI Projects client library for .NET - version 1.0.0-beta.8

Article • 04/23/2025

Use the AI Projects client library to:

- **Develop Agents using the Azure AI Agent Service**, leveraging an extensive ecosystem of models, tools, and capabilities from OpenAI, Microsoft, and other LLM providers. The Azure AI Agent Service enables the building of Agents for a wide range of generative AI use cases. The package is currently in preview.
- **Enumerate connections** in your Azure AI Foundry project and get connection properties. For example, get the inference endpoint URL and credentials associated with your Azure OpenAI connection.

Product documentation | Samples | API reference documentation | Package (NuGet) | SDK source code

Table of contents

- Getting started
  - Prerequisites
  - Install the package
- Key concepts
  - Create and authenticate the client
- Examples
  - Agents
    - Create an Agent
      - Create thread
      - Create message
      - Create and execute run
      - Retrieve messages
    - File search
    - Enterprise File Search
    - Code interpreter attachment
    - Create Agent with Bing Grounding
    - Azure AI Search
    - Function call
    - Azure function Call
    - OpenAPI
- Troubleshooting

- Next steps
- Contributing

Getting started

Prerequisites

To use Azure AI Projects capabilities, you must have an Azure subscription. This will allow you to create an Azure AI resource and get a connection URL.

Install the package

Install the client library for .NET with NuGet:

.NET CLI

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

Authenticate the client

A secure, keyless authentication approach is to use Microsoft Entra ID (formerly Azure Active Directory) via the Azure Identity library. To use this library, you need to install the Azure.Identity package:

.NET CLI

```
dotnet add package Azure.Identity
```

Key concepts

Create and authenticate the client

To interact with Azure AI Projects, you'll need to create an instance of `AIProjectClient`. Use the appropriate credential type from the Azure Identity library. For example, DefaultAzureCredential:

C#

```
var connectionString =
Environment.GetEnvironmentVariable("PROJECT_CONNECTION_STRING");
```

```
AIProjectClient projectClient = new AIProjectClient(connectionString, new
DefaultAzureCredential());
```

Once the `AIProjectClient` is created, you can call methods in the form of `GetXxxClient()` on this client to retrieve instances of specific sub-clients.

Examples

Agents

Agents in the Azure AI Projects client library are designed to facilitate various interactions and operations within your AI projects. They serve as the core components that manage and execute tasks, leveraging different tools and resources to achieve specific goals. The following steps outline the typical sequence for interacting with agents:

Create an Agent

First, you need to create an `AgentsClient`

C#

```
var connectionString =
System.Environment.GetEnvironmentVariable("PROJECT_CONNECTION_STRING");
var modelDeploymentName =
System.Environment.GetEnvironmentVariable("MODEL_DEPLOYMENT_NAME");
AgentsClient client = new(connectionString, new DefaultAzureCredential());
```

With an authenticated client, an agent can be created:

C#

```
Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "Math Tutor",
    instructions: "You are a personal math tutor. Write and run code to answer
math questions."
);
```

Create thread

Next, create a thread:

C#

```
AgentThread thread = await client.CreateThreadAsync();
```

Create message

With a thread created, messages can be created on it:

C#

```
ThreadMessage message = await client.CreateMessageAsync(
    thread.Id,
    MessageRole.User,
    "I need to solve the equation `3x + 11 = 14`. Can you help me?");
```

Create and execute run

A run can then be started that evaluates the thread against an agent:

C#

```
ThreadRun run = await client.CreateRunAsync(
    thread.Id,
    agent.Id,
    additionalInstructions: "Please address the user as Jane Doe. The user has a
premium account.");
```

Once the run has started, it should then be polled until it reaches a terminal status:

C#

```
do
{
    await Task.Delay(TimeSpan.FromMilliseconds(500));
    run = await client.GetRunAsync(thread.Id, run.Id);
}
while (run.Status == RunStatus.Queued
    || run.Status == RunStatus.InProgress);
Assert.AreEqual(
    RunStatus.Completed,
    run.Status,
    run.LastError?.Message);
```

Retrieve messages

Assuming the run successfully completed, listing messages from the thread that was run will now reflect new information added by the agent:

C#

```
PageableList<ThreadMessage> messages
    = await client.GetMessagesAsync(
        threadId: thread.Id, order: ListSortOrder.Ascending);

foreach (ThreadMessage threadMessage in messages)
{
    Console.Write($"{threadMessage.CreatedAt:yyyy-MM-dd HH:mm:ss} -
{threadMessage.Role,10}: ");
    foreach (MessageContent contentItem in threadMessage.ContentItems)
    {
        if (contentItem is MessageTextContent textItem)
        {
            Console.Write(textItem.Text);
        }
        else if (contentItem is MessageImageFileContent imageFileItem)
        {
            Console.Write($"<image from ID: {imageFileItem.FileId}");
        }
        Console.WriteLine();
    }
}
```

Example output from this sequence:

```
 2024-10-15 23:12:59 - assistant: Yes, Jane Doe, the solution to the equation \(3x
+ 11 = 14\) is \(x = 1\).
 2024-10-15 23:12:51 - user: I need to solve the equation `3x + 11 = 14`. Can you
help me?
```

File search

Files can be uploaded and then referenced by agents or messages. First, use the generalized upload API with a purpose of 'agents' to make a file ID available:

C#

```
// Upload a file and wait for it to be processed
File.WriteAllText(
    path: "sample_file_for_upload.txt",
    contents: "The word 'apple' uses the code 442345, while the word 'banana' uses
the code 673457.");
AgentFile uploadedAgentFile = await client.UploadFileAsync(
```

```
    filePath: "sample_file_for_upload.txt",
    purpose: AgentFilePurpose.Agents);
Dictionary<string, string> fileIds = new()
{
    { uploadedAgentFile.Id, uploadedAgentFile.Filename }
};
```

Once uploaded, the file ID can then be provided to create a vector store for it

C#

```
// Create a vector store with the file and wait for it to be processed.
// If you do not specify a vector store, create_message will create a vector store
with a default expiration policy of seven days after they were last active
VectorStore vectorStore = await client.CreateVectorStoreAsync(
    fileIds:  new List<string> { uploadedAgentFile.Id },
    name: "my_vector_store");
```

The vectorStore ID can then be provided to an agent upon creation. Note that file search will only be used if an appropriate tool like Code Interpreter is enabled. Also, you do not need to provide toolResources if you did not create a vector store above

C#

```
FileSearchToolResource fileSearchToolResource = new FileSearchToolResource();
fileSearchToolResource.VectorStoreIds.Add(vectorStore.Id);

// Create an agent with toolResources and process assistant run
Agent agent = await client.CreateAgentAsync(
        model: modelDeploymentName,
        name: "SDK Test Agent - Retrieval",
        instructions: "You are a helpful agent that can help fetch data from files
you know about.",
        tools: new List<ToolDefinition> { new FileSearchToolDefinition() },
        toolResources: new ToolResources() { FileSearch = fileSearchToolResource
});
```

With a file ID association and a supported tool enabled, the agent will then be able to consume the associated data when running threads.

Create Agent with Enterprise File Search

We can upload file to Azure as it is shown in the example, or use the existing Azure blob storage. In the code below we demonstrate how this can be achieved. First we upload file to azure and create `VectorStoreDataSource`, which then is used to create vector store. This vector store is then given to the `FileSearchTool` constructor.

C#

```
var ds = new VectorStoreDataSource(
    assetIdentifier: blobURI,
    assetType: VectorStoreDataSourceAssetType.UriAsset
);
VectorStore vectorStore = await client.CreateVectorStoreAsync(
    name: "sample_vector_store",
    storeConfiguration: new VectorStoreConfiguration(
        dataSources: [ ds ]
    )
);

FileSearchToolResource fileSearchResource = new([vectorStore.Id], null);

List<ToolDefinition> tools = [new FileSearchToolDefinition()];
Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "my-assistant",
    instructions: "You are helpful assistant.",
    tools: tools,
    toolResources: new ToolResources() { FileSearch = fileSearchResource }
);
```

We also can attach files to the existing vector store. In the code snippet below, we first create an empty vector store and add file to it.

C#

```
var ds = new VectorStoreDataSource(
    assetIdentifier: blobURI,
    assetType: VectorStoreDataSourceAssetType.UriAsset
);
VectorStore vectorStore = await client.CreateVectorStoreAsync(
    name: "sample_vector_store"
);

VectorStoreFileBatch vctFile = await client.CreateVectorStoreFileBatchAsync(
    vectorStoreId: vectorStore.Id,
    dataSources: [ ds ]
);
Console.WriteLine($"Created vector store file batch, vector store file batch ID:
{vctFile.Id}");

FileSearchToolResource fileSearchResource = new([vectorStore.Id], null);
```

Create Message with Code Interpreter Attachment

To attach a file with the context to the message, use the `MessageAttachment` class. To be able to process the attached file contents we need to provide the `List` with the single element

`CodeInterpreterToolDefinition` as a `tools` parameter to both `CreateAgent` method and `MessageAttachment` class constructor.

Here is an example to pass `CodeInterpreterTool` as tool:

C#

```
List<ToolDefinition> tools = [ new CodeInterpreterToolDefinition() ];
Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "my-assistant",
    instructions: "You are a helpful agent that can help fetch data from files you
know about.",
    tools: tools
);

File.WriteAllText(
    path: "sample_file_for_upload.txt",
    contents: "The word 'apple' uses the code 442345, while the word 'banana' uses
the code 673457.");
AgentFile uploadedAgentFile = await client.UploadFileAsync(
    filePath: "sample_file_for_upload.txt",
    purpose: AgentFilePurpose.Agents);
var fileId = uploadedAgentFile.Id;

var attachment = new MessageAttachment(
    fileId: fileId,
    tools: tools
);

AgentThread thread = await client.CreateThreadAsync();

ThreadMessage message = await client.CreateMessageAsync(
    threadId: thread.Id,
    role: MessageRole.User,
    content: "Can you give me the documented codes for 'banana' and 'orange'?",
    attachments: [ attachment ]
);
```

Azure blob storage can be used as a message attachment. In this case, use `VectorStoreDataSource` as a data source:

C#

```
var ds = new VectorStoreDataSource(
    assetIdentifier: blobURI,
    assetType: VectorStoreDataSourceAssetType.UriAsset
);

var attachment = new MessageAttachment(
    ds: ds,
```

```
    tools: tools
);
```

Create Agent with Bing Grounding

To enable your Agent to perform search through Bing search API, you use `BingGroundingTool` along with a connection.

Here is an example:

C#

```
ConnectionResponse bingConnection = await
projectClient.GetConnectionsClient().GetConnectionAsync(bingConnectionName);
var connectionId = bingConnection.Id;

ToolConnectionList connectionList = new()
{
    ConnectionList = { new ToolConnection(connectionId) }
};
BingGroundingToolDefinition bingGroundingTool = new(connectionList);
```

C#

```
Agent agent = await agentClient.CreateAgentAsync(
   model: modelDeploymentName,
   name: "my-assistant",
   instructions: "You are a helpful assistant.",
   tools: [ bingGroundingTool ]);
```

Create Agent with Azure AI Search

Azure AI Search is an enterprise search system for high-performance applications. It integrates with Azure OpenAI Service and Azure Machine Learning, offering advanced search technologies like vector search and full-text search. Ideal for knowledge base insights, information discovery, and automation. Creating an Agent with Azure AI Search requires an existing Azure AI Search Index. For more information and setup guides, see Azure AI Search Tool Guide.

C#

```
ListConnectionsResponse connections = await
projectClient.GetConnectionsClient().GetConnectionsAsync(ConnectionType.AzureAISea
rch).ConfigureAwait(false);
```

```
if (connections?.Value == null || connections.Value.Count == 0)
{
    throw new InvalidOperationException("No connections found for the Azure AI
Search.");
}

ConnectionResponse connection = connections.Value[0];

AzureAISearchResource searchResource = new(
    connection.Id,
    "sample_index",
    5,
    "category eq 'sleeping bag'",
    AzureAISearchQueryType.Simple
);
ToolResources toolResource = new()
{
    AzureAISearch = searchResource
};

AgentsClient agentClient = projectClient.GetAgentsClient();

Agent agent = await agentClient.CreateAgentAsync(
   model: modelDeploymentName,
   name: "my-assistant",
   instructions: "You are a helpful assistant.",
   tools: [ new AzureAISearchToolDefinition() ],
   toolResources: toolResource);
```

If the agent has found the relevant information in the index, the reference and annotation will be provided in the message response. In the example above, we replace the reference placeholder by the actual reference and url. Please note, that to get sensible result, the index needs to have fields "title" and "url".

C#

```
PageableList<ThreadMessage> messages = await agentClient.GetMessagesAsync(
    threadId: thread.Id,
    order: ListSortOrder.Ascending
);

foreach (ThreadMessage threadMessage in messages)
{
    Console.Write($"{threadMessage.CreatedAt:yyyy-MM-dd HH:mm:ss} -
{threadMessage.Role,10}: ");
    foreach (MessageContent contentItem in threadMessage.ContentItems)
    {
        if (contentItem is MessageTextContent textItem)
        {
            // We need to annotate only Agent messages.
            if (threadMessage.Role == MessageRole.Agent &&
textItem.Annotations.Count > 0)
```

```
            {
                string annotatedText = textItem.Text;
                foreach (MessageTextAnnotation annotation in textItem.Annotations)
                {
                    if (annotation is MessageTextUrlCitationAnnotation
urlAnnotation)
                    {
                        annotatedText = annotatedText.Replace(
                            urlAnnotation.Text,
                            $" [see {urlAnnotation.UrlCitation.Title}]
({urlAnnotation.UrlCitation.Url})");
                    }
                }
                Console.Write(annotatedText);
            }
            else
            {
                Console.Write(textItem.Text);
            }
        }
        else if (contentItem is MessageImageFileContent imageFileItem)
        {
            Console.Write($"<image from ID: {imageFileItem.FileId}");
        }
        Console.WriteLine();
    }
}
```

Function call

Tools that reference caller-defined capabilities as functions can be provided to an agent to allow it to dynamically resolve and disambiguate during a run.

Here, outlined is a simple agent that "knows how to," via caller-provided functions:

1. Get the user's favorite city
2. Get a nickname for a given city
3. Get the current weather, optionally with a temperature unit, in a city

To do this, begin by defining the functions to use -- the actual implementations here are merely representative stubs.

C#

```
// Example of a function that defines no parameters
string GetUserFavoriteCity() => "Seattle, WA";
FunctionToolDefinition getUserFavoriteCityTool = new("getUserFavoriteCity", "Gets
the user's favorite city.");
// Example of a function with a single required parameter
string GetCityNickname(string location) => location switch
```

```
{
    "Seattle, WA" => "The Emerald City",
    _ => throw new NotImplementedException(),
};
FunctionToolDefinition getCityNicknameTool = new(
    name: "getCityNickname",
    description: "Gets the nickname of a city, e.g. 'LA' for 'Los Angeles, CA'.",
    parameters: BinaryData.FromObjectAsJson(
        new
        {
            Type = "object",
            Properties = new
            {
                Location = new
                {
                    Type = "string",
                    Description = "The city and state, e.g. San Francisco, CA",
                },
            },
            Required = new[] { "location" },
        },
        new JsonSerializerOptions() { PropertyNamingPolicy =
JsonNamingPolicy.CamelCase }));
// Example of a function with one required and one optional, enum parameter
string GetWeatherAtLocation(string location, string temperatureUnit = "f") =>
location switch
{
    "Seattle, WA" => temperatureUnit == "f" ? "70f" : "21c",
    _ => throw new NotImplementedException()
};
FunctionToolDefinition getCurrentWeatherAtLocationTool = new(
    name: "getCurrentWeatherAtLocation",
    description: "Gets the current weather at a provided location.",
    parameters: BinaryData.FromObjectAsJson(
        new
        {
            Type = "object",
            Properties = new
            {
                Location = new
                {
                    Type = "string",
                    Description = "The city and state, e.g. San Francisco, CA",
                },
                Unit = new
                {
                    Type = "string",
                    Enum = new[] { "c", "f" },
                },
            },
            Required = new[] { "location" },
        },
        new JsonSerializerOptions() { PropertyNamingPolicy =
JsonNamingPolicy.CamelCase }));
```

With the functions defined in their appropriate tools, an agent can be now created that has those tools enabled:

C#

```
// note: parallel function calling is only supported with newer models like gpt-4-1106-preview
Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "SDK Test Agent - Functions",
        instructions: "You are a weather bot. Use the provided functions to help answer questions. "
            + "Customize your responses to the user's preferences as much as possible and use friendly "
            + "nicknames for cities whenever possible.",
    tools: [ getUserFavoriteCityTool, getCityNicknameTool, getCurrentWeatherAtLocationTool ]
    );
```

If the agent calls tools, the calling code will need to resolve `ToolCall` instances into matching `ToolOutput` instances. For convenience, a basic example is extracted here:

C#

```
ToolOutput GetResolvedToolOutput(RequiredToolCall toolCall)
{
    if (toolCall is RequiredFunctionToolCall functionToolCall)
    {
        if (functionToolCall.Name == getUserFavoriteCityTool.Name)
        {
            return new ToolOutput(toolCall, GetUserFavoriteCity());
        }
        using JsonDocument argumentsJson = JsonDocument.Parse(functionToolCall.Arguments);
        if (functionToolCall.Name == getCityNicknameTool.Name)
        {
            string locationArgument = argumentsJson.RootElement.GetProperty("location").GetString();
            return new ToolOutput(toolCall, GetCityNickname(locationArgument));
        }
        if (functionToolCall.Name == getCurrentWeatherAtLocationTool.Name)
        {
            string locationArgument = argumentsJson.RootElement.GetProperty("location").GetString();
            if (argumentsJson.RootElement.TryGetProperty("unit", out JsonElement unitElement))
            {
                string unitArgument = unitElement.GetString();
                return new ToolOutput(toolCall, GetWeatherAtLocation(locationArgument, unitArgument));
            }
            return new ToolOutput(toolCall,
```

```
GetWeatherAtLocation(locationArgument));
        }
    }
    return null;
}
```

To handle user input like "what's the weather like right now in my favorite city?", polling the response for completion should be supplemented by a `RunStatus` check for `RequiresAction` or, in this case, the presence of the `RequiredAction` property on the run. Then, the collection of `ToolOutputSubmissions` should be submitted to the run via the `SubmitRunToolOutputs` method so that the run can continue:

C#

```
do
{
    await Task.Delay(TimeSpan.FromMilliseconds(500));
    run = await client.GetRunAsync(thread.Id, run.Id);

    if (run.Status == RunStatus.RequiresAction
        && run.RequiredAction is SubmitToolOutputsAction submitToolOutputsAction)
    {
        List<ToolOutput> toolOutputs = [];
        foreach (RequiredToolCall toolCall in submitToolOutputsAction.ToolCalls)
        {
            toolOutputs.Add(GetResolvedToolOutput(toolCall));
        }
        run = await client.SubmitToolOutputsToRunAsync(run, toolOutputs);
    }
}
while (run.Status == RunStatus.Queued
    || run.Status == RunStatus.InProgress);
Assert.AreEqual(
    RunStatus.Completed,
    run.Status,
    run.LastError?.Message);
```

Calling function with streaming requires small modification of the code above. Streaming updates contain one ToolOutput per update and now the GetResolvedToolOutput function will look like it is shown on the code snippet below:

C#

```
ToolOutput GetResolvedToolOutput(string functionName, string toolCallId, string
functionArguments)
{
    if (functionName == getUserFavoriteCityTool.Name)
    {
        return new ToolOutput(toolCallId, GetUserFavoriteCity());
```

```
    }
    using JsonDocument argumentsJson = JsonDocument.Parse(functionArguments);
    if (functionName == getCityNicknameTool.Name)
    {
        string locationArgument =
argumentsJson.RootElement.GetProperty("location").GetString();
        return new ToolOutput(toolCallId, GetCityNickname(locationArgument));
    }
    if (functionName == getCurrentWeatherAtLocationTool.Name)
    {
        string locationArgument =
argumentsJson.RootElement.GetProperty("location").GetString();
        if (argumentsJson.RootElement.TryGetProperty("unit", out JsonElement
unitElement))
        {
            string unitArgument = unitElement.GetString();
            return new ToolOutput(toolCallId,
GetWeatherAtLocation(locationArgument, unitArgument));
        }
        return new ToolOutput(toolCallId, GetWeatherAtLocation(locationArgument));
    }
    return null;
}
```

We create a stream and wait for the stream update of the `RequiredActionUpdate` type. This update will mark the point, when we need to submit tool outputs to the stream. We will submit outputs in the inner cycle. Please note that `RequiredActionUpdate` keeps only one required action, while our run may require multiple function calls, this case is handled in the inner cycle, so that we can add tool output to the existing array of outputs. After all required actions were submitted we clean up the array of required actions.

C#

```
List<ToolOutput> toolOutputs = [];
ThreadRun streamRun = null;
AsyncCollectionResult<StreamingUpdate> stream =
client.CreateRunStreamingAsync(thread.Id, agent.Id);
do
{
    toolOutputs.Clear();
    await foreach (StreamingUpdate streamingUpdate in stream)
    {
        if (streamingUpdate.UpdateKind == StreamingUpdateReason.RunCreated)
        {
            Console.WriteLine("--- Run started! ---");
        }
        else if (streamingUpdate is RequiredActionUpdate submitToolOutputsUpdate)
        {
            RequiredActionUpdate newActionUpdate = submitToolOutputsUpdate;
            toolOutputs.Add(
                GetResolvedToolOutput(
```

```
                    newActionUpdate.FunctionName,
                    newActionUpdate.ToolCallId,
                    newActionUpdate.FunctionArguments
            ));
            streamRun = submitToolOutputsUpdate.Value;
        }
        else if (streamingUpdate is MessageContentUpdate contentUpdate)
        {
            Console.Write(contentUpdate.Text);
        }
        else if (streamingUpdate.UpdateKind == StreamingUpdateReason.RunCompleted)
        {
            Console.WriteLine();
            Console.WriteLine("--- Run completed! ---");
        }
    }
    if (toolOutputs.Count > 0)
    {
        stream = client.SubmitToolOutputsToStreamAsync(streamRun, toolOutputs);
    }
}
while (toolOutputs.Count > 0);
```

Azure function call

We can use Azure Function from inside the agent. In the example below we are calling function "foo", which responds "Bar". In this example we create `AzureFunctionToolDefinition` object, with the function name, description, input and output queues, followed by function parameters. See below for the instructions on function deployment.

C#

```
var connectionString =
System.Environment.GetEnvironmentVariable("PROJECT_CONNECTION_STRING");
var modelDeploymentName =
System.Environment.GetEnvironmentVariable("MODEL_DEPLOYMENT_NAME");
var storageQueueUri =
System.Environment.GetEnvironmentVariable("STORAGE_QUEUE_URI");

AgentsClient client = new(connectionString, new DefaultAzureCredential());

AzureFunctionToolDefinition azureFnTool = new(
    name: "foo",
    description: "Get answers from the foo bot.",
    inputBinding: new AzureFunctionBinding(
        new AzureFunctionStorageQueue(
            queueName: "azure-function-foo-input",
            storageServiceEndpoint: storageQueueUri
        )
    ),
    outputBinding: new AzureFunctionBinding(
```

```
        new AzureFunctionStorageQueue(
            queueName: "azure-function-tool-output",
            storageServiceEndpoint: storageQueueUri
        )
    ),
    parameters: BinaryData.FromObjectAsJson(
            new
            {
                Type = "object",
                Properties = new
                {
                    query = new
                    {
                        Type = "string",
                        Description = "The question to ask.",
                    },
                    outputqueueuri = new
                    {
                        Type = "string",
                        Description = "The full output queue uri."
                    }
                },
            },
        new JsonSerializerOptions() { PropertyNamingPolicy =
JsonNamingPolicy.CamelCase }
    )
);
```

Note that in this scenario we are asking agent to supply storage queue URI to the azure function whenever it is called.

C#

```
Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "azure-function-agent-foo",
        instructions: "You are a helpful support agent. Use the provided function
any "
        + "time the prompt contains the string 'What would foo say?'. When you
invoke "
        + "the function, ALWAYS specify the output queue uri parameter as "
        + $"'{storageQueueUri}/azure-function-tool-output'. Always responds with "
        + "\"Foo says\" and then the response from the tool.",
    tools: [ azureFnTool ]
    );
```

After we have created a message with request to ask "What would foo say?", we need to wait while the run is in queued, in progress or requires action states.

C#

```
AgentThread thread = await client.CreateThreadAsync();

ThreadMessage message = await client.CreateMessageAsync(
    thread.Id,
    MessageRole.User,
    "What is the most prevalent element in the universe? What would foo say?");

ThreadRun run = await client.CreateRunAsync(thread, agent);

do
{
    await Task.Delay(TimeSpan.FromMilliseconds(500));
    run = await client.GetRunAsync(thread.Id, run.Id);
}
while (run.Status == RunStatus.Queued
    || run.Status == RunStatus.InProgress
    || run.Status == RunStatus.RequiresAction);
Assert.AreEqual(
    RunStatus.Completed,
    run.Status,
    run.LastError?.Message);
```

To make a function call we need to create and deploy the Azure function. In the code snippet below, we have an example of function on C# which can be used by the code above.

C#

```
namespace FunctionProj
{
    public class Response
    {
        public required string Value { get; set; }
        public required string CorrelationId { get; set; }
    }

    public class Arguments
    {
        public required string OutputQueueUri { get; set; }
        public required string CorrelationId { get; set; }
    }

    public class Foo
    {
        private readonly ILogger<Foo> _logger;

        public Foo(ILogger<Foo> logger)
        {
            _logger = logger;
        }

        [Function("Foo")]
        public void Run([QueueTrigger("azure-function-foo-input")] Arguments
```

```
input, FunctionContext executionContext)
        {
            var logger = executionContext.GetLogger("Foo");
            logger.LogInformation("C# Queue function processed a request.");

            // We have to provide the Managed identity for function resource
            // and allow this identity a Queue Data Contributor role on the
storage account.
            var cred = new DefaultAzureCredential();
            var queueClient = new QueueClient(new Uri(input.OutputQueueUri), cred,
                    new QueueClientOptions { MessageEncoding =
QueueMessageEncoding.Base64 });

            var response = new Response
            {
                Value = "Bar",
                // Important! Correlation ID must match the input correlation ID.
                CorrelationId = input.CorrelationId
            };

            var jsonResponse = JsonSerializer.Serialize(response);
            queueClient.SendMessage(jsonResponse);
        }
    }
}
```

In this code we define function input and output class: `Arguments` and `Response` respectively. These two data classes will be serialized in JSON. It is important that these both contain field `CorrelationId`, which is the same between input and output.

In our example the function will be stored in the storage account, created with the AI hub. For that we need to allow key access to that storage. In Azure portal go to Storage account > Settings > Configuration and set "Allow storage account key access" to Enabled. If it is not done, the error will be displayed "The remote server returned an error: (403) Forbidden." To create the function resource that will host our function, install azure-cli python package and run the next command:

shell

```
pip install -U azure-cli
az login
az functionapp create --resource-group your-resource-group --consumption-plan-
location region --runtime dotnet-isolated --functions-version 4 --name
function_name --storage-account storage_account_already_present_in_resource_group
--app-insights existing_or_new_application_insights_name
```

This function writes data to the output queue and hence needs to be authenticated to Azure, so we will need to assign the function system identity and provide it `Storage Queue Data`

`Contributor`. To do that in Azure portal select the function, located in `your-resource-group` resource group and in Settings>Identity, switch it on and click Save. After that assign the `Storage Queue Data Contributor` permission on storage account used by our function (`storage_account_already_present_in_resource_group` in the script above) for just assigned System Managed identity.

Now we will create the function itself. Install .NET and Core Tools and create the function project using next commands.

```
func init FunctionProj --worker-runtime dotnet-isolated --target-framework net8.0
cd FunctionProj
func new --name foo --template "HTTP trigger" --authlevel "anonymous"
dotnet add package Azure.Identity
dotnet add package Microsoft.Azure.Functions.Worker.Extensions.Storage.Queues --
prerelease
```

Note: There is a "Azure Queue Storage trigger", however the attempt to use it results in error for now. We have created a project, containing HTTP-triggered azure function with the logic in `Foo.cs` file. As far as we need to trigger Azure function by a new message in the queue, we will replace the content of a Foo.cs by the C# sample code above. To deploy the function run the command from dotnet project folder:

```
func azure functionapp publish function_name
```

In the `storage_account_already_present_in_resource_group` select the `Queue service` and create two queues: `azure-function-foo-input` and `azure-function-tool-output`. Note that the same queues are used in our sample. To check that the function is working, place the next message into the `azure-function-foo-input` and replace `storage_account_already_present_in_resource_group` by the actual resource group name, or just copy the output queue address.

JSON

```
{
  "OutputQueueUri":
"https://storage_account_already_present_in_resource_group.queue.core.windows.net/
azure-function-tool-output",
  "CorrelationId": "42"
}
```

Next, we will monitor the output queue or the message. You should receive the next message.

JSON

```
{
  "Value": "Bar",
  "CorrelationId": "42"
}
```

Please note that the input `CorrelationId` is the same as output. *Hint:* Place multiple messages to input queue and keep second internet browser window with the output queue open and hit the refresh button on the portal user interface, so that you will not miss the message. If the message instead went to `azure-function-foo-input-poison` queue, the function completed with error, please check your setup. After we have tested the function and made sure it works, please make sure that the Azure AI Project have the next roles for the storage account: `Storage Account Contributor`, `Storage Blob Data Contributor`, `Storage File Data Privileged Contributor`, `Storage Queue Data Contributor` and `Storage Table Data Contributor`. Now the function is ready to be used by the agent.

Create Agent With OpenAPI

OpenAPI specifications describe REST operations against a specific endpoint. Agents SDK can read an OpenAPI spec, create a function from it, and call that function against the REST endpoint without additional client-side execution.

Here is an example creating an OpenAPI tool (using anonymous authentication):

C#

```
OpenApiAnonymousAuthDetails oaiAuth = new();
OpenApiToolDefinition openapiTool = new(
    name: "get_weather",
    description: "Retrieve weather information for a location",
    spec: BinaryData.FromBytes(File.ReadAllBytes(file_path)),
    auth: oaiAuth,
    defaultParams: ["format"]
);

Agent agent = await client.CreateAgentAsync(
    model: modelDeploymentName,
    name: "azure-function-agent-foo",
    instructions: "You are a helpful assistant.",
    tools: [ openapiTool ]
);
```

In this example we are using the `weather_openapi.json` file and agent will request the wttr.in website for the weather in a location fron the prompt.

C#

```
AgentThread thread = await client.CreateThreadAsync();
ThreadMessage message = await client.CreateMessageAsync(
    thread.Id,
    MessageRole.User,
    "What's the weather in Seattle?");

ThreadRun run = await client.CreateRunAsync(thread, agent);

do
{
    await Task.Delay(TimeSpan.FromMilliseconds(500));
    run = await client.GetRunAsync(thread.Id, run.Id);
}
while (run.Status == RunStatus.Queued
    || run.Status == RunStatus.InProgress
    || run.Status == RunStatus.RequiresAction);
Assert.AreEqual(
    RunStatus.Completed,
    run.Status,
    run.LastError?.Message);
```

Troubleshooting

Any operation that fails will throw a RequestFailedException. The exception's `code` will hold the HTTP response status code. The exception's `message` contains a detailed message that may be helpful in diagnosing the issue:

C#

```
try
{
    client.CreateMessage(
    "thread1234",
    MessageRole.User,
    "I need to solve the equation `3x + 11 = 14`. Can you help me?");
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    Console.WriteLine($"Exception status code: {ex.Status}");
    Console.WriteLine($"Exception message: {ex.Message}");
}
```

To further diagnose and troubleshoot issues, you can enable logging following the Azure SDK logging documentation. This allows you to capture additional insights into request and response details, which can be particularly helpful when diagnosing complex issues.

Next steps

Beyond the introductory scenarios discussed, the AI Projects client library offers support for additional scenarios to help take advantage of the full feature set of the AI services. In order to help explore some of these scenarios, the AI Projects client library offers a set of samples to serve as an illustration for common scenarios. Please see the Samples for details.

Contributing

See the Azure SDK CONTRIBUTING.md for details on building, testing, and contributing to this library.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit cla.microsoft.com.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Azure AI Projects client library for Python - version 1.0.0b10

Article • 04/23/2025

Use the AI Projects client library (in preview) to:

- **Enumerate connections** in your Azure AI Foundry project and get connection properties. For example, get the inference endpoint URL and credentials associated with your Azure OpenAI connection.
- **Get an authenticated Inference client** to do chat completions, for the default Azure OpenAI or AI Services connections in your Azure AI Foundry project. Supports the AzureOpenAI client from the `openai` package, or clients from the `azure-ai-inference` package.
- **Develop Agents using the Azure AI Agent Service**, leveraging an extensive ecosystem of models, tools, and capabilities from OpenAI, Microsoft, and other LLM providers. The Azure AI Agent Service enables the building of Agents for a wide range of generative AI use cases. The package is currently in preview.
- **Run Evaluations** to assess the performance of generative AI applications using various evaluators and metrics. It includes built-in evaluators for quality, risk, and safety, and allows custom evaluators for specific needs.
- **Enable OpenTelemetry tracing**.

Product documentation | Samples | API reference documentation | Package (PyPI) | SDK source code | AI Starter Template

Reporting issues

To report an issue with the client library, or request additional features, please open a GitHub issue here. Mention the package name "azure-ai-projects" in the title or content.

Table of contents

- Getting started
  - Prerequisite
  - Install the package
- Key concepts
  - Create and authenticate the client
- Examples
  - Enumerate connections
    - Get properties of all connections

        - Get properties of all connections of a particular type
        - Get properties of a default connection
        - Get properties of a connection by its connection name
    - Get an authenticated ChatCompletionsClient
    - Get an authenticated AzureOpenAI client
    - Agents (Preview)
        - Create an Agent with:
            - File Search
            - Enterprise File Search
            - Code interpreter
            - Bing grounding
            - Azure AI Search
            - Function call
            - Azure Function Call
            - OpenAPI
            - Fabric data
        - Create thread with
            - Tool resource
        - Create message with:
            - File search attachment
            - Code interpreter attachment
            - Create Message with Image Inputs
        - Execute Run, Run_and_Process, or Stream
        - Retrieve message
        - Retrieve file
        - Tear down by deleting resource
        - Tracing
    - Evaluation
        - Evaluator
        - Run Evaluation in the cloud
            - Evaluators
            - Data to be evaluated
                - [Optional] Azure OpenAI Model
                - Example Remote Evaluation
    - Tracing
        - Installation
        - How to enable tracing
        - How to trace your own functions
- Troubleshooting
    - Exceptions
    - Logging

    - Reporting issues
- Next steps
- Contributing

Getting started

Prerequisite

- Python 3.8 or later.
- An Azure subscription.
- A project in Azure AI Foundry.
- The project connection string. It can be found in your Azure AI Foundry project overview page, under "Project details". Below we will assume the environment variable `PROJECT_CONNECTION_STRING` was defined to hold this value.
- Entra ID is needed to authenticate the client. Your application needs an object that implements the TokenCredential interface. Code samples here use DefaultAzureCredential. To get that working, you will need:
    - An appropriate role assignment. see Role-based access control in Azure AI Foundry portal. Role assigned can be done via the "Access Control (IAM)" tab of your Azure AI Project resource in the Azure portal.
    - Azure CLI installed.
    - You are logged into your Azure account by running `az login`.
    - Note that if you have multiple Azure subscriptions, the subscription that contains your Azure AI Project resource must be your default subscription. Run `az account list --output table` to list all your subscription and see which one is the default. Run `az account set --subscription "Your Subscription ID or Name"` to change your default subscription.

Install the package

Bash

```
pip install azure-ai-projects
```

Key concepts

Create and authenticate the client

The class factory method `from_connection_string` is used to construct the client. To construct a synchronous client:

Python

```python
import os
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential

project_client = AIProjectClient.from_connection_string(
    credential=DefaultAzureCredential(),
    conn_str=os.environ["PROJECT_CONNECTION_STRING"],
)
```

To construct an asynchronous client, Install the additional package aiohttp:

Bash

```
pip install aiohttp
```

and update the code above to import `asyncio`, and import `AIProjectClient` from the `azure.ai.projects.aio` namespace:

Python

```python
import os
import asyncio
from azure.ai.projects.aio import AIProjectClient
from azure.core.credentials import AzureKeyCredential

project_client = AIProjectClient.from_connection_string(
    credential=DefaultAzureCredential(),
    conn_str=os.environ["PROJECT_CONNECTION_STRING"],
)
```

Examples

Enumerate connections

Your Azure AI Foundry project has a "Management center". When you enter it, you will see a tab named "Connected resources" under your project. The `.connections` operations on the client allow you to enumerate the connections and get connection properties. Connection properties include the resource URL and authentication credentials, among other things.

Below are code examples of the connection operations. Full samples can be found under the "connetions" folder in the package samples.

Get properties of all connections

To list the properties of all the connections in the Azure AI Foundry project:

Python

```
connections = project_client.connections.list()
for connection in connections:
    print(connection)
```

Get properties of all connections of a particular type

To list the properties of connections of a certain type (here Azure OpenAI):

Python

```
connections = project_client.connections.list(
    connection_type=ConnectionType.AZURE_OPEN_AI,
)
for connection in connections:
    print(connection)
```

Get properties of a default connection

To get the properties of the default connection of a certain type (here Azure OpenAI), with its authentication credentials:

Python

```
connection = project_client.connections.get_default(
    connection_type=ConnectionType.AZURE_OPEN_AI,
    include_credentials=True,  # Optional. Defaults to "False".
)
print(connection)
```

If the call was made with `include_credentials=True`, depending on the value of `connection.authentication_type`, either `connection.key` or `connection.token_credential` will be populated. Otherwise both will be `None`.

Get properties of a connection by its connection name

To get the connection properties of a connection named `connection_name`:

Python

```
connection = project_client.connections.get(
    connection_name=connection_name,
    include_credentials=True  # Optional. Defaults to "False"
)
print(connection)
```

Get an authenticated ChatCompletionsClient

Your Azure AI Foundry project may have one or more AI models deployed that support chat completions. These could be OpenAI models, Microsoft models, or models from other providers. Use the code below to get an already authenticated ChatCompletionsClient from the azure-ai-inference package, and execute a chat completions call.

First, install the package:

Bash

```
pip install azure-ai-inference
```

Then run this code (replace "gpt-4o" with your model deployment name):

Python

```
inference_client = project_client.inference.get_chat_completions_client()

response = inference_client.complete(
    model="gpt-4o", # Model deployment name
    messages=[UserMessage(content="How many feet are in a mile?")]
)

print(response.choices[0].message.content)
```

See the "inference" folder in the package samples for additional samples, including getting an authenticated EmbeddingsClient and ImageEmbeddingsClient.

Get an authenticated AzureOpenAI client

Your Azure AI Foundry project may have one or more OpenAI models deployed that support chat completions. Use the code below to get an already authenticated AzureOpenAI from

the openai package, and execute a chat completions call.

First, install the package:

```bash
pip install openai
```

Then run the code below. Replace `gpt-4o` with your model deployment name, and update the `api_version` value with one found in the "Data plane - inference" row in this table.

```python
aoai_client = project_client.inference.get_azure_openai_client(api_version="2024-06-01")

response = aoai_client.chat.completions.create(
    model="gpt-4o", # Model deployment name
    messages=[
        {
            "role": "user",
            "content": "How many feet are in a mile?",
        },
    ],
)

print(response.choices[0].message.content)
```

See the "inference" folder in the package samples for additional samples.

Agents (Preview)

Agents in the Azure AI Projects client library are designed to facilitate various interactions and operations within your AI projects. They serve as the core components that manage and execute tasks, leveraging different tools and resources to achieve specific goals. The following steps outline the typical sequence for interacting with Agents. See the "agents" folder in the package samples for additional Agent samples.

Create Agent

Before creating an Agent, you need to set up Azure resources to deploy your model. Create a New Agent Quickstart details selecting and deploying your Agent Setup.

Here is an example of how to create an Agent:

Python

```
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are helpful assistant",
)
```

To allow Agents to access your resources or custom functions, you need tools. You can pass tools to `create_agent` by either `toolset` or combination of `tools` and `tool_resources`.

Here is an example of `toolset`:

Python

```
functions = FunctionTool(user_functions)
code_interpreter = CodeInterpreterTool()

toolset = ToolSet()
toolset.add(functions)
toolset.add(code_interpreter)

# To enable tool calls executed automatically
project_client.agents.enable_auto_function_calls(toolset=toolset)

agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are a helpful assistant",
    toolset=toolset,
)
```

Also notice that if you use the asynchronous client, use `AsyncToolSet` instead. Additional information related to `AsyncFunctionTool` be discussed in the later sections.

Here is an example to use `tools` and `tool_resources`:

Python

```
file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])

# Notices that FileSearchTool as tool and tool_resources must be added or the
assistant unable to search the file
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are helpful assistant",
    tools=file_search_tool.definitions,
```

```
    tool_resources=file_search_tool.resources,
)
```

In the following sections, we show you sample code in either `toolset` or combination of `tools` and `tool_resources`.

Create Agent with File Search

To perform file search by an Agent, we first need to upload a file, create a vector store, and associate the file to the vector store. Here is an example:

Python

```
file = project_client.agents.upload_file_and_poll(file_path="product_info_1.md",
purpose="assistants")
print(f"Uploaded file, file ID: {file.id}")

vector_store = project_client.agents.create_vector_store_and_poll(file_ids=
[file.id], name="my_vectorstore")
print(f"Created vector store, vector store ID: {vector_store.id}")

# Create file search tool with resources followed by creating agent
file_search = FileSearchTool(vector_store_ids=[vector_store.id])

agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="Hello, you are helpful assistant and can search information from
uploaded files",
    tools=file_search.definitions,
    tool_resources=file_search.resources,
)
```

Create Agent with Enterprise File Search

We can upload file to Azure as it is shown in the example, or use the existing Azure blob storage. In the code below we demonstrate how this can be achieved. First we upload file to azure and create `VectorStoreDataSource`, which then is used to create vector store. This vector store is then given to the `FileSearchTool` constructor.

Python

```
# We will upload the local file to Azure and will use it for vector store
creation.
_, asset_uri = project_client.upload_file("./product_info_1.md")

# Create a vector store with no file and wait for it to be processed
```

```
ds = VectorStoreDataSource(asset_identifier=asset_uri,
asset_type=VectorStoreDataSourceAssetType.URI_ASSET)
vector_store = project_client.agents.create_vector_store_and_poll(data_sources=
[ds], name="sample_vector_store")
print(f"Created vector store, vector store ID: {vector_store.id}")

# Create a file search tool
file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])

# Notices that FileSearchTool as tool and tool_resources must be added or the
assistant unable to search the file
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are helpful assistant",
    tools=file_search_tool.definitions,
    tool_resources=file_search_tool.resources,
)
```

We also can attach files to the existing vector store. In the code snippet below, we first create an empty vector store and add file to it.

Python

```
# Create a vector store with no file and wait for it to be processed
vector_store = project_client.agents.create_vector_store_and_poll(data_sources=[],
name="sample_vector_store")
print(f"Created vector store, vector store ID: {vector_store.id}")

ds = VectorStoreDataSource(asset_identifier=asset_uri,
asset_type=VectorStoreDataSourceAssetType.URI_ASSET)
# Add the file to the vector store or you can supply data sources in the vector
store creation
vector_store_file_batch =
project_client.agents.create_vector_store_file_batch_and_poll(
    vector_store_id=vector_store.id, data_sources=[ds]
)
print(f"Created vector store file batch, vector store file batch ID:
{vector_store_file_batch.id}")

# Create a file search tool
file_search_tool = FileSearchTool(vector_store_ids=[vector_store.id])
```

Create Agent with Code Interpreter

Here is an example to upload a file and use it for code interpreter by an Agent:

Python

```
file = project_client.agents.upload_file_and_poll(
    file_path="nifty_500_quarterly_results.csv", purpose=FilePurpose.AGENTS
)
print(f"Uploaded file, file ID: {file.id}")

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

# Create agent with code interpreter tool and tools_resources
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are helpful assistant",
    tools=code_interpreter.definitions,
    tool_resources=code_interpreter.resources,
)
```

Create Agent with Bing Grounding

To enable your Agent to perform search through Bing search API, you use `BingGroundingTool` along with a connection.

Here is an example:

Python

```
bing_connection =
project_client.connections.get(connection_name=os.environ["BING_CONNECTION_NAME"])
conn_id = bing_connection.id

print(conn_id)

# Initialize agent bing tool and add the connection id
bing = BingGroundingTool(connection_id=conn_id)

# Create agent with the bing tool and process assistant run
with project_client:
    agent = project_client.agents.create_agent(
        model=os.environ["MODEL_DEPLOYMENT_NAME"],
        name="my-assistant",
        instructions="You are a helpful assistant",
        tools=bing.definitions,
        headers={"x-ms-enable-preview": "true"},
    )
```

Create Agent with Azure AI Search

Azure AI Search is an enterprise search system for high-performance applications. It integrates with Azure OpenAI Service and Azure Machine Learning, offering advanced search

technologies like vector search and full-text search. Ideal for knowledge base insights, information discovery, and automation. Creating an Agent with Azure AI Search requires an existing Azure AI Search Index. For more information and setup guides, see Azure AI Search Tool Guide.

Here is an example to integrate Azure AI Search:

Python

```
connection =
project_client.connections.get(connection_name=os.environ["AI_SEARCH_CONNECTION_NA
ME"])
conn_id = connection.id

print(conn_id)

# Initialize agent AI search tool and add the search index connection id
ai_search = AzureAISearchTool(
    index_connection_id=conn_id, index_name="sample_index",
query_type=AzureAISearchQueryType.SIMPLE, top_k=3, filter=""
)

# Create agent with AI search tool and process assistant run
with project_client:
    agent = project_client.agents.create_agent(
        model=os.environ["MODEL_DEPLOYMENT_NAME"],
        name="my-assistant",
        instructions="You are a helpful assistant",
        tools=ai_search.definitions,
        tool_resources=ai_search.resources,
    )
```

If the agent has found the relevant information in the index, the reference and annotation will be provided in the message response. In the example above, we replace the reference placeholder by the actual reference and url. Please note, that to get sensible result, the index needs to have fields "title" and "url".

Python

```
# Fetch and log all messages
messages = project_client.agents.list_messages(thread_id=thread.id,
order=ListSortOrder.ASCENDING)
for message in messages.data:
    if message.role == MessageRole.AGENT and message.url_citation_annotations:
        placeholder_annotations = {
            annotation.text: f" [see {annotation.url_citation.title}]
({annotation.url_citation.url})"
            for annotation in message.url_citation_annotations
        }
        for message_text in message.text_messages:
```

```
            message_str = message_text.text.value
            for k, v in placeholder_annotations.items():
                message_str = message_str.replace(k, v)
            print(f"{message.role}: {message_str}")
    else:
        for message_text in message.text_messages:
            print(f"{message.role}: {message_text.text.value}")
```

Create Agent with Function Call

You can enhance your Agents by defining callback functions as function tools. These can be provided to `create_agent` via either the `toolset` parameter or the combination of `tools` and `tool_resources`.

For more details about requirements and specification of functions, refer to Function Tool Specifications

Here is an example to use user functions in `toolset`:

Python

```
functions = FunctionTool(user_functions)
toolset = ToolSet()
toolset.add(functions)
project_client.agents.enable_auto_function_calls(toolset=toolset)

agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are a helpful assistant",
    toolset=toolset,
)
```

For asynchronous functions, you must import `AIProjectClient` from `azure.ai.projects.aio` and use `AsyncFunctionTool`. Here is an example using asynchronous user functions:

Python

```
from azure.ai.projects.aio import AIProjectClient
```

Python

```
functions = AsyncFunctionTool(user_async_functions)

toolset = AsyncToolSet()
toolset.add(functions)
project_client.agents.enable_auto_function_calls(toolset=toolset)
```

```
agent = await project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are a helpful assistant",
    toolset=toolset,
)
```

Notice that if `enable_auto_function_calls` is called, the SDK will invoke the functions automatically during `create_and_process_run` or streaming. If you prefer to execute them manually, refer to sample_agents_stream_eventhandler_with_functions.py or sample_agents_functions.py

Create Agent With Azure Function Call

The AI agent leverages Azure Functions triggered asynchronously via Azure Storage Queues. To enable the agent to perform Azure Function calls, you must set up the corresponding `AzureFunctionTool`, specifying input and output queues as well as parameter definitions.

Example Python snippet illustrating how you create an agent utilizing the Azure Function Tool:

Python

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

azure_function_tool = AzureFunctionTool(
    name="get_weather",
    description="Get weather information using Azure Function",
    parameters={
            "type": "object",
            "properties": {
                "location": {"type": "string", "description": "The location of the
weather."},
            },
            "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 = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are a helpful assistant",
```

```
    tools=azure_function_tool.definitions,
)
```

Limitations

Currently, the Azure Function integration for the AI Agent has the following limitations:

- Supported trigger for Azure Function is currently limited to **Queue triggers** only. HTTP or other trigger types and streaming responses are not supported at this time.

Create and Deploy Azure Function

Before you can use the agent with AzureFunctionTool, you need to create and deploy Azure Function.

Below is an example Python Azure Function responding to queue-triggered messages and placing responses on the output queue:

Python

```
import azure.functions as func
import logging
import json

app = func.FunctionApp()

@app.get_weather(arg_name="inputQueue",
                   queue_name="input",
                   connection="AzureWebJobsStorage")
@app.queue_output(arg_name="outputQueue",
                  queue_name="output",
                  connection="AzureWebJobsStorage")
def get_weather(inputQueue: func.QueueMessage, outputQueue: func.Out[str]):
    try:
        messagepayload = json.loads(inputQueue.get_body().decode("utf-8"))
        location = messagepayload["location"]
        weather_result = f"Weather is 82 degrees and sunny in {location}."

        response_message = {
            "Value": weather_result,
            "CorrelationId": messagepayload["CorrelationId"]
        }

        outputQueue.set(json.dumps(response_message))

        logging.info(f"Sent message to output queue with message
{response_message}")
    except Exception as e:
```

```
        logging.error(f"Error processing message: {e}")
        return
```

> **Important:** Both input and output payloads must contain the `CorrelationId`, which must match in request and response.

Azure Function Project Creation and Deployment

To deploy your function to Azure properly, follow Microsoft's official documentation step by step:

Azure Functions Python Developer Guide

Summary of required steps:

- Use the Azure CLI or Azure Portal to create an Azure Function App.
- Enable System Managed Identity for your Azure Function App.
- Assign appropriate permissions to your Azure Function App identity as outlined in the Role Assignments section below
- Create input and output queues in Azure Storage.
- Deploy your Function code.

Verification and Testing Azure Function

To ensure that your Azure Function deployment functions correctly:

1. Place the following style message manually into the input queue (`input`):

{ "location": "Seattle", "CorrelationId": "42" }

Check the output queue (`output`) and validate the structured message response:

{ "Value": "The weather in Seattle is sunny and warm.", "CorrelationId": "42" }

Required Role Assignments (IAM Configuration)

Clearly assign the following Azure IAM roles to ensure correct permissions:

1. **Azure Function App's identity:**
    - Enable system managed identity through Azure Function App > Settings > Identity.
    - Add permission to storage account:

    - Go to **Storage Account** > **Access control (IAM)** and add role assignment:
        - `Storage Queue Data Contributor` assigned to Azure Function managed identity

2. **Azure AI Project Identity:**

Ensure your Azure AI Project identity has the following storage account permissions:

- `Storage Account Contributor`
- `Storage Blob Data Contributor`
- `Storage File Data Privileged Contributor`
- `Storage Queue Data Contributor`
- `Storage Table Data Contributor`

Additional Important Configuration Notes

- The Azure Function configured above uses the `AzureWebJobsStorage` connection string for queue connectivity. You may alternatively use managed identity-based connections as described in the official Azure Functions Managed Identity documentation.
- Storage queues you specify (`input` & `output`) should already exist in the storage account before the Function deployment or invocation, created manually via Azure portal or CLI.
- When using Azure storage account connection strings, make sure the account has enabled storage account key access (`Storage Account > Settings > Configuration`).

With the above steps complete, your Azure Function integration with your AI Agent is ready for use.

Create Agent With Logic Apps

Logic Apps allow HTTP requests to trigger actions. For more information, refer to the guide Logic App Workflows for Function Calling.

Your Logic App must be in the same resource group as your Azure AI Project, shown in the Azure Portal. Agents SDK accesses Logic Apps through Workflow URLs, which are fetched and called as requests in functions.

Below is an example of how to create an Azure Logic App utility tool and register a function with it.

Python

```
# Create the project client
project_client = AIProjectClient.from_connection_string(
    credential=DefaultAzureCredential(),
    conn_str=os.environ["PROJECT_CONNECTION_STRING"],
)

# Extract subscription and resource group from the project scope
subscription_id = project_client.scope["subscription_id"]
resource_group = project_client.scope["resource_group_name"]

# Logic App details
logic_app_name = "<LOGIC_APP_NAME>"
trigger_name = "<TRIGGER_NAME>"

# Create and initialize AzureLogicAppTool utility
logic_app_tool = AzureLogicAppTool(subscription_id, resource_group)
logic_app_tool.register_logic_app(logic_app_name, trigger_name)
print(f"Registered logic app '{logic_app_name}' with trigger '{trigger_name}'.")

# Create the specialized "send_email_via_logic_app" function for your agent tools
send_email_func = create_send_email_function(logic_app_tool, logic_app_name)

# Prepare the function tools for the agent
functions_to_use: Set = {
    fetch_current_datetime,
    send_email_func,  # This references the AzureLogicAppTool instance via closure
}
```

After this the functions can be incorporated normally into code using `FunctionTool`.

Create Agent With OpenAPI

OpenAPI specifications describe REST operations against a specific endpoint. Agents SDK can read an OpenAPI spec, create a function from it, and call that function against the REST endpoint without additional client-side execution.

Here is an example creating an OpenAPI tool (using anonymous authentication):

Python

```
with open("./weather_openapi.json", "r") as f:
    openapi_weather = jsonref.loads(f.read())

with open("./countries.json", "r") as f:
    openapi_countries = jsonref.loads(f.read())

# Create Auth object for the OpenApiTool (note that connection or managed identity
auth setup requires additional setup in Azure)
```

```
auth = OpenApiAnonymousAuthDetails()

# Initialize agent OpenApi tool using the read in OpenAPI spec
openapi_tool = OpenApiTool(
    name="get_weather",
    spec=openapi_weather,
    description="Retrieve weather information for a location",
    auth=auth,
    default_parameters=["format"],
)
openapi_tool.add_definition(
    name="get_countries",
    spec=openapi_countries,
    description="Retrieve a list of countries",
    auth=auth,
)

# Create agent with OpenApi tool and process assistant run
with project_client:
    agent = project_client.agents.create_agent(
        model=os.environ["MODEL_DEPLOYMENT_NAME"],
        name="my-assistant",
        instructions="You are a helpful assistant",
        tools=openapi_tool.definitions,
    )
```

Create an Agent with Fabric

To enable your Agent to answer queries using Fabric data, use `FabricTool` along with a connection to the Fabric resource.

Here is an example:

Python

```
fabric_connection =
project_client.connections.get(connection_name=os.environ["FABRIC_CONNECTION_NAME"
])
conn_id = fabric_connection.id

print(conn_id)

# Initialize an Agent Fabric tool and add the connection id
fabric = FabricTool(connection_id=conn_id)

# Create an Agent with the Fabric tool and process an Agent run
with project_client:
    agent = project_client.agents.create_agent(
        model=os.environ["MODEL_DEPLOYMENT_NAME"],
        name="my-agent",
        instructions="You are a helpful agent",
```

```
        tools=fabric.definitions,
        headers={"x-ms-enable-preview": "true"},
    )
```

Create Thread

For each session or conversation, a thread is required. Here is an example:

Python

```
thread = project_client.agents.create_thread()
```

Create Thread with Tool Resource

In some scenarios, you might need to assign specific resources to individual threads. To achieve this, you provide the `tool_resources` argument to `create_thread`. In the following example, you create a vector store and upload a file, enable an Agent for file search using the `tools` argument, and then associate the file with the thread using the `tool_resources` argument.

Python

```
file = project_client.agents.upload_file_and_poll(file_path="product_info_1.md",
purpose="assistants")
print(f"Uploaded file, file ID: {file.id}")

vector_store = project_client.agents.create_vector_store_and_poll(file_ids=
[file.id], name="my_vectorstore")
print(f"Created vector store, vector store ID: {vector_store.id}")

# Create file search tool with resources followed by creating agent
file_search = FileSearchTool(vector_store_ids=[vector_store.id])

agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="Hello, you are helpful assistant and can search information from
uploaded files",
    tools=file_search.definitions,
)

print(f"Created agent, ID: {agent.id}")

# Create thread with file resources.
# If the agent has multiple threads, only this thread can search this file.
thread = project_client.agents.create_thread(tool_resources=file_search.resources)
```

List Threads

To list all threads attached to a given agent, use the list_threads API:

Python

```
threads = project_client.agents.list_threads()
```

Create Message

To create a message for assistant to process, you pass `user` as `role` and a question as `content`:

Python

```
message = project_client.agents.create_message(thread_id=thread.id, role="user",
content="Hello, tell me a joke")
```

Create Message with File Search Attachment

To attach a file to a message for content searching, you use `MessageAttachment` and `FileSearchTool`:

Python

```
attachment = MessageAttachment(file_id=file.id,
tools=FileSearchTool().definitions)
message = project_client.agents.create_message(
    thread_id=thread.id, role="user", content="What feature does Smart Eyewear
offer?", attachments=[attachment]
)
```

Create Message with Code Interpreter Attachment

To attach a file to a message for data analysis, use `MessageAttachment` and `CodeInterpreterTool` classes. You must pass `CodeInterpreterTool` as `tools` or `toolset` in `create_agent` call or the file attachment cannot be opened for code interpreter.

Here is an example to pass `CodeInterpreterTool` as tool:

Python

```
# Notice that CodeInterpreter must be enabled in the agent creation,
# otherwise the agent will not be able to see the file attachment for code
```

```
interpretation
agent = project_client.agents.create_agent(
    model=os.environ["MODEL_DEPLOYMENT_NAME"],
    name="my-assistant",
    instructions="You are helpful assistant",
    tools=CodeInterpreterTool().definitions,
)
print(f"Created agent, agent ID: {agent.id}")

thread = project_client.agents.create_thread()
print(f"Created thread, thread ID: {thread.id}")

# Create an attachment
attachment = MessageAttachment(file_id=file.id,
tools=CodeInterpreterTool().definitions)

# Create a message
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content="Could you please create bar chart in TRANSPORTATION sector for the
operating profit from the uploaded csv file and provide file to me?",
    attachments=[attachment],
)
```

Azure blob storage can be used as a message attachment. In this case, use `VectorStoreDataSource` as a data source:

Python

```
# We will upload the local file to Azure and will use it for vector store
creation.
_, asset_uri = project_client.upload_file("./product_info_1.md")
ds = VectorStoreDataSource(asset_identifier=asset_uri,
asset_type=VectorStoreDataSourceAssetType.URI_ASSET)

# Create a message with the attachment
attachment = MessageAttachment(data_source=ds, tools=code_interpreter.definitions)
message = project_client.agents.create_message(
    thread_id=thread.id, role="user", content="What does the attachment say?",
attachments=[attachment]
)
```

Create Message with Image Inputs

You can send messages to Azure agents with image inputs in following ways:

- **Using an image stored as a uploaded file**
- **Using a public image accessible via URL**

- **Using a base64 encoded image string**

The following examples demonstrate each method:

Create message using uploaded image file

Python

```
# Upload the local image file
image_file =
project_client.agents.upload_file_and_poll(file_path="image_file.png",
purpose="assistants")

# Construct content using uploaded image
file_param = MessageImageFileParam(file_id=image_file.id, detail="high")
content_blocks = [
    MessageInputTextBlock(text="Hello, what is in the image?"),
    MessageInputImageFileBlock(image_file=file_param),
]

# Create the message
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content=content_blocks
)
```

Create message with an image URL input

Python

```
# Specify the public image URL
image_url = "https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-
wisconsin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-
nature-boardwalk.jpg"

# Create content directly referencing image URL
url_param = MessageImageUrlParam(url=image_url, detail="high")
content_blocks = [
    MessageInputTextBlock(text="Hello, what is in the image?"),
    MessageInputImageUrlBlock(image_url=url_param),
]

# Create the message
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content=content_blocks
)
```

Create message with base64-encoded image input

Python

```python
import base64

def image_file_to_base64(path: str) -> str:
    with open(path, "rb") as f:
        return base64.b64encode(f.read()).decode("utf-8")

# Convert your image file to base64 format
image_base64 = image_file_to_base64("image_file.png")

# Prepare the data URL
img_data_url = f"{image_base64}"

# Use base64 encoded string as image URL parameter
url_param = MessageImageUrlParam(url=img_data_url, detail="high")
content_blocks = [
    MessageInputTextBlock(text="Hello, what is in the image?"),
    MessageInputImageUrlBlock(image_url=url_param),
]

# Create the message
message = project_client.agents.create_message(
    thread_id=thread.id,
    role="user",
    content=content_blocks
)
```

Create Run, Run_and_Process, or Stream

To process your message, you can use `create_run`, `create_and_process_run`, or `create_stream`.

`create_run` requests the Agent to process the message without polling for the result. If you are using `function tools` regardless as `toolset` or not, your code is responsible for polling for the result and acknowledging the status of `Run`. When the status is `requires_action`, your code is responsible for calling the function tools. For a code sample, visit sample_agents_functions.py.

Here is an example of `create_run` and poll until the run is completed:

Python

```python
run = project_client.agents.create_run(thread_id=thread.id, agent_id=agent.id)

# Poll the run as long as run status is queued or in progress
while run.status in ["queued", "in_progress", "requires_action"]:
    # Wait for a second
```

```
    time.sleep(1)
    run = project_client.agents.get_run(thread_id=thread.id, run_id=run.id)
```

To have the SDK poll on your behalf and call `function tools`, use the `create_and_process_run` method.

Here is an example:

Python

```
run = project_client.agents.create_and_process_run(thread_id=thread.id,
agent_id=agent.id)
```

With streaming, polling need not be considered. If `function tools` are provided as `toolset` during the `create_agent` call, they will be invoked by the SDK.

Here is an example of streaming:

Python

```
with project_client.agents.create_stream(thread_id=thread.id, agent_id=agent.id)
as stream:

    for event_type, event_data, _ in stream:

        if isinstance(event_data, MessageDeltaChunk):
            print(f"Text delta received: {event_data.text}")

        elif isinstance(event_data, ThreadMessage):
            print(f"ThreadMessage created. ID: {event_data.id}, Status:
{event_data.status}")

        elif isinstance(event_data, ThreadRun):
            print(f"ThreadRun status: {event_data.status}")

        elif isinstance(event_data, RunStep):
            print(f"RunStep type: {event_data.type}, Status: {event_data.status}")

        elif event_type == AgentStreamEvent.ERROR:
            print(f"An error occurred. Data: {event_data}")

        elif event_type == AgentStreamEvent.DONE:
            print("Stream completed.")
            break

        else:
            print(f"Unhandled Event Type: {event_type}, Data: {event_data}")
```

In the code above, because an `event_handler` object is not passed to the `create_stream` function, the SDK will instantiate `AgentEventHandler` or `AsyncAgentEventHandler` as the default event handler and produce an iterable object with `event_type` and `event_data`. `AgentEventHandler` and `AsyncAgentEventHandler` are overridable. Here is an example:

Python

```
# With AgentEventHandler[str], the return type for each event functions is
optional string.
class MyEventHandler(AgentEventHandler[str]):

    def on_message_delta(self, delta: "MessageDeltaChunk") -> Optional[str]:
        return f"Text delta received: {delta.text}"

    def on_thread_message(self, message: "ThreadMessage") -> Optional[str]:
        return f"ThreadMessage created. ID: {message.id}, Status:
{message.status}"

    def on_thread_run(self, run: "ThreadRun") -> Optional[str]:
        return f"ThreadRun status: {run.status}"

    def on_run_step(self, step: "RunStep") -> Optional[str]:
        return f"RunStep type: {step.type}, Status: {step.status}"

    def on_error(self, data: str) -> Optional[str]:
        return f"An error occurred. Data: {data}"

    def on_done(self) -> Optional[str]:
        return "Stream completed."

    def on_unhandled_event(self, event_type: str, event_data: Any) ->
Optional[str]:
        return f"Unhandled Event Type: {event_type}, Data: {event_data}"
```

Python

```
with project_client.agents.create_stream(
    thread_id=thread.id, agent_id=agent.id, event_handler=MyEventHandler()
) as stream:
    for event_type, event_data, func_return in stream:
        print(f"Received data.")
        print(f"Streaming receive Event Type: {event_type}")
        print(f"Event Data: {str(event_data)[:100]}...")
        print(f"Event Function return: {func_return}\n")
```

As you can see, this SDK parses the events and produces various event types similar to OpenAI assistants. In your use case, you might not be interested in handling all these types and may decide to parse the events on your own. To achieve this, please refer to override base event handler.

```
Note: Multiple streaming processes may be chained behind the scenes.

When the SDK receives a `ThreadRun` event with the status `requires_action`, the
next event will be `Done`, followed by termination. The SDK will submit the tool
calls using the same event handler. The event handler will then chain the main
stream with the tool stream.

Consequently, when you iterate over the streaming using a for loop similar to the
example above, the for loop will receive events from the main stream followed by
events from the tool stream.
```

Retrieve Message

To retrieve messages from agents, use the following example:

Python

```python
messages = project_client.agents.list_messages(thread_id=thread.id)

# The messages are following in the reverse order,
# we will iterate them and output only text contents.
for data_point in reversed(messages.data):
    last_message_content = data_point.content[-1]
    if isinstance(last_message_content, MessageTextContent):
        print(f"{data_point.role}: {last_message_content.text.value}")
```

In addition, `messages` and `messages.data[]` offer helper properties such as `text_messages`, `image_contents`, `file_citation_annotations`, and `file_path_annotations` to quickly retrieve content from one message or all messages.

Retrieve File

Files uploaded by Agents cannot be retrieved back. If your use case need to access the file content uploaded by the Agents, you are advised to keep an additional copy accessible by your application. However, files generated by Agents are retrievable by `save_file` or `get_file_content`.

Here is an example retrieving file ids from messages and save to the local drive:

Python

```python
messages = project_client.agents.list_messages(thread_id=thread.id)
print(f"Messages: {messages}")
```

```
for image_content in messages.image_contents:
    file_id = image_content.image_file.file_id
    print(f"Image File ID: {file_id}")
    file_name = f"{file_id}_image_file.png"
    project_client.agents.save_file(file_id=file_id, file_name=file_name)
    print(f"Saved image file to: {Path.cwd() / file_name}")

for file_path_annotation in messages.file_path_annotations:
    print(f"File Paths:")
    print(f"Type: {file_path_annotation.type}")
    print(f"Text: {file_path_annotation.text}")
    print(f"File ID: {file_path_annotation.file_path.file_id}")
    print(f"Start Index: {file_path_annotation.start_index}")
    print(f"End Index: {file_path_annotation.end_index}")
```

Here is an example to use `get_file_content`:

Python

```
from pathlib import Path

async def save_file_content(client, file_id: str, file_name: str, target_dir:
Optional[Union[str, Path]] = None):
    # Determine the target directory
    path = Path(target_dir).expanduser().resolve() if target_dir else Path.cwd()
    path.mkdir(parents=True, exist_ok=True)

    # Retrieve the file content
    file_content_stream = await client.get_file_content(file_id)
    if not file_content_stream:
        raise RuntimeError(f"No content retrievable for file ID '{file_id}'.")

    # Collect all chunks asynchronously
    chunks = []
    async for chunk in file_content_stream:
        if isinstance(chunk, (bytes, bytearray)):
            chunks.append(chunk)
        else:
            raise TypeError(f"Expected bytes or bytearray, got
{type(chunk).__name__}")

    target_file_path = path / file_name

    # Write the collected content to the file synchronously
    with open(target_file_path, "wb") as file:
        for chunk in chunks:
            file.write(chunk)
```

Teardown

To remove resources after completing tasks, use the following functions:

Python

```
# Delete the file when done
project_client.agents.delete_vector_store(vector_store.id)
print("Deleted vector store")

project_client.agents.delete_file(file_id=file.id)
print("Deleted file")

# Delete the agent when done
project_client.agents.delete_agent(agent.id)
print("Deleted agent")
```

Evaluation

Evaluation in Azure AI Project client library is designed to assess the performance of generative AI applications in the cloud. The output of Generative AI application is quantitively measured with mathematical based metrics, AI-assisted quality and safety metrics. Metrics are defined as evaluators. Built-in or custom evaluators can provide comprehensive insights into the application's capabilities and limitations.

Evaluator

Evaluators are custom or prebuilt classes or functions that are designed to measure the quality of the outputs from language models or generative AI applications.

Evaluators are made available via azure-ai-evaluation SDK for local experience and also in Evaluator Library in Azure AI Foundry for using them in the cloud.

More details on built-in and custom evaluators can be found here.

Run Evaluation in the cloud

To run evaluation in the cloud the following are needed:

- Evaluators
- Data to be evaluated
- [Optional] Azure Open AI model.

Evaluators

For running evaluator in the cloud, evaluator `ID` is needed. To get it via code you use azure-ai-evaluation

Python

```
# pip install azure-ai-evaluation

from azure.ai.evaluation import RelevanceEvaluator

evaluator_id = RelevanceEvaluator.id
```

Data to be evaluated

Evaluation in the cloud supports data in form of `jsonl` file. Data can be uploaded via the helper method `upload_file` on the project client.

Python

```
# Upload data for evaluation and get dataset id
data_id, _ = project_client.upload_file("<data_file.jsonl>")
```

[Optional] Azure OpenAI Model

Azure AI Foundry project comes with a default Azure Open AI endpoint which can be easily accessed using following code. This gives you the endpoint details for you Azure OpenAI endpoint. Some of the evaluators need model that supports chat completion.

Python

```
default_connection =
project_client.connections.get_default(connection_type=ConnectionType.AZURE_OPEN_A
I)
```

Example Remote Evaluation

Python

```
import os
from azure.ai.projects import AIProjectClient
from azure.identity import DefaultAzureCredential
from azure.ai.projects.models import Evaluation, Dataset, EvaluatorConfiguration,
ConnectionType
from azure.ai.evaluation import F1ScoreEvaluator, RelevanceEvaluator,
HateUnfairnessEvaluator


# Create project client
project_client = AIProjectClient.from_connection_string(
```

```
    credential=DefaultAzureCredential(),
    conn_str=os.environ["PROJECT_CONNECTION_STRING"],
)

# Upload data for evaluation and get dataset id
data_id, _ = project_client.upload_file("<data_file.jsonl>")

deployment_name = "<deployment_name>"
api_version = "<api_version>"

# Create an evaluation
evaluation = Evaluation(
    display_name="Remote Evaluation",
    description="Evaluation of dataset",
    data=Dataset(id=data_id),
    evaluators={
        "f1_score": EvaluatorConfiguration(
            id=F1ScoreEvaluator.id,
        ),
        "relevance": EvaluatorConfiguration(
            id=RelevanceEvaluator.id,
            init_params={
                "model_config": default_connection.to_evaluator_model_config(
                    deployment_name=deployment_name, api_version=api_version
                )
            },
        ),
        "violence": EvaluatorConfiguration(
            id=ViolenceEvaluator.id,
            init_params={"azure_ai_project": project_client.scope},
        ),
    },
)


evaluation_response = project_client.evaluations.create(
    evaluation=evaluation,
)

# Get evaluation
get_evaluation_response = project_client.evaluations.get(evaluation_response.id)

print("----------------------------------------------------------------")
print("Created evaluation, evaluation ID: ", get_evaluation_response.id)
print("Evaluation status: ", get_evaluation_response.status)
if isinstance(get_evaluation_response.properties, dict):
    print("AI Foundry URI: ",
get_evaluation_response.properties["AiStudioEvaluationUri"])
print("----------------------------------------------------------------")
```

NOTE: For running evaluators locally refer to Evaluate with the Azure AI Evaluation SDK.

Tracing

You can add an Application Insights Azure resource to your Azure AI Foundry project. See the Tracing tab in your AI Foundry project. If one was enabled, you can get the Application Insights connection string, configure your Agents, and observe the full execution path through Azure Monitor. Typically, you might want to start tracing before you create an Agent.

Installation

Make sure to install OpenTelemetry and the Azure SDK tracing plugin via

```Bash
pip install opentelemetry
pip install azure-ai-projects azure-identity opentelemetry-sdk azure-core-tracing-opentelemetry
```

You will also need an exporter to send telemetry to your observability backend. You can print traces to the console or use a local viewer such as Aspire Dashboard.

To connect to Aspire Dashboard or another OpenTelemetry compatible backend, install OTLP exporter:

```Bash
pip install opentelemetry-exporter-otlp
```

How to enable tracing

Here is a code sample that shows how to enable Azure Monitor tracing:

```Python
from opentelemetry import trace
from azure.monitor.opentelemetry import configure_azure_monitor

# Enable Azure Monitor tracing
application_insights_connection_string =
project_client.telemetry.get_connection_string()
if not application_insights_connection_string:
    print("Application Insights was not enabled for this project.")
    print("Enable it via the 'Tracing' tab in your AI Foundry project page.")
    exit()
configure_azure_monitor(connection_string=application_insights_connection_string)

# enable additional instrumentations
```

```
project_client.telemetry.enable()

scenario = os.path.basename(__file__)
tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span(scenario):
    with project_client:
```

In addition, you might find helpful to see the tracing logs in console. You can achieve by the following code:

Python

```
project_client.telemetry.enable(destination=sys.stdout)
```

How to trace your own functions

The decorator `trace_function` is provided for tracing your own function calls using OpenTelemetry. By default the function name is used as the name for the span. Alternatively you can provide the name for the span as a parameter to the decorator.

This decorator handles various data types for function parameters and return values, and records them as attributes in the trace span. The supported data types include:

- Basic data types: str, int, float, bool
- Collections: list, dict, tuple, set
  - Special handling for collections:
    - If a collection (list, dict, tuple, set) contains nested collections, the entire collection is converted to a string before being recorded as an attribute.
    - Sets and dictionaries are always converted to strings to ensure compatibility with span attributes.

Object types are omitted, and the corresponding parameter is not traced.

The parameters are recorded in attributes `code.function.parameter.<parameter_name>` and the return value is recorder in attribute `code.function.return.value`

Troubleshooting

Exceptions

Client methods that make service calls raise an HttpResponseError exception for a non-success HTTP status code response from the service. The exception's `status_code` will hold the HTTP response status code (with `reason` showing the friendly name). The exception's `error.message` contains a detailed message that may be helpful in diagnosing the issue:

Python

```python
from azure.core.exceptions import HttpResponseError

...

try:
    result = project_client.connections.list()
except HttpResponseError as e:
    print(f"Status code: {e.status_code} ({e.reason})")
    print(e.message)
```

For example, when you provide wrong credentials:

text

```
Status code: 401 (Unauthorized)
Operation returned an invalid status 'Unauthorized'
```

Logging

The client uses the standard Python logging library. The SDK logs HTTP request and response details, which may be useful in troubleshooting. To log to stdout, add the following:

Python

```python
import sys
import logging

# Acquire the logger for this client library. Use 'azure' to affect both
# 'azure.core` and `azure.ai.inference' libraries.
logger = logging.getLogger("azure")

# Set the desired logging level. logging.INFO or logging.DEBUG are good options.
logger.setLevel(logging.DEBUG)

# Direct logging output to stdout:
handler = logging.StreamHandler(stream=sys.stdout)
# Or direct logging output to a file:
# handler = logging.FileHandler(filename="sample.log")
logger.addHandler(handler)

# Optional: change the default logging format. Here we add a timestamp.
```

```
#formatter = logging.Formatter("%(asctime)s:%(levelname)s:%(name)s:%(message)s")
#handler.setFormatter(formatter)
```

By default logs redact the values of URL query strings, the values of some HTTP request and response headers (including `Authorization` which holds the key or token), and the request and response payloads. To create logs without redaction, add `logging_enable = True` to the client constructor:

Python

```
project_client = AIProjectClient.from_connection_string(
    credential=DefaultAzureCredential(),
    conn_str=os.environ["PROJECT_CONNECTION_STRING"],
    logging_enable = True
)
```

Note that the log level must be set to `logging.DEBUG` (see above code). Logs will be redacted with any other log level.

Be sure to protect non redacted logs to avoid compromising security.

For more information, see Configure logging in the Azure libraries for Python

Reporting issues

To report an issue with the client library, or request additional features, please open a GitHub issue here. Mention the package name "azure-ai-projects" in the title or content.

Next steps

Have a look at the Samples folder, containing fully runnable Python code for synchronous and asynchronous clients.

Explore the AI Starter Template. This template creates an Azure AI Foundry hub, project and connected resources including Azure OpenAI Service, AI Search and more. It also deploys a simple chat application to Azure Container Apps.

Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit https://cla.microsoft.com.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information, see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Azure AI Projects client library for JavaScript - version 1.0.0-beta.5

Article • 04/22/2025

Use the AI Projects client library (in preview) to:

- **Enumerate connections** in your Azure AI Foundry project and get connection properties. For example, get the inference endpoint URL and credentials associated with your Azure OpenAI connection.
- **Develop Agents using the Azure AI Agent Service**, leveraging an extensive ecosystem of models, tools, and capabilities from OpenAI, Microsoft, and other LLM providers. The Azure AI Agent Service enables the building of Agents for a wide range of generative AI use cases. The package is currently in private preview.
- **Enable OpenTelemetry tracing**.

Product documentation

| Samples | Package (npm) | API reference documentation

Table of contents

- Getting started
  - Prerequisite
  - Install the package
- Key concepts
  - Create and authenticate the client
- Examples
  - Enumerate connections
    - Get properties of all connections
    - Get properties of all connections of a particular type
    - Get properties of a default connection
    - Get properties of a connection by its connection name
  - Agents (Preview)
    - Create an Agent with:
      - File Search
      - Code interpreter
      - Bing grounding
      - Azure AI Search
      - Function call
      - Fabric Data
    - Create thread with

        - Tool resource
      - Create message with:
        - File search attachment
        - Code interpreter attachment
      - Execute Run, Create Thread and Run, or Stream
      - Retrieve message
      - Retrieve file
      - Tear down by deleting resource
      - Tracing
    - Tracing
      - Installation
      - Tracing example
- Troubleshooting
  - Exceptions
  - Reporting issues
  - Next steps
- Contributing

Getting started

Prerequisite

- LTS versions of Node.js
- An Azure subscription .
- A project in Azure AI Foundry.
- The project connection string. It can be found in your Azure AI Foundry project overview page, under "Project details". Below we will assume the environment variable `AZURE_AI_PROJECTS_CONNECTION_STRING` was defined to hold this value.
- Entra ID is needed to authenticate the client. Your application needs an object that implements the TokenCredential interface. Code samples here use DefaultAzureCredential. To get that working, you will need:
  - The `Contributor` role. Role assigned can be done via the "Access Control (IAM)" tab of your Azure AI Project resource in the Azure portal.
  - Azure CLI installed.
  - You are logged into your Azure account by running `az login`.
  - Note that if you have multiple Azure subscriptions, the subscription that contains your Azure AI Project resource must be your default subscription. Run `az account list --output table` to list all your subscription and see which one is the default. Run `az`

`account set --subscription "Your Subscription ID or Name"` to change your default subscription.

Install the package

Bash

```
npm install @azure/ai-projects @azure/identity
```

Key concepts

Create and authenticate the client

The class factory method `fromConnectionString` is used to construct the client. To construct a client:

ts

```ts
import { AIProjectsClient } from "@azure/ai-projects";
import { DefaultAzureCredential } from "@azure/identity";

const connectionString = process.env.AZURE_AI_PROJECTS_CONNECTION_STRING ?? "
<connectionString>";
const client = AIProjectsClient.fromConnectionString(
  connectionString,
  new DefaultAzureCredential(),
);
```

Examples

Enumerate connections

Your Azure AI Foundry project has a "Management center". When you enter it, you will see a tab named "Connected resources" under your project. The `.connections` operations on the client allow you to enumerate the connections and get connection properties. Connection properties include the resource URL and authentication credentials, among other things.

Below are code examples of the connection operations. Full samples can be found under the "connections" folder in the package samples.

Get properties of all connections

To list the properties of all the connections in the Azure AI Foundry project:

```ts
const connections = await client.connections.listConnections();
for (const connection of connections) {
  console.log(connection);
}
```

Get properties of all connections of a particular type

To list the properties of connections of a certain type (here Azure OpenAI):

```ts
const connections = await client.connections.listConnections({ category:
"AzureOpenAI" });
for (const connection of connections) {
  console.log(connection);
}
```

Get properties of a connection by its connection name

To get the connection properties of a connection named `connectionName`:

```ts
const connection = await client.connections.getConnection("connectionName");
console.log(connection);
```

To get the connection properties with its authentication credentials:

```ts
const connection = await
client.connections.getConnectionWithSecrets("connectionName");
console.log(connection);
```

Agents (Preview)

Agents in the Azure AI Projects client library are designed to facilitate various interactions and operations within your AI projects. They serve as the core components that manage and

execute tasks, leveraging different tools and resources to achieve specific goals. The following steps outline the typical sequence for interacting with Agents. See the "agents" folder in the package samples for additional Agent samples.

Agents are actively being developed. A sign-up form for private preview is coming soon.

Create Agent

Here is an example of how to create an Agent:

ts

```ts
const agent = await client.agents.createAgent("gpt-4o", {
  name: "my-agent",
  instructions: "You are a helpful assistant",
});
```

To allow Agents to access your resources or custom functions, you need tools. You can pass tools to `createAgent` through the `tools` and `toolResources` arguments.

You can use `ToolSet` to do this:

ts

```ts
import { ToolSet } from "@azure/ai-projects";

const toolSet = new ToolSet();
toolSet.addFileSearchTool([vectorStore.id]);
toolSet.addCodeInterpreterTool([codeInterpreterFile.id]);
const agent = await client.agents.createAgent("gpt-4o", {
  name: "my-agent",
  instructions: "You are a helpful agent",
  tools: toolSet.toolDefinitions,
  toolResources: toolSet.toolResources,
});
console.log(`Created agent, agent ID: ${agent.id}`);
```

Create Agent with File Search

To perform file search by an Agent, we first need to upload a file, create a vector store, and associate the file to the vector store. Here is an example:

ts

```ts
import { ToolUtility } from "@azure/ai-projects";
```

```
const localFileStream = fs.createReadStream(filePath);
const file = await client.agents.uploadFile(localFileStream, "assistants", {
  fileName: "sample_file_for_upload.txt",
});
console.log(`Uploaded file, ID: ${file.id}`);
const vectorStore = await client.agents.createVectorStore({
  fileIds: [file.id],
  name: "my_vector_store",
});
console.log(`Created vector store, ID: ${vectorStore.id}`);
const fileSearchTool = ToolUtility.createFileSearchTool([vectorStore.id]);
const agent = await client.agents.createAgent("gpt-4o", {
  name: "SDK Test Agent - Retrieval",
  instructions: "You are helpful agent that can help fetch data from files you
know about.",
  tools: [fileSearchTool.definition],
  toolResources: fileSearchTool.resources,
});
console.log(`Created agent, agent ID : ${agent.id}`);
```

Create Agent with Code Interpreter

Here is an example to upload a file and use it for code interpreter by an Agent:

ts

```
import { ToolUtility } from "@azure/ai-projects";

const localFileStream = fs.createReadStream(filePath);
const localFile = await client.agents.uploadFile(localFileStream, "assistants", {
  fileName: "localFile",
});
console.log(`Uploaded local file, file ID : ${localFile.id}`);
const codeInterpreterTool = ToolUtility.createCodeInterpreterTool([localFile.id]);
// Notice that CodeInterpreter must be enabled in the agent creation, otherwise
the agent will not be able to see the file attachment
const agent = await client.agents.createAgent("gpt-4o-mini", {
  name: "my-agent",
  instructions: "You are a helpful agent",
  tools: [codeInterpreterTool.definition],
  toolResources: codeInterpreterTool.resources,
});
console.log(`Created agent, agent ID: ${agent.id}`);
```

Create Agent with Bing Grounding

To enable your Agent to perform search through Bing search API, you use `ToolUtility.createConnectionTool()` along with a connection.

Here is an example:

```ts
import { ToolUtility, connectionToolType } from "@azure/ai-projects";

const bingConnection = await client.connections.getConnection(
  process.env.BING_CONNECTION_NAME ?? "<connection-name>",
);
const connectionId = bingConnection.id;
const bingTool =
ToolUtility.createConnectionTool(connectionToolType.BingGrounding,
[connectionId]);
const agent = await client.agents.createAgent("gpt-4o", {
  name: "my-agent",
  instructions: "You are a helpful agent",
  tools: [bingTool.definition],
});
console.log(`Created agent, agent ID : ${agent.id}`);
```

Create Agent with Azure AI Search

Azure AI Search is an enterprise search system for high-performance applications. It integrates with Azure OpenAI Service and Azure Machine Learning, offering advanced search technologies like vector search and full-text search. Ideal for knowledge base insights, information discovery, and automation

Here is an example to integrate Azure AI Search:

```ts
import { ToolUtility } from "@azure/ai-projects";

const connectionName =
  process.env.AZURE_AI_SEARCH_CONNECTION_NAME ?? "<AzureAISearchConnectionName>";
const connection = await client.connections.getConnection(connectionName);
const azureAISearchTool = ToolUtility.createAzureAISearchTool(connection.id,
connection.name);
const agent = await client.agents.createAgent("gpt-4-0125-preview", {
  name: "my-agent",
  instructions: "You are a helpful agent",
  tools: [azureAISearchTool.definition],
  toolResources: azureAISearchTool.resources,
});
console.log(`Created agent, agent ID : ${agent.id}`);
```

Create Agent with Function Call

You can enhance your Agents by defining callback functions as function tools. These can be provided to `createAgent` via the combination of `tools` and `toolResources`. Only the function definitions and descriptions are provided to `createAgent`, without the implementations. The `Run` or `event handler of stream` will raise a `requires_action` status based on the function definitions. Your code must handle this status and call the appropriate functions.

Here is an example:

ts

```
import {
  FunctionToolDefinition,
  ToolUtility,
  RequiredToolCallOutput,
  FunctionToolDefinitionOutput,
  ToolOutput,
} from "@azure/ai-projects";

class FunctionToolExecutor {
  private functionTools: {
    func: Function;
    definition: FunctionToolDefinition;
  }[];
  constructor() {
    this.functionTools = [
      {
        func: this.getUserFavoriteCity,
        ...ToolUtility.createFunctionTool({
          name: "getUserFavoriteCity",
          description: "Gets the user's favorite city.",
          parameters: {},
        }),
      },
      {
        func: this.getCityNickname,
        ...ToolUtility.createFunctionTool({
          name: "getCityNickname",
          description: "Gets the nickname of a city, e.g. 'LA' for 'Los Angeles,
CA'.",
          parameters: {
            type: "object",
            properties: {
              location: { type: "string", description: "The city and state, e.g.
Seattle, Wa" },
            },
          },
        }),
      },
      {
        func: this.getWeather,
        ...ToolUtility.createFunctionTool({
          name: "getWeather",
```

```
          description: "Gets the weather for a location.",
          parameters: {
            type: "object",
            properties: {
              location: { type: "string", description: "The city and state, e.g.
Seattle, Wa" },
              unit: { type: "string", enum: ["c", "f"] },
            },
          },
        }),
      },
    ];
  }
  private getUserFavoriteCity(): {} {
    return { location: "Seattle, WA" };
  }
  private getCityNickname(location: string): {} {
    return { nickname: "The Emerald City" };
  }
  private getWeather(location: string, unit: string): {} {
    return { weather: unit === "f" ? "72f" : "22c" };
  }
  public invokeTool(
    toolCall: RequiredToolCallOutput & FunctionToolDefinitionOutput,
  ): ToolOutput | undefined {
    console.log(`Function tool call - ${toolCall.function.name}`);
    const args = [];
    if (toolCall.function.parameters) {
      try {
        const params = JSON.parse(toolCall.function.parameters);
        for (const key in params) {
          if (Object.prototype.hasOwnProperty.call(params, key)) {
            args.push(params[key]);
          }
        }
      } catch (error) {
        console.error(`Failed to parse parameters:
${toolCall.function.parameters}`, error);
        return undefined;
      }
    }
    const result = this.functionTools
      .find((tool) => tool.definition.function.name === toolCall.function.name)
      ?.func(...args);
    return result
      ? {
          toolCallId: toolCall.id,
          output: JSON.stringify(result),
        }
      : undefined;
  }
  public getFunctionDefinitions(): FunctionToolDefinition[] {
    return this.functionTools.map((tool) => {
      return tool.definition;
    });
```

```
  }
}
const functionToolExecutor = new FunctionToolExecutor();
const functionTools = functionToolExecutor.getFunctionDefinitions();
const agent = await client.agents.createAgent("gpt-4o", {
  name: "my-agent",
  instructions:
    "You are a weather bot. Use the provided functions to help answer questions.
Customize your responses to the user's preferences as much as possible and use
friendly nicknames for cities whenever possible.",
  tools: functionTools,
});
console.log(`Created agent, agent ID: ${agent.id}`);
```

Create Agent With OpenAPI

OpenAPI specifications describe REST operations against a specific endpoint. Agents SDK can read an OpenAPI spec, create a function from it, and call that function against the REST endpoint without additional client-side execution. Here is an example creating an OpenAPI tool (using anonymous authentication):

ts

```ts
import { ToolUtility } from "@azure/ai-projects";

// Read in OpenApi spec
const filePath = "./data/weatherOpenApi.json";
const openApiSpec = JSON.parse(fs.readFileSync(filePath, "utf-8"));
// Define OpenApi function
const openApiFunction = {
  name: "getWeather",
  spec: openApiSpec,
  description: "Retrieve weather information for a location",
  auth: {
    type: "anonymous",
  },
  default_params: ["format"], // optional
};
// Create OpenApi tool
const openApiTool = ToolUtility.createOpenApiTool(openApiFunction);
// Create agent with OpenApi tool
const agent = await client.agents.createAgent("gpt-4o-mini", {
  name: "myAgent",
  instructions: "You are a helpful agent",
  tools: [openApiTool.definition],
});
console.log(`Created agent, agent ID: ${agent.id}`);
```

Create an Agent with Fabric

To enable your Agent to answer queries using Fabric data, use `FabricTool` along with a connection to the Fabric resource.

Here is an example:

```ts
import { ToolUtility } from "@azure/ai-projects";

const fabricConnection = await client.connections.getConnection(
  process.env["FABRIC_CONNECTION_NAME"] || "<connection-name>",
);
const connectionId = fabricConnection.id;
// Initialize agent Microsoft Fabric tool with the connection id
const fabricTool = ToolUtility.createFabricTool(connectionId);
// Create agent with the Microsoft Fabric tool and process assistant run
const agent = await client.agents.createAgent("gpt-4o", {
  name: "my-agent",
  instructions: "You are a helpful agent",
  tools: [fabricTool.definition],
});
console.log(`Created agent, agent ID : ${agent.id}`);
```

Create Thread

For each session or conversation, a thread is required. Here is an example:

```ts
const thread = await client.agents.createThread();
```

Create Thread with Tool Resource

In some scenarios, you might need to assign specific resources to individual threads. To achieve this, you provide the `toolResources` argument to `createThread`. In the following example, you create a vector store and upload a file, enable an Agent for file search using the `tools` argument, and then associate the file with the thread using the `toolResources` argument.

```ts
import { ToolUtility } from "@azure/ai-projects";

const localFileStream = fs.createReadStream(filePath);
const file = await client.agents.uploadFile(localFileStream, "assistants", {
  fileName: "sample_file_for_upload.csv",
});
```

```
console.log(`Uploaded file, ID: ${file.id}`);
const vectorStore = await client.agents.createVectorStore({
  fileIds: [file.id],
});
console.log(`Created vector store, ID: ${vectorStore.id}`);
const fileSearchTool = ToolUtility.createFileSearchTool([vectorStore.id]);
const agent = await client.agents.createAgent("gpt-4o", {
  name: "myAgent",
  instructions: "You are helpful agent that can help fetch data from files you
know about.",
  tools: [fileSearchTool.definition],
});
console.log(`Created agent, agent ID : ${agent.id}`);
// Create thread with file resources.
// If the agent has multiple threads, only this thread can search this file.
const thread = await client.agents.createThread({ toolResources:
fileSearchTool.resources });
```

List Threads

To list all threads attached to a given agent, use the list_threads API:

ts

```
const threads = await client.agents.listThreads();
console.log(`Threads for agent ${agent.id}:`);
for await (const t of (await threads).data) {
  console.log(`Thread ID: ${t.id}`);
  console.log(`Created at: ${t.createdAt}`);
  console.log(`Metadata: ${t.metadata}`);
  console.log(`---- `);
}
```

Create Message

To create a message for assistant to process, you pass `user` as `role` and a question as `content`:

ts

```
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content: "hello, world!",
});
console.log(`Created message, message ID: ${message.id}`);
```

Create Message with File Search Attachment

To attach a file to a message for content searching, you use `ToolUtility.createFileSearchTool()` and the `attachments` argument:

ts

```ts
import { ToolUtility } from "@azure/ai-projects";

const fileSearchTool = ToolUtility.createFileSearchTool();
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content: "What feature does Smart Eyewear offer?",
  attachments: {
    fileId: file.id,
    tools: [fileSearchTool.definition],
  },
});
```

Create Message with Code Interpreter Attachment

To attach a file to a message for data analysis, you use `ToolUtility.createCodeInterpreterTool()` and the `attachment` argument.

Here is an example:

ts

```ts
import { ToolUtility } from "@azure/ai-projects";

// notice that CodeInterpreter must be enabled in the agent creation,
// otherwise the agent will not be able to see the file attachment for code
interpretation
const codeInterpreterTool = ToolUtility.createCodeInterpreterTool();
const agent = await client.agents.createAgent("gpt-4-1106-preview", {
  name: "my-assistant",
  instructions: "You are helpful assistant",
  tools: [codeInterpreterTool.definition],
});
console.log(`Created agent, agent ID: ${agent.id}`);
const thread = await client.agents.createThread();
console.log(`Created thread, thread ID: ${thread.id}`);
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content:
    "Could you please create bar chart in TRANSPORTATION sector for the operating
profit from the uploaded csv file and provide file to me?",
  attachments: {
    fileId: file.id,
    tools: [codeInterpreterTool.definition],
  },
```

```
});
console.log(`Created message, message ID: ${message.id}`);
```

Create Message with Image Inputs

You can send messages to Azure agents with image inputs in following ways:

- **Using an image stored as a uploaded file**
- **Using a public image accessible via URL**
- **Using a base64 encoded image string**

The following examples demonstrate each method:

Create message using uploaded image file

ts

```ts
// Upload the local image file
const fileStream = fs.createReadStream(imagePath);
const imageFile = await client.agents.uploadFile(fileStream, "assistants", {
  fileName: "image_file.png",
});
console.log(`Uploaded file, file ID: ${imageFile.id}`);
// Create a message with both text and image content
console.log("Creating message with image content...");
const inputMessage = "Hello, what is in the image?";
const content = [
  {
    type: "text",
    text: inputMessage,
  },
  {
    type: "image_file",
    image_file: {
      file_id: imageFile.id,
      detail: "high",
    },
  },
];
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content: content,
});
console.log(`Created message, message ID: ${message.id}`);
```

Create message with an image URL input

```
```

ts

```ts
// Specify the public image URL
const imageUrl =
  "https://github.com/Azure/azure-sdk-for-
js/blob/0aa88ceb18d865726d423f73b8393134e783aea6/sdk/ai/ai-
projects/data/image_file.png?raw=true";
// Create content directly referencing image URL
const inputMessage = "Hello, what is in the image?";
const content = [
  {
    type: "text",
    text: inputMessage,
  },
  {
    type: "image_url",
    image_url: {
      url: imageUrl,
      detail: "high",
    },
  },
];
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content: content,
});
console.log(`Created message, message ID: ${message.id}`);
```

Create message with base64-encoded image input

ts

```ts
function imageToBase64DataUrl(imagePath: string, mimeType: string): string {
  try {
    // Read the image file as binary
    const imageBuffer = fs.readFileSync(imagePath);
    // Convert to base64
    const base64Data = imageBuffer.toString("base64");
    // Format as a data URL
    return `data:${mimeType};base64,${base64Data}`;
  } catch (error) {
    console.error(`Error reading image file at ${imagePath}:`, error);
    throw error;
  }
}
// Convert your image file to base64 format
const imageDataUrl = imageToBase64DataUrl(filePath, "image/png");
// Create a message with both text and image content
const inputMessage = "Hello, what is in the image?";
const content = [
  {
    type: "text",
```

```
    text: inputMessage,
  },
  {
    type: "image_url",
    image_url: {
      url: imageDataUrl,
      detail: "high",
    },
  },
];
const message = await client.agents.createMessage(thread.id, {
  role: "user",
  content: content,
});
console.log(`Created message, message ID: ${message.id}`);
```

Create Run, Run_and_Process, or Stream

Here is an example of `createRun` and poll until the run is completed:

ts

```
let run = await client.agents.createRun(thread.id, agent.id);
// Poll the run as long as run status is queued or in progress
while (
  run.status === "queued" ||
  run.status === "in_progress" ||
  run.status === "requires_action"
) {
  // Wait for a second
  await new Promise((resolve) => setTimeout(resolve, 1000));
  run = await client.agents.getRun(thread.id, run.id);
}
```

To have the SDK poll on your behalf, use the `createThreadAndRun` method.

Here is an example:

ts

```
const run = await client.agents.createThreadAndRun(agent.id, {
  thread: {
    messages: [
      {
        role: "user",
        content: "hello, world!",
      },
    ],
```

```
  },
});
```

With streaming, polling also need not be considered.

Here is an example:

ts

```ts
const streamEventMessages = await client.agents.createRun(thread.id,
agent.id).stream();
```

Event handling can be done as follows:

ts

```ts
import {
  RunStreamEvent,
  ThreadRunOutput,
  MessageStreamEvent,
  MessageDeltaChunk,
  MessageDeltaTextContent,
  DoneEvent,
} from "@azure/ai-projects";

const streamEventMessages = await client.agents.createRun(thread.id,
agent.id).stream();
for await (const eventMessage of streamEventMessages) {
  switch (eventMessage.event) {
    case RunStreamEvent.ThreadRunCreated:
      console.log(`ThreadRun status: ${(eventMessage.data as
ThreadRunOutput).status}`);
      break;
    case MessageStreamEvent.ThreadMessageDelta:
      {
        const messageDelta = eventMessage.data as MessageDeltaChunk;
        messageDelta.delta.content.forEach((contentPart) => {
          if (contentPart.type === "text") {
            const textContent = contentPart as MessageDeltaTextContent;
            const textValue = textContent.text?.value || "No text";
            console.log(`Text delta received:: ${textValue}`);
          }
        });
      }
      break;
    case RunStreamEvent.ThreadRunCompleted:
      console.log("Thread Run Completed");
      break;
    case ErrorEvent.Error:
      console.log(`An error occurred. Data ${eventMessage.data}`);
      break;
```

```
    case DoneEvent.Done:
      console.log("Stream completed.");
      break;
  }
}
```

Retrieve Message

To retrieve messages from agents, use the following example:

ts

```ts
import { MessageContentOutput, isOutputOfType, MessageTextContentOutput } from
"@azure/ai-projects";

const messages = await client.agents.listMessages(thread.id);
while (messages.hasMore) {
  const nextMessages = await client.agents.listMessages(currentRun.threadId, {
    after: messages.lastId,
  });
  messages.data = messages.data.concat(nextMessages.data);
  messages.hasMore = nextMessages.hasMore;
  messages.lastId = nextMessages.lastId;
}
// The messages are following in the reverse order,
// we will iterate them and output only text contents.
for (const dataPoint of messages.data.reverse()) {
  const lastMessageContent: MessageContentOutput =
dataPoint.content[dataPoint.content.length - 1];
  console.log(lastMessageContent);
  if (isOutputOfType<MessageTextContentOutput>(lastMessageContent, "text")) {
    console.log(
      `${dataPoint.role}: ${(lastMessageContent as
MessageTextContentOutput).text.value}`,
    );
  }
}
```

Retrieve File

Files uploaded by Agents cannot be retrieved back. If your use case needs to access the file content uploaded by the Agents, you are advised to keep an additional copy accessible by your application. However, files generated by Agents are retrievable by `getFileContent`.

Here is an example retrieving file ids from messages:

ts

```ts
import {
  isOutputOfType,
  MessageTextContentOutput,
  MessageImageFileContentOutput,
} from "@azure/ai-projects";

const messages = await client.agents.listMessages(thread.id);
// Get most recent message from the assistant
const assistantMessage = messages.data.find((msg) => msg.role === "assistant");
if (assistantMessage) {
  const textContent = assistantMessage.content.find((content) =>
    isOutputOfType<MessageTextContentOutput>(content, "text"),
  ) as MessageTextContentOutput;
  if (textContent) {
    console.log(`Last message: ${textContent.text.value}`);
  }
}
const imageFile = (messages.data[0].content[0] as
MessageImageFileContentOutput).imageFile;
const imageFileName = (await client.agents.getFile(imageFile.fileId)).filename;
const fileContent = await (
  await client.agents.getFileContent(imageFile.fileId).asNodeStream()
).body;
if (fileContent) {
  const chunks: Buffer[] = [];
  for await (const chunk of fileContent) {
    chunks.push(Buffer.from(chunk));
  }
  const buffer = Buffer.concat(chunks);
  fs.writeFileSync(imageFileName, buffer);
} else {
  console.error("Failed to retrieve file content: fileContent is undefined");
}
console.log(`Saved image file to: ${imageFileName}`);
```

Teardown

To remove resources after completing tasks, use the following functions:

ts

```ts
await client.agents.deleteVectorStore(vectorStore.id);
console.log(`Deleted vector store, vector store ID: ${vectorStore.id}`);
await client.agents.deleteFile(file.id);
console.log(`Deleted file, file ID: ${file.id}`);
client.agents.deleteAgent(agent.id);
console.log(`Deleted agent, agent ID: ${agent.id}`);
```

Tracing

You can add an Application Insights Azure resource to your Azure AI Foundry project. See the Tracing tab in your studio. If one was enabled, you can get the Application Insights connection string, configure your Agents, and observe the full execution path through Azure Monitor. Typically, you might want to start tracing before you create an Agent.

Installation

Make sure to install OpenTelemetry and the Azure SDK tracing plugin via

Bash

```
npm install @opentelemetry/api \
  @opentelemetry/instrumentation \
  @opentelemetry/sdk-trace-node \
  @azure/opentelemetry-instrumentation-azure-sdk \
  @azure/monitor-opentelemetry-exporter
```

You will also need an exporter to send telemetry to your observability backend. You can print traces to the console or use a local viewer such as Aspire Dashboard.

To connect to Aspire Dashboard or another OpenTelemetry compatible backend, install OTLP exporter:

Bash

```
npm install @opentelemetry/exporter-trace-otlp-proto \
  @opentelemetry/exporter-metrics-otlp-proto
```

Tracing example

Here is a code sample to be included above `createAgent`:

ts

```ts
import {
  NodeTracerProvider,
  SimpleSpanProcessor,
  ConsoleSpanExporter,
} from "@opentelemetry/sdk-trace-node";
import { trace } from "@opentelemetry/api";
import { AzureMonitorTraceExporter } from "@azure/monitor-opentelemetry-exporter";

const provider = new NodeTracerProvider();
provider.addSpanProcessor(new SimpleSpanProcessor(new ConsoleSpanExporter()));
provider.register();
const tracer = trace.getTracer("Agents Sample", "1.0.0");
```

```ts
let appInsightsConnectionString =
  process.env.APP_INSIGHTS_CONNECTION_STRING ?? "<appInsightsConnectionString>";
if (appInsightsConnectionString == "<appInsightsConnectionString>") {
  appInsightsConnectionString = await client.telemetry.getConnectionString();
}
if (appInsightsConnectionString) {
  const exporter = new AzureMonitorTraceExporter({
    connectionString: appInsightsConnectionString,
  });
  provider.addSpanProcessor(new SimpleSpanProcessor(exporter));
}
await tracer.startActiveSpan("main", async (span) => {
  client.telemetry.updateSettings({ enableContentRecording: true });
  // ...
});
```

Troubleshooting

Exceptions

Client methods that make service calls raise an RestError for a non-success HTTP status code response from the service. The exception's `code` will hold the HTTP response status code. The exception's `error.message` contains a detailed message that may be helpful in diagnosing the issue:

ts

```ts
import { RestError } from "@azure/core-rest-pipeline";

try {
  const result = await client.connections.listConnections();
} catch (e) {
  if (e instanceof RestError) {
    console.log(`Status code: ${e.code}`);
    console.log(e.message);
  } else {
    console.error(e);
  }
}
```

For example, when you provide wrong credentials:

text

```
Status code: 401 (Unauthorized)
Operation returned an invalid status 'Unauthorized'
```

Reporting issues

To report issues with the client library, or request additional features, please open a GitHub issue here

Next steps

Have a look at the package samples folder, containing fully runnable code.

Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit https://cla.microsoft.com.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information, see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Azure AI Agent Service monitoring data reference

Article • 04/28/2025

This article contains all the monitoring reference information for this service.

See Monitor Azure AI Agent Service for details on the data you can collect on your agents.

Metrics

Here are the most important metrics we think you should monitor for Azure AI Agent Service. Later in this article is a longer list of all available metrics which contains more details on metrics in this shorter list. *See the below list for most up to date information. We're working on refreshing the tables in the following sections.*

- Category: Agents

Supported metrics

This section lists all the automatically collected platform metrics for this service. These metrics are also part of the global list of all platform metrics supported in Azure Monitor.

- All columns might not be present in every table.
- Some columns might be beyond the viewing area of the page. Select **Expand table** to view all available columns.

Table headings

- **Category** - The metrics group or classification.
- **Metric** - The metric display name as it appears in the Azure portal.
- **Name in REST API** - The metric name as referred to in the REST API.
- **Unit** - Unit of measure.
- **Aggregation** - The default aggregation type. Valid values: Average (Avg), Minimum (Min), Maximum (Max), Total (Sum), Count.
- **Dimensions** - Dimensions available for the metric.
- **Time Grains** - Intervals at which the metric is sampled. For example, `PT1M` indicates that the metric is sampled every minute, `PT30M` every 30 minutes, `PT1H` every hour, and so on.
- **DS Export**- Whether the metric is exportable to Azure Monitor Logs via diagnostic settings. For information on exporting metrics, see Create diagnostic settings in Azure Monitor.

Category: Agents

Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Agents	`Agents`	Count	Average, Maximum, Minimum, Total (Sum)	`EventType`	PT1M	No

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Number of events for AI Agents in this workspace						
IndexedFiles Number of files indexed for file search in this workspace	`IndexedFiles`	Count	Average, Maximum, Minimum, Total (Sum)	`ErrorCode`, `Status`, `VectorStoreId`	PT1M	No
Messages Number of events for AI Agent messages in this workspace	`Messages`	Count	Average, Maximum, Minimum, Total (Sum)	`EventType`, `ThreadId`	PT1M	No
Runs Number of runs by AI Agents in this workspace	`Runs`	Count	Average, Maximum, Minimum, Total (Sum)	`AgentId`, `RunStatus`, `StatusCode`, `StreamType`	PT1M	No
Threads Number of events for AI Agent threads in this workspace	`Threads`	Count	Average, Maximum, Minimum, Total (Sum)	`EventType`	PT1M	No
Tokens Count of tokens by AI Agents in this workspace	`Tokens`	Count	Average, Maximum, Minimum, Total (Sum)	`AgentId`, `TokenType`	PT1M	No
ToolCalls Tool calls made by AI Agents in this workspace	`ToolCalls`	Count	Average, Maximum, Minimum, Total (Sum)	`AgentId`, `ToolName`	PT1M	No

Category: Model

Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Model Deploy Failed Number of model deployments that failed in this workspace	`Model Deploy Failed`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `StatusCode`	PT1M	Yes
Model Deploy Started Number of model deployments started in this workspace	`Model Deploy Started`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Model Deploy Succeeded Number of model deployments that succeeded in this workspace	`Model Deploy Succeeded`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`	PT1M	Yes
Model Register Failed Number of model registrations that failed in this workspace	`Model Register Failed`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `StatusCode`	PT1M	Yes
Model Register Succeeded Number of model registrations that succeeded in this workspace	`Model Register Succeeded`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`	PT1M	Yes

Category: Quota

Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Active Cores Number of active cores	`Active Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Active Nodes Number of Acitve nodes. These are the nodes which are actively running a job.	`Active Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Idle Cores Number of idle cores	`Idle Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Idle Nodes Number of idle nodes. Idle nodes are the nodes which are not running any jobs but can accept new job if available.	`Idle Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Leaving Cores Number of leaving cores	`Leaving Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Leaving Nodes Number of leaving nodes. Leaving nodes are the nodes which just finished processing a job and will go to Idle state.	`Leaving Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Preempted Cores Number of preempted cores	`Preempted Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Preempted Nodes Number of preempted nodes. These nodes are the low priority nodes which are taken away from the available node pool.	`Preempted Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Quota Utilization Percentage Percent of quota utilized	`Quota Utilization Percentage`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`, `VmFamilyName`, `VmPriority`	PT1M	Yes
Total Cores Number of total cores	`Total Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Total Nodes Number of total nodes. This total includes some of Active Nodes, Idle Nodes, Unusable Nodes, Premepted Nodes, Leaving Nodes	`Total Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Unusable Cores Number of unusable cores	`Unusable Cores`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes
Unusable Nodes Number of unusable nodes. Unusable nodes are not functional due to some unresolvable issue. Azure will recycle these nodes.	`Unusable Nodes`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `ClusterName`	PT1M	Yes

Category: Resource

Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
CpuCapacityMillicores Maximum capacity of a CPU node in millicores. Capacity is aggregated in one minute intervals.	`CpuCapacityMillicores`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
CpuMemoryCapacityMegabytes Maximum memory utilization of a CPU node in megabytes. Utilization is aggregated in one minute intervals.	`CpuMemoryCapacityMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
CpuMemoryUtilizationMegabytes Memory utilization of a CPU node in megabytes. Utilization is aggregated in one minute intervals.	`CpuMemoryUtilizationMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
CpuMemoryUtilizationPercentage Memory utilization percentage of a CPU node. Utilization is aggregated in one minute intervals.	`CpuMemoryUtilizationPercentage`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
CpuUtilization Percentage of utilization on a CPU node. Utilization is reported at one minute intervals.	`CpuUtilization`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `runId`, `NodeId`, `ClusterName`	PT1M	Yes
CpuUtilizationMillicores Utilization of a CPU node in millicores. Utilization is aggregated in one minute intervals.	`CpuUtilizationMillicores`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
CpuUtilizationPercentage Utilization percentage of a CPU node. Utilization is aggregated in one minute intervals.	`CpuUtilizationPercentage`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
DiskAvailMegabytes Available disk space in megabytes. Metrics are aggregated in one minute intervals.	`DiskAvailMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
DiskReadMegabytes Data read from disk in megabytes.	`DiskReadMegabytes`	Count	Average, Maximum,	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Metrics are aggregated in one minute intervals.			Minimum, Total (Sum)			
DiskUsedMegabytes Used disk space in megabytes. Metrics are aggregated in one minute intervals.	`DiskUsedMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
DiskWriteMegabytes Data written into disk in megabytes. Metrics are aggregated in one minute intervals.	`DiskWriteMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
GpuCapacityMilliGPUs Maximum capacity of a GPU device in milli-GPUs. Capacity is aggregated in one minute intervals.	`GpuCapacityMilliGPUs`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
GpuEnergyJoules Interval energy in Joules on a GPU node. Energy is reported at one minute intervals.	`GpuEnergyJoules`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `runId`, `rootRunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
GpuMemoryCapacityMegabytes Maximum memory capacity of a GPU device in megabytes. Capacity aggregated in at one minute intervals.	`GpuMemoryCapacityMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
GpuMemoryUtilization Percentage of memory utilization on a GPU node. Utilization is reported at one minute intervals.	`GpuMemoryUtilization`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `runId`, `NodeId`, `DeviceId`, `ClusterName`	PT1M	Yes
GpuMemoryUtilizationMegabytes Memory utilization of a GPU device in megabytes. Utilization aggregated in at one minute intervals.	`GpuMemoryUtilizationMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
GpuMemoryUtilizationPercentage Memory utilization percentage of a GPU device. Utilization aggregated in at one minute intervals.	`GpuMemoryUtilizationPercentage`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
GpuUtilization Percentage of utilization on a GPU node. Utilization is reported at one minute intervals.	`GpuUtilization`	Count	Average, Maximum, Minimum, Total (Sum)	`Scenario`, `runId`, `NodeId`, `DeviceId`, `ClusterName`	PT1M	Yes
GpuUtilizationMilliGPUs Utilization of a GPU device in milli-GPUs. Utilization is aggregated in one minute intervals.	`GpuUtilizationMilliGPUs`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
GpuUtilizationPercentage Utilization percentage of a GPU device. Utilization is aggregated in one minute intervals.	`GpuUtilizationPercentage`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `DeviceId`, `ComputeName`	PT1M	Yes
IBReceiveMegabytes Network data received over InfiniBand in megabytes. Metrics are aggregated in one minute intervals.	`IBReceiveMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`, `DeviceId`	PT1M	Yes
IBTransmitMegabytes Network data sent over InfiniBand in megabytes. Metrics are aggregated in one minute intervals.	`IBTransmitMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`, `DeviceId`	PT1M	Yes
NetworkInputMegabytes Network data received in megabytes. Metrics are aggregated in one minute intervals.	`NetworkInputMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`, `DeviceId`	PT1M	Yes
NetworkOutputMegabytes Network data sent in megabytes. Metrics are aggregated in one minute intervals.	`NetworkOutputMegabytes`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`, `DeviceId`	PT1M	Yes
StorageAPIFailureCount Azure Blob Storage API calls failure count.	`StorageAPIFailureCount`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes
StorageAPISuccessCount Azure Blob Storage API calls success count.	`StorageAPISuccessCount`	Count	Average, Maximum, Minimum, Total (Sum)	`RunId`, `InstanceId`, `ComputeName`	PT1M	Yes

Category: Run

Expand table

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Cancel Requested Runs Number of runs where cancel was requested for this workspace. Count is updated when cancellation request has been received for a run.	`Cancel Requested Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Cancelled Runs Number of runs cancelled for this workspace. Count is updated when a run is successfully cancelled.	`Cancelled Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Completed Runs Number of runs completed successfully for this workspace. Count is updated when a run has completed and output has been collected.	`Completed Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Errors Number of run errors in this workspace. Count is updated whenever run encounters an error.	`Errors`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`	PT1M	Yes
Failed Runs Number of runs failed for this workspace. Count is updated when a run fails.	`Failed Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Finalizing Runs Number of runs entered finalizing state for this workspace. Count is updated when a run has completed but output collection still in progress.	`Finalizing Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Not Responding Runs Number of runs not responding for this workspace. Count is updated when a run enters Not Responding state.	`Not Responding Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Not Started Runs Number of runs in Not Started state for this workspace. Count is updated when a request is received to create a run but run information has not yet been populated.	`Not Started Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Preparing Runs Number of runs that are preparing for this workspace. Count is updated when a run enters Preparing state while the run environment is being prepared.	`Preparing Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Provisioning Runs Number of runs that are provisioning for this workspace. Count is updated when a run is waiting on compute target creation or provisioning.	`Provisioning Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Queued Runs Number of runs that are queued for this workspace. Count is updated when a run is queued in compute target. Can occure when waiting for required compute nodes to be ready.	`Queued Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Started Runs Number of runs running for this workspace. Count is updated when run starts running on required resources.	`Started Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes

Metric	Name in REST API	Unit	Aggregation	Dimensions	Time Grains	DS Export
Starting Runs Number of runs started for this workspace. Count is updated after request to create run and run info, such as the Run Id, has been populated	`Starting Runs`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`, `RunType`, `PublishedPipelineId`, `ComputeType`, `PipelineStepType`, `ExperimentName`	PT1M	Yes
Warnings Number of run warnings in this workspace. Count is updated whenever a run encounters a warning.	`Warnings`	Count	Total (Sum), Average, Minimum, Maximum, Count	`Scenario`	PT1M	Yes

Related content

- See Monitor Azure AI Agent Service for a description of monitoring Azure AI Agent Service.
- See Monitor Azure resources with Azure Monitor for details on monitoring Azure resources.

Azure AI services support and help options

Article • 05/02/2025

Here are the options for getting support, staying up to date, giving feedback, and reporting bugs for Azure AI services.

Get solutions to common issues

In the Azure portal, you can find answers to common AI service issues.

1. Go to your Azure AI services resource in the Azure portal. You can find it on the list on this page: Azure AI services. If you're a United States government customer, use the Azure portal for the United States government.
2. In the left pane, under **Help**, select **Support + Troubleshooting**.
3. Describe your issue in the text box, and answer the remaining questions in the form.
4. You'll find Learn articles and other resources that might help you resolve your issue.

Create an Azure support request

Explore the range of Azure support options and choose the plan that best fits, whether you're a developer just starting your cloud journey or a large organization deploying business-critical, strategic applications. Azure customers can create and manage support requests in the Azure portal.

To submit a support request for Azure AI services, follow the instructions on the New support request page in the Azure portal. After choosing your **Issue type**, select **Cognitive Services** in the **Service type** dropdown field.

Post a question on Microsoft Q&A

For quick and reliable answers on your technical product questions from Microsoft Engineers, Azure Most Valuable Professionals (MVPs), or our expert community, engage with us on Microsoft Q&A, Azure's preferred destination for community support.

If you can't find an answer to your problem using search, submit a new question to Microsoft Q&A. Use one of the following tags when you ask your question:

- Azure AI services

Azure OpenAI

- Azure OpenAI

Vision

- Azure AI Vision
- Azure AI Custom Vision
- Azure Face
- Azure AI Document Intelligence
- Video Indexer

Language

- Azure AI Immersive Reader
- Language Understanding (LUIS)
- Azure QnA Maker
- Azure AI Language
- Azure Translator

Speech

- Azure AI Speech

Decision

- Azure AI Anomaly Detector
- Content Moderator
- Azure AI Metrics Advisor
- Azure AI Personalizer

Post a question to Stack Overflow

For answers to your developer questions from the largest community developer ecosystem, ask your question on Stack Overflow.

If you submit a new question to Stack Overflow, use one or more of the following tags when you create the question:

- Azure AI services

Azure OpenAI

- Azure OpenAI

Vision

- Azure AI Vision
- Azure AI Custom Vision
- Azure Face
- Azure AI Document Intelligence
- Video Indexer

Language

- Azure AI Immersive Reader
- Language Understanding (LUIS)
- Azure QnA Maker
- Azure AI Language service
- Azure Translator

Speech

- Azure AI Speech service

Decision

- Azure AI Anomaly Detector
- Content Moderator
- Azure AI Metrics Advisor
- Azure AI Personalizer

Submit feedback

To request new features, post them on `https://feedback.azure.com`. Share your ideas for making Azure AI services and its APIs work better for the applications you develop.

- Azure AI services

Vision

- Azure AI Vision
- Azure AI Custom Vision
- Azure Face
- Azure AI Document Intelligence
- Video Indexer

Language

- Azure AI Immersive Reader
- Language Understanding (LUIS)

- Azure QnA Maker
- Azure AI Language
- Azure Translator

Speech

- Azure AI Speech service

Decision

- Azure AI Anomaly Detector
- Content Moderator
- Azure AI Metrics Advisor
- Azure AI Personalizer

Stay informed

You can learn about the features in a new release or get the latest news on the Azure blog. Staying informed can help you find the difference between a programming error, a service bug, or a feature not yet available in Azure AI services.

- Learn more about product updates, roadmap, and announcements in Azure Updates.
- News about Azure AI services is shared in the Azure AI blog.
- Join the conversation on Reddit about Azure AI services.

Next step

What are Azure AI services?