Microsoft 365 Agents SDK documentation

Microsoft 365 Agents SDK provides an integrated environment that is purpose-built for agent development.

About the SDK

OVERVIEW

What is the Microsoft 365 Agents SDK?

Choose the right agent solution

Get started

QUICKSTART

Create a basic agent

Create .NET agent in Visual Studio with Microsoft 365 Agents Toolkit

Create JavaScript agent in Visual Studio Code with Microsoft 365 Agents Toolkit

SAMPLE

Agents SDK samples on GitHub (.NET and JavaScript)

Agents SDK Python samples on GitHub

Deploy and test your agent

HOW-TO GUIDE

Test your Agents SDK agent

Test your agent locally in Agents Playground

Deploy your agent to Azure and register with Azure Bot Service manually

Languages

REFERENCE

.NET SDK

JavaScript SDK

Python SDK

Microsoft 365 Agents SDK overview (preview)

Article • 05/12/2025

[This article is prerelease documentation and is subject to change.]

With the Microsoft 365 Agents SDK, you can create agents deployable to channels of your choice, such as Microsoft 365 Copilot, Microsoft Teams, Web & Custom Apps and more, with scaffolding to handle the required communication. Developers can use the AI Services of their choice, and make the agents they build available using the channel management capabilities of the SDK.

Key features of the Agents SDK

Developers need the flexibility to integrate agents from any provider or technology stack into their enterprise systems. The Agents SDK simplifies the implementation of agentic patterns using the AI of their choice, allowing them to select one or more services, models, or agents to meet their specific requirements.

Use the Agents SDK to:

1. Quickly build an agent 'container' with state, storage, and the ability to manage activities and events. Deploy this container across any channel, such as Microsoft 365 Copilot or Microsoft Teams.
2. Implement agentic patterns without being restricted to a specific technology stack. The Agents SDK is agnostic regarding the AI you choose.
3. Customize your agent to align with the specific behaviors of clients, such as Microsoft Teams.

Supported languages

The Agents SDK is currently in Public Preview and supports:

- C# using the .NET 8.0 SDK
- JavaScript using Node.js version 18 and above
- Python 3.9 to 3.11

Create an agent

It is easy to get the starter sample in C#, JavaScript, or Python from Github

To create an agent in C#:

C#

```
builder.AddAgent( sp =>
{
    var agent = new
AgentApplication(sp.GetRequiredService<AgentApplicationOptions>());
    agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken) =>
    {
        var text = turnContext.Activity.Text;
        await turnContext.SendActivityAsync(MessageFactory.Text($"Echo: {text}"),
cancellationToken);
    });
});
```

This creates a new agent, listens for a message type activity and sends a message back

From here, you can add your chosen custom AI Services (e.g Azure Foundry or OpenAI Agents) & Orchestration (e.g. Semantic Kernel).

Important terms

Some specific concepts that are important to the SDK are:

- `Turns` - A turn is a unit of work that is done by the agent. It can be a single message or a series of messages. Developers will work with 'turns' and manage the data between them
- `Activity` - An activity is one or more type of unit that is managed by the agent
- `Messages` - A message is a type of activity that is sent to the agent. It can be a single message or a series of messages.

Get Started

Before you get started, you need to take care of some prerequisites. The prerequisites depend on the language you are using to develop your application.

C#

- .NET 8.0 SDK
- Bot Framework Emulator
- Knowledge of ASP.NET Core and asynchronous programming in C#

Download and install

Download the files needed to get started.

C#

To get started with C#/.NET, clone the Agents GitHub repo repo locally. The repo contains SDK source libraries and samples to help you get started building applications using the SDK. Installing the samples installs needed packages for the SDK.

Next steps

- Quickstart
- Building agents with Agents SDK

Choose the right agent solution to support your use case

Article • 05/20/2025

Microsoft provides products and services for build agents, with different audiences and capabilities depending on your requirements & use cases.The following sections describe each product in more detail:

Expand table

Product	Audience	Description
Copilot Studio	Fusion teams, citizen developers	Copilot Studio is an end-to-end agent-building tool, with built-in natural language understanding models, data connectivity through Power Automates, and support for multiple channels.
Microsoft 365 Agents SDK	Developers	Provides a framework for building agents, including tools, templates, and related AI services. The SDK is ideal for developers who want to build agents that are publicly available on the Microsoft Teams app store.
Azure AI Foundry SDK	Developers	Azure agents provide agent development capabilities under a unified Azure AI SDK.
Semantic Kernel SDK	Developers	The Semantic Kernel SDK allows developers to build agents and integrates various AI services to define orchestration chains for rapid development.

Copilot Studio

Copilot Studio is a tool for agent development included in Microsoft Power Platform —a business-application platform that incorporates data analysis, solution building, and process automation. You don't need to write code or understand the details of the underlying AI technologies to build agents in Copilot Studio. Agents built in Copilot Studio can apply automation and other capabilities within the Power Platform, and you can rapidly develop sophisticated agent experiences.

You can connect agents to various user platforms, such as Microsoft 365 and Microsoft Dynamics 365 .

For more information about Copilot Studio, see the product overview page . For details about pricing, see Copilot Studio pricing .

Microsoft 365 Agents SDK

With the Microsoft 365 Agents SDK, you can create agents deployable to channels of your choice, such as Microsoft 365 Copilot, Microsoft Teams, Web & Custom Apps and more, with scaffolding to handle the required communication. Developers can use the AI Services of their choice, and make the agents they build available using the channel management capabilities of the SDK.

Use the Agents SDK to:

1. Quickly build an agent 'container' with state, storage, and the ability to manage activities and events. Deploy this container across any channel, such as Microsoft 365 Copilot or Microsoft Teams.
2. Implement agentic patterns without being restricted to a specific technology stack. The Agents SDK is agnostic regarding the AI you choose.
3. Customize your agent to align with the specific behaviors of clients, such as Microsoft Teams.

The Agents SDK utilizes the Azure Bot Service. For details about pricing, see Azure AI Bot Service pricing. Costs vary depending on what AI services are used for implementing and hosting your code.

Azure AI Foundry SDK

Azure AI Foundry includes many development capabilities under a unified Azure AI SDK tailored to suit specific needs or work with specific services. Azure Agents is the name of the feature within the Unified Azure AI SDK that also includes fine-tuning and evaluations.

The agent capabilities include API layers such as Multi-agent API, Assistants API, Responses API and Completion API. These API layer provide various amounts of capabilities to support multi-agent, RAG, and AI services.

Developers can add and use Azure Agents APIs from Assistants and broader APIs including Azure OpenAI Services and Azure Cognitive Services to add functionality and capabilities to the agent.

Semantic Kernel SDK

Semantic Kernel SDK allows developers to build agents and integrates various AI services to define orchestration chains for rapid development.

As with Azure Agents, developers can choose to use Semantic Kernel in the middleware or agent logic components of an Agents SDK agent to add capabilities such as the ability to easily orchestrate between APIs or plugins.

Skills in Copilot Studio (formerly Bot Framework skills)

Copilot Studio provides a range of functionality to create different types of agents, including copilot agents to extend Copilot for Microsoft 365.

A skill is developed, and deployed and hosted (for example, on Azure), and then configured within Copilot Studio. Here is some additional information about skills:

- About skills in the SDK
- Use a classic chatbot as a skill
- Configure a Skill
- Implement a skill with the Agents SDK

Quickstart: Create an agent with the Agents SDK

07/18/2025

> **Note**
>
> The Agents SDK is generally available ((GA)) for C#/.NET and JavaScript/Node.js. Python is planned to be GA in June 2025.

This example shows you how to:

- Download and run the sample agent code from GitHub sample repos
- Create an agent using the Agents SDK
- Add events and responses
- Send messages
- Add AI services and orchestration

If you use the sample empty agent sample, the agent echoes the message you enter. This lets get started with the SDK. It isn't intended for use in production scenarios.

Prerequisites

You need a few things before you get started.

C#

- .NET 8.0 SDK
- Visual Studio or Visual Studio Code
- Knowledge of ASP.NET Core and asynchronous programming in C#
- Download the Empty Agent/Echo Agent sample from GitHub

Set up and run the sample agent

There are three ways to get started:

- Clone and run the Empty/echo agent sample available on GitHub
- Use the Microsoft 365 Agents Toolkit. The Agents Toolkit has two built-in templates for Visual Studio and Visual Studio Code that use the Microsoft 365 Agents SDK for starting

with an Echo/Empty Agent and a Weather Agent that uses Azure Foundry or OpenAI Services with either Semantic Kernel or LangChain
- Use the CLI, as shown in the quickstart

The steps depend on the language you're using. Follow the instructions for your language of choice.

C#

1. Open the solution file `EmptyAgent.csproj` in Visual Studio.
2. Run the project without debugging.
3. Visual Studio builds the application, and runs it in localhost, opening an application page in a browser. The application page should be blank, with some text. You should also see a Windows Terminal window open with messages indicating that the application is running.

At this point, your agent is running locally on the port specified in the terminal window (5000, by default).

The app creates a simple agent that echoes back the messages you send to it. You can now test your agent locally and build on it as desired.

Create an Agents SDK agent using CLI and code

You can also set up an agent using the Microsoft 365 Agents SDK directly via command line and code editor. Instructions are provided for .NET.

Set up a new project

First, create a new .NET Console Application project.

Bash

```
mkdir my_project
cd my_project
dotnet new console -n myexample
cd myexample
```

Next, install the Microsoft 365 Agents SDK:

Bash

```
dotnet add package microsoft.agents.hosting.aspnet.core
```

You may need to update the Agents SDK packages from NuGet.

Next, build, and run your project.

Bash

```
dotnet build
dotnet run
```

You should now see your console app running which outputs "Hello World." You can also navigate to your new folder and open the .csproj file you created and run it from Visual Studio (myexample.csproj).

With the project created, and Agents SDK libraries added to the project, you can now add elements from the Microsoft 365 Agents SDK.

Create an Agents SDK agent

An Agents SDK agent acts like a container (AgentApplication) and can also be created with other client services (for example, Microsoft Teams).

To begin, create some initial setup.

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddHttpClient();

builder.Services.AddSingleton<IStorage, MemoryStorage>();
builder.AddAgentApplicationOptions();
```

Next, add your agent:

C#

```
builder.AddAgent( sp =>
{
    // Setup the Agent.
    var agent = new
```

```
AgentApplication(sp.GetRequiredService<AgentApplicationOptions>());

    // Respond to message events.
    // We will add an event in the next step in 'Events and Responses' below
    return agent;
});
```

Finish building HTTP components:

C#

```
var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseRouting();
app.MapPost("/api/messages", async (HttpRequest request, HttpResponse response,
IAgentHttpAdapter adapter, IAgent agent, CancellationToken cancellationToken) =>
{
    await adapter.ProcessAsync(request, response, agent, cancellationToken);
});


// Setup port and listening address.
var port = args.Length > 0 ? args[0] : "5000";
app.Urls.Add($"http://localhost:{port}");

// Start listening.
await app.RunAsync();
```

This agent also listens for a message type activity and sends a message back

The container itself is an 'agent' implementing your chosen AI Service (for example, Azure OpenAI). The container manages the conversation using turns and activities.

Add events and responses

Use the `OnActivity` event to trigger when a new activity is sent from the client (for example, a user message).

Add your AI service or chat endpoint & use the `SendActivityAsync` method to send a response back to the requesting client (which can be from an AI service):

C#

```
    agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken) =>
    {
        var text = turnContext.Activity.Text;
        await turnContext.SendActivityAsync(MessageFactory.Text($"Echo: {text}"),
```

```
cancellationToken);
    });
```

You need to update the packages from NuGet. Specifically, pull the Hosting package and it also adds dependencies. If you want to add auth, additionally install the MSAL package `microsoft.agents.authentication.msal`.

To learn more about the agent and some of the basics, review the following sections.

Send Messages

The Agents SDK makes it easy to send different types of messages back, when required. The support for different types of messages saves developers time formatting different types, and takes advantage of common types of messages supported by a broad set of clients.

`Messages` can be sent using `MessageFactory` that supports types like text, images, cards, attachments, adaptive cards, and more.

`Messages` are constructed and sent back within the `SendActivityAsync` method:

C#

```
    await turnContext.SendActivityAsync(MessageFactory.Text($"{response}"),
cancellationToken);
```

Add AI services and orchestration to your agent

If you begin with the Empty/Echo agent sample or build an agent using the CLI, you need to add AI services and orchestration to enable more sophisticated behavior. A key feature of the Agents SDK is that you can make your choice of AI services and orchestrator and integrate these into your agent using their SDKs. Choose from Azure AI Foundry, Semantic Kernel, LangChain and more.

C#

```
      var builder = Kernel.CreateBuilder();

      builder.AddAzureOpenAIChatCompletion(
      deploymentName: "gpt-4o-mini",
      endpoint: "url",
      apiKey: apiKey,
      apiVersion: apiVersion);
```

```
        var kernel = builder.Build();
        var result = await kernel.InvokePromptAsync(userInput);
```

Once there's a response or behavior from the orchestrator you want to send back to the user or service, you can send the response in the `SendActivityAsync` method.

C#

```
        await turnContext.SendActivityAsync(MessageFactory.Text(result),
cancellationToken);
```

There are multiple patterns to integrate AI services and orchestrators into your agent, depending on the functionality they bring and how you want to surface them in your agent. Check out an example of integration where we integrate Semantic Kernel directly into the agent at runtime.

Test your agent locally

C#

Next, start your agent locally. We provide instructions for Visual Studio and Visual Studio Code / CLI.

Quickstart: Create an agent with the Agents SDK for JavaScript

07/25/2025

This quickstart guides you through creating a custom engine agent that just replies with whatever you send to it.

Prerequisites

- Node.js v22 or newer
- A code editor of your choice. These instructions use Visual Studio Code.
- A client to interact with the agent, such as the Agents Playground
- An Azure subscription to deploy to Azure Bot Services (optional)

Initialize the project and install the SDK

Use `npm` to initialize a node.js project by creating a package.json and installing the required dependencies

1. Open a terminal and create a new folder

```
mkdir echo
cd echo
```

2. Initialize the node.js project

```
npm init -y
```

3. Install the Agents SDK

```
npm install @microsoft/agents-hosting-express
```

4. Open the folder using Visual Studio Code using the this command:

```
code .
```

Import the required libraries

Create the file `index.mjs` and import the following NPM packages into your application code:

- @microsoft/agents-hosting
- @microsoft/agents-hosting-express

JavaScript

```
// index.mjs
import { startServer } from '@microsoft/agents-hosting-express'
import { AgentApplication, MemoryStorage } from '@microsoft/agents-hosting'
```

Implement the EchoAgent as an AgentApplication

In `index.mjs`, add the following code to create the `EchoAgent` extending the AgentApplication, and implement three routes to respond to three events:

- Conversation Update
- the message `/help`
- any other activity

JavaScript

```
class EchoAgent extends AgentApplication {
  constructor (storage) {
    super({ storage })

    this.onConversationUpdate('membersAdded', this._help)
    this.onMessage('/help', this._help)
    this.onActivity('message', this._echo)
  }

  _help = async context =>
    await context.sendActivity(`Welcome to the Echo Agent sample 🚀.
      Type /help for help or send a message to see the echo feature in action.`)

  _echo = async (context, state) => {
    let counter= state.getValue('conversation.counter') || 0
    await context.sendActivity(`[${counter++}]You said: ${context.activity.text}`)
    state.setValue('conversation.counter', counter)
```

```
  }
}
```

Start the web server to listen in localhost:3978

At the end of `index.mjs` start the web server using startServer based on express using MemoryStorage as the turn state storage.

JavaScript

```
startServer(new EchoAgent(new MemoryStorage()))
```

Run the Agent locally in anonymous mode

From your terminal, run this command:

```
node index.mjs
```

The terminal should return this:

```
Server listening to port 3978 on sdk 0.6.18 for appId undefined debug undefined
```

Test the agent locally

1. From another terminal (to keep the agent running) install the Microsoft 365 Agents Playground with this command:

   ```
   npm install -D @microsoft/teams-app-test-tool
   ```

   The terminal should return something like:

   ```
   added 1 package, and audited 130 packages in 1s
   ```

```
19 packages are looking for funding
run `npm fund` for details

found 0 vulnerabilities
```

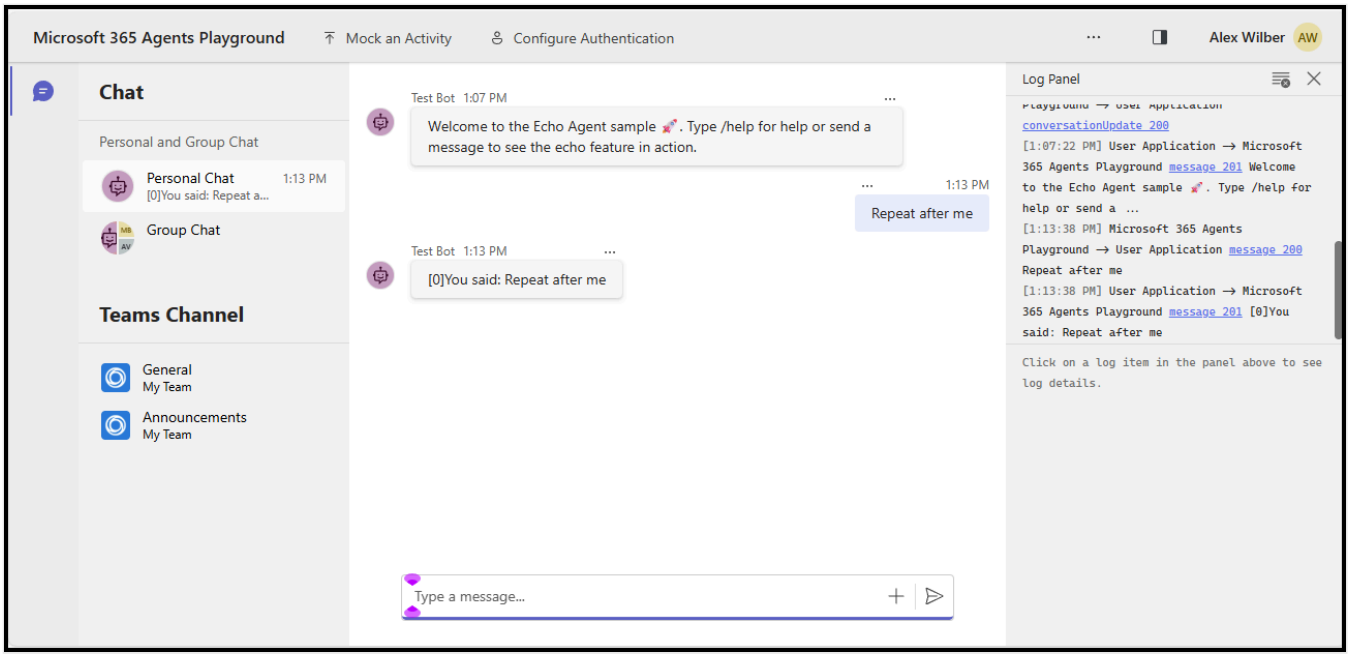
2. Run the test tool to interact with your agent using this command:

```
node_modules/.bin/teamsapptester
```

   The terminal should return something like:

```
Telemetry: agents-playground-cli/serverStart {"cleanProperties":{"options":"
{\"configFileOptions\":{\"path\":\"<REDACTED: user-file-
path>\"},\"appConfig\":{},\"port\":56150,\"disableTelemetry\":false}"}}

Telemetry: agents-playground-cli/cliStart {"cleanProperties":
{"isExec":"false","argv":"<REDACTED: user-file-path>,<REDACTED: user-file-
path>"}}

Listening on 56150
Microsoft 365 Agents Playground is being launched for you to debug the app:
http://localhost:56150
started web socket client
started web socket client
Waiting for connection of endpoint: http://127.0.0.1:3978/api/messages
waiting for 1 resources: http://127.0.0.1:3978/api/messages
wait-on(37568) complete
Telemetry: agents-playground-server/getConfig {"cleanProperties":
{"internalConfig":"{\"locale\":\"en-
US\",\"localTimezone\":\"America/Los_Angeles\",\"channelId\":\"msteams\"}"}}

Telemetry: agents-playground-server/sendActivity {"cleanProperties":
{"activityType":"installationUpdate","conversationId":"5305bb42-59c9-4a4c-
a2b6-e7a8f4162ede","headers":"{\"x-ms-agents-playground\":\"true\"}"}}

Telemetry: agents-playground-server/sendActivity {"cleanProperties":
{"activityType":"conversationUpdate","conversationId":"5305bb42-59c9-4a4c-
a2b6-e7a8f4162ede","headers":"{\"x-ms-agents-playground\":\"true\"}"}}
```

The `teamsapptester` command opens your default browser and connects to your agent.



Now you can send any message to see the echo reply, or send the message `/help` to see how that message is routed to the `_help` handler.

Create .NET agents in Visual Studio using the Microsoft 365 Agents Toolkit

Article • 05/28/2025

In this article, you learn how to create a new Agents SDK .NET project in Visual Studio, using the Microsoft 365 Agents Toolkit.

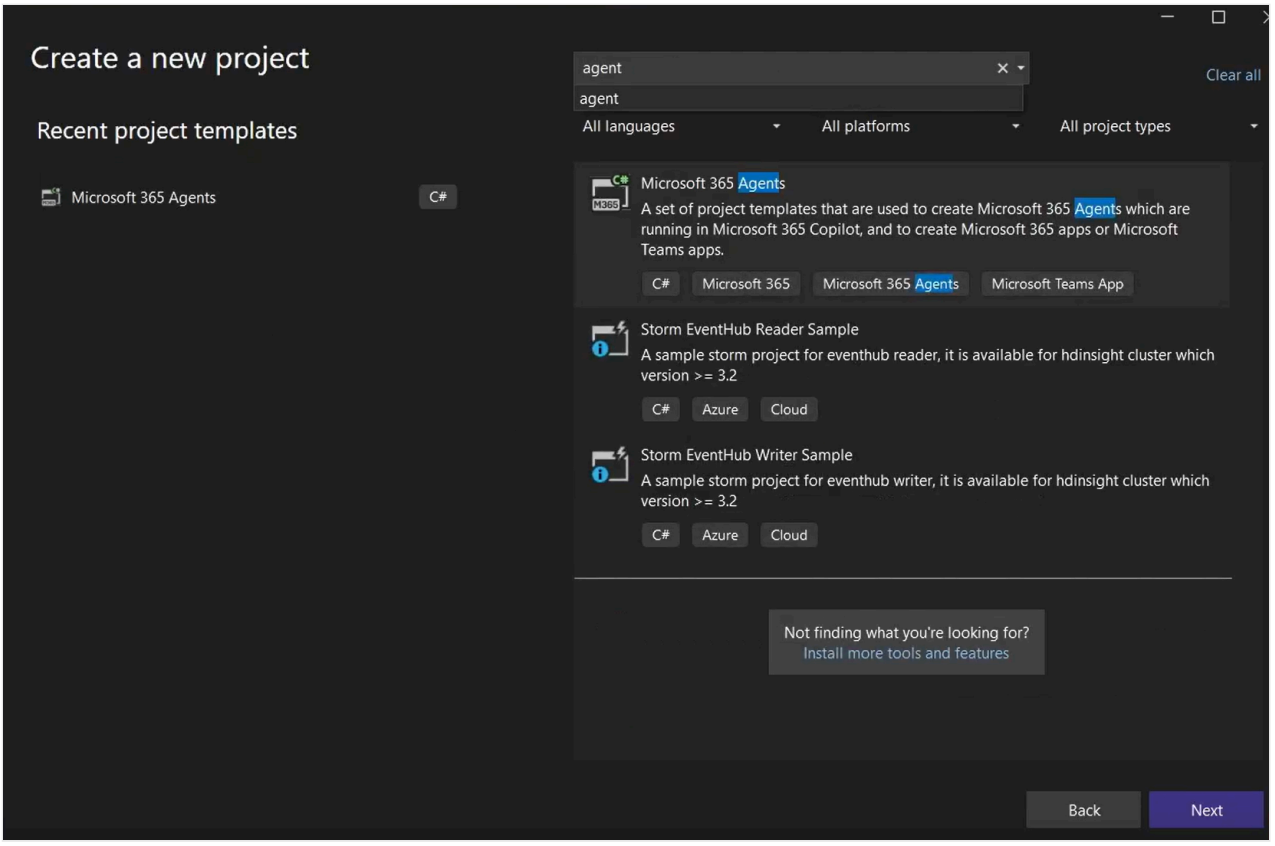
> ⓘ **Note**
>
> Before you begin, you need to **install the Agents Toolkit extension** for Visual Studio.

Create a new project

The Agents Toolkit provides a project template to help you get started with building an agent. You can start from a template in the toolkit or from samples in the Agents SDK. This document focuses on the templates available in the Agents Toolkit.

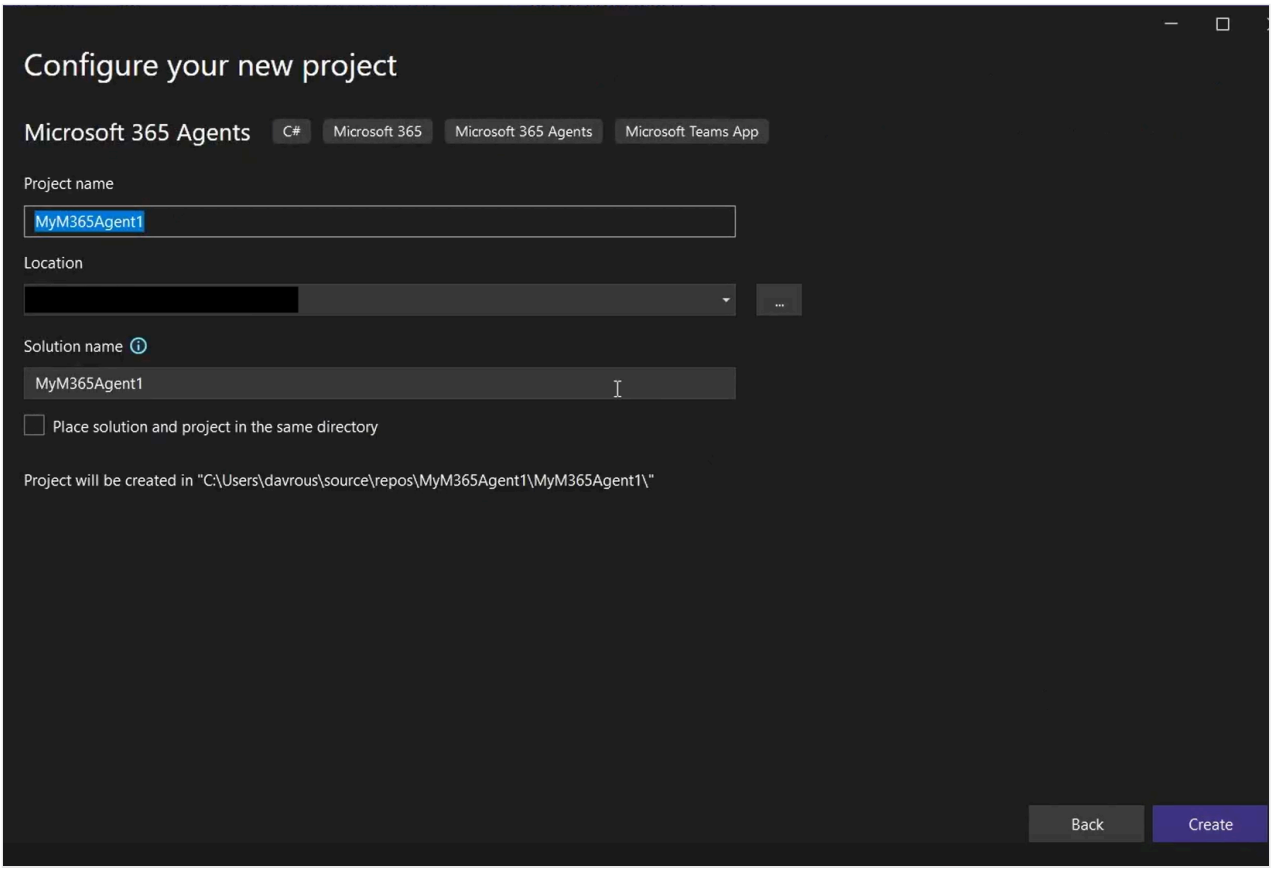
> ⓘ **Note**
>
> The procedure that follows currently works for .NET only.

1. To build a new agent project, open Visual Studio and select **Create a new project**. Search for "agent" to find new templates using the Microsoft 365 Agent Toolkit and Agents SDK Select **Microsoft 365 Agents** > **Next**.

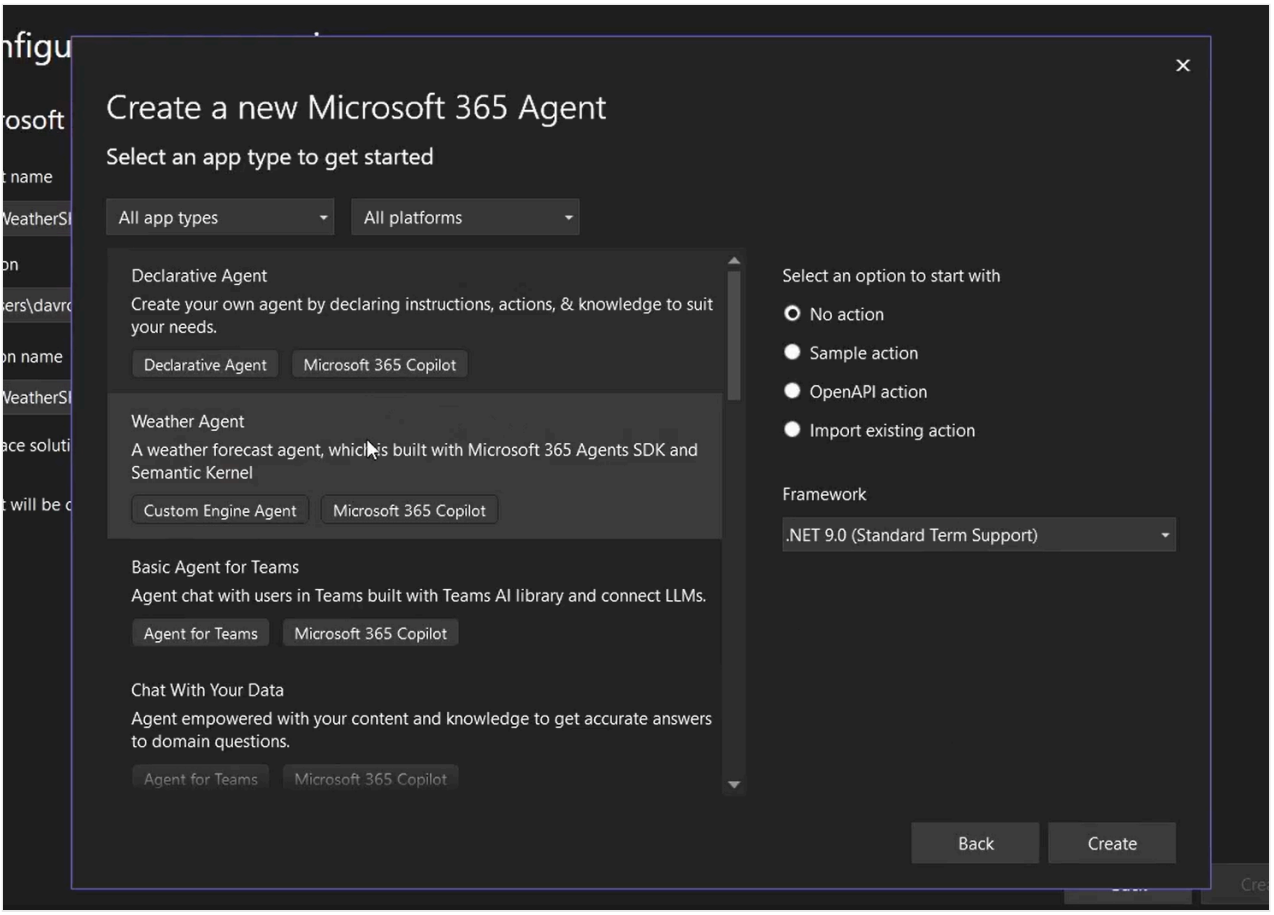


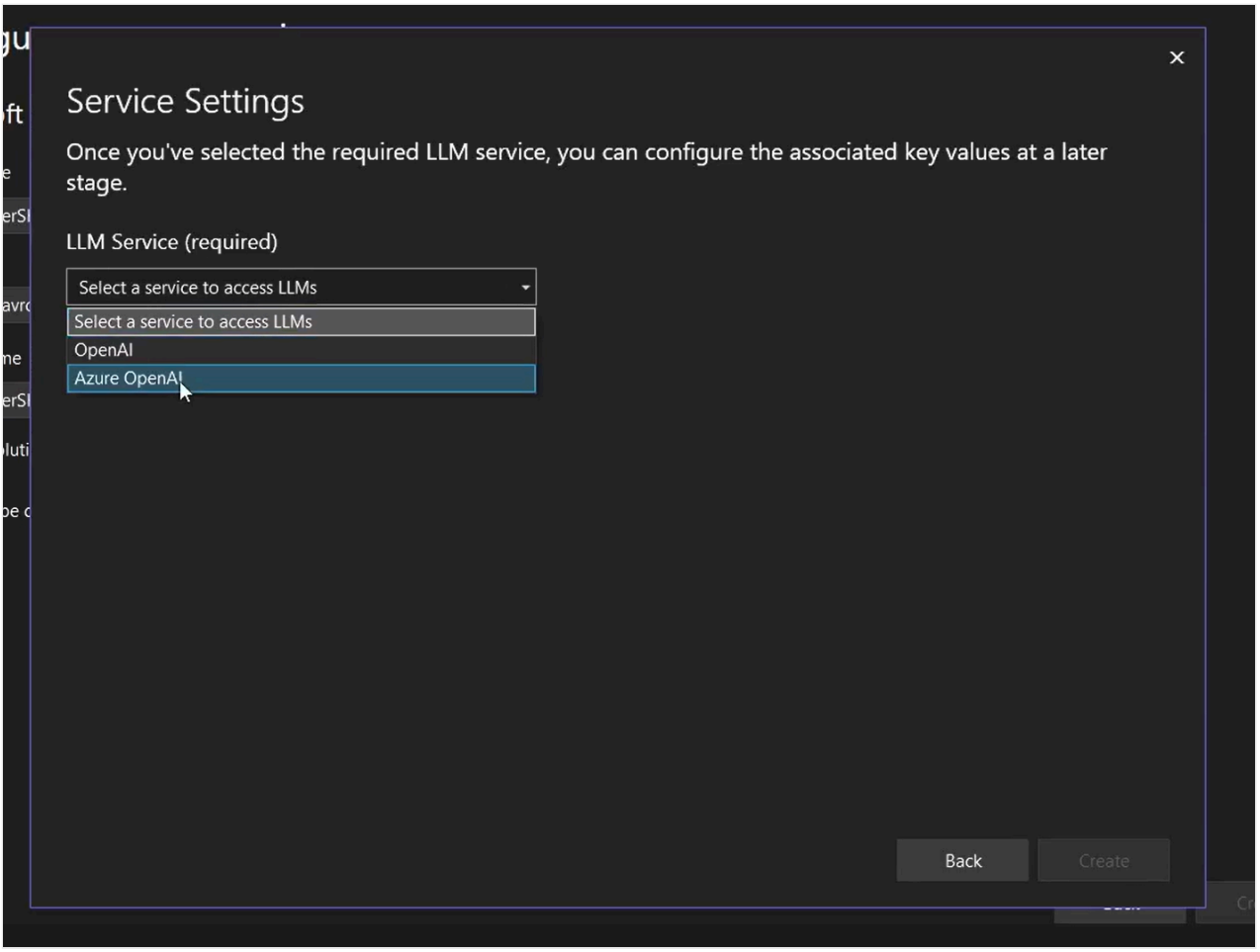
A dialog appears so that you can name the agent. You can also change the location and solution name if you want to.

2. When you are finished, select **Next**

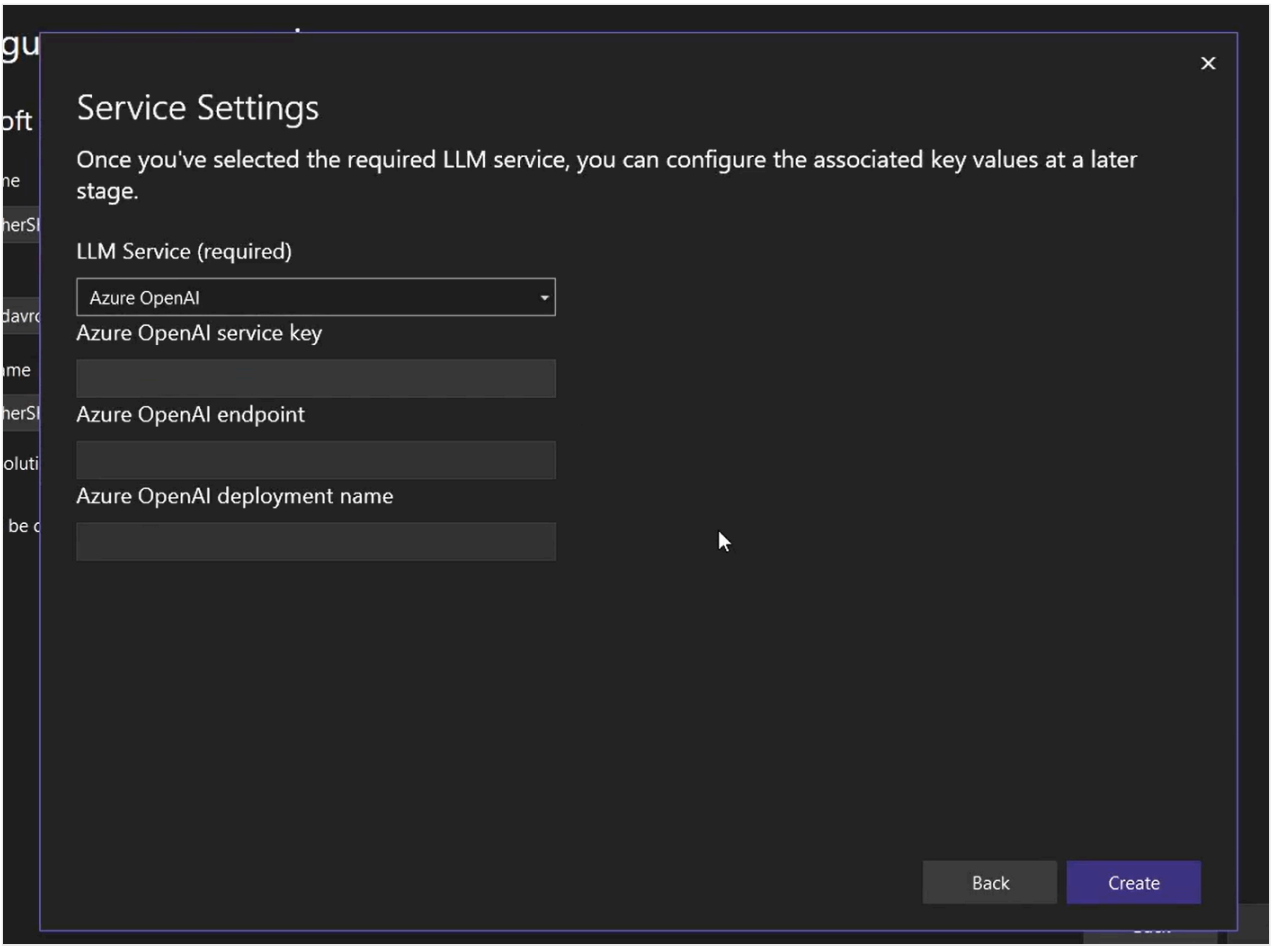


3. The next dialog prompts you for the agent type to choose. You can select a few different options the toolkit provides for creating agents. To use the Microsoft 365 Agents SDK, you should select the **Weather Agent** sample. This prebuilt sample implements Semantic Kernel for orchestration with Azure AI Foundry or Azure OpenAI models. You can also use the Empty agent sample if you want to get started without a model or orchestrator set up.



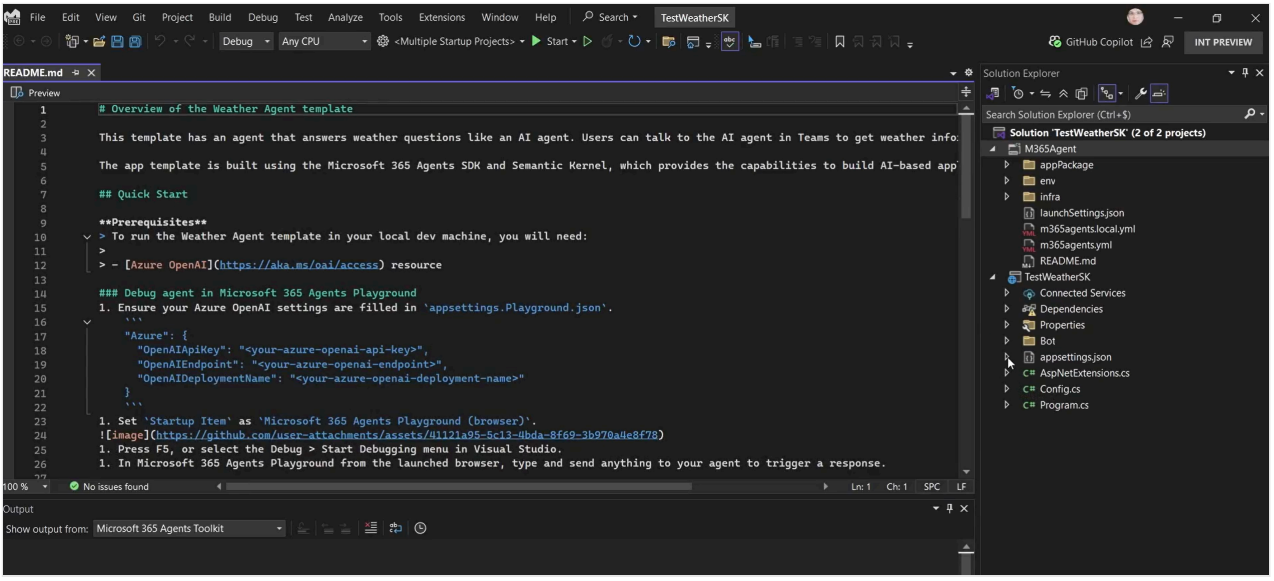


4. For this walkthrough, select Azure AI Foundry and complete the required inputs. Including the key, endpoint and deployment name. This information can be found in Azure AI Foundry under **Models and endpoints**.



5. Select **Create**. The toolkit creates the project from a template for you, ready to get started.

6. In a short amount of time, you should have a new project created which utilizes the Agents SDK.



Test your agent in Microsoft 365 Agents Playground

To get started, you can test locally using the Microsoft 365 Agents Playground in the Toolkit.

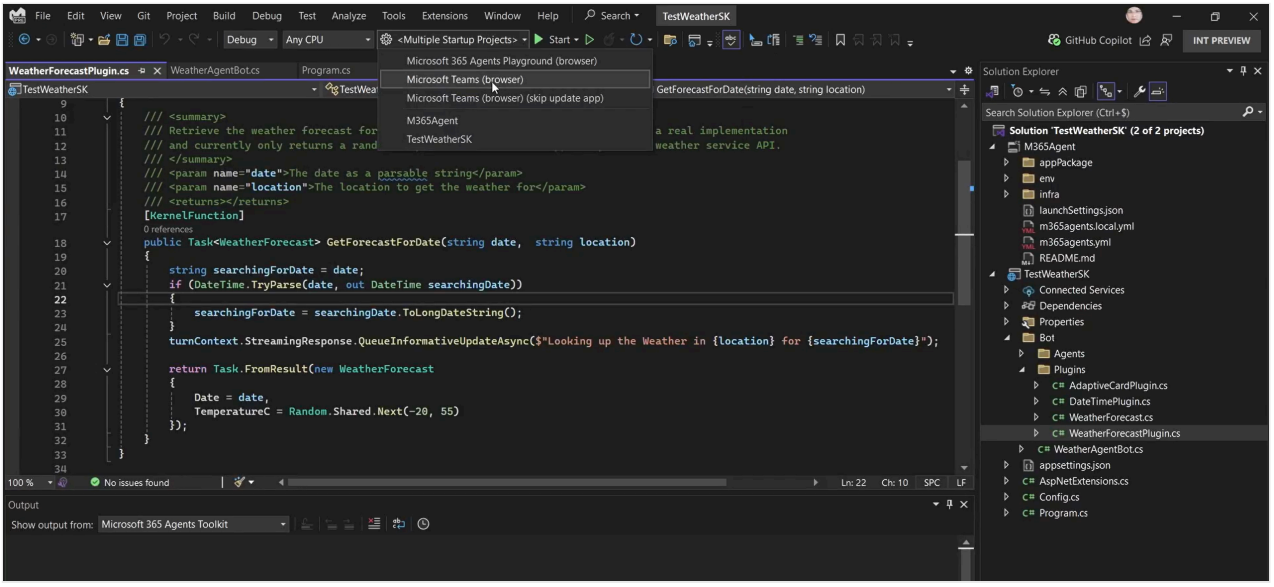
To start testing, select the debug target in your project to be **Microsoft 365 Agents Playground**."

The playground opens for you to test in a new browser window with your local host and shows the playground with it ready to test. Start sending messages to your agent to test it's behaviour.

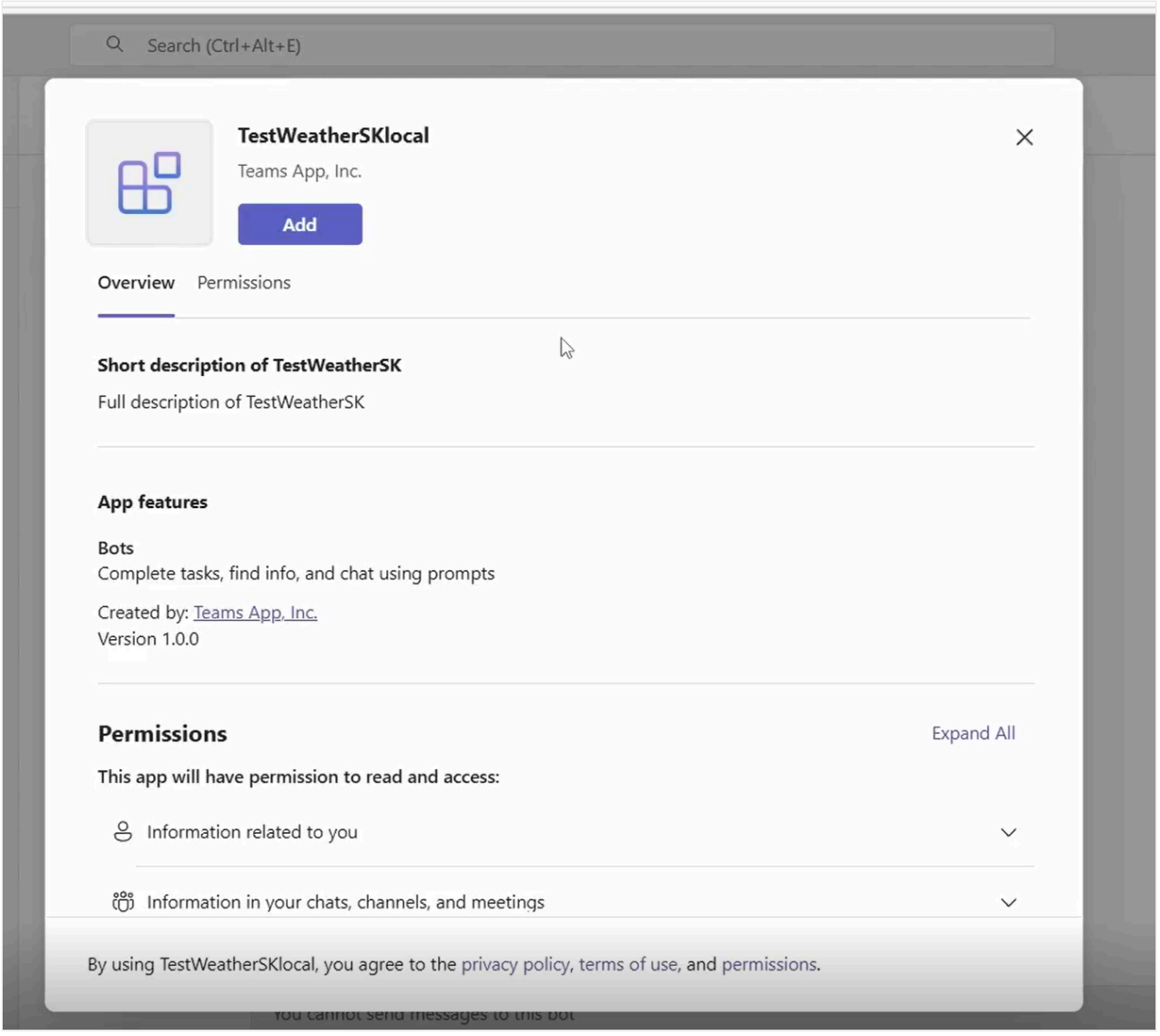
Debug and test your agent in Microsoft Teams or Microsoft 365 Copilot

You can also set the debug target to be directly in Microsoft Teams or Microsoft 365 Copilot.

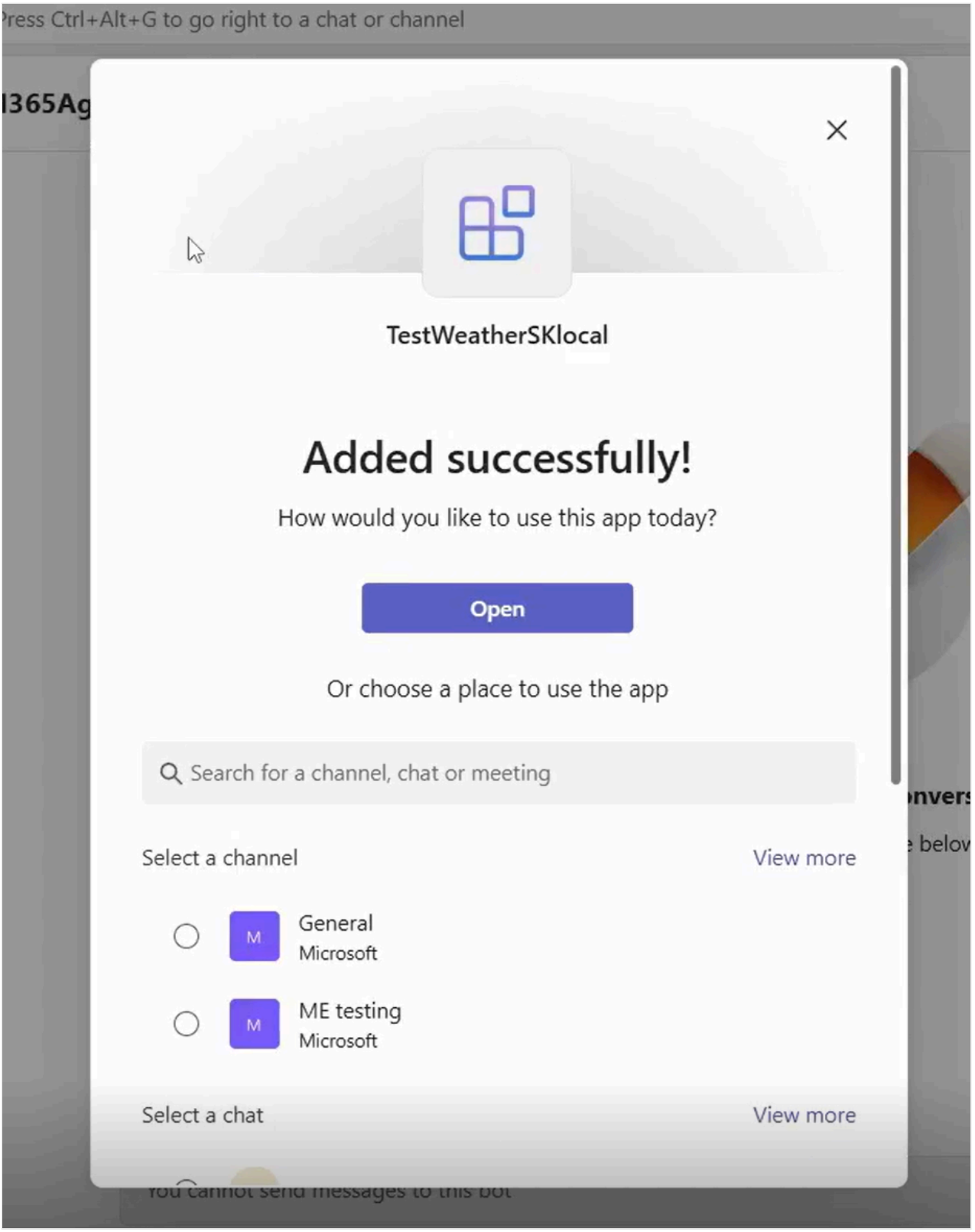
1. Select one of the debugging options as the debug target from the list of targets.



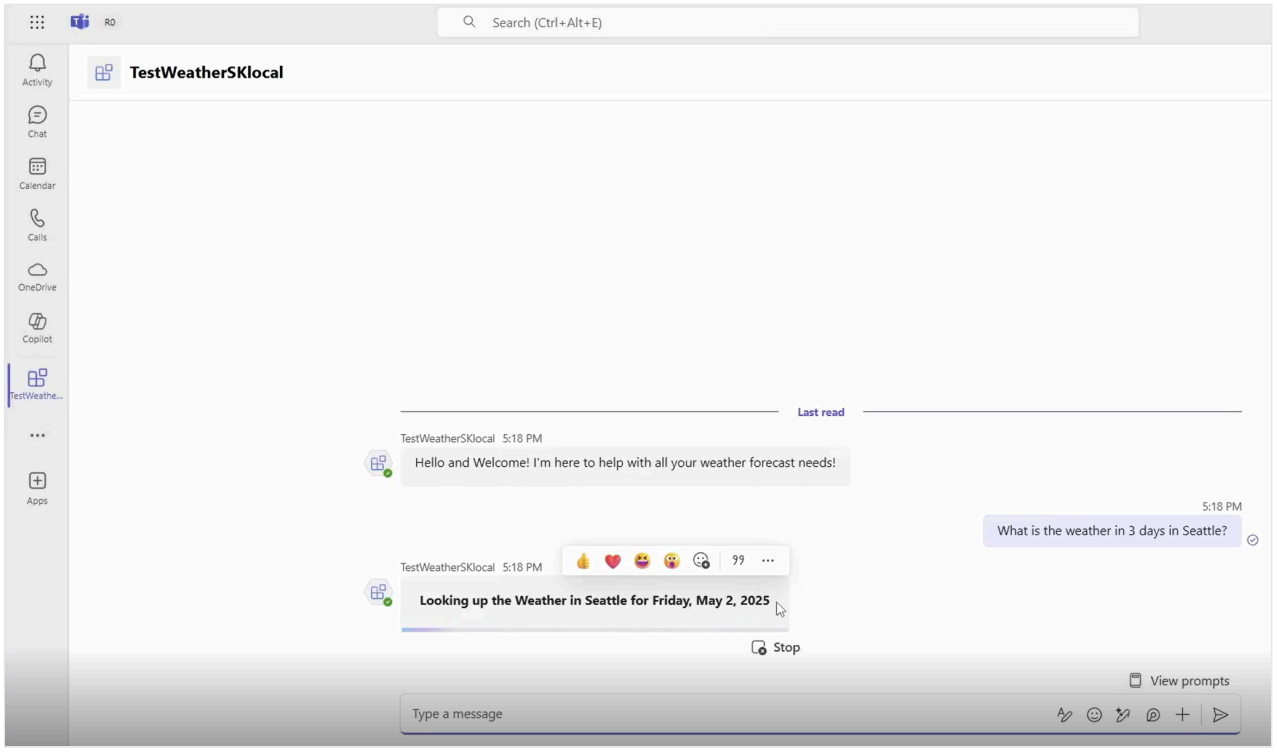
   It takes a few moments to Microsoft Teams. You're prompted to add your agent in the Teams Client that opens.



2. Select **Add.** A notification pane appears, indicating that the agent is added successfully.



3. Select **Open** on your new agent to open in Teams. You can ask questions of your agent, directly in Teams. You can set breakpoints to work through debugging your experience when required.



Summary

You have now successfully:

- Started a new Microsoft 365 Agents project and agent using the Agents Toolkit
- Tested the agent locally using the Microsoft 365 Agents Playground
- Deployed the agent for debugging directly in the Teams or Microsoft 365 channel

Create JavaScript agents in Visual Studio Code with the Microsoft 365 Agents Toolkit

07/15/2025

In this article, you learn how to create a new Agents SDK JavaScript project in Visual Studio, using the Microsoft 365 Agents Toolkit.

Prerequisites

- Install the Agents Toolkit extension for Visual Studio Code.
- You need an Azure model from the Azure AI Foundry portal. You need this data about the model:
  - **Name**
  - **Target URI**
  - **Key**

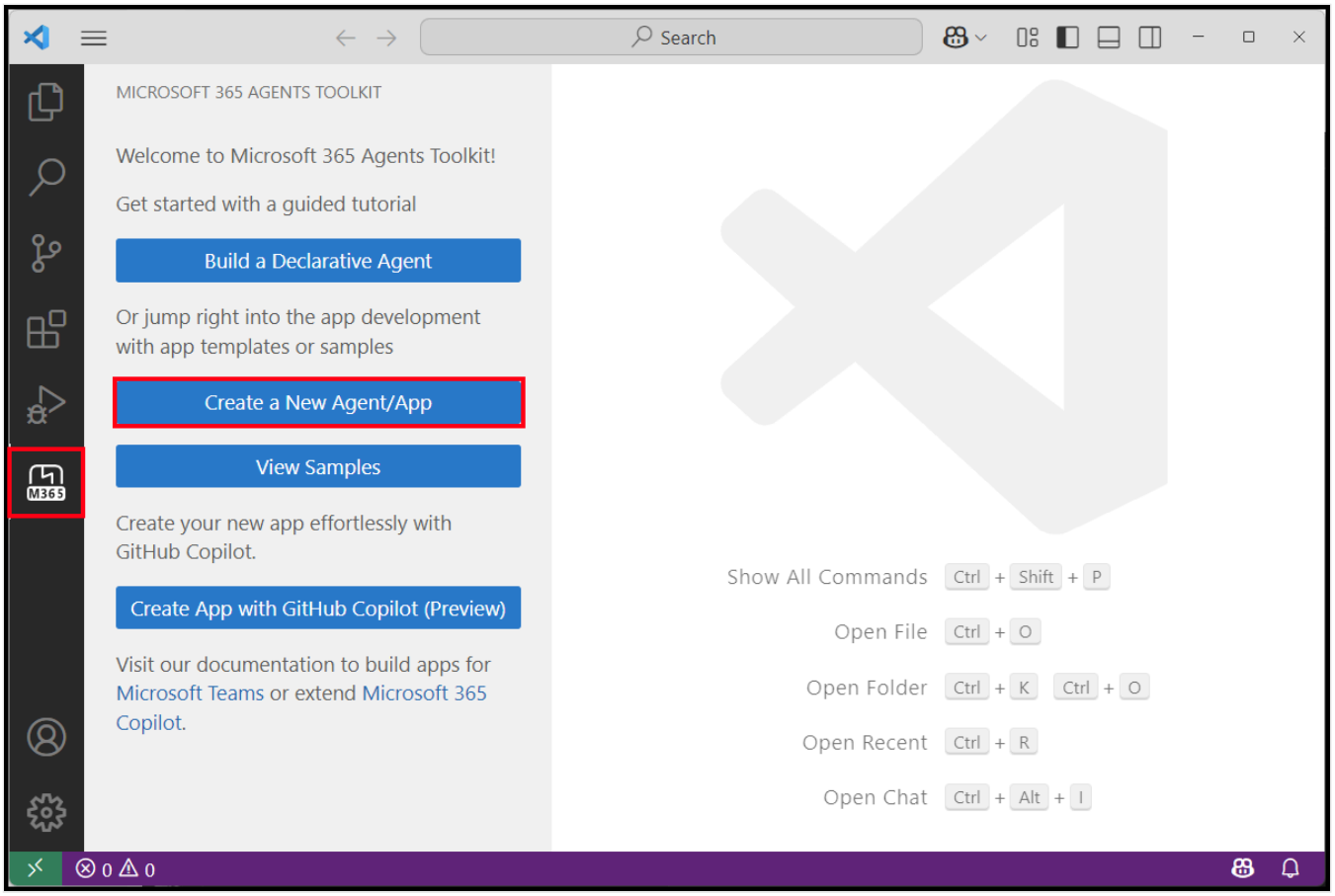
  Learn how to add and configure models to Azure AI Foundry Models

Create a new project

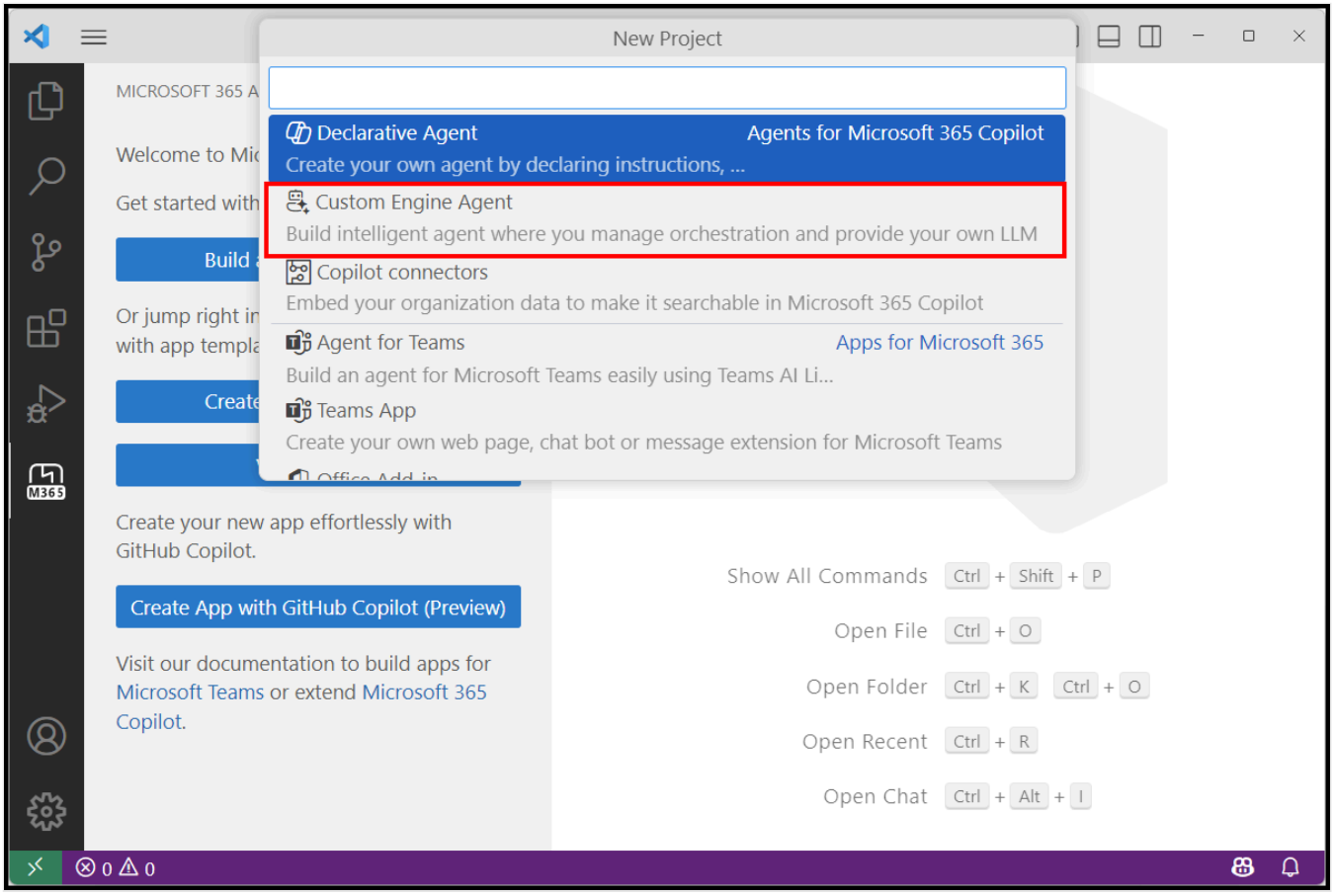
The Agents Toolkit provides a project template to help you get started with building an agent. You can start from a template in the toolkit or from samples in the Agents SDK.

> **Note**
>
> The procedure that follows currently works for JavaScript and TypeScript only. Support is planned for Python.

You can build a new agent project by selecting **Create a New Agent/App** in the Microsoft 365 Agents Toolkit. You can start from a template in the toolkit or from samples in the Agents SDK. This guide covers starting with the Agents Toolkit.



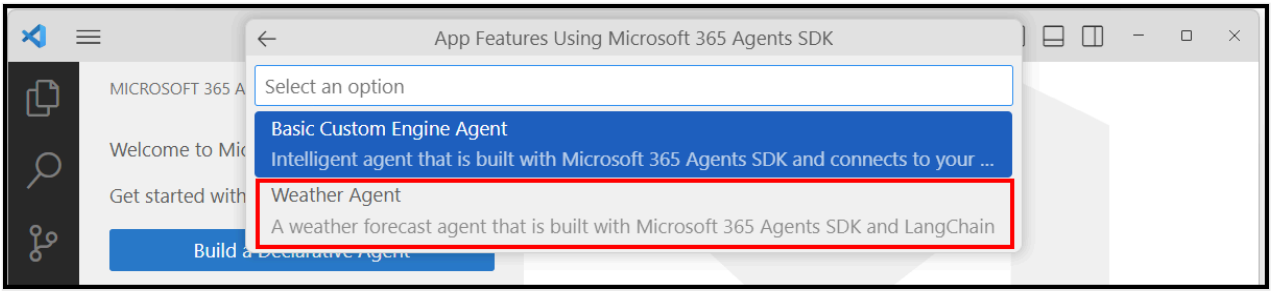
To start with building an agent with the Agents SDK, select **Custom Engine Agent** from the first menu:



Create a new agent

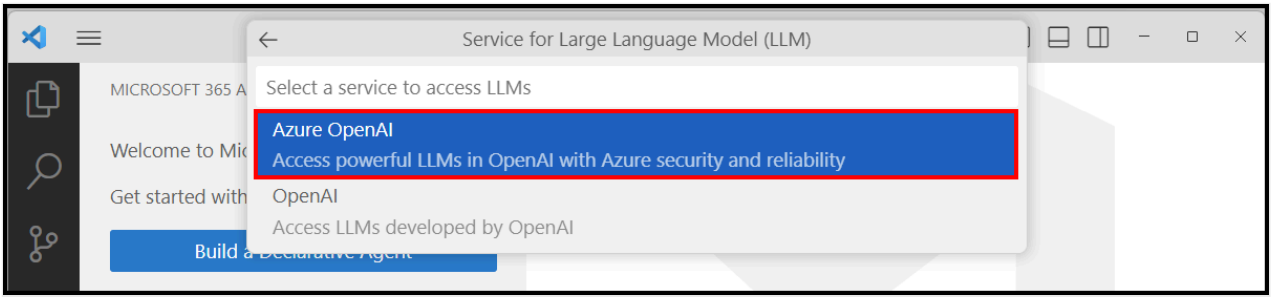
With custom engine agent selected as an option, you're guided down a series of prompts to add in your own AI services.

1. You have two templates to select from: **Basic Custom Engine Agent** or **Weather Agent**. The basic custom engine agent is an agent without anything prebuilt. You need to add an AI orchestrator, like Semantic Kernel or LangChain, and your knowledge.



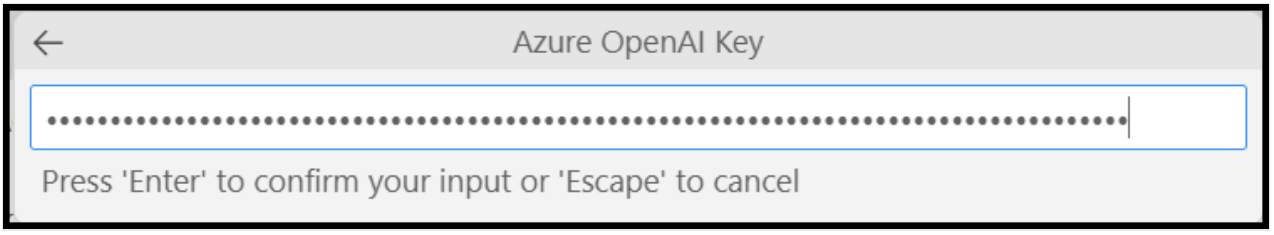
2. In this example, select **Weather Agent** to create an agent that uses LangChain and Azure AI Foundry depending on your chosen language.

3. Select **Azure OpenAI** for your model.



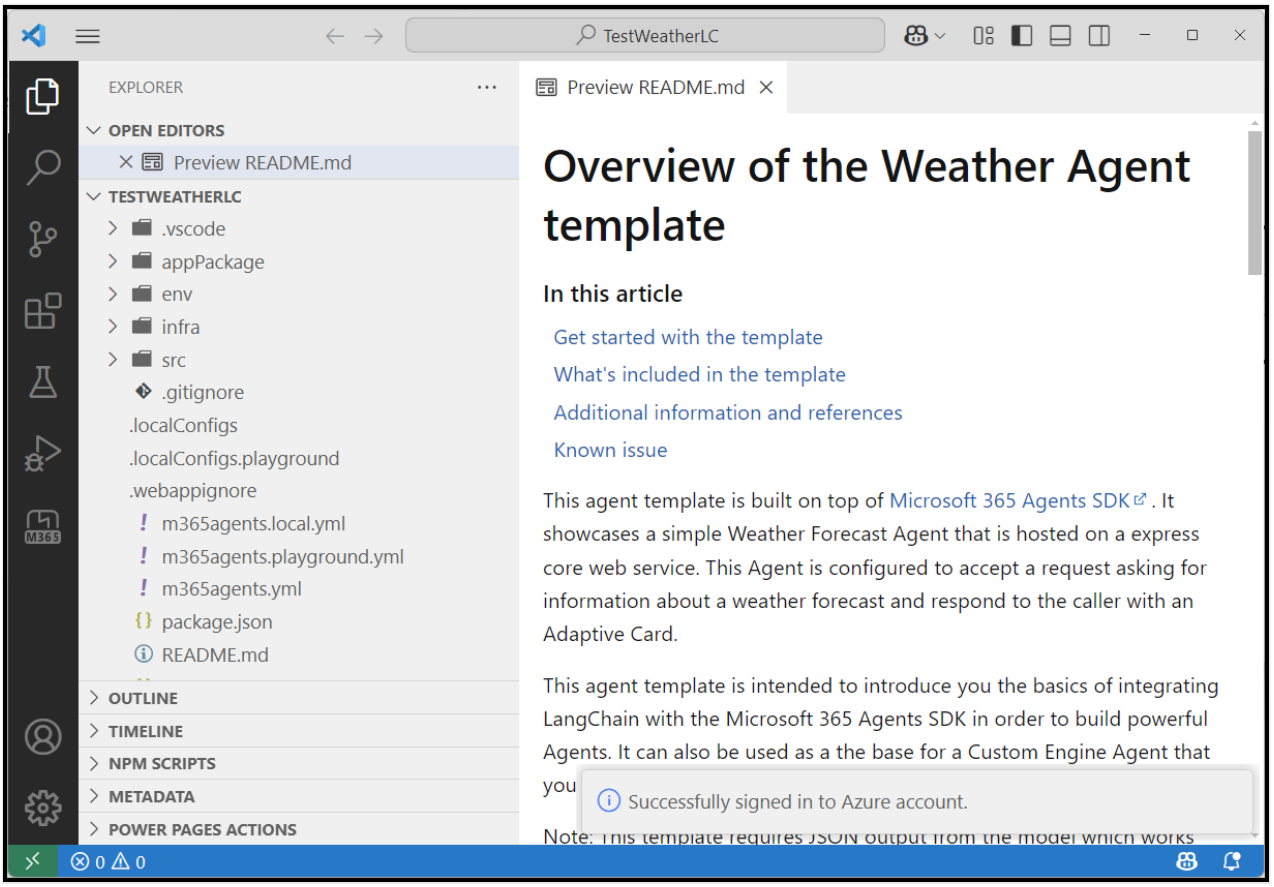
   You're prompted for your **Key**, **Target URI**, and the **Name** of your Azure model from the Azure AI Foundry portal. You can find these pieces of information under **My assets** and **Models and endpoints** in the Foundry portal.

4. Enter the details, starting with the **Key**:

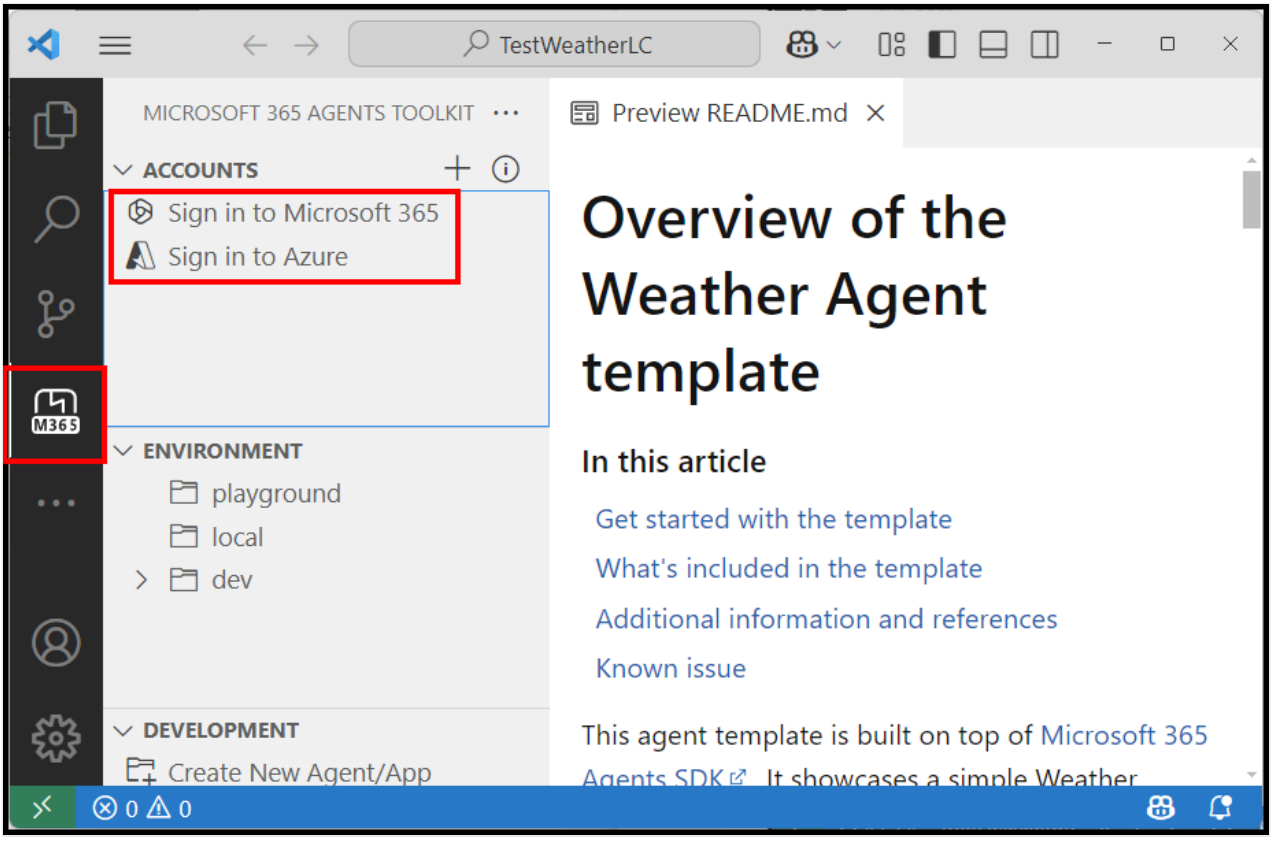


5. Select **JavaScript** or **TypeScript**, select the **Default folder**, and enter an **Application Name** to store your project root folder in the default location.

   Your new project opens.



6. Confirm you're signed in using the extension by selecting the Microsoft 365 logo on the toolbar in Visual Studio Code. Ensure you're signed in to the tenant you want to connect to.

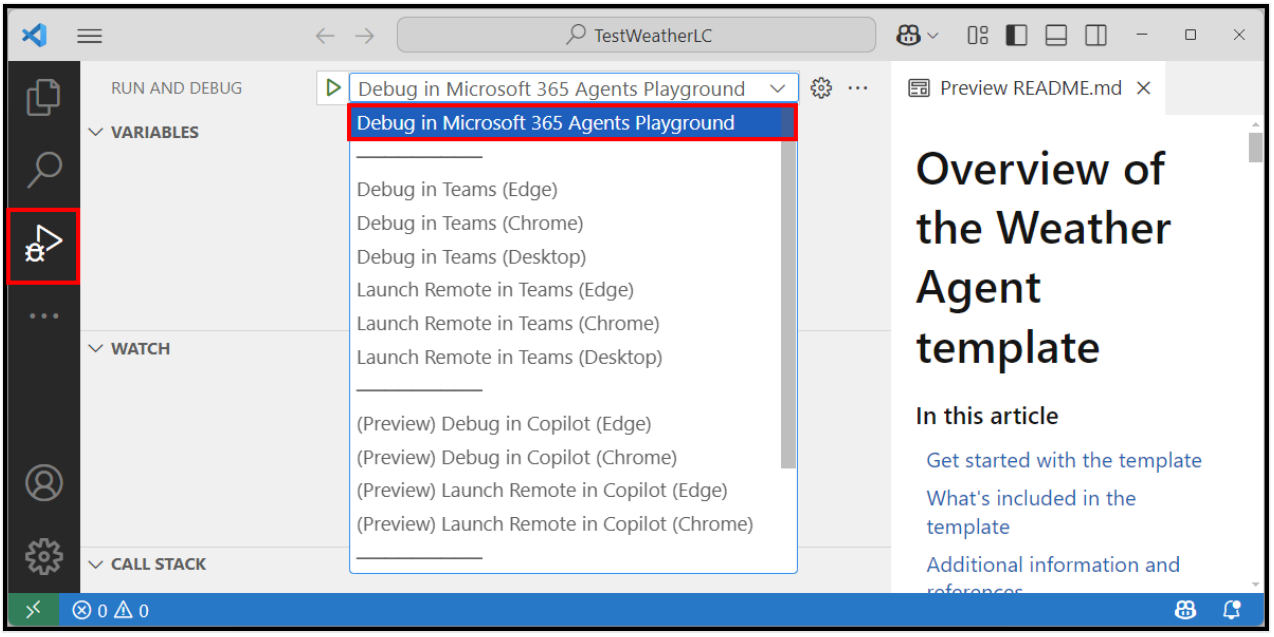


Debug and test your agent in Agents Playground

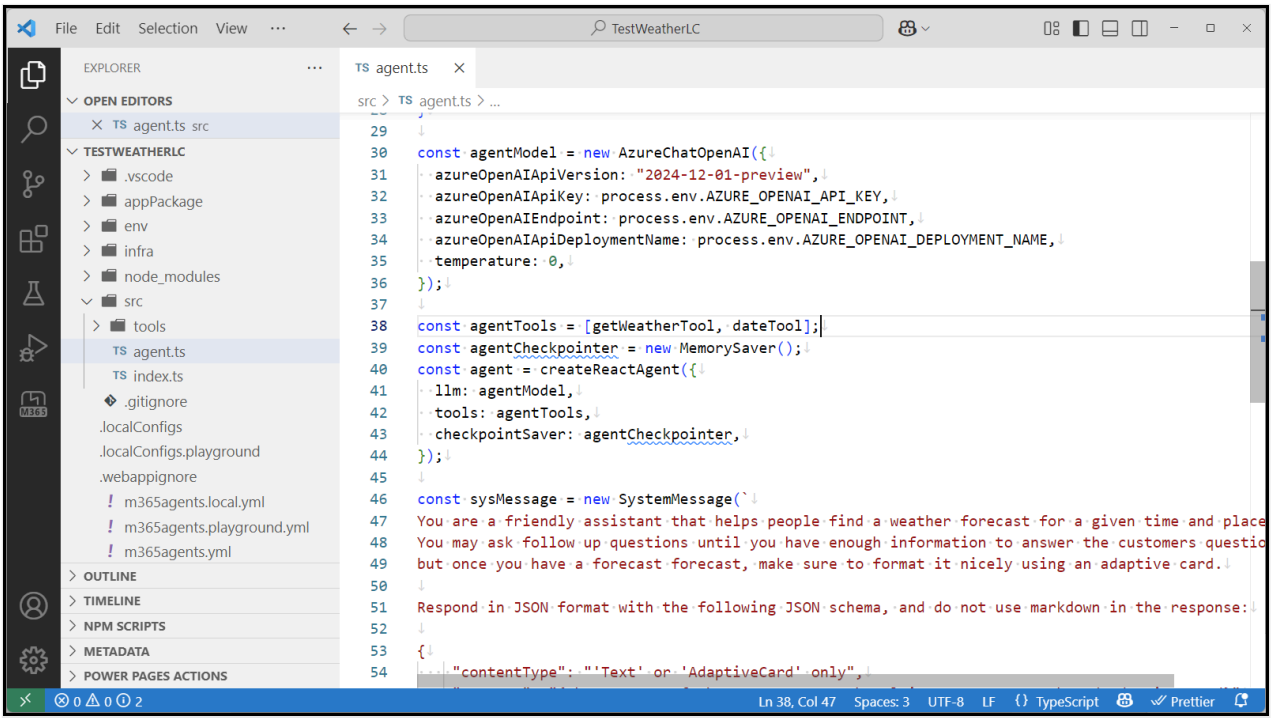
You can debug and test your code with the new Microsoft 365 Agent Playground available in the toolkit. The playground helps you debug your code easily and without having to go through a full deployment cycle.

1. Select **Debug in Microsoft 365 Agents Playground**.

   When you select the playground, wait a short time while it prepares your local machine with the required components. The preparation takes a few minutes.

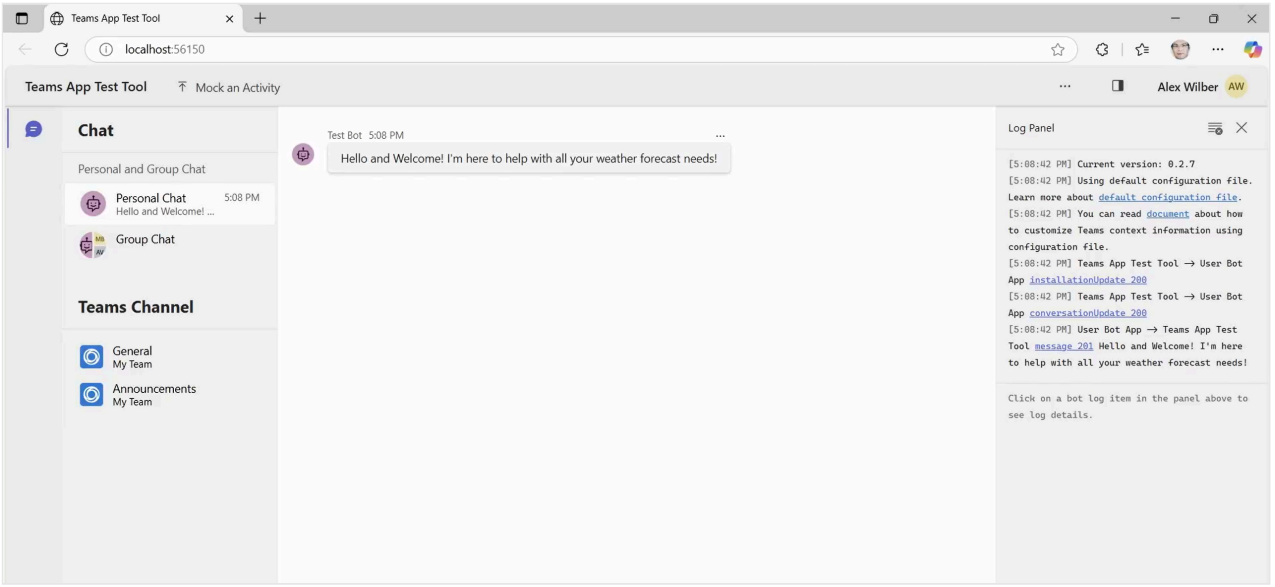


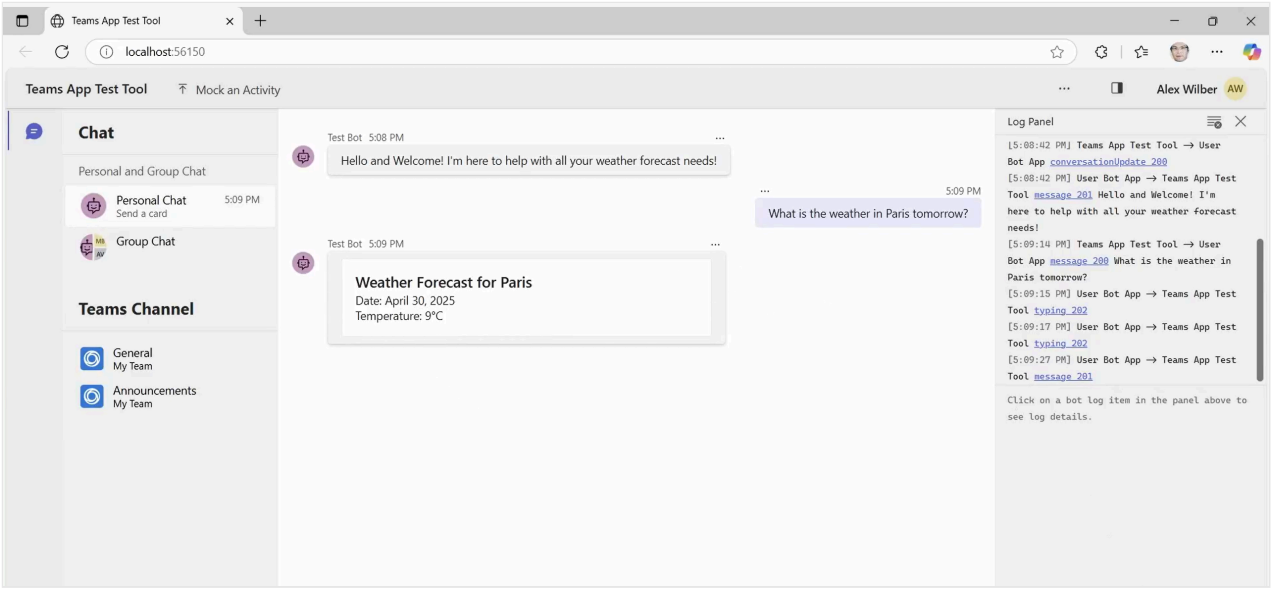
2. While you wait for the deployment, check your folder for the code and review it to familiarize yourself.



3. Once the playground for debug and testing finishes loading, a browser opens and you're ready to interact with your agent using the playground. If you followed the guide and

used the prebuilt template with LangChain and Azure AI Foundry, you can ask "What is the weather in {your location} tomorrow?" The agent responds with an adaptive card with the weather, using your chosen AI Service.

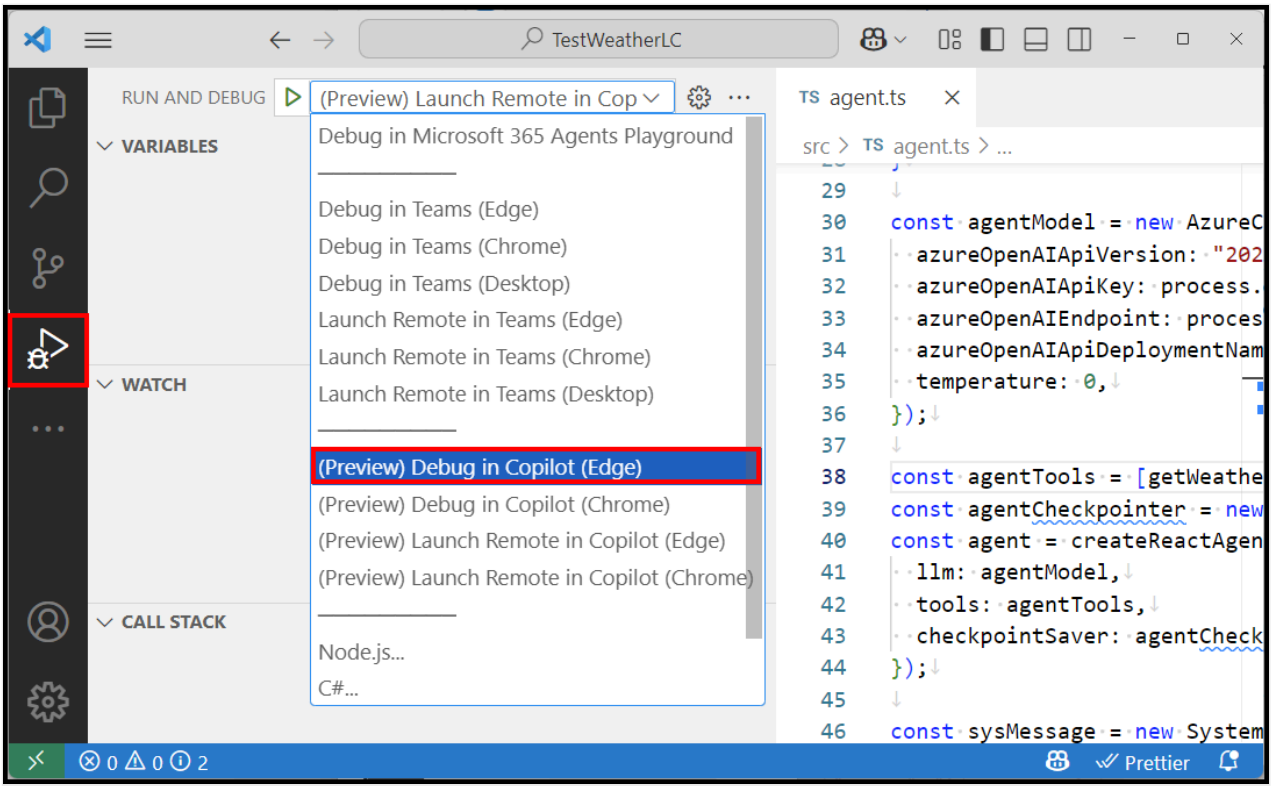




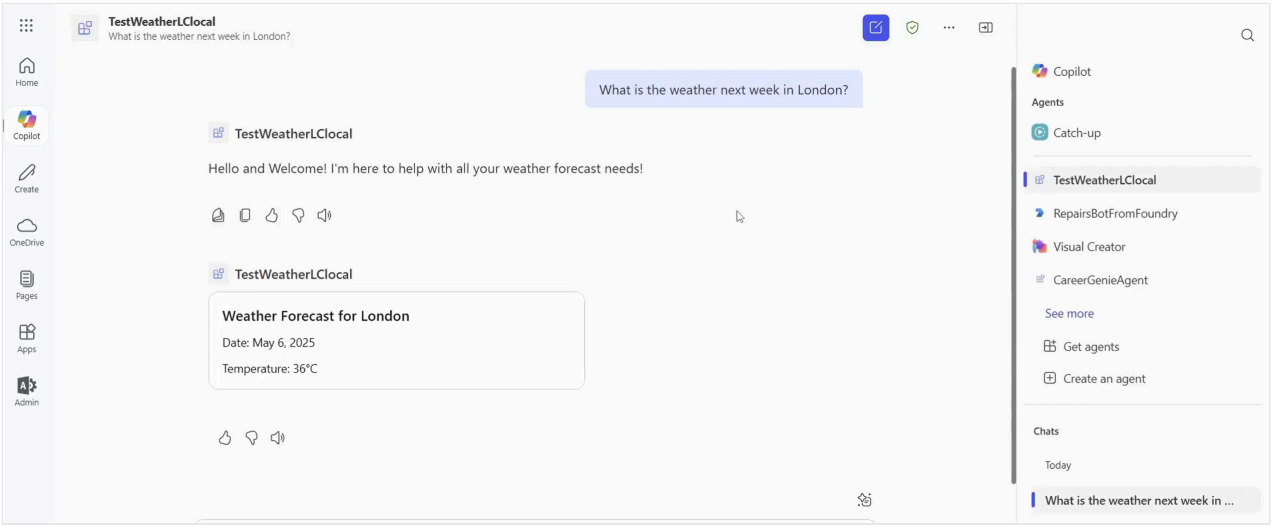
Debug and test your agent in Microsoft 365 Copilot

When you finish testing locally in the Agents Playground, you can deploy to Azure Bot Service and configure for the Microsoft 365 Copilot channel. Ensure you're logged into a tenant that has access to Microsoft 365 Copilot.

1. Change the debug target to Copilot, so that you can debug using Microsoft 365 Copilot. Select `F5` or **Debug** to test. It takes a few minutes of preparation to make the agent available to Microsoft 365. Behind the scenes, the toolkit creates an App Registration and Bot Service record in Azure Bot Service, and deploys your project to your tenant along with a manifest.



2. Once you do this, you should see Microsoft 365 Copilot load and be able to ask questions, add breakpoints, and debug, as required, directly in Microsoft 365 Copilot:



Summary

You have now successfully:

- Started a new Microsoft 365 Agents project and agent using the Agents Toolkit
- Tested the agent locally using the Microsoft 365 Agents Playground
- Deployed the agent for debugging directly in the Microsoft 365 Channel

Test your agent locally in Agents Playground

Article • 05/19/2025

The details of how to test your agent locally depend on how you created your agent.

You can create an agent using the Microsoft 365 Agents SDK in three ways:

- Start with the Microsoft 365 Agents Toolkit in C#, JS, or Python using Visual Studio or Visual Studio Code
- Clone from a sample and open in your IDE
- Use the CLI, as shown in the quickstart

Using the Agents Playground

You started your project with the toolkit

If you start with the toolkit, you have everything set up to test using the Agents Playground straight away. You can test in the Agents Playground either locally, or in Microsoft 365 Copilot or Microsoft Teams. This scenario is covered in the Visual Studio Code walkthrough and the Visual Studio walkthrough, respectively.

You started your project by cloning or CLI

If you start your project using the CLI or by cloning a sample and opening the sample in your IDE, you can use the local Agents Playground to test. The Agents Playground connects to your local code.

1. Once your quickstart agent is created or a sample is cloned from the repo, you can use it with the test tool from the Agents Toolkit.

2. Make sure you download and install Agents Toolkit before beginning on Visual Studio or Visual Studio Code

3. Ensure you have your config settings in the right project folder and that they're referenced in your agent to be used when the app is built.

   For example, in .NET:

   ```
   C#
   ```

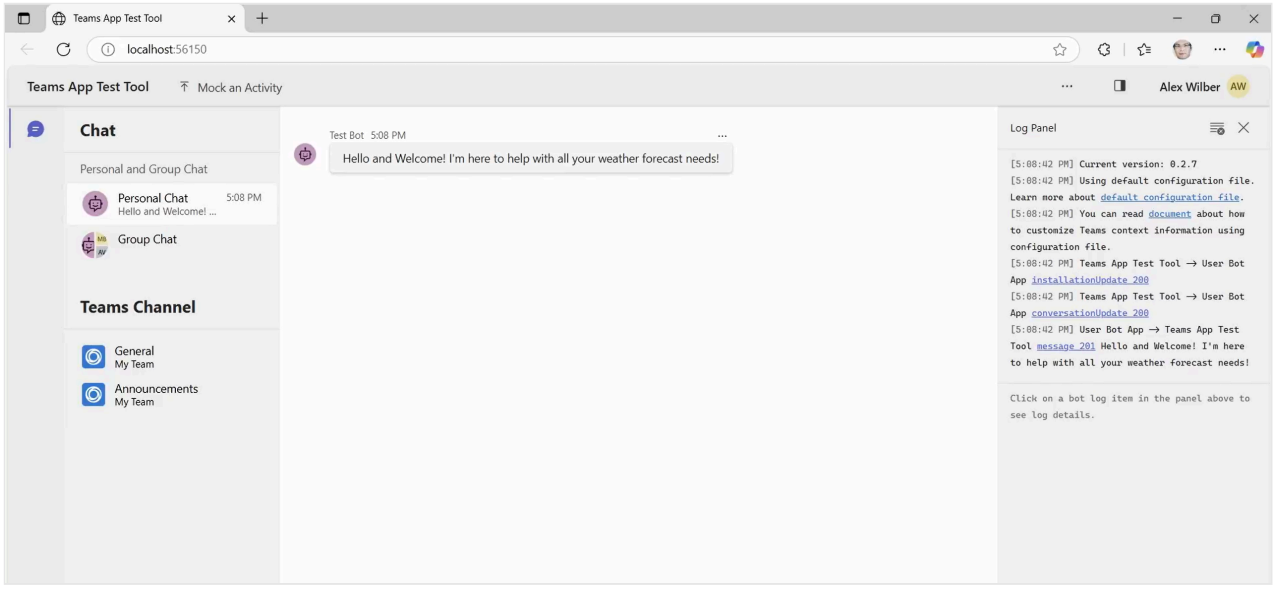
```
var config = new
ConfigurationBuilder().AddJsonFile("appsettings.json").Build();
```

4. Configure your ports correctly in your app. For example, currently, port must be 3978, the port the local test app tool is listening on.

5. Your configuration settings need to be configured with the `Connections` and `ConnectionsMap` set with a valid client ID and secret, app ID, and tenant.

6. Run your code.

7. Open the test tool. It should open as seen in the following image. You can ask questions and test your agent.



Wherever possible, we recommend that you start with the Microsoft 365 Agents Toolkit. The toolkit makes getting started, testing locally, and deploying *easier and quicker*. It abstracts much the manual setup of the Azure Bot Service and Azure App Registrations so you don't have to. By starting manually, you must perform these manual steps yourself.

Summary

You have successfully tested your Microsoft 365 Agents SDK locally using the Agents Playground when starting with a cloned sample from the GitHub repo or from the CLI.

Deploy your agent to Azure and register with Azure Bot Service manually

07/18/2025

The details for deploying your agent depend on how you create your agent.

You can create an agent using the Microsoft 365 Agents SDK in three ways:

- Clone from a sample and open in your IDE
- Start with the Microsoft 365 Agents Toolkit in C#, JS, or Python using Visual Studio or Visual Studio Code
- Use the CLI, as shown in the quickstart

This guide covers how to deploy your agent manually to Microsoft 365 Copilot and Microsoft Teams.

Create your agent with Agents SDK

To deploy your agent, you need to create your agent and be ready to deploy it. This means having access to your target deployment channel—for example, Microsoft 365 Copilot or Microsoft Teams.

You also need access to set up Azure Resources, and admin access to Microsoft 365 Admin Center (MAC) or Teams Admin Center (TAC).

You also need to manually upload a manifest to either MAC or TAC

This process is commonly referred to as *side-loading*.

Create an Azure Bot Service app and app registration

Create your Azure Bot Service (ABS) app and ensure it has the same App ID as your ABS record.

An agent can work with either single or multitenant configurations. However, for this test, it's best to stay with single-tenant.

In your app registration, create a client ID and secret.

Ensure the configuration file in your Agents SDK agent (for example, `appSettings.json` in .NET agents) is updated with the client ID, secret and tenant information:

JSON

```
  "TokenValidation": {
    "Audiences": [
      "{ClientID}" // this is the Client ID used for the Azure Bot Service
    ],
    "TenantId": "{TenantID}"
  },

  "Connections": {
    "BotServiceConnection": {
      "Settings": {
        "AuthType": "ClientSecret", // this is the AuthType for the connection,
valid values can be found in Microsoft.Agents.Authentication.Msal.Model.AuthTypes.
The default is ClientSecret.
        "AuthorityEndpoint": "https://login.microsoftonline.com/common", // common
is used for multi tenant. If single tenant replace 'common' with tenantID
        "ClientId": "{ClientID}", // this is the Client ID used for the
connection.
        "ClientSecret": "{ClientSecret}", // this is the Client Secret used for
the connection.
        "Scopes": [
          "https://api.botframework.com/.default"
        ]
      }
    }
  },
```

Publish your agent to a web app

Publish your agent into your web app so that you have an endpoint like `example.azurewebsites.net`.

Once your app is published, navigate to your ABS instance and under the **Configuration** heading change the **Messaging endpoint** to `https://{yourwebsite}/api/messages`.

Test in Web Chat

To see your message in web chat, select **Test in Web Chat** in your ABS instance and trigger your agent.

Test using Dev Tunnels

If your website isn't deployed, but you want to test from your local host, run your code, and get the port number it runs on (for example, port 5000).

Open DevTunnels and enter:

```
host -a -p port {portnumber}
```

In the console that opens, you should see a URL to paste into your ABS app under the **Messaging Endpoint** section in the configuration.

Prepare your Teams and Microsoft 365 Copilot .zip package

For Microsoft Teams and Microsoft 365 Copilot, you need to create and upload a manifest file with two images.

1. Create an empty folder, and make a copy of the Microsoft 365 Copilot manifest template file from the Agents SDL GitHub.
2. In the template, replace the Client IDs with your Client ID. This is the Client ID agent you made earlier. The manifest file references the bot ID and runs it from where it's hosted.

> **Note**
>
> This manifest file may change.

1. Add in the color and outline images and name them correctly. Ensure the name of your app is how you expect it to show up in Microsoft 365 Copilot and Teams. Also, ensure you review all the related information in the manifest and update it from the templates. The template should contain information related to your agent, such as terms and privacy policy.

Deploy to Microsoft 365

1. In the ABS instance in Azure, select **Channels** and turn on Microsoft Teams.
2. Navigate to the Microsoft Admin Portal (MAC). Under **Settings** and **Integrated Apps,** select **Upload Custom App**.
3. Upload your app. The manifest should pass and you should be able to deploy your app.
4. After a short period of time, the app shows up in Microsoft Teams and Microsoft 365 Copilot. You need to have the $30 SKU for Microsoft 365 Copilot at the moment to support CEAs and have a tenant enabled for private preview.

Summary

You have successfully deployed your agent. You can now test the agent directly from the deployed channel.

Building Agents

Article • 05/20/2025

Learn more about how and what happens when you build an agent with the Microsoft 365 Agents SDK.

In the Agents SDK, an agent is built using `AgentApplication` and `AgentApplicationOptions`

`AgentApplicationOptions` allows for different configuration options including authentication. It's a parameter in the constructor of `AgentApplication`

C#

```
builder.AddAgentApplicationOptions();

builder.AddAgent(sp =>
{
    // Setup the Agent.
    var agent = new
AgentApplication(sp.GetRequiredService<AgentApplicationOptions>());

    // Respond to message events.
    agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken) =>
    {
        var text = turnContext.Activity.Text;
        await turnContext.SendActivityAsync(MessageFactory.Text($"Echo: {text}"),
cancellationToken);
    });
    return agent;
});
```

The endpoint of your application needs to be configured to handle incoming requests and is designed to specifically process messages to your agent, using an adapter `IAgentHttpAdapter` and `IAgent`. This configuration converts the HTTP request and response into understandable formats between the web server and the agent.

C#

```
app.MapPost("/api/messages", async (HttpRequest request, HttpResponse response,
IAgentHttpAdapter adapter, IAgent agent, CancellationToken cancellationToken) =>
{
    await adapter.ProcessAsync(request, response, agent, cancellationToken);
});
```

Now that the agent is created, you can register to listen for events, add your AI services and custom logic.

In the starter samples on the Agents SDK GitHub repo, you will see that the agent is registered automatically for the generic `OnActivity` event.

C#

```
    agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken) =>
    {
        var text = turnContext.Activity.Text;
        await turnContext.SendActivityAsync(MessageFactory.Text($"Echo: {text}"),
cancellationToken);
    });
```

Managing Turns

Article • 05/02/2025

The Microsoft 365 Agents SDK is based on turns. A turn is the roundtrip starting with the incoming `activity` and then sending any outgoing `activity` that you choose to send back.

A turn can contain many `Activities` where multiple messages are sent back to a person or other agent. A turn doesn't mean the conversation or interaction is done, only that specific turn. Typically within a conversation multiple 'turns' take place. A turn therefore represents the workflow from the incoming to an outgoing action of the agent.

Multiple internal actions can take place within a turn, such as using internal services, AI services, APIs and more to generate plans, responses and get information to use & format.

Within the Agents SDK, developers can access information available for the agent to use, per turn based on the `turn context`:

C#

```
    await turnContext.SendActivityAsync(MessageFactory.Text({response}"),
cancellationToken);
```

The `turn context` object is how activities are sent back to the channel using the `SendActivityAync` method

State & Memory

Agents created with the Agents SDK are stateless by default and can be added.

State is supported as an optional parameter when the agent is built, and includes `Private State`, `User State`, and `Conversation State`. State requires storage to store state (and other information you require). The SDK supports `Memory` storage, `Blob`, `Cosmos`, or custom storage.

Because by default, agents are stateless, any information is lost on the next turn, unless you save state. Developers must specifically configure state (or memory) and choose where to store or persist that state, and retrive it on the next turn. A sample will be linked soon on how to do this.

A note on using state across multiple agents

When you work with multiple agents, you might want to manage multiple states of agents. How you manage the state and memory of those agents depends on a few factors, including:

- The types of state and storage supported by the agents
- What information needs to be stored persistently
- What data you want or need to store across turns
- What data needs to be accessed across agents to perform the required actions

Using Activities

Article • 05/02/2025

`Activities` are the main object sent and received by your agent. This back-and-forth interaction uses something commonly referred to as the `Activity Protocol`.

Your agent listens for events that are types of `Activities` created between the client and your agent. The Microsoft 365 Agents SDK has `Channel Adapters` that translate the different channel languages into `Activities`. Be aware that typically channels will have the 'Message' type as a core method of communication, and beyond that it is channel dependent what types of activities are supported based on the client.

Activities are used in two places:

1. When listening for events. The agent listens for `Activities`. In the following example, the agent listens for an `Activity` of type `Message` that can contain any type of text, media, and so on.

C#

```
    agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken) =>
    {
        // custom logic here
    });
```

1. When sending responses. Developers can construct a `Message` using `MessageFactory` and send it in the `SendActivityAsync` method.

C#

```
    await turnContext.SendActivityAsync(MessageFactory.Text({response}"),
cancellationToken);
```

Activity types

The Agents SDK supports various activity types, each with its own purpose. Here are some of the most common activity types.

Message

Message is the most common type of activity. A message activity is used to exchange text, media, and rich content between the bot and the user. For example, a user sends a text message to the agent, and the agent replies with a text message.

C#

```
agent.OnActivity(ActivityTypes.Message, async (turnContext, turnState,
cancellationToken))
```

Conversation Update

A conversation update activity is used to notify the agent when members are added to or removed from a conversation. For example, a user joins a group chat, and the agent is notified of the new participant.

C#

```
agent.OnActivity(ActivityTypes.ConversationUpdate, async (turnContext, turnState,
cancellationToken) =>
{
    var membersAdded = turnContext.Activity.MembersAdded;
    if (membersAdded != null && membersAdded.Any())
    {
        foreach (var member in membersAdded)
        {
            if (member.Id != turnContext.Activity.Recipient.Id)
            {
                await turnContext.SendActivityAsync(MessageFactory.Text($"Welcome
to the chat, {member.Name}!"), cancellationToken);
            }
        }
    }
});
```

Typing

A typing activity indicates that the sender is actively typing a message. For example, the agent sends a typing indicator to show that it's processing the user's message.

C#

```
agent.OnActivity(ActivityTypes.Typing, async (turnContext, turnState,
cancellationToken) =>
{
    Console.WriteLine("User is typing...");
```

```
    await Task.CompletedTask;
});
```

End of Conversation

An end of conversation activity indicates that the conversation is finished. For example, the agent sends an end-of-conversation activity when it completes its interaction with the user. This activity type is useful if you want to specifically send prebuilt feedback forms or show that a conversation is done.

C#

```
    agent.OnActivity(ActivityTypes.EndOfConversation, async (turnContext,
turnState, cancellationToken) =>
    {
        await turnContext.SendActivityAsync(MessageFactory.Text("Goodbye!"),
cancellationToken);
    });
```

Custom Event

A custom event activity lets you send and receive events. Events are lightweight messages that can carry custom data. For example, the agent sends an event to trigger a specific action in the client application.

C#

```
    agent.OnActivity(ActivityTypes.Event, async (turnContext, turnState,
cancellationToken) =>
    {
        var eventActivity = turnContext.Activity.AsEventActivity();
        if (eventActivity.Name == "customEvent")
        {
            var value = eventActivity.Value?.ToString();
            await turnContext.SendActivityAsync(MessageFactory.Text($"Received
custom event with value: {value}"), cancellationToken);
        }
    });
```

There are other types of activities that are specific to clients. For example, Microsoft Teams includes a `Message Reaction` activity. Custom apps or websites can also have their own events. To interact with these events, developers can use the `customEvent` activity type.

Creating Messages

Article • 05/02/2025

Messages are one of the most common types of activities to be used in agents as they're used to communicate to humans or other agents.

The `MessageFactory` object provides methods that help developers create different types of message activities. Using the `MessageFactory` makes it faster and easier to send responses. Using `MessageFactory` is also less prone to errors in the construction of more complex types of messages.

Typically, the way a message is rendered in a UI depends on the client. Most clients accept a message of type text and adaptive cards (Clients vary in the supported version of adaptive cards).

Text

C#

```
 var textMessage = MessageFactory.Text("Hello, world!");
```

Adaptive cards

C#

```
var adaptiveCardAttachment = new Attachment
{
    ContentType = "application/vnd.microsoft.card.adaptive",
    Content = JsonConvert.DeserializeObject(adaptiveCardJson)
};

var adaptiveCardMessage = MessageFactory.Attachment(adaptiveCardAttachment);

await turnContext.SendActivityAsync(adaptiveCardMessage, cancellationToken);
```

Typing indicators

Typing indicators use a combination of a Text `Message` and a Typing `Activity`:

C#

```
var typingMessage = MessageFactory.Text(string.Empty);
typingMessage.Type = ActivityTypes.Typing;
```

There are other types of supported messages in `MessageFactory`, including `Carousel`, `List`, and `SuggestedActions`.

Add AI services and orchestration

Article • 05/19/2025

The Microsoft 365 Agents SDK creates an agent that provides a container for developers to add their chosen AI Services. The agent can:

- Manage state across turns
- Store state or data persistently in the storage of your choice (with built-in options for `Blob` and `Cosmos`)
- Manage activities and events

The agent can be deployed in any channel, including Microsoft 365 Copilot and Microsoft Teams.

AI services typically contain chat endpoints, but can also include assistants or hosted agents, such as agents built in Copilot Studio. Orchestration can be added based on the business logic and how developers want to manage state and invoke tools/plugins. Developers can choose to implement multi-agent scenarios and patterns depending on their requirements, different tech platforms as required based on their defined business logic.

Add Semantic Kernel and Azure OpenAI

You have the choice where you want to add AI services to the agent.

You can choose to implement AI services at the same time the core agent container gets built, together with the services in a web application. You can see this behavior in the WeatherForecastAgent sample on the GitHub samples repo. The sample has one agent called `WeatherForecastAgent`. The agent is passed into the `MyAgent` object as a parameter at runtime as the application is built.

C#

```
builder.Services.AddKernel();

builder.Services.AddAzureOpenAIChatCompletion(
    deploymentName:
builder.Configuration.GetSection("AIServices:AzureOpenAI").GetValue<string>
("DeploymentName"),
    endpoint:
builder.Configuration.GetSection("AIServices:AzureOpenAI").GetValue<string>
("Endpoint"),
    apiKey:
builder.Configuration.GetSection("AIServices:AzureOpenAI").GetValue<string>
("ApiKey"),
    modelId:
```

```
builder.Configuration.GetSection("AIServices:OpenAI").GetValue<string>("ModelId")
);
```

Developers can create different agents using these services, each with their own plugin. You can then pass in the agents as parameters to the Agents SDK container.

You can also choose to implement AI services elsewhere. For example, you can implement the services in the agent class that gets built and passed to the agent at runtime. This class appears as `MyAgent` or `MyBot` in some of the samples. This approach allows you to use Semantic Kernel or even multiple orchestrators, and is built separately from the main agent.

C#

```
var builder = Kernel.CreateBuilder();

builder.AddAzureOpenAIChatCompletion(
    deploymentName: deploymentName,
    endpoint: endpoint,
    apiKey: apiKey);

var kernel = builder.Build();
```

Integrate with Copilot Studio

Article • 05/19/2025

As a developer, you have the flexibility to use the technology stack of your choice for AI Services when working with the Microsoft 365 Agents SDK. This flexibility includes the ability to use agents built in Microsoft Copilot Studio. Copilot Studio lets business users and makers create agents easily in an SaaS-based environment. Makers can then share the agents they build with developers to integrate in their own custom web or native applications. You can carry out integration in either of two ways:

1. Reference one or more agents built in Copilot Studio from your agent built with the Agents SDK. You can also integrate with other agents, such as those created with Azure.
2. Use the Microsoft 365 Agents SDK to integrate Copilot Studio directly within your web or native apps.

Using the Copilot Studio client library

There is a full guide and sample available to get started with the Copilot Studio client library in the Agents SDK in .NET and JavaScript. A Python client is coming soon.

Clone the Copilot Studio client sample from the Agents SDK repo and access the relevant readme.

The sample is configured as a console app so you can get started quickly. In this console app, you interact with a live Copilot Studio agent.

Integrate with web or native apps using Microsoft 365 Agents SDK

07/16/2025

> **Important**
>
> This content is intended for experienced IT professionals, such as IT admins or developers, who are familiar with developer tools, utilities, and integrated development environments (IDEs). It requires software development expertise.

After you create and test your agent in Copilot Studio, deploy it to your preferred channel to test how your target users will interact with it.

> **Note**
>
> If you want to use the Embed code for the web app in Copilot Studio, you must set your security authentication options to **No authentication**. Navigate to **Settings** > **Security** > **Authentication** and select **No authentication** to make it publicly available.

This guide covers how to take your Copilot Studio agent and integrate it with your existing web application (typically a website) or native application using the Copilot Studio client with the Microsoft 365 Agents SDK.

> **Note**
>
> For another option to connect a Copilot Studio or Microsoft 365 Agents SDK agent to a native mobile or Windows app, see Integrate with native apps using the Agents Client SDK.

Different parts of this guide are relevant depending on whether you have an existing UX/UI that you want to integrate the agent into, or if you plan to use a Microsoft-provided UX/UI. Use the following table to find the section that fits your scenario.

Expand table

Deployment Method	How, where, and why	Quick Link
Use the Default Web Chat Embed Code	With an agent that has **No Authentication** security setting enabled, it's available on the **Web** channel publishing pane in Copilot Studio. **Note**: This option only appears when you have **No Authentication** selected in Copilot Studio.	Use the default Web Chat Embed code (without development/code)
Connect to Copilot Studio with Agents SDK User sign-in	Use the Agents SDK connection string or configuration settings to directly integrate to your agent using user credentials using the copilot.	Configure your app registration for user interactive sign-in
Connect to Copilot Studio with Agents SDK Service Principal sign-in	To be used where you want the agent to have its own identity and not use on behalf of for the user accessing the agent. Useful in scenarios where you don't require user context but you still need to access privileged APIs or information and requires authentication. **Note**: To perform this task, you must have authentication for this agent in Copilot Studio set to **No Authentication**.	Configure your app registration for Service Principal
Use legacy DirectLine to connect to Copilot Studio	You can connect to DirectLine using the DirectLine API and should be used where the Agents SDK doesn't support your scenario.	Publish an agent to mobile or custom apps

Prerequisites

- .NET Core / JS/ Python
- Packages–Copilot Studio client
- An agent

Methods to Integrate your Copilot Studio agent

- **Copilot Studio client (using the Agents SDK)**: This method is the preferred way to integrate with Copilot Studio.
- **DirectLine**: DirectLine is the legacy way to integrate with Copilot Studio and should be used where there's no support for your use case with Copilot Studio client.
  - Doesn't support service principal token

Get Started: Basic configuration and agent connection testing using the Microsoft 365 Agents SDK

1. Download the Copilot Studio client Sample from the Microsoft 365 Agents SDK.

   We simplified integration with the Agents SDK for your web and native apps by providing a client library that allows developers to enter a few details about your agent and to easily integrate it into your applications.

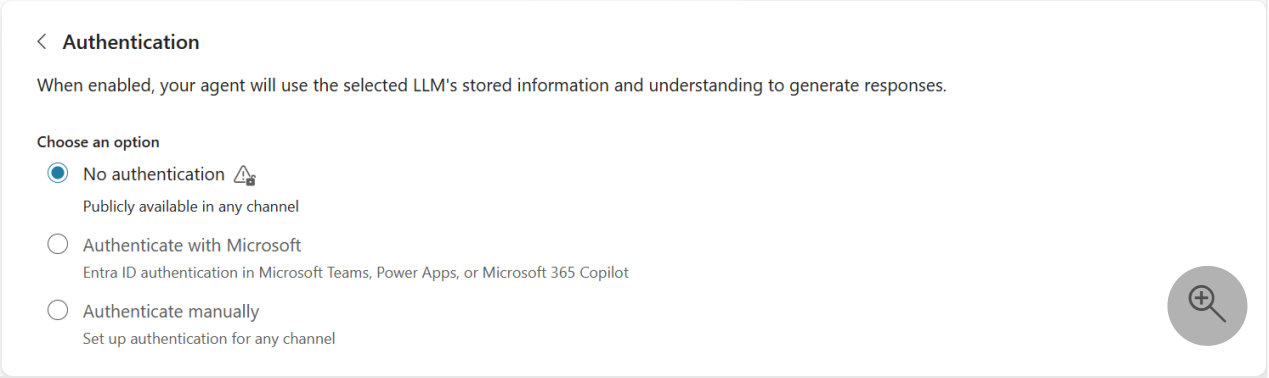
2. Get the Copilot Studio client sample from the Agents SDK GitHub repo in either C#, JavaScript or Python.

3. Get the Embed code or connection string for your agent in Copilot Studio.

   You need access to your agent in Copilot Studio to get the connection string details (or details for the configuration setting) to be able to connect to it.

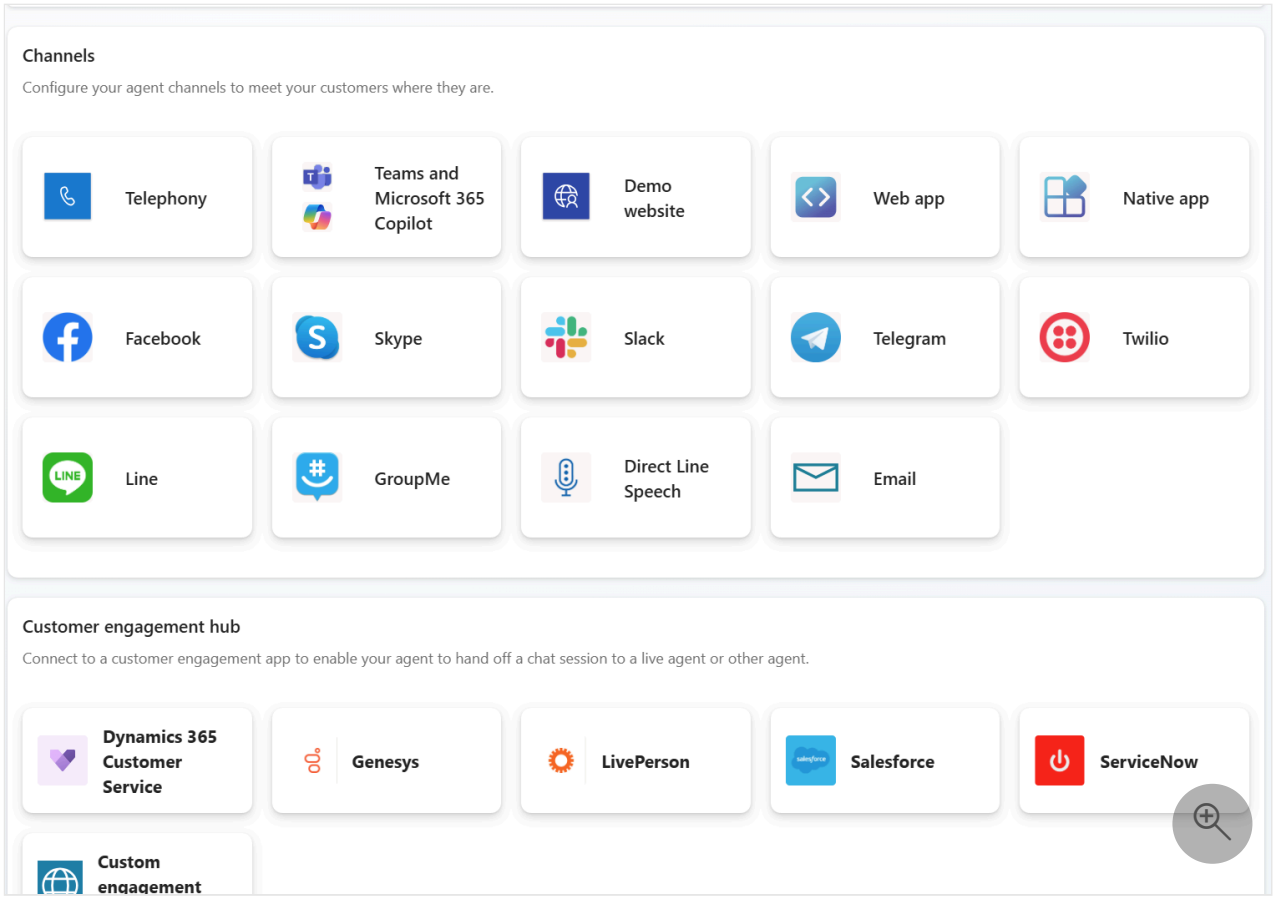
   In Copilot Studio, open your agent, select **Settings** > **Security** > **Authentication**, and then review your agent's settings.

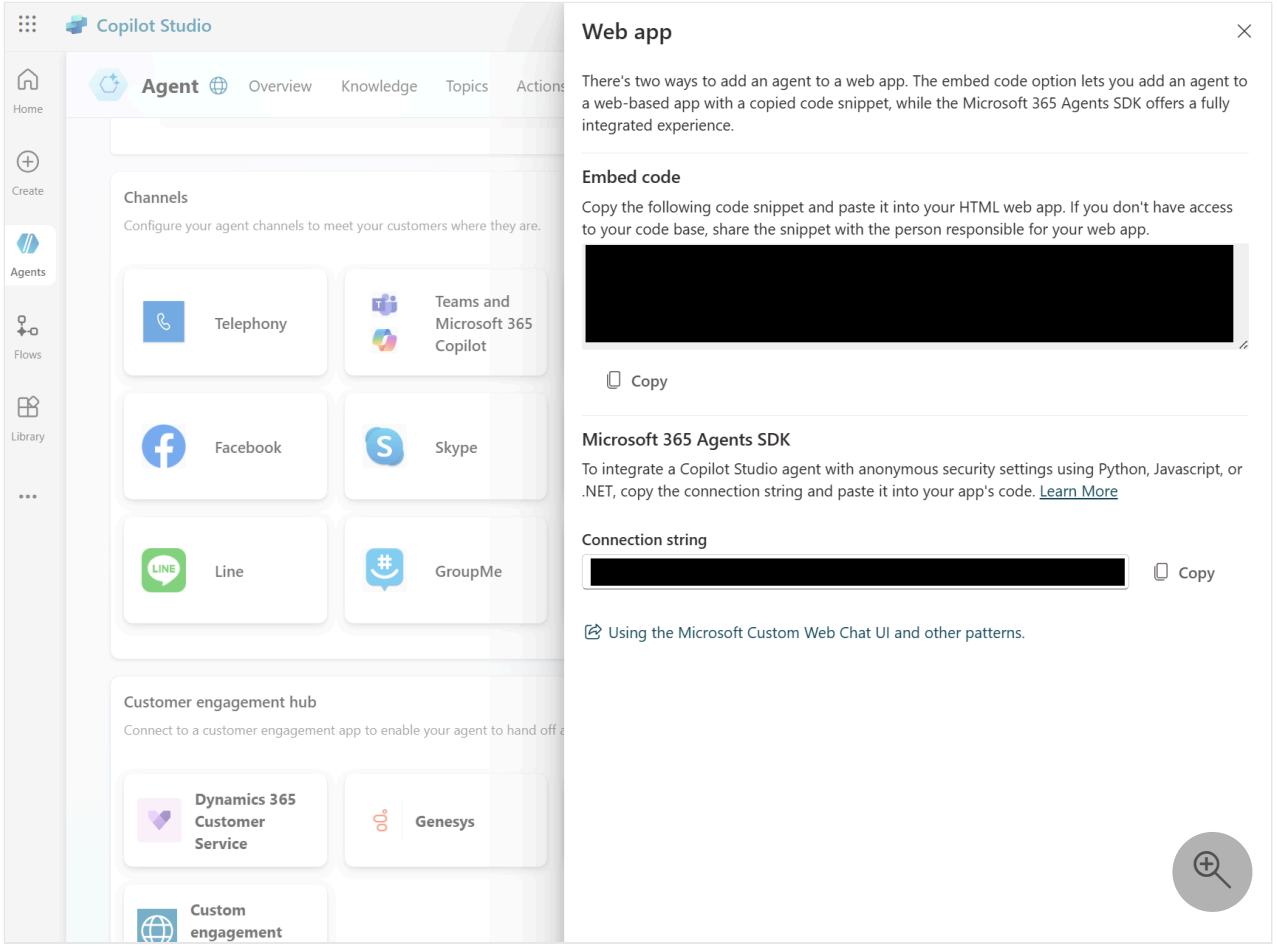
   - If either **Authenticate with Microsoft** or **Authenticate manually** is selected, you see only the *connection string* option to integrate with Agents SDK.
   - If **No authentication** is selected, you see both the Embed code that you can add to your website and the connection string. The Embed code option doesn't use the Agents SDK and uses the standard out-of-the-box Web Chat component.



4. Get your connection string.

   To get the connection string for your agent in Copilot Studio, select either **Web app** or **Native app** on the **Channels** page. Select **Copy** next to the connection string under **Microsoft 365 Agents SDK**. Use this string to connect to this agent from your web or native app's code.

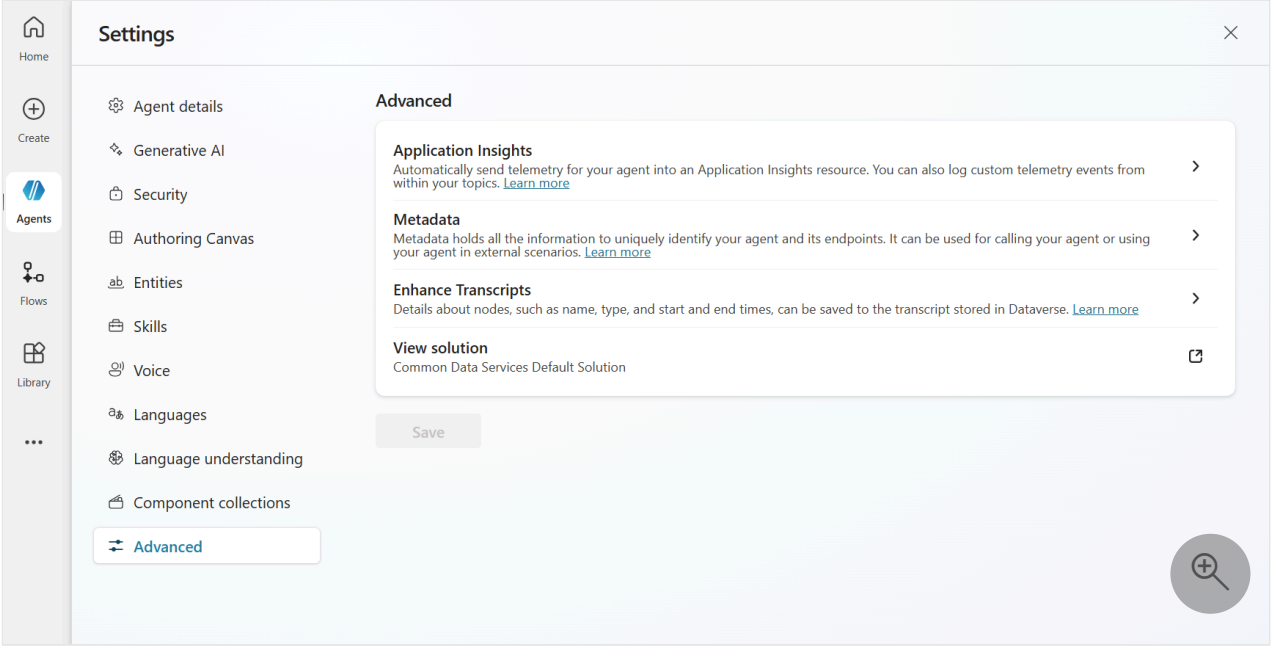




5. Get information for traditional configuration settings.

If you don't want to use the *connection string* method, and instead use the traditional configuration settings, you need some other metadata from your Copilot Studio agent.

Select **Settings** > **Advanced**. Under **Metadata**, you need the **Environment ID**, **Tenant ID**, and **Schema name**. Record these metadata values for later.



6. Configure your application registration for the correct permissions to access Copilot Studio.

   Your web or native app needs to have an app registration in Azure. If you don't have an app registration in Azure, you can follow the full guide on setting one up using the Readme or in the Azure documentation. For more information, see Register an application in Microsoft Entra ID.

   > **Note**
   >
   > Most likely you already have an existing app registration for your registration, and you need to use that instead. You can configure *User* or *Service Principal* authentication methods to access your agent.

Configure your app registration for user interactive sign-in

1. Navigate to **API permission** > **Add permissions**, select **APIs my organization uses**, and search for `Power Platform API`.

2. Select **delegated permissions** > **Copilot Studio** > ***Copilot Studio.Copilots.Invoke*** permission. Select **Add Permissions**.

For user sign-in, you can test the sample work with your application registration and your Copilot Studio agent, by:

Adding the connection string into the Copilot Studio client settings in the *appsettings.config* file:

- `DirectConnectUrl`
- `TenantID:`
- `ClientID:`
- `ClientSecret:`

Adding the Copilot Studio client settings in the *appsettings.config* file:

- `EnvironmentID:`
- `SchemaName:`
- `TenantID:`
- `AppClientID:`
- `AppClientSecret:`

You should now be able to run the sample and connect to the agent via your app registration settings using the sample console application.

Configure your app registration for Service Principal

Alternatively, you might choose to configure your app registration for service authentication rather than for user authentication.

1. On your app registration, go to API permission, add permissions, select **APIs my organization uses**, and search for `Power Platform API`.
2. Select **Application permissions** > **Copilot Studio**, and check the **Copilot Studio.Copilots.Invoke** permission.
3. Select **Add Permissions**.

For user sign-in, you can test the sample work with your application registration and your Copilot Studio agent, by:

1. Adding the Connection String into the Copilot Studio client settings in the *appsettings.config* file:

    - `DirectConnectUrl`
    - `TenantID:`
    - `UseS2SConnection: true`
    - `ClientID:`

- `ClientSecret:`

2. Adding the Copilot Studio client settings in the *appsettings.config* file:

   - `EnvironmentID:`
   - `SchemaName:`
   - `TenantID:`
   - `UseS2SConnection: true`
   - `AppClientID:`
   - `AppClientSecret:`

You should now be able to run the sample and connect to the agent with your app registration settings using the sample console application.

Integrate the Copilot Studio client into your existing UI/UX

Now you have tested your agent with the Copilot Studio client. Your testing confirms your agent is connected with the sample console app, you're ready to integrate the library into your existing website or app, and connect/surface it with your existing UI.

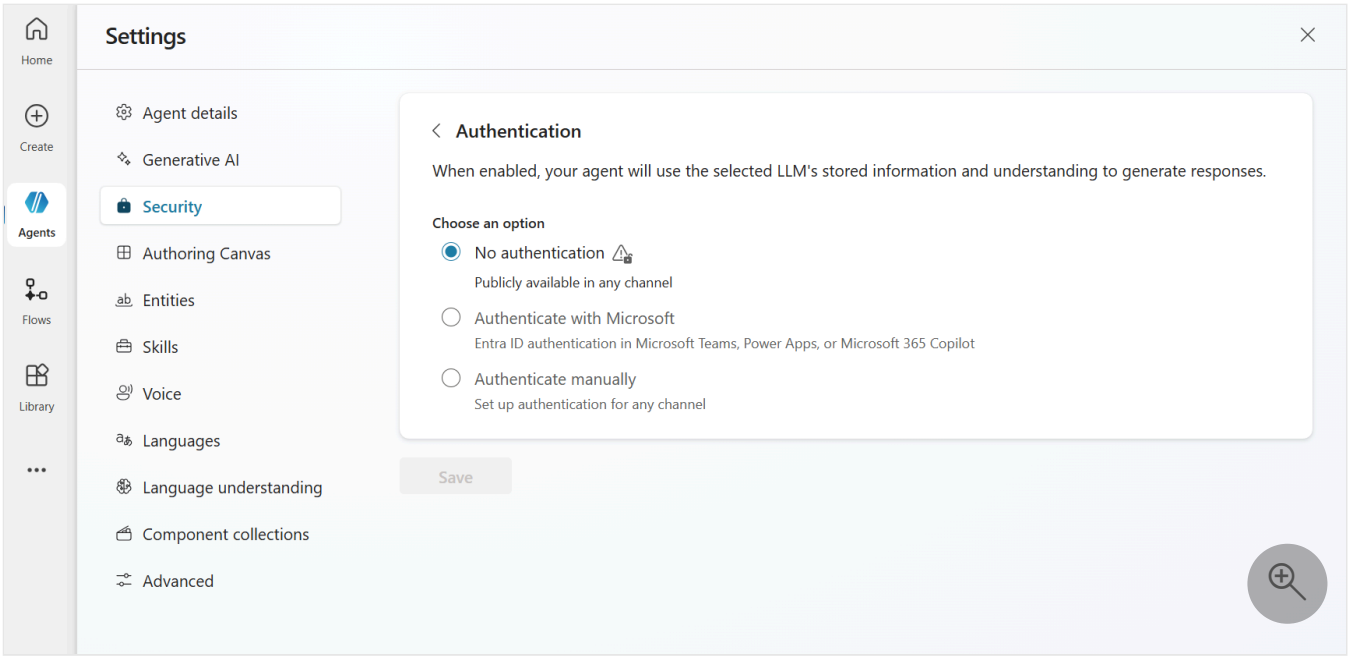
How you integrate your agent into the application is up to you and your existing code base. Typically, steps to integrate your agent might include:

1. Referencing the library in your application.
2. Implementing objects and methods from the client library in your project.
3. Injecting with DI or managing the client based on the existing design of your application, ensuring you reference the app settings for the configuration details for the agent.

Use the default Web Chat Embed code (without development/code)

You can add the Copilot Studio agent to your website using an iFrame, embeddable in the HTML of the website/web app.

This code is visible only if the **No authentication** option is selected in the agent's settings in Copilot Studio under **Security**. If **Authenticate with Microsoft** or **Authenticate manually** is selected, the embed code isn't visible.



Use DirectLine to connect to Copilot Studio

See Publish an agent to mobile or custom apps to use DirectLine to integrate with your Copilot Studio agent.

Microsoft 365 Copilot APIs client libraries (preview)

06/16/2025

The Microsoft 365 Copilot APIs client libraries are designed to facilitate the development of high-quality, efficient, and resilient AI solutions that access the Copilot APIs. These libraries include service and core libraries.

The service libraries offer models and request builders that provide a rich, typed experience for working with Microsoft 365 Copilot APIs.

The core libraries offer advanced features to facilitate interactions with the Copilot APIs. These features include embedded support for retry handling, secure redirects, transparent authentication, and payload compression. These capabilities help you enhance the quality of your AI solution's communications with the Copilot APIs without adding complexity. Additionally, the core libraries simplify routine tasks such as paging through collections and creating batch requests.

Supported languages

The Copilot APIs client libraries are currently available for the following languages:

- C#
- TypeScript
- Python

Client libraries in preview status

The Copilot APIs client libraries can be in preview status when initially released or after a significant update. Avoid using the preview release of these libraries in production solutions, regardless of whether your solution uses version 1.0 or the beta version of the Copilot APIs.

Client libraries support

The Copilot API libraries are open-source projects on GitHub. If you encounter a bug, file an issue with the details on the Issues tab. Contributors review and release fixes as needed.

Install the libraries

The Copilot API client libraries are included as a module in the Microsoft 365 Agents SDK. These libraries can be included in your projects via GitHub and popular platform package managers.

Install the Copilot APIs .NET client libraries

The Copilot APIs .NET client libraries are available in the following NuGet packages:

- Microsoft.Agents.M365Copilot.Beta - Contains the models and request builders for accessing the beta endpoint. Microsoft.Agents.M365Copilot.Beta has a dependency on Microsoft.Agents.M365Copilot.Core. The same dependency structure applies to both the TypeScript and Python libraries as well.
- Microsoft.Agents.M365Copilot.Core - The core library for making calls to the Copilot APIs.

To install the Microsoft.Agents.M365Copilot packages into your project, use the dotnet CLI, the Package Manager UI in Visual Studio, or the Package Manager Console in Visual Studio.

dotnet CLI

.NET CLI

```
dotnet add package Microsoft.Agent.M365Copilot.Beta
```

Package Manager Console

PowerShell

```
Install-Package Microsoft.Agent.M365Copilot.Beta
```

Create a Copilot APIs client and make an API call

The following code example shows how to create an instance of a Microsoft 365 Copilot APIs client with an authentication provider in the supported languages. The authentication provider handles acquiring access tokens for the application. Many different authentication providers are available for each language and platform. The different authentication providers support different client scenarios. For details about which provider and options are appropriate for your scenario, see Choose an Authentication Provider.

The example also shows how to make a call to the Retrieval API. To call this API, you first need to create a request object, and then run the POST method on the request.

The client ID is the app registration ID that is generated when you register your app in the Azure portal.

C#

C#

```
using Azure.Identity;
using Microsoft.Agents.M365Copilot.Beta;
using Microsoft.Agents.M365Copilot.Beta.Models;
using Microsoft.Agents.M365Copilot.Beta.Copilot.Retrieval;

var scopes = new[] {"Files.Read.All", "Sites.Read.All"};

// Multi-tenant apps can use "common",
// single-tenant apps must use the tenant ID from the Azure portal
var tenantId = "YOUR_TENANT_ID";

// Value from app registration
var clientId = "YOUR_CLIENT_ID";

// using Azure.Identity;
var deviceCodeCredentialOptions = new DeviceCodeCredentialOptions
{
   ClientId = clientId,
   TenantId = tenantId,
   // Callback function that receives the user prompt
   // Prompt contains the generated device code that user must
   // enter during the auth process in the browser
   DeviceCodeCallback = (deviceCodeInfo, cancellationToken) =>
   {
       Console.WriteLine(deviceCodeInfo.Message);
       return Task.CompletedTask;
   },
};

// https://learn.microsoft.com/dotnet/api/azure.identity.devicecodecredential
var deviceCodeCredential = new
DeviceCodeCredential(deviceCodeCredentialOptions);


//Create the client with explicit base URL
var baseURL = “https://graph.microsoft.com/beta”;
AgentsM365CopilotBetaServiceClient client = new
AgentsM365CopilotBetaServiceClient (deviceCodeCredential, scopes, baseURL);

try
{
```

```
  var requestBody = new RetrievalPostRequestBody
  {
      DataSource = RetrievalDataSource.SharePoint,
      QueryString = "What is the latest in my organization?",
      MaximumNumberOfResults = 10
  };

  var result = await client.Copilot.Retrieval.PostAsync(requestBody);
  Console.WriteLine($"Retrieval post: {result}");

  if (result != null)
  {
    Console.WriteLine("Retrieval response received successfully");
    Console.WriteLine("\nResults:");
    Console.WriteLine(result.RetrievalHits.Count.ToString());
    if (result.RetrievalHits != null)
    {
       foreach (var hit in result.RetrievalHits)
       {
         Console.WriteLine("\n---");
         Console.WriteLine($"Web URL: {hit.WebUrl}");
         Console.WriteLine($"Resource Type: {hit.ResourceType}");
         if (hit.Extracts != null && hit.Extracts.Any())
         {
            Console.WriteLine("\nExtracts:");
            foreach (var extract in hit.Extracts)
            {
              Console.WriteLine($"  {extract.Text}");
            }
          }
          if (hit.SensitivityLabel != null)
          {
            Console.WriteLine("\nSensitivity Label:");
            Console.WriteLine($"  Display Name:
{hit.SensitivityLabel.DisplayName}");
            Console.WriteLine($"  Tooltip: {hit.SensitivityLabel.Tooltip}");
            Console.WriteLine($"  Priority: {hit.SensitivityLabel.Priority}");
            Console.WriteLine($"  Color: {hit.SensitivityLabel.Color}");
            if (hit.SensitivityLabel.IsEncrypted.HasValue)
            {
              Console.WriteLine($"  Is Encrypted:
{hit.SensitivityLabel.IsEncrypted.Value}");
            }
          }
        }
      }
      else
      {
        Console.WriteLine("No retrieval hits found in the response");
      }
  }
}
catch (Exception ex)
{
    Console.WriteLine($"Error making retrieval request: {ex.Message}");
```

```
    Console.Error.WriteLine(ex);
}
```

Related content

- Overview of the Microsoft 365 Copilot Retrieval API
- Use the Retrieval API

Azure Bot Framework SDK to Agents SDK migration guidance

07/16/2025

The Microsoft 365 Agents SDK provides developers the ability to create and tailor agents using the AI stack of their choice. Developers can create a custom engine agent (CEA) and deploy it to Microsoft 365 Copilot. Developers can get started with the Microsoft 365 Agents Toolkit for Visual Studio and Visual Studio Code to get started quickly with scaffolding and templates. Developers can add their chosen models and orchestrator from Azure Foundry and Semantic Kernel, OpenAI Agents, LangChain, or even a custom built orchestrator. Developers can even choose to bring multiple agents built with different technologies, and surface the agents through Microsoft 365 Copilot.

Using the Microsoft 365 Agents SDK, we want you to be able to build an agent quickly and surface it on any channel, including Microsoft 365 Copilot and Microsoft Teams.

The SDK is designed to be un-opinionated about the AI you use. You can implement agentic patterns without being locked into a tech stack.

The SDK takes advantage of specific client channel behavior, such as Microsoft 365 Copilot, Teams, and other non-Microsoft channels, to allow you to tailor your agent to client channels, including specific events or actions.

Unsupported and deprecated packages

The Microsoft 365 Agents SDK is the evolution of the Azure Bot Framework SDK. The Azure Bot Framework was previously the way for a developer to build bots with a primary focus on conversational AI around topics, dialogs, and messages. The industry norm is now to use generative AI functionality, grounded on knowledge located across the enterprise. Companies need to be able to orchestrate actions, and answer questions, from within a conversational experience. The Microsoft 365 Agents SDK provides capabilities for modern agent development, bringing together the creation of conversational agents with conversation management and orchestration. Agents built with the SDK can connect to numerous AI services and clients, including agents created with non-Microsoft software or technology.

This article is written to provide guidance and awareness if you're considering, or in the process of, migrating from the Azure Bot Framework to the Agents SDK. With this information, you can stay informed and make more informed decisions.

The functionalities in the following list aren't supported in Agents SDK. Bots requiring these functionalities won't be able to migrate without implementing alternatives:

1. **Adaptive Dialogs**: The Adaptive Dialog System (implemented in C# in the Bot Framework) is no longer directly relevant. We don't plan to move this system into the Agents SDK
2. **AdaptiveExpressions**: This includes `Microsoft.Bot.AdaptiveExpressions.Core`. However, these packages are still available for use from the .NET BotBuilder packages. These packages don't take dependencies on SDK packages, and can be used if you wish with Agents SDK. The AdaptiveExpressions functionality, however, isn't actively supported.
3. **Bot FrameworkComposer Artifacts**: Artifacts from Composer (adaptive dialogs, adaptive expressions, and so on, implemented in C# in the Bot Framework) are no longer required and aren't being brought forward.
4. **Previous Generation AI Tooling**: Tools such as LUIS, Orchestrator, and QnA Maker are no longer needed. The online services for these tools are already disabled. Existing bots that depend on these services need to migrate to different tooling.
5. **Language Understanding**: This includes `Microsoft.Bot.Builder.Parsers.LU`.
6. **Language Generation**: LG tooling, template tooling, and related parsers aren't needed. General purpose LLMs replace these tools.
7. **TemplateManager**
8. **ASP.Net WebAPI**: ASP.NET WebAPI and similar older technologies are no longer needed. This impacts only C#, where only the current generation of ASP.NET Core (ironically called "ASP.NET Core Web API") is supported.
9. **Application Insights**
10. **Streaming Connections**
11. **QueueStorage**
12. **Inspection**
13. **BotFrameworkAdapter**: Replaced by `CloudAdapter` in Bot Framework, and removed from the Agents SDK.
14. **Deprecated Activities**. Older activities are deprecated, such as payments activities.
15. **Generators**. The legacy generators for each of the languages (Yeoman, and so on), aren't being brought forward.
16. **CLI** All commands in the Bot Framework CLI ("bf") are deprecated.

DotNet SDK Code changes

Azure resources

- Your Azure resources remain unchanged.
- You need to reference your appsettings properties for `MicrosoftAppType`, `MicrosoftAppId`, `MicrosoftAppPassword`, and `MicrosoftAppTenantId`. However those setting names are no longer used and can be deleted later.

First steps

- Update package dependencies. This won't get all required namespaces settled, but it will get the bulk of them.

Expand table

Bot Framework SDK	Agents SDK
Microsoft.Bot.Builder.Integration.AspNet.Core	Microsoft.Agents.Hosting.AspNetCore and Microsoft.Agents.Authentication.Msal
Microsoft.Bot.Builder	Microsoft.Agents.Builder
Microsoft.Bot.Builder.Dialogs	Microsoft.Agents.Builder.Dialogs
Microsoft.Bot.Builder.Azure.Blobs	Microsoft.Agents.Storage.Blobs
Microsoft.Bot.Builder.Azure	Microsoft.Agents.Storage.CosmosDb
Microsoft.Bot.Schema	Microsoft.Agents.Core

- If your bot uses Teams, add a package dependency for `Microsoft.Agents.Extensions.Teams`
- Remove dependency on NewtonSoft. Agents SDK uses `System.Text.Json`.
- Update namespaces
  - Easiest to replace in files across solution (find and replace exact text)

Expand table

Bot Framework Namespace	Agents SDK Namespace
using Microsoft.Bot.Builder.Integration.AspNet.Core;	using Microsoft.Agents.Hosting.AspNetCore;
using Microsoft.Bot.Builder;	using Microsoft.Agents.Builder;
Microsoft.Bot.Builder.Dialogs	Microsoft.Agents.Builder.Dialogs
using Microsoft.Bot.Schema;	using Microsoft.Agents.Core.Models;
Microsoft.Bot.Connector.Authentication	Microsoft.Agents.Connector
using Microsoft.Bot.Builder.Teams;	using Microsoft.Agents.Extensions.Teams.Compat;

Bot Framework Namespace	Agents SDK Namespace
using Microsoft.Bot.Schema.Teams;	using Microsoft.Agents.Extensions.Teams.Models;
using Newtonsoft.Json;	using System.Text.Json;
using Newtonsoft.Json.Linq;	**blank to remove**

- Make changes to type/property names

Expand table

Bot Framework	Agents SDK
BotState	AgentState
OAuthPromptSettings.ConenctionName	OAuthPromptSettings.AzureBotOAuthConnectionName
IAttachments.GetAttachmentInfo	IAttachments.GetAttachmentInfoAsync
IBotFrameworkHttpAdapter	IAgentHttpAdapter
BotAdapter	ChannelAdapter
CloudAdapterBase	ChannelServiceAdapterBase

- Make the following replacements in your code:
  - `TurnState.Get<ConnectorClient>` to `.Services.Get<IConnectorClient>`
  - `.TurnState.Get<IUserTokenClient>` to `.Services.Get<IUserTokenClient>`
  - `.TurnState.` to `.StackState.`
  - `.StackState.Add` to `.StackState.Set`

- Startup
  - Initializing (Startup) is organized differently in Agents SDK
    - In most Bot Framework SDK bots, DI registration happens in `Startup.cs`, which is referenced from `Program.cs`
    - In Agents SDK, we default to everything in `Program.cs`. We recommend you switch to this.

- Authentication
  - Bot Framework SDK included incoming JWT token authentication in the stack. Agents SDK does not.
  - Copy AspNetExtensions to your project folder. This provides AspNet Authentication for incoming requests.

- Paste the following into your `Program.cs`, replacing the current contents.

chsarp

```
using Microsoft.Agents.Builder;
using Microsoft.Agents.Builder.State;
using Microsoft.Agents.Hosting.AspNetCore;
using Microsoft.Agents.Storage;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System.Threading;
using FullAuthentication;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddHttpClient();

// Register your bot.
// A Dialog based bot would look something like:
// builder.AddAgent<MyBot<MainDialog>>();
//
// If you used custom (CloudAdpater subclass):
// builder.AddAgent<MyBot, MyAdapter>();
builder.AddAgent<MyBot>();

// This is the same as in BF SDK and can be replaced with what you had in
Startup.cs
builder.Services.AddSingleton<IStorage, MemoryStorage>();

builder.Services.AddSingleton<ConversationState>();
builder.Services.AddSingleton<UserState>();

// If you are using dialogs, copy your equivelent from Startup.cs
builder.Services.AddSingleton<MainDialog>();

// Perform any other DI included in your Startup.cs
// Do not include:
//    services.AddSingleton<BotFrameworkAuthentication,
ConfigurationBotFrameworkAuthentication>();
//    services.AddSingleton<IBotFrameworkHttpAdapter, ???>();
//    services.AddTransient<IBot, ???>();
//    services.AddSingleton<ServiceClientCredentialsFactory>(...)
//    services.AddHttpClient().AddControllers().AddNewtonsoftJson

// Configure the HTTP request pipeline.

// Add AspNet token validation for Azure Bot Service and Entra.
Authentication is
// configured in the appsettings.json "TokenValidation" section.
builder.Services.AddControllers();
builder.Services.AddAgentAspNetAuthentication(builder.Configuration);

var app = builder.Build();
```

```
// Enable AspNet authentication and authorization
app.UseAuthentication();
app.UseAuthorization();

app.MapPost("/api/messages", async (HttpRequest request, HttpResponse
response, IAgentHttpAdapter adapter, IAgent agent, CancellationToken
cancellationToken) =>
{
    await adapter.ProcessAsync(request, response, agent, cancellationToken);
}).RequireAuthorization(); ;

app.Run();
```

- appsettings
  - Reference your existing appsettings to add the following
  - TokenValidation

JSON

```
"TokenValidation": {
  "Audiences": [
    "{{MicrosoftAppId-value}}"
  ],
  "TenantId": "{{MicrosoftTenantId-value}}"
},
```

  - Connections (sample setup using SingleTenant & ClientSecrets)

JSON

```
"Connections": {
  "ServiceConnection": {
    "Settings": {
      "AuthType": "ClientSecret",
      "AuthorityEndpoint":
"https://login.microsoftonline.com/{{MicrosoftTenantId-value}}",
      "ClientId": "{{MicrosoftAppId-value}}",
      "ClientSecret": "{{MicrosoftAppPassword-value}}",
      "Scopes": [
        "https://api.botframework.com/.default"
      ]
    }
  }
},
"ConnectionsMap": [
  {
    "ServiceUrl": "*",
    "Connection": "ServiceConnection"
```

```
    }
  ],
```

Serialization changes

- Some `TeamsActivityHandler` methods required `JObject` arguments. Replaced with `JsonElement`
- Remove line `builder.Services.AddControllers().AddNewtonsoftJson();` from Program.cs unless your bot code is using it. Not needed by Agents SDK.

State

- `ConversationState`, `UserState`, and `PrivateConversationState` are compatible with the following differences
  - `IStatePropertyAccessor` is deprecated, but still functions as before. You should change to using the `IAgentState` methods.

- `Dialog.RunAsync` still accepts an `IStatePropertyAccessor` argument. However, it also supports just passing `ConversationState` which is recommended.

- `AutoSaveStateMiddleware` has been enhanced to support auto load/save of State. This should NOT be used for `AgentApplication` based bots. But for `ActiviyHandler` based bots, it can be added to your Adapter to perform auto load/save of all:

  C#

  ```
  adapter.Use(new AutoSaveStateMiddleware(true, conversationState, userState));
  ```

Skills

- Skills functionality exists in `Microsoft.Agents.Client`
  - `SkillConversationIdFactoryBase` replaced with `IConversationIdFactory`
  - `SkillConversationIdFactory` replaced with `ConversationIdFactory`
  - `SkillConversationIdFactoryOptions` replaced with `ConversationIdFactoryOptions`
  - `SkillsConfiguration` replaced with `IChannelHost` & `ConfigurationChannelHost`
  - `CloudSkillHandler` replaced with `HttpBotChannel` and associated `HttpBotChannelFactory`
  - Response handling now handled by `IChannelApiHandler`
  - The expected Bot Framework Skill handling is in `ProxyChannelApiHandler`

    - Different configuration naming and format

Related Content

For more information on specifics of how the Microsoft 365 Agents SDK works, check out the official documentation:

- Managing Turns in the Agents SDK
- Using Activities
- Creating Messages

Microsoft 365 Agents SDK JavaScript reference

07/22/2025

Version 1.0.0

The Microsoft 365 Agent SDK simplifies building full stack, multichannel, trusted agents for platforms including Microsoft 365, Teams, Copilot Studio, and Web chat. We also offer integrations with third parties such as Facebook Messenger, Slack, or Twilio. The SDK provides developers with the building blocks to create agents that handle user interactions, orchestrate requests, reason about responses, and collaborate with other agents.

The Agents SDK is a comprehensive framework for building enterprise-grade agents, enabling developers to integrate components from the Azure AI Foundry SDK, Semantic Kernel, and AI components from other vendors.

For more information, see the Microsoft 365 Agents SDK documentation.

Getting started

Sample code is a great way to get started. Look at these samples located at https://github.com/microsoft/Agents/samples

Expand table

Name	Description	Location
Empty Agent	Simplest agent	basic/empty-agent/nodejs
AI: Weather Agent	WeatherAgent with LangChain tools	basic/weather-agent/nodejs
AI: Poem Agent	Agent calling OpenAI with Vercel AI	basic/azureai-streaming-poem-agent
Auth: Auto Signin	Shows how to apply auth handlers to routes to integrate with ABS OAuth connections	basic/authorization/auto-signin
Auth: On Behalf Of (OBO)	Shows how to request a token on behalf of	basic/authorization/auto-signin
Copilot Studio Client	Consume Copilot Studio Agent from a console	basic/authorization/obo-authorization
Copilot Studio	Consume Copilot Studio Agent from WebChat	basic/copilotstudio-webclient

Name	Description	Location
WebClient		
Copilot Studio WebClient React	Consume Copilot Studio Agent from WebChat with React	basic/copilotstudio-webchat-react
Copilot Studio Skill	Call the empty agent from a Copilot Studio skill	complex/copilotstudio-skill/nodejs

Packages Overview

We offer the following NPM packages to create conversational experiences based on agents:

Expand table

Package	Description
@microsoft/agents-activity	Types and validators implementing the Activity protocol spec.
@microsoft/agents-copilotstudio-client	Direct to Engine client to interact with Agents created in Copilot Studio.
@microsoft/agents-hosting	Provides classes to implement and host agents.
@microsoft/agents-hosting-express	Provides a startServer method to host an agent in express.
@microsoft/agents-hosting-dialogs	Provides classes to host an Agent in express.
@microsoft/agents-hosting-storage-blob	Extension to use Azure Blob as storage.
@microsoft/agents-hosting-storage-cosmos	Extension to use Cosmos DB as storage.

Environment requirements

The packages should target node20 or greater, and can be used from JavaScript using CommonJS or ES6 modules, or from TypeScript.

> **Note**
>
> We're using node 22 to be able to initialize the process from a `.env` file without adding the dependency to **dotenv** by using the `--env-file` flag. Previous node versions should set the env vars explicitly before running.

Microsoft 365 Agents SDK Python reference

Article • 04/29/2025

The Microsoft 365 Agent SDK simplifies building full stack, multichannel, trusted agents for platforms including M365, Teams, Copilot Studio, and Webchat. We also offer integrations with 3rd parties such as Facebook Messenger, Slack, or Twilio. The SDK provides developers with the building blocks to create agents that handle user interactions, orchestrate requests, reason about responses, and collaborate with other agents.

The Agents SDK is a comprehensive framework for building enterprise-grade agents, enabling developers to integrate components from the Azure AI Foundry SDK, Semantic Kernel, as well as AI components from other vendors.