Microsoft Agent Framework

10/01/2025

The Microsoft Agent Framework is an open-source development kit for building **AI agents** and **multi-agent workflows** for .NET and Python. It brings together and extends ideas from the Semantic Kernel and AutoGen projects, combining their strengths while adding new capabilities. Built by the same teams, it is the unified foundation for building AI agents going forward.

The Agent Framework offers two primary categories of capabilities:

- AI Agents: individual agents that use LLMs to process user inputs, call tools and MCP servers to perform actions, and generate responses. Agents support model providers including Azure OpenAI, OpenAI, and Azure AI.
- Workflows: graph-based workflows that connect multiple agents and functions to perform complex, multi-step tasks. Workflows support type-based routing, nesting, checkpointing, and request/response patterns for human-in-the-loop scenarios.

The framework also provides foundational building blocks, including model clients (chat completions and responses), an agent thread for state management, context providers for agent memory, middleware for intercepting agent actions, and MCP clients for tool integration. Together, these components give you the flexibility and power to build interactive, robust, and safe AI applications.

Why another agent framework?

Semantic Kernel and AutoGen pioneered the concepts of AI agents and multi-agent orchestration. The Agent Framework is the direct successor, created by the same teams. It combines AutoGen's simple abstractions for single- and multi-agent patterns with Semantic Kernel's enterprise-grade features such as thread-based state management, type safety, filters, telemetry, and extensive model and embedding support. Beyond merging the two, the Agent Framework introduces workflows that give developers explicit control over multi-agent execution paths, plus a robust state management system for long-running and human-in-the-loop scenarios. In short, the Agent Framework is the next generation of both Semantic Kernel and AutoGen.

To learn more about migrating from either Semantic Kernel or AutoGen, see the Migration Guide from Semantic Kernel and Migration Guide from AutoGen.

Both Semantic Kernel and AutoGen have benefited significantly from the open-source community, and we expect the same for the Agent Framework. The Microsoft Agent

Framework will continue to welcome contributions and will keep improving with new features and capabilities.

> **Note**
>
> Microsoft Agent Framework is currently in public preview. Please submit any feedback or issues on the GitHub repository.

> **Important**
>
> If you use the Microsoft Agent Framework to build applications that operate with third-party servers or agents, you do so at your own risk. We recommend reviewing all data being shared with third-party servers or agents and being cognizant of third-party practices for retention and location of data. It is your responsibility to manage whether your data will flow outside of your organization's Azure compliance and geographic boundaries and any related implications.

Installation

Python:

```Bash
pip install agent-framework
```

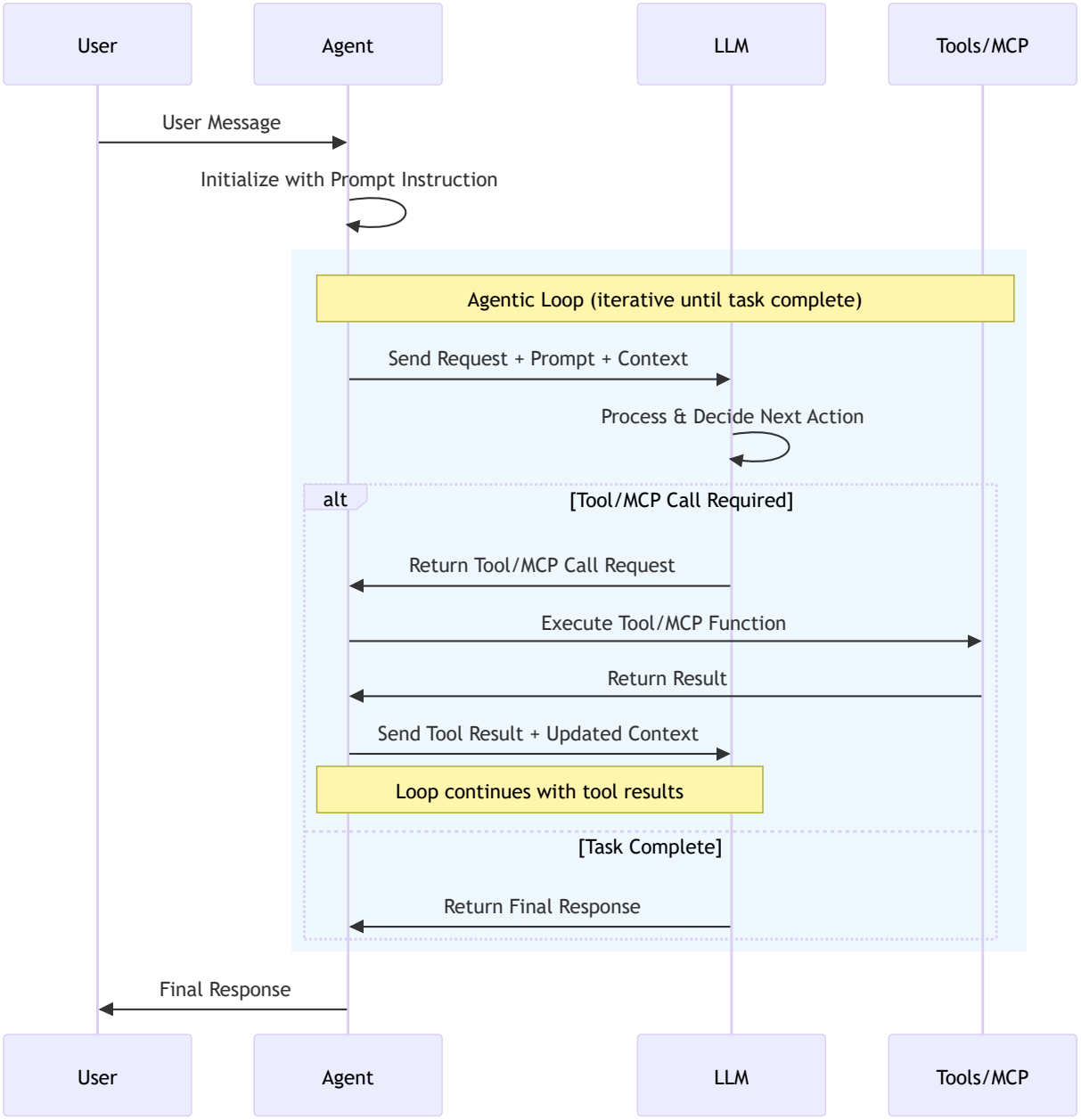
.NET:

```Bash
dotnet add package Microsoft.Agents.AI
```

AI Agents

What is an AI agent?

An **AI agent** uses an LLM to process user inputs, make decisions, call tools and MCP servers to perform actions, and generate responses. The following diagram illustrates the core components and their interactions in an AI agent:



An AI agent can also be augmented with additional components such as a thread, a context provider, and middleware to enhance its capabilities.

When to use an AI agent?

AI agents are suitable for applications that require autonomous decision-making, ad hoc planning, trial-and-error exploration, and conversation-based user interactions. They are particularly useful for scenarios where the input task is unstructured and cannot be easily defined in advance.

Here are some common scenarios where AI agents excel:

- **Customer Support**: AI agents can handle multi-modal queries (text, voice, images) from customers, use tools to look up information, and provide natural language responses.

- **Education and Tutoring**: AI agents can leverage external knowledge bases to provide personalized tutoring and answer student questions.
- **Code Generation and Debugging**: For software developers, AI agents can assist with implementation, code reviews, and debugging by using various programming tools and environments.
- **Research Assistance**: For researchers and analysts, AI agents can search the web, summarize documents, and piece together information from multiple sources.

The key is that AI agents are designed to operate in a dynamic and underspecified setting, where the exact sequence of steps to fulfill a user request is not known in advance and may require exploration and close collaboration with users.

When not to use an AI agent?

AI agents are not well-suited for tasks that are highly structured and require strict adherence to predefined rules. If your application anticipates a specific kind of input and has a well-defined sequence of operations to perform, using AI agents may introduce unnecessary uncertainty, latency, and cost.

If you can write a function to handle the task, do that instead of using an AI agent. You can use AI to help you write that function.

A single AI agent may struggle with complex tasks that involve multiple steps and decision points. Such tasks may require a large number of tools (e.g., over 20), which a single agent cannot feasibly manage.

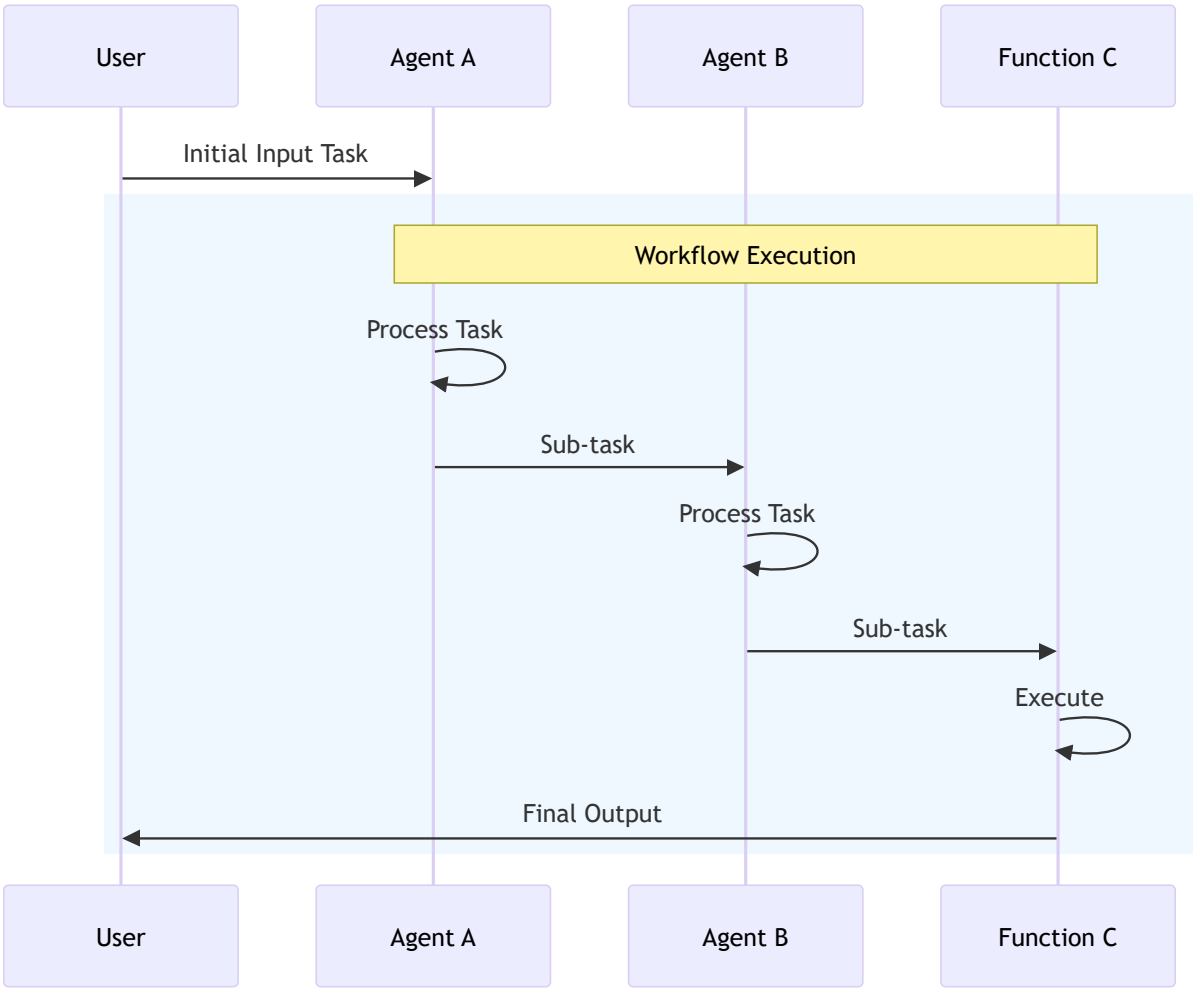
In these cases, consider using workflows instead.

Workflows

What is a Workflow?

A **workflow** can express a predefined sequence of operations that can include AI agents as components while maintaining consistency and reliability. Workflows are designed to handle complex and long-running processes that may involve multiple agents, human interactions, and integrations with external systems.

The execution sequence of a workflow can be explicitly defined, allowing for more control over the execution path. The following diagram illustrates an example of a workflow that connects two AI agents and a function:



Workflows can also express dynamic sequences using conditional routing, model-based decision making, and concurrent execution. This is how our multi-agent orchestration patterns are implemented. The orchestration patterns provide mechanisms to coordinate multiple agents to work on complex tasks that require multiple steps and decision points, addressing the limitations of single agents.

What problems do Workflows solve?

Workflows provide a structured way to manage complex processes that involve multiple steps, decision points, and interactions with various systems or agents. The types of tasks workflows are designed to handle often require more than one AI agent.

Here are some of the key benefits of Agent Framework workflows:

- **Modularity**: Workflows can be broken down into smaller, reusable components, making it easier to manage and update individual parts of the process.
- **Agent Integration**: Workflows can incorporate multiple AI agents alongside non-agentic components, allowing for sophisticated orchestration of tasks.
- **Type Safety**: Strong typing ensures messages flow correctly between components, with comprehensive validation that prevents runtime errors.

- **Flexible Flow**: Graph-based architecture allows for intuitive modeling of complex workflows with `executors` and `edges`. Conditional routing, parallel processing, and dynamic execution paths are all supported.
- **External Integration**: Built-in request/response patterns enable seamless integration with external APIs and support human-in-the-loop scenarios.
- **Checkpointing**: Save workflow states via checkpoints, enabling recovery and resumption of long-running processes on the server side.
- **Multi-Agent Orchestration**: Built-in patterns for coordinating multiple AI agents, including sequential, concurrent, hand-off, and Magentic.
- **Composability**: Workflows can be nested or combined to create more complex processes, allowing for scalability and adaptability.

Next steps

- Quickstart Guide
- Migration Guide from Semantic Kernel
- Migration Guide from AutoGen

Microsoft Agent Framework Quick Start

10/01/2025

This guide will help you get up and running quickly with a basic agent using the Agent Framework and Azure OpenAI.

Prerequisites

Before you begin, ensure you have the following:

- .NET 8.0 SDK or later
- Azure OpenAI resource with a deployed model (e.g., `gpt-4o-mini`)
- Azure CLI installed and authenticated (`az login`)
- User has the Cognitive Services OpenAI User or Cognitive Services OpenAI Contributor roles for the Azure OpenAI resource.

Note: The Microsoft Agent Framework is supported with all actively supported versions of .Net. For the purposes of this sample we are recommending the .NET 8.0 SDK or higher.

Note: This demo uses Azure CLI credentials for authentication. Make sure you're logged in with `az login` and have access to the Azure OpenAI resource. For more information, see the Azure CLI documentation. It is also possible to replace the `AzureCliCredential` with an `ApiKeyCredential` if you have an api key and do not wish to use role based authentication, in which case `az login` is not required.

Installing Packages

Packages will be published to NuGet Gallery | Microsoft.Agent.AI.

First, add the following Microsoft Agent Framework NuGet packages into your application, using the following commands:

PowerShell

```
dotnet add package Azure.AI.OpenAI
dotnet add package Azure.Identity
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

Running a Basic Agent Sample

This sample demonstrates how to create and use a simple AI agent with Azure OpenAI Chat Completion as the backend. It will create a basic agent using `AzureOpenAIClient` with `gpt-4o-mini` and custom instructions.

Sample Code

Make sure to replace `https://your-resource.openai.azure.com/` with the endpoint of your Azure OpenAI resource.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
  new Uri("https://your-resource.openai.azure.com/"),
  new AzureCliCredential())
    .GetChatClient("gpt-4o-mini")
    .CreateAIAgent(instructions: "You are good at telling jokes.");

Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate."));
```

(Optional) Installing Nightly Packages

If you need to get a package containing the latest enhancements or fixes nightly builds of the Agent Framework are available here.

To download nightly builds follow the following steps:

1. You will need a GitHub account to complete these steps.
2. Create a GitHub Personal Access Token with the `read:packages` scope using these instructions.
3. If your account is part of the Microsoft organization then you must authorize the `Microsoft` organization as a single sign-on organization.
   a. Click the "Configure SSO" next to the Personal Access Token you just created and then authorize `Microsoft`.
4. Use the following command to add the Microsoft GitHub Packages source to your NuGet configuration:

PowerShell

```
dotnet nuget add source --username GITHUBUSERNAME --password
GITHUBPERSONALACCESSTOKEN --store-password-in-clear-text --name
GitHubMicrosoft "https://nuget.pkg.github.com/microsoft/index.json"
```

5. Or you can manually create a `NuGet.Config` file.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="nuget.org" value="https://api.nuget.org/v3/index.json"
protocolVersion="3" />
    <add key="github"
value="https://nuget.pkg.github.com/microsoft/index.json" />
  </packageSources>

  <packageSourceMapping>
    <packageSource key="nuget.org">
      <package pattern="*" />
    </packageSource>
    <packageSource key="github">
      <package pattern="*nightly"/>
      <package pattern="Microsoft.Agents.AI" />
    </packageSource>
  </packageSourceMapping>

  <packageSourceCredentials>
    <github>
        <add key="Username" value="<Your GitHub Id>" />
        <add key="ClearTextPassword" value="<Your Personal Access Token>" />
      </github>
  </packageSourceCredentials>
</configuration>
```

- If you place this file in your project folder make sure to have Git (or whatever source control you use) ignore it.
- For more information on where to store this file go here.

6. You can now add packages from the nightly build to your project.

- E.g. use this command `dotnet add package Microsoft.Agents.AI --prerelease`

7. And the latest package release can be referenced in the project like this:

- `<PackageReference Include="Microsoft.Agents.AI" Version="*-*" />`

For more information see: https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-nuget-registry

Next steps

Create and run agents

Agent Framework Tutorials

09/25/2025

Welcome to the Agent Framework tutorials! This section is designed to help you quickly learn how to build, run, and extend agents using the Agent Framework. Whether you're new to agents or looking to deepen your understanding, these step-by-step guides will walk you through essential concepts such as creating agents, managing conversations, integrating function tools, handling approvals, producing structured output, persisting state, and adding telemetry. Start with the basics and progress to more advanced scenarios to unlock the full potential of agent-based solutions.

Agent getting started tutorials

These samples cover the essential capabilities of the Agent Framework. You'll learn how to create agents, enable multi-turn conversations, integrate function tools, add human-in-the-loop approvals, generate structured outputs, persist conversation history, and monitor agent activity with telemetry. Each tutorial is designed to help you build practical solutions and understand the core features step by step.

Create and run an agent with Agent Framework

10/01/2025

This tutorial shows you how to create and run an agent with the Agent Framework, based on the Azure OpenAI Chat Completion service.

> **Important**
>
> The agent framework supports many different types of agents. This tutorial uses an agent based on a Chat Completion service, but all other agent types are run in the same way. See the Agent Framework user guide for more information on other agent types and how to construct them.

Prerequisites

Before you begin, ensure you have the following prerequisites:

- .NET 8.0 SDK
- Azure OpenAI service endpoint and deployment configured
- Azure CLI installed and authenticated (for Azure credential authentication)
- User has the Cognitive Services OpenAI User or Cognitive Services OpenAI Contributor roles for the Azure OpenAI resource.

> **Note**
>
> The Microsoft Agent Framework is supported with all actively supported versions of .net. For the purposes of this sample we are recommending the .NET 8.0 SDK or higher.

> **Important**
>
> For this tutorial we are using Azure OpenAI for the Chat Completion service, but you can use any inference service that provides a Microsoft.Extensions.AI.IChatClient implementation.

Installing Nuget packages

To use the Microsoft Agent Framework with Azure OpenAI, you need to install the following NuGet packages:

PowerShell

```
dotnet add package Azure.Identity
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

Creating the agent

- First we create a client for Azure OpenAI, by providing the Azure OpenAI endpoint and using the same login as was used when authenticating with the Azure CLI in the Prerequisites step.
- Then we get a chat client for communicating with the chat completion service, where we also specify the specific model deployment to use. Use one of the deployments that you created in the Prerequisites step.
- Finally we create the agent, providing instructions and a name for the agent.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
        .GetChatClient("gpt-4o-mini")
        .CreateAIAgent(instructions: "You are good at telling jokes.", name:
"Joker");
```

Running the agent

To run the agent, call the `RunAsync` method on the agent instance, providing the user input. The agent will return an `AgentRunResponse` object, and calling `.ToString()` or `.Text` on this response object, provides the text result from the agent.

C#

```
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate."));
```

Sample output:

```text
Why did the pirate go to school?

Because he wanted to improve his "arrr-ticulation"! 🏴‍☠️
```

Running the agent with streaming

To run the agent with streaming, call the `RunStreamingAsync` method on the agent instance, providing the user input. The agent will return a stream `AgentRunResponseUpdate` objects, and calling `.ToString()` or `.Text` on each update object provides the part of the text result contained in that update.

```C#
await foreach (var update in agent.RunStreamingAsync("Tell me a joke about a pirate."))
{
    Console.WriteLine(update);
}
```

Sample output:

```text
Why
 did
 the
 pirate
 go
 to
 school
?


To
 improve
 his
 "
ar
rrrr
rr
```

```
tic
ulation
!"
```

Running the agent with ChatMessages

Instead of a simple string, you can also provide one or more `ChatMessage` objects to the `RunAsync` and `RunStreamingAsync` methods.

Here is an example with a single user message:

C#

```
ChatMessage message = new(ChatRole.User, [
    new TextContent("Tell me a joke about this image?"),
    new
UriContent("https://upload.wikimedia.org/wikipedia/commons/1/11/Joseph_Grimaldi.jp
g", "image/jpeg")
]);

Console.WriteLine(await agent.RunAsync(message));
```

Sample output:

text

```
Why did the clown bring a bottle of sparkling water to the show?

Because he wanted to make a splash!
```

Here is an example with a system and user message:

C#

```
ChatMessage systemMessage = new(
    ChatRole.System,
    """
    If the user asks you to tell a joke, refuse to do so, explaining that you are
not a clown.
    Offer the user an interesting fact instead.
    """);
ChatMessage userMessage = new(ChatRole.User, "Tell me a joke about a pirate.");

Console.WriteLine(await agent.RunAsync([systemMessage, userMessage]));
```

Sample output:

text

```
I’m not a clown, but I can share an interesting fact! Did you know that pirates
often revised the Jolly Roger flag? Depending on the pirate captain, it could
feature different symbols like skulls, bones, or hourglasses, each representing
their unique approach to piracy.
```

Next steps

Using images with an agent

Using images with an agent

10/01/2025

This tutorial shows you how to use images with an agent, allowing the agent to analyze and respond to image content.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Passing images to the agent

You can send images to an agent by creating a `ChatMessage` that includes both text and image content. The agent can then analyze the image and respond accordingly.

First, create an `AIAgent` that is able to analyze images.

C#

```
AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
    .GetChatClient("gpt-4o")
    .CreateAIAgent(
        name: "VisionAgent",
        instructions: "You are a helpful agent that can analyze images");
```

Next, create a `ChatMessage` that contains both a text prompt and an image URL. Use `TextContent` for the text and `UriContent` for the image.

C#

```
ChatMessage message = new(ChatRole.User, [
    new TextContent("What do you see in this image?"),
    new UriContent("https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg", "image/jpeg")
]);
```

Run the agent with the message. You can use streaming to receive the response as it is generated.

C#

```
Console.WriteLine(await agent.RunAsync(message));
```

This will print the agent's analysis of the image to the console.

Next steps

Having a multi-turn conversation with an agent

Multi-turn conversations with an agent

10/01/2025

This tutorial step shows you how to have a multi-turn conversation with an agent, where the agent is built on the Azure OpenAI Chat Completion service.

> **ⓘ Important**
>
> The agent framework supports many different types of agents. This tutorial uses an agent based on a Chat Completion service, but all other agent types are run in the same way. See the Agent Framework user guide for more information on other agent types and how to construct them.

Prerequisites

For prerequisites and creating the agent, see the Create and run a simple agent step in this tutorial.

Running the agent with a multi-turn conversation

Agents are stateless and do not maintain any state internally between calls. To have a multi-turn conversation with an agent, you need to create an object to hold the conversation state and pass this object to the agent when running it.

To create the conversation state object, call the `GetNewThread` method on the agent instance.

C#

```
AgentThread thread = agent.GetNewThread();
```

You can then pass this thread object to the `RunAsync` and `RunStreamingAsync` methods on the agent instance, along with the user input.

C#

```
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate.", thread));
Console.WriteLine(await agent.RunAsync("Now add some emojis to the joke and tell
it in the voice of a pirate's parrot.", thread));
```

This will maintain the conversation state between the calls, and the agent will be able to refer to previous input and response messages in the conversation when responding to new input.

> **ⓘ Important**
>
> The type of service that is used by the `AIAgent` will determine how conversation history is stored. E.g. when using a ChatCompletion service, like in this example, the conversation history is stored in the AgentThread object and sent to the service on each call. When using the Azure AI Agent service on the other hand, the conversation history is stored in the Azure AI Agent service and only a reference to the conversation is sent to the service on each call.

Single agent with multiple conversations

It is possible to have multiple, independent conversations with the same agent instance, by creating multiple `AgentThread` objects. These threads can then be used to maintain separate conversation states for each conversation. The conversations will be fully independent of each other, since the agent does not maintain any state internally.

C#

```
AgentThread thread1 = agent.GetNewThread();
AgentThread thread2 = agent.GetNewThread();
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate.",
thread1));
Console.WriteLine(await agent.RunAsync("Tell me a joke about a robot.", thread2));
Console.WriteLine(await agent.RunAsync("Now add some emojis to the joke and tell
it in the voice of a pirate's parrot.", thread1));
Console.WriteLine(await agent.RunAsync("Now add some emojis to the joke and tell
it in the voice of a robot.", thread2));
```

Next steps

Using function tools with an agent

Using function tools with an agent

10/01/2025

This tutorial step shows you how to use function tools with an agent, where the agent is built on the Azure OpenAI Chat Completion service.

> **Important**
>
> Not all agent types support function tools. Some may only support custom built-in tools, without allowing the caller to provide their own functions. In this step we are using a `ChatClientAgent`, which does support function tools.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Creating the agent with function tools

Function tools are just custom code that you want the agent to be able to call when needed. You can turn any C# method into a function tool, by using the `AIFunctionFactory.Create` method to create an `AIFunction` instance from the method.

If you need to provide additional descriptions about the function or its parameters to the agent, so that it can more accurately choose between different functions, you can use the `System.ComponentModel.DescriptionAttribute` attribute on the method and its parameters.

Here is an example of a simple function tool that fakes getting the weather for a given location. It is decorated with description attributes to provide additional descriptions about itself and its location parameter to the agent.

C#

```
using System.ComponentModel;

[Description("Get the weather for a given location.")]
static string GetWeather([Description("The location to get the weather for.")]
string location)
    => $"The weather in {location} is cloudy with a high of 15°C.";
```

When creating the agent, we can now provide the function tool to the agent, by passing a list of tools to the `CreateAIAgent` method.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(instructions: "You are a helpful assistant", tools:
[AIFunctionFactory.Create(GetWeather)]);
```

Now we can just run the agent as normal, and the agent will be able to call the `GetWeather` function tool when needed.

C#

```
Console.WriteLine(await agent.RunAsync("What is the weather like in Amsterdam?"));
```

Next steps

Using function tools with human in the loop approvals

Using function tools with human in the loop approvals

10/01/2025

This tutorial step shows you how to use function tools that require human approval with an agent, where the agent is built on the Azure OpenAI Chat Completion service.

When agents require any user input, for example to approve a function call, this is referred to as a human-in-the-loop pattern. An agent run that requires user input, will complete with a response that indicates what input is required from the user, instead of completing with a final answer. The caller of the agent is then responsible for getting the required input from the user, and passing it back to the agent as part of a new agent run.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Creating the agent with function tools

When using functions, it's possible to indicate for each function, whether it requires human approval before being executed. This is done by wrapping the `AIFunction` instance in an `ApprovalRequiredAIFunction` instance.

Here is an example of a simple function tool that fakes getting the weather for a given location. For simplicity we are also listing all required usings for this sample here.

C#

```
using System;
using System.ComponentModel;
using System.Linq;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

[Description("Get the weather for a given location.")]
static string GetWeather([Description("The location to get the weather for.")]
string location)
    => $"The weather in {location} is cloudy with a high of 15°C.";
```

To create an `AIFunction` and then wrap it in an `ApprovalRequiredAIFunction`, you can do the following:

```
C#

AIFunction weatherFunction = AIFunctionFactory.Create(GetWeather);
AIFunction approvalRequiredWeatherFunction = new
ApprovalRequiredAIFunction(weatherFunction);
```

When creating the agent, we can now provide the approval requiring function tool to the agent, by passing a list of tools to the `CreateAIAgent` method.

```
C#

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(instructions: "You are a helpful assistant", tools:
[approvalRequiredWeatherFunction]);
```

Since we now have a function that requires approval, the agent may respond with a request for approval, instead of executing the function directly and returning the result. We can check the response content for any `FunctionApprovalRequestContent` instances, which indicates that the agent requires user approval for a function.

```
C#

AgentThread thread = agent.GetNewThread();
AgentRunResponse response = await agent.RunAsync("What is the weather like in
Amsterdam?", thread);

var functionApprovalRequests = response.Messages
    .SelectMany(x => x.Contents)
    .OfType<FunctionApprovalRequestContent>()
    .ToList();
```

If there are any function approval requests, the detail of the function call including name and arguments can be found in the `FunctionCall` property on the `FunctionApprovalRequestContent` instance. This can be shown to the user, so that they can decide whether to approve or reject the function call. For our example, we will assume there is one request.

```
C#

FunctionApprovalRequestContent requestContent = functionApprovalRequests.First();
Console.WriteLine($"We require approval to execute
```

```
'{requestContent.FunctionCall.Name}'");
```

Once the user has provided their input, we can create a `FunctionApprovalResponseContent` instance using the `CreateResponse` method on the `FunctionApprovalRequestContent`. Pass `true` to approve the function call, or `false` to reject it.

The response content can then be passed to the agent in a new `User` `ChatMessage`, along with the same thread object to get the result back from the agent.

C#

```
var approvalMessage = new ChatMessage(ChatRole.User,
[requestContent.CreateResponse(true)]);
Console.WriteLine(await agent.RunAsync(approvalMessage, thread));
```

Whenever you are using function tools with human in the loop approvals, remember to check for `FunctionApprovalRequestContent` instances in the response, after each agent run, until all function calls have been approved or rejected.

Next steps

Producing Structured Output with agents

Producing Structured Output with Agents

10/01/2025

This tutorial step shows you how to produce structured output with an agent, where the agent is built on the Azure OpenAI Chat Completion service.

> **Important**
>
> Not all agent types support structured output. In this step we are using a `ChatClientAgent`, which does support structured output.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Creating the agent with structured output

The `ChatClientAgent` is built on top of any `Microsoft.Extensions.AI.IChatClient` implementation. The `ChatClientAgent` uses the support for structured output that is provided by the underlying chat client.

When creating the agent, we have the option to provide the default `ChatOptions` instance to use for the underlying chat client. This `ChatOptions` instance allows us to pick a preferred ChatResponseFormat.

Various options are supported:

- `ChatResponseFormat.Text`: The response will be plain text.
- `ChatResponseFormat.Json`: The response will be a JSON object without any particular schema.
- `ChatResponseFormatJson.ForJsonSchema`: The response will be a JSON object that conforms to the provided schema.

Let's look at an example of creating an agent that produces structured output in the form of a JSON object that conforms to a specific schema.

The easiest way to produce the schema is to define a C# class that represents the structure of the output you want from the agent, and then use the `AIJsonUtilities.CreateJsonSchema` method to create a schema from the type.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.Extensions.AI;

public class PersonInfo
{
    [JsonPropertyName("name")]
    public string? Name { get; set; }

    [JsonPropertyName("age")]
    public int? Age { get; set; }

    [JsonPropertyName("occupation")]
    public string? Occupation { get; set; }
}

JsonElement schema = AIJsonUtilities.CreateJsonSchema(typeof(PersonInfo));
```

We can then create a `ChatOptions` instance that uses this schema for the response format.

C#

```
using Microsoft.Extensions.AI;

ChatOptions chatOptions = new()
{
    ResponseFormat = ChatResponseFormatJson.ForJsonSchema(
        schema: schema,
        schemaName: "PersonInfo",
        schemaDescription: "Information about a person including their name, age,
and occupation")
};
```

This `ChatOptions` instance can be used when creating the agent.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
        .GetChatClient("gpt-4o-mini")
        .CreateAIAgent(new ChatClientAgentOptions()
        {
```

```
            Name = "HelpfulAssistant",
            Instructions = "You are a helpful assistant.",
            ChatOptions = chatOptions
        });
```

Now we can just run the agent with some textual information that the agent can use to fill in the structured output.

C#

```
var response = await agent.RunAsync("Please provide information about John Smith,
who is a 35-year-old software engineer.");
```

The agent response can then be deserialized into the `PersonInfo` class using the `Deserialize<T>` method on the response object.

C#

```
var personInfo = response.Deserialize<PersonInfo>(JsonSerializerOptions.Web);
Console.WriteLine($"Name: {personInfo.Name}, Age: {personInfo.Age}, Occupation:
{personInfo.Occupation}");
```

When streaming, the agent response is streamed as a series of updates, and we can only deserialize the response once we have received all the updates. We therefore need to assemble all the updates into a single response, before deserializing it.

C#

```
var updates = agent.RunStreamingAsync("Please provide information about John
Smith, who is a 35-year-old software engineer.");
personInfo = (await updates.ToAgentRunResponseAsync()).Deserialize<PersonInfo>
(JsonSerializerOptions.Web);
```

Next steps

Using an agent as a function tool

Using an agent as a function tool

10/01/2025

This tutorial shows you how to use an agent as a function tool, so that one agent can call another agent as a tool.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Creating and using an agent as a function tool

You can use an `AIAgent` as a function tool by calling `.AsAIFunction()` on the agent and providing it as a tool to another agent. This allows you to compose agents and build more advanced workflows.

First, create a function tool as a C# method, and decorate it with descriptions if needed. This tool will be used by our agent that is exposed as a function.

C#

```
using System.ComponentModel;

[Description("Get the weather for a given location.")]
static string GetWeather([Description("The location to get the weather for.")]
string location)
    => $"The weather in {location} is cloudy with a high of 15°C.";
```

Create an `AIAgent` that uses the function tool.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

AIAgent weatherAgent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(
```

```
        instructions: "You answer questions about the weather.",
        name: "WeatherAgent",
        description: "An agent that answers questions about the weather.",
        tools: [AIFunctionFactory.Create(GetWeather)]);
```

Now, create a main agent and provide the `weatherAgent` as a function tool by calling `.AsAIFunction()` to convert `weatherAgent` to a function tool.

C#

```
AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(instructions: "You are a helpful assistant who responds in
French.", tools: [weatherAgent.AsAIFunction()]);
```

Invoke the main agent as normal. It can now call the weather agent as a tool, and should respond in French.

C#

```
Console.WriteLine(await agent.RunAsync("What is the weather like in Amsterdam?"));
```

Next steps

Exposing an agent as an MCP tool

Exposing an agent as an MCP tool

10/01/2025

This tutorial shows you how to expose an agent as a tool over the Model Context Protocol (MCP), so it can be used by other systems that support MCP tools.

Prerequisites

For prerequisites see the Create and run a simple agent step in this tutorial.

Installing Nuget packages

To use the Microsoft Agent Framework with Azure OpenAI, you need to install the following NuGet packages:

PowerShell

```
dotnet add package Azure.Identity
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

To add support for hosting a tool over the Model Context Protocol (MCP), add the following Nuget packages

PowerShell

```
dotnet add package Microsoft.Extensions.Hosting --prerelease
dotnet add package ModelContextProtocol --prerelease
```

Exposing an agent as an MCP tool

You can expose an `AIAgent` as an MCP tool by wrapping it in a function and using `McpServerTool`. You then need to register it with an MCP server. This allows the agent to be invoked as a tool by any MCP-compatible client.

First, create an agent that we will expose as an MCP tool.

C#

```
using System;
using Azure.AI.OpenAI;
```

```
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
        .GetChatClient("gpt-4o-mini")
        .CreateAIAgent(instructions: "You are good at telling jokes.", name:
"Joker");
```

Turn the agent into a function tool and then an MCP tool. The agent name and description will be used as the mcp tool name and description.

C#

```
using ModelContextProtocol.Server;

McpServerTool tool = McpServerTool.Create(agent.AsAIFunction());
```

Setup the MCP server to listen for incoming requests over standard input/output and expose the MCP tool:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

HostApplicationBuilder builder = Host.CreateEmptyApplicationBuilder(settings:
null);
builder.Services
    .AddMcpServer()
    .WithStdioServerTransport()
    .WithTools([tool]);

await builder.Build().RunAsync();
```

This will start an MCP server that exposes the agent as a tool over the MCP protocol.

Next steps

Enabling observability for agents

Enabling observability for Agents

10/01/2025

This tutorial shows how to enable OpenTelemetry on an agent so that interactions with the agent are automatically logged and exported. In this tutorial, output is written to the console using the OpenTelemetry console exporter.

> **Note**
>
> See Semantic Conventions for GenAI agent and framework spans from Open Telemetry for more information about the standards followed by the Microsoft Agent Framework.

Prerequisites

For prerequisites, see the Create and run a simple agent step in this tutorial.

Installing Nuget packages

To use the Agent Framework with Azure OpenAI, you need to install the following NuGet packages:

PowerShell

```
dotnet add package Azure.Identity
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

To also add OpenTelemetry support, with support for writing to the console, install these additional packages:

PowerShell

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Exporter.Console
```

Enable OpenTelemetry in your app

Enable the agent framework telemetry and create an OpenTelemetry `TracerProvider` that exports to the console. Note that the `TracerProvider` must remain alive while you run the agent so traces are exported.

C#

```
using System;
using OpenTelemetry;
using OpenTelemetry.Trace;

// Create a TracerProvider that exports to the console
using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddSource("agent-telemetry-source")
    .AddConsoleExporter()
    .Build();
```

Create and instrument the agent

Create an agent, and using the builder pattern, call `UseOpenTelemetry` to provide a source name. Note that the string literal "agent-telemetry-source" is the OpenTelemetry source name that we used above, when we created the tracer provider.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

// Create the agent and enable OpenTelemetry instrumentation
AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
        .GetChatClient("gpt-4o-mini")
        .CreateAIAgent(instructions: "You are good at telling jokes.", name:
"Joker")
        .AsBuilder()
        .UseOpenTelemetry(sourceName: "agent-telemetry-source")
        .Build();
```

Run the agent and print the text response. The console exporter will show trace data on the console.

C#

```
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate."));
```

The expected output will be something like this, where the agent invocation trace is shown first, followed by the text response from the agent.

PowerShell

```
Activity.TraceId:            f2258b51421fe9cf4c0bd428c87b1ae4
Activity.SpanId:             2cad6fc139dcf01d
Activity.TraceFlags:         Recorded
Activity.DisplayName:        invoke_agent Joker
Activity.Kind:               Client
Activity.StartTime:          2025-09-18T11:00:48.6636883Z
Activity.Duration:           00:00:08.6077009
Activity.Tags:
    gen_ai.operation.name: invoke_agent
    gen_ai.system: openai
    gen_ai.agent.id: e1370f89-3ca8-4278-bce0-3a3a2b22f407
    gen_ai.agent.name: Joker
    gen_ai.request.instructions: You are good at telling jokes.
    gen_ai.response.id: chatcmpl-CH6fgKwMRGDtGNO3H88gA3AG2o7c5
    gen_ai.usage.input_tokens: 26
    gen_ai.usage.output_tokens: 29
Instrumentation scope (ActivitySource):
    Name: c8aeb104-0ce7-49b3-bf45-d71e5bf782d1
Resource associated with Activity:
    telemetry.sdk.name: opentelemetry
    telemetry.sdk.language: dotnet
    telemetry.sdk.version: 1.12.0
    service.name: unknown_service:Agent_Step08_Telemetry

Why did the pirate go to school?

Because he wanted to improve his "arrr-ticulation"! ?????
```

Next steps

Persisting conversations

Adding Middleware to Agents

10/01/2025

Learn how to add middleware to your agents in a few simple steps. Middleware allows you to intercept and modify agent interactions for logging, security, and other cross-cutting concerns.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Step 1: Create a Simple Agent

First, let's create a basic agent with a function tool.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

[Description("The current datetime offset.")]
static string GetDateTime()
    => DateTimeOffset.Now.ToString();

AIAgent baseAgent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
        .GetChatClient("gpt-4o-mini")
        .CreateAIAgent(
            instructions: "You are an AI assistant that helps people find
information.",
            tools: [AIFunctionFactory.Create(GetDateTime, name:
nameof(GetDateTime))]);
```

Step 2: Create Your Agent Run Middleware

Next, we'll create a function that will get invoked for each agent run. It allows us to inspect the input and output from the agent.

Unless the intention is to use the middleware to stop executing the run, the function should call `RunAsync` on the provided `innerAgent`.

This sample middleware just inspects the input and output from the agent run and outputs the number of messages passed into and out of the agent.

C#

```
async Task<AgentRunResponse> CustomAgentRunMiddleware(
    IEnumerable<ChatMessage> messages,
    AgentThread? thread,
    AgentRunOptions? options,
    AIAgent innerAgent,
    CancellationToken cancellationToken)
{
    Console.WriteLine(messages.Count());
    var response = await innerAgent.RunAsync(messages, thread, options,
cancellationToken).ConfigureAwait(false);
    Console.WriteLine(response.Messages.Count);
    return response;
}
```

Step 3: Add Agent Run Middleware to Your Agent

To add this middleware function to the `baseAgent` we created in step 1, we should use the builder pattern. This creates a new agent that has the middleware applied. The original `baseAgent` is not modified.

C#

```
var middlewareEnabledAgent = baseAgent
    .AsBuilder()
        .Use(CustomAgentRunMiddleware)
    .Build();
```

Step 4: Create Function calling Middleware

> ⓘ **Note**
>
> Function calling middleware is currently only supported with an `AIAgent` that uses `Microsoft.Extensions.AI.FunctionInvokingChatClient`, e.g. `ChatClientAgent`.

We can also create middleware that gets called for each function tool that is invoked. Here is an example of function calling middleware, that can inspect and/or modify the function being called, and the result from the function call.

Unless the intention is to use the middleware to not execute the function tool, the middleware should call the provided `next` `Func`.

C#

```
async ValueTask<object?> CustomFunctionCallingMiddleware(
    AIAgent agent,
    FunctionInvocationContext context,
    Func<FunctionInvocationContext, CancellationToken, ValueTask<object?>> next,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"Function Name: {context!.Function.Name}");
    var result = await next(context, cancellationToken);
    Console.WriteLine($"Function Call Result: {result}");

    return result;
}
```

Step 5: Add Function calling Middleware to Your Agent

Same as with adding agent run middleware, we can add function calling middleware as follows:

C#

```
var middlewareEnabledAgent = baseAgent
    .AsBuilder()
        .Use(CustomFunctionCallingMiddleware)
    .Build();
```

Now, when executing the agent with a query that invokes a function, the middleware should get invoked, outputting the function name and call result.

C#

```
await middlewareEnabledAgent.RunAsync("What's the current time?");
```

Step 6: Create Chat Client Middleware

For agents that are built using `IChatClient` developers may want to intercept calls going from the agent to the `IChatClient`. In this case it is possible to use middleware for the `IChatClient`.

Here is an example of chat client middleware, that can inspect and/or modify the input and output for the request to the inference service that the chat client provides.

C#

```
async Task<ChatResponse> CustomChatClientMiddleware(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    IChatClient innerChatClient,
    CancellationToken cancellationToken)
{
    Console.WriteLine(messages.Count());
    var response = await innerChatClient.GetResponseAsync(messages, options,
cancellationToken);
    Console.WriteLine(response.Messages.Count);

    return response;
}
```

> **Note**
>
> For more information about `IChatClient` middleware, see Custom IChatClient middleware in the Microsoft.Extensions.AI documentation.

Step 7: Add Chat client Middleware to an `IChatClient`

To add middleware to your `IChatClient`, you can use the builder pattern. After adding the middleware, you can use the `IChatClient` with your agent as usual.

C#

```
var chatClient = new AzureOpenAIClient(new
Uri("https://<myresource>.openai.azure.com"), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient();

var middlewareEnabledChatClient = chatClient
    .AsBuilder()
        .Use(getResponseFunc: CustomChatClientMiddleware,
getStreamingResponseFunc: null)
    .Build();
```

```
var agent = new ChatClientAgent(middlewareEnabledChatClient, instructions: "You
are a helpful assistant.");
```

`IChatClient` middleware can also be registered using a factory method when constructing an agent via one of the helper methods on SDK clients.

C#

```
var agent = new AzureOpenAIClient(new Uri(endpoint), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .CreateAIAgent("You are a helpful assistant.", clientFactory: (chatClient) =>
chatClient
        .AsBuilder()
            .Use(getResponseFunc: CustomChatClientMiddleware,
getStreamingResponseFunc: null)
        .Build());
```

Persisting and Resuming Agent Conversations

10/01/2025

This tutorial shows how to persist an agent conversation (AgentThread) to storage and reload it later.

When hosting an agent in a service or even in a client application, you often want to maintain conversation state across multiple requests or sessions. By persisting the `AgentThread`, you can save the conversation context and reload it later.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Persisting and resuming the conversation

Create an agent and obtain a new thread that will hold the conversation state.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(instructions: "You are a helpful assistant.", name:
"Assistant");

AgentThread thread = agent.GetNewThread();
```

Run the agent, passing in the thread, so that the `AgentThread` includes this exchange.

C#

```
// Run the agent and append the exchange to the thread
Console.WriteLine(await agent.RunAsync("Tell me a short pirate joke.", thread));
```

Call the SerializeAsync method on the thread to serialize it to a JsonElement. It can then be converted to a string for storage and saved to a database, blob storage, or file.

C#

```
using System.IO;
using System.Text.Json;

// Serialize the thread state
JsonElement serializedThread = thread.Serialize();
string serializedJson = JsonSerializer.Serialize(serializedThread,
JsonSerializerOptions.Web);

// Example: save to a local file (replace with DB or blob storage in production)
string filePath = Path.Combine(Path.GetTempPath(), "agent_thread.json");
await File.WriteAllTextAsync(filePath, serializedJson);
```

Load the persisted JSON from storage and recreate the AgentThread instance from it. Note that the thread must be deserialized using an agent instance. This should be the same agent type that was used to create the original thread. This is because agents may have their own thread types and may construct threads with additional functionality that is specific to that agent type.

C#

```
// Read persisted JSON
string loadedJson = await File.ReadAllTextAsync(filePath);
JsonElement reloaded = JsonSerializer.Deserialize<JsonElement>(loadedJson);

// Deserialize the thread into an AgentThread tied to the same agent type
AgentThread resumedThread = agent.DeserializeThread(reloaded);
```

Use the resumed thread to continue the conversation.

C#

```
// Continue the conversation with resumed thread
Console.WriteLine(await agent.RunAsync("Now tell that joke in the voice of a
pirate.", resumedThread));
```

Next steps

Third Party chat history storage

Storing Chat History in 3rd Party Storage

10/01/2025

This tutorial shows how to store agent chat history in external storage by implementing a custom `ChatMessageStore` and using it with a `ChatClientAgent`.

By default, when using `ChatClientAgent`, chat history is stored either in memory in the `AgentThread` object or the underlying inference service, if the service supports it.

Where services do not require chat history to be stored in the service, it is possible to provide a custom store for persisting chat history instead of relying on the default in-memory behavior.

Prerequisites

For prerequisites, see the Create and run a simple agent step in this tutorial.

Installing Nuget packages

To use the Microsoft Agent Framework with Azure OpenAI, you need to install the following NuGet packages:

PowerShell

```
dotnet add package Azure.Identity
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

In addition to this, we will use the in-memory vector store to store chat messages and a utility package for async LINQ operations.

PowerShell

```
dotnet add package Microsoft.SemanticKernel.Connectors.InMemory --prerelease
dotnet add package System.Linq.Async
```

Creating a custom ChatMessage Store

To create a custom `ChatMessageStore`, you need to implement the abstract `ChatMessageStore` class and provide implementations for the required methods.

Message storage and retrieval methods

The most important methods to implement are:

- `AddMessagesAsync` - called to add new messages to the store.
- `GetMessagesAsync` - called to retrieve the messages from the store.

`GetMessagesAsync` should return the messages in ascending chronological order. All messages returned by it will be used by the `ChatClientAgent` when making calls to the underlying `IChatClient`. It's therefore important that this method considers the limits of the underlying model, and only returns as many messages as can be handled by the model.

Any chat history reduction logic, such as summarization or trimming, should be done before returning messages from `GetMessagesAsync`.

Serialization

`ChatMessageStore` instances are created and attached to an `AgentThread` when the thread is created, and when a thread is resumed from a serialized state.

While the actual messages making up the chat history are stored externally, the `ChatMessageStore` instance may need to store keys or other state to identify the chat history in the external store.

To allow persisting threads, you need to implement the `SerializeStateAsync` method of the `ChatMessageStore` class. You also need to provide a constructor that takes a `JsonElement` parameter, which can be used to deserialize the state when resuming a thread.

Sample ChatMessageStore implementation

Let's look at a sample implementation that stores chat messages in a vector store.

In `AddMessagesAsync` it upserts messages into the vector store, using a unique key for each message.

`GetMessagesAsync` retrieves the messages for the current thread from the vector store, orders them by timestamp, and returns them in ascending order.

When the first message is received, the store generates a unique key for the thread, which is then used to identify the chat history in the vector store for subsequent calls.

The unique key is stored in the `ThreadDbKey` property, which is serialized and deserialized using the `SerializeStateAsync` method and the constructor that takes a `JsonElement`. This key will

therefore be persisted as part of the `AgentThread` state, allowing the thread to be resumed later and continue using the same chat history.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.Json;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Connectors.InMemory;

internal sealed class VectorChatMessageStore : ChatMessageStore
{
    private readonly VectorStore _vectorStore;

    public VectorChatMessageStore(
        VectorStore vectorStore,
        JsonElement serializedStoreState,
        JsonSerializerOptions? jsonSerializerOptions = null)
    {
        this._vectorStore = vectorStore ?? throw new
ArgumentNullException(nameof(vectorStore));
        if (serializedStoreState.ValueKind is JsonValueKind.String)
        {
            this.ThreadDbKey = serializedStoreState.Deserialize<string>();
        }
    }

    public string? ThreadDbKey { get; private set; }

    public override async Task AddMessagesAsync(
        IEnumerable<ChatMessage> messages,
        CancellationToken cancellationToken)
    {
        this.ThreadDbKey ??= Guid.NewGuid().ToString("N");
        var collection = this._vectorStore.GetCollection<string, ChatHistoryItem>
("ChatHistory");
        await collection.EnsureCollectionExistsAsync(cancellationToken);
        await collection.UpsertAsync(messages.Select(x => new ChatHistoryItem()
        {
            Key = this.ThreadDbKey + x.MessageId,
            Timestamp = DateTimeOffset.UtcNow,
            ThreadId = this.ThreadDbKey,
            SerializedMessage = JsonSerializer.Serialize(x),
            MessageText = x.Text
        }), cancellationToken);
    }

    public override async Task<IEnumerable<ChatMessage>> GetMessagesAsync(
        CancellationToken cancellationToken)
```

```
    {
        var collection = this._vectorStore.GetCollection<string, ChatHistoryItem>
("ChatHistory");
        await collection.EnsureCollectionExistsAsync(cancellationToken);
        var records = await collection
            .GetAsync(
                x => x.ThreadId == this.ThreadDbKey, 10,
                new() { OrderBy = x => x.Descending(y => y.Timestamp) },
                cancellationToken)
            .ToListAsync(cancellationToken);
        var messages = records.ConvertAll(x =>
JsonSerializer.Deserialize<ChatMessage>(x.SerializedMessage!)!);
        messages.Reverse();
        return messages;
    }

    public override JsonElement Serialize(JsonSerializerOptions?
jsonSerializerOptions = null) =>
        // We have to serialize the thread id, so that on deserialization we can
retrieve the messages using the same thread id.
        JsonSerializer.SerializeToElement(this.ThreadDbKey);

    private sealed class ChatHistoryItem
    {
        [VectorStoreKey]
        public string? Key { get; set; }
        [VectorStoreData]
        public string? ThreadId { get; set; }
        [VectorStoreData]
        public DateTimeOffset? Timestamp { get; set; }
        [VectorStoreData]
        public string? SerializedMessage { get; set; }
        [VectorStoreData]
        public string? MessageText { get; set; }
    }
}
```

Using the custom ChatMessageStore with a ChatClientAgent

To use the custom `ChatMessageStore`, you need to provide a `ChatMessageStoreFactory` when creating the agent. This factory allows the agent to create a new instance of the desired `ChatMessageStore` for each thread.

When creating a `ChatClientAgent` it is possible to provide a `ChatClientAgentOptions` object that allows providing the `ChatMessageStoreFactory` in addition to all other agent options.

```
C#
```

```
AIAgent agent = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
     .GetChatClient("gpt-4o-mini")
     .CreateAIAgent(new ChatClientAgentOptions
     {
         Name = "Joker",
         Instructions = "You are good at telling jokes.",
         ChatMessageStoreFactory = ctx =>
         {
             // Create a new chat message store for this agent that stores the
messages in a vector store.
             return new VectorChatMessageStore(
                new InMemoryVectorStore(),
                ctx.SerializedState,
                ctx.JsonSerializerOptions);
         }
     });
```

Next steps

Adding Memory to an Agent

Adding Memory to an Agent

10/01/2025

This tutorial shows how to add memory to an agent by implementing an `AIContextProvider` and attaching it to the agent.

> **Important**
>
> Not all agent types support `AIContextProvider`. In this step we are using a `ChatClientAgent`, which does support `AIContextProvider`.

Prerequisites

For prerequisites and installing nuget packages, see the Create and run a simple agent step in this tutorial.

Creating an AIContextProvider

`AIContextProvider` is an abstract class that you can inherit from, and which can be associated with the `AgentThread` for a `ChatClientAgent`. It allows you to:

1. run custom logic before and after the agent invokes the underlying inference service
2. provide additional context to the agent before it invokes the underlying inference service
3. inspect all messages provided to and produced by the agent

Pre and post invocation events

The `AIContextProvider` class has two methods that you can override to run custom logic before and after the agent invokes the underlying inference service:

- `InvokingAsync` - called before the agent invokes the underlying inference service. You can provide additional context to the agent by returning an `AIContext` object. This context will be merged with the agent's existing context before invoking the underlying service. It is possible to provide instructions, tools, and messages to add to the request.
- `InvokedAsync` - called after the agent has received a response from the underlying inference service. You can inspect the request and response messages, and update the state of the context provider.

Serialization

`AIContextProvider` instances are created and attached to an `AgentThread` when the thread is created, and when a thread is resumed from a serialized state.

The `AIContextProvider` instance may have its own state that needs to be persisted between invocations of the agent. E.g. a memory component that remembers information about the user may have memories as part of its state.

To allow persisting threads, you need to implement the `SerializeAsync` method of the `AIContextProvider` class. You also need to provide a constructor that takes a `JsonElement` parameter, which can be used to deserialize the state when resuming a thread.

Sample AIContextProvider implementation

Let's look at an example of a custom memory component that remembers a user's name and age, and provides it to the agent before each invocation.

First we'll create a model class to hold the memories.

C#

```
internal sealed class UserInfo
{
    public string? UserName { get; set; }
    public int? UserAge { get; set; }
}
```

Then we can implement the `AIContextProvider` to manage the memories. The `UserInfoMemory` class below contains the following behavior:

1. It uses a `IChatClient` to look for the user's name and age in user messages when new messages are added to the thread at the end of each run.
2. It provides any current memories to the agent before each invocation.
3. If not memories are available, it instructs the agent to ask the user for the missing information, and not to answer any questions until the information is provided.
4. It also implements serialization to allow persisting the memories as part of the thread state.

C#

```
internal sealed class UserInfoMemory : AIContextProvider
{
    private readonly IChatClient _chatClient;
```

```
    public UserInfoMemory(IChatClient chatClient, UserInfo? userInfo = null)
    {
        this._chatClient = chatClient;
        this.UserInfo = userInfo ?? new UserInfo();
    }

    public UserInfoMemory(IChatClient chatClient, JsonElement serializedState,
JsonSerializerOptions? jsonSerializerOptions = null)
    {
        this._chatClient = chatClient;
        this.UserInfo = serializedState.ValueKind == JsonValueKind.Object ?
            serializedState.Deserialize<UserInfo>(jsonSerializerOptions)! :
            new UserInfo();
    }

    public UserInfo UserInfo { get; set; }

    public override async ValueTask InvokedAsync(
        InvokedContext context,
        CancellationToken cancellationToken = default)
    {
        if ((this.UserInfo.UserName is null || this.UserInfo.UserAge is null) &&
context.RequestMessages.Any(x => x.Role == ChatRole.User))
        {
            var result = await this._chatClient.GetResponseAsync<UserInfo>(
                context.RequestMessages,
                new ChatOptions()
                {
                    Instructions = "Extract the user's name and age from the
message if present. If not present return nulls."
                },
                cancellationToken: cancellationToken);
            this.UserInfo.UserName ??= result.Result.UserName;
            this.UserInfo.UserAge ??= result.Result.UserAge;
        }
    }

    public override ValueTask<AIContext> InvokingAsync(
        InvokingContext context,
        CancellationToken cancellationToken = default)
    {
        StringBuilder instructions = new();
        instructions
            .AppendLine(
                this.UserInfo.UserName is null ?
                    "Ask the user for their name and politely decline to answer
any questions until they provide it." :
                    $"The user's name is {this.UserInfo.UserName}.")
            .AppendLine(
                this.UserInfo.UserAge is null ?
                    "Ask the user for their age and politely decline to answer any
questions until they provide it." :
                    $"The user's age is {this.UserInfo.UserAge}.");
        return new ValueTask<AIContext>(new AIContext
        {
```

```
            Instructions = instructions.ToString()
        });
    }

    public override JsonElement Serialize(JsonSerializerOptions?
jsonSerializerOptions = null)
    {
        return JsonSerializer.SerializeToElement(this.UserInfo,
jsonSerializerOptions);
    }
}
```

Using the AIContextProvider with an agent

To use the custom `AIContextProvider`, you need to provide an `AIContextProviderFactory` when creating the agent. This factory allows the agent to create a new instance of the desired `AIContextProvider` for each thread.

When creating a `ChatClientAgent` it is possible to provide a `ChatClientAgentOptions` object that allows providing the `AIContextProviderFactory` in addition to all other agent options.

C#

```
ChatClient chatClient = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential())
    .GetChatClient("gpt-4o-mini");

AIAgent agent = chatClient.CreateAIAgent(new ChatClientAgentOptions()
{
    Instructions = "You are a friendly assistant. Always address the user by their
name.",
    AIContextProviderFactory = ctx => new UserInfoMemory(
        chatClient.AsIChatClient(),
        ctx.SerializedState,
        ctx.JsonSerializerOptions)
});
```

When creating a new thread, the `AIContextProvider` will be created by `GetNewThread` and attached to the thread. Once memories are extracted it is therefore possible to access the memory component via the thread's `GetService` method and inspect the memories.

C#

```
// Create a new thread for the conversation.
AgentThread thread = agent.GetNewThread();
```

```
Console.WriteLine(await agent.RunAsync("Hello, what is the square root of 9?",
thread));
Console.WriteLine(await agent.RunAsync("My name is Ruaidhrí", thread));
Console.WriteLine(await agent.RunAsync("I am 20 years old", thread));

// Access the memory component via the thread's GetService method.
var userInfo = thread.GetService<UserInfoMemory>()?.UserInfo;
Console.WriteLine($"MEMORY - User Name: {userInfo?.UserName}");
Console.WriteLine($"MEMORY - User Age: {userInfo?.UserAge}");
```

Next steps

Create a simple workflow

Create a Simple Sequential Workflow

10/01/2025

This tutorial demonstrates how to create a simple sequential workflow using the Agent Framework Workflows.

Sequential workflows are the foundation of building complex AI agent systems. This tutorial shows how to create a simple two-step workflow where each step processes data and passes it to the next step.

Overview

In this tutorial, you'll create a workflow with two executors:

1. **Uppercase Executor** - Converts input text to uppercase
2. **Reverse Text Executor** - Reverses the text and outputs the final result

The workflow demonstrates core concepts like:

- Creating custom executors that implement `IMessageHandler<TInput, TOutput>`
- Using `WorkflowBuilder` to connect executors with edges
- Processing data through sequential steps
- Observing workflow execution through events

Prerequisites

- .NET 9.0 or later
- Microsoft.Agents.AI.Workflows NuGet package
- No external AI services required for this basic example

Step-by-Step Implementation

Let's build the sequential workflow step by step.

Step 1: Add Required Using Statements

First, add the necessary using statements:

```
C#
```

```
using System;
using System.Threading.Tasks;
using Microsoft.Agents.AI.Workflows;
using Microsoft.Agents.AI.Workflows.Reflection;
```

Step 2: Create the Uppercase Executor

Create an executor that converts text to uppercase:

C#

```
/// <summary>
/// First executor: converts input text to uppercase.
/// </summary>
internal sealed class UppercaseExecutor() : ReflectingExecutor<UppercaseExecutor>
("UppercaseExecutor"),
    IMessageHandler<string, string>
{
    public ValueTask<string> HandleAsync(string input, CancellationToken
cancellationToken = default)
    {
        // Convert input to uppercase and pass to next executor
        return ValueTask.FromResult(input.ToUpper());
    }
}
```

Key Points:

- Inherits from `ReflectingExecutor<T>` for basic executor functionality
- Implements `IMessageHandler<string, string>` - takes string input, produces string output
- The `HandleAsync` method processes the input and returns the result
- Result is automatically passed to the next connected executor

Step 3: Create the Reverse Text Executor

Create an executor that reverses the text:

C#

```
/// <summary>
/// Second executor: reverses the input text and completes the workflow.
/// </summary>
internal sealed class ReverseTextExecutor() :
ReflectingExecutor<ReverseTextExecutor>("ReverseTextExecutor"),
    IMessageHandler<string, string>
{
```

```
    public ValueTask<string> HandleAsync(string input, CancellationToken
cancellationToken = default)
    {
        // Reverse the input text
        return ValueTask.FromResult(new string(input.Reverse().ToArray()));
    }
}
```

Key Points:

- Same pattern as the first executor
- Reverses the string using LINQ's `Reverse()` method
- This will be the final executor in our workflow

Step 4: Build and Connect the Workflow

Connect the executors using `WorkflowBuilder`:

C#

```
// Create the executors
UppercaseExecutor uppercase = new();
ReverseTextExecutor reverse = new();

// Build the workflow by connecting executors sequentially
WorkflowBuilder builder = new(uppercase);
builder.AddEdge(uppercase, reverse).WithOutputFrom(reverse);
var workflow = builder.Build();
```

Key Points:

- `WorkflowBuilder` constructor takes the starting executor
- `AddEdge()` creates a directed connection from uppercase to reverse
- `WithOutputFrom()` specifies which executor produces the final workflow output
- `Build()` creates the immutable workflow

Step 5: Execute the Workflow

Run the workflow and observe the results:

C#

```
// Execute the workflow with input data
Run run = await InProcessExecution.RunAsync(workflow, "Hello, World!");
foreach (WorkflowEvent evt in run.NewEvents)
{
```

```
    if (evt is ExecutorCompletedEvent executorComplete)
    {
        Console.WriteLine($"{executorComplete.ExecutorId}:
{executorComplete.Data}");
    }
}
```

Step 6: Understanding the Workflow Output

When you run the workflow, you'll see output like:

```
text
```

```
UppercaseExecutor: HELLO, WORLD!
ReverseTextExecutor: !DLROW ,OLLEH
```

The input "Hello, World!" is first converted to uppercase ("HELLO, WORLD!"), then reversed ("!DLROW ,OLLEH").

Key Concepts Explained

Executor Interface

Executors implement `IMessageHandler<TInput, TOutput>`:

- **TInput**: The type of data this executor accepts
- **TOutput**: The type of data this executor produces
- **HandleAsync**: The method that processes the input and returns the output

Workflow Builder Pattern

The `WorkflowBuilder` provides a fluent API for constructing workflows:

- **Constructor**: Takes the starting executor
- **AddEdge()**: Creates directed connections between executors
- **WithOutputFrom()**: Specifies which executors produce workflow outputs
- **Build()**: Creates the final immutable workflow

.NET Event Types

During execution, you can observe these event types:

- `ExecutorCompletedEvent` - When an executor finishes processing
- `WorkflowOutputEvent` - Contains the final workflow result (for streaming execution)

.NET Workflow Builder Pattern

Running the .NET Example

1. Create a new console application
2. Install the `Microsoft.Agents.AI.Workflows` NuGet package
3. Combine all the code snippets from the steps above into your `Program.cs`
4. Run the application

The workflow will process your input through both executors and display the results.

Complete .NET Example

For the complete, ready-to-run implementation, see the 01_ExecutorsAndEdges sample in the Agent Framework repository.

This sample includes:

- Full implementation with all using statements and class structure
- Additional comments explaining the workflow concepts
- Complete project setup and configuration

Next Steps

Learn about creating a simple concurrent workflow

Create a Simple Concurrent Workflow

10/01/2025

This tutorial demonstrates how to create a concurrent workflow using the Agent Framework. You'll learn to implement fan-out and fan-in patterns that enable parallel processing, allowing multiple executors or agents to work simultaneously and then aggregate their results.

What You'll Build

You'll create a workflow that:

- Takes a question as input (e.g., "What is temperature?")
- Sends the same question to two expert AI agents simultaneously (Physicist and Chemist)
- Collects and combines responses from both agents into a single output
- Demonstrates concurrent execution with AI agents using fan-out/fan-in patterns

Prerequisites

- .NET 9.0 or later
- Agent Framework NuGet package: `Microsoft.Agents.AI.Workflows`
- Azure OpenAI access with an endpoint and deployment configured

Step 1: Setup Dependencies and Azure OpenAI

Start by setting up your project with the required NuGet packages and Azure OpenAI client:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Agents.AI.Workflows;
using Microsoft.Agents.AI.Workflows.Reflection;
using Microsoft.Extensions.AI;

public static class Program
{
    private static async Task Main()
    {
        // Set up the Azure OpenAI client
```

```
        var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")
??
            throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not
set.");
        var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
        var chatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
            .GetChatClient(deploymentName).AsIChatClient();
```

Step 2: Create Expert AI Agents

Create two specialized AI agents that will provide expert perspectives:

C#

```
        // Create the AI agents with specialized expertise
        ChatClientAgent physicist = new(
            chatClient,
            name: "Physicist",
            instructions: "You are an expert in physics. You answer questions from
a physics perspective."
        );

        ChatClientAgent chemist = new(
            chatClient,
            name: "Chemist",
            instructions: "You are an expert in chemistry. You answer questions
from a chemistry perspective."
        );
```

Step 3: Create the Start Executor

Create an executor that initiates the concurrent processing by sending input to multiple agents:

C#

```
        var startExecutor = new ConcurrentStartExecutor();
```

The `ConcurrentStartExecutor` implementation:

C#

```
/// <summary>
/// Executor that starts the concurrent processing by sending messages to the
agents.
```

```
/// </summary>
internal sealed class ConcurrentStartExecutor() :
    ReflectingExecutor<ConcurrentStartExecutor>("ConcurrentStartExecutor"),
    IMessageHandler<string>
{
    /// <summary>
    /// Handles the input string and forwards it to connected agents.
    /// </summary>
    /// <param name="message">The input message to process</param>
    /// <param name="context">Workflow context for sending messages</param>
    public async ValueTask HandleAsync(string message, IWorkflowContext context)
    {
        await context.SendMessageAsync(new ChatMessage(ChatRole.User, message));
    }
}
```

Step 4: Create the Aggregation Executor

Create an executor that collects and combines responses from multiple agents:

C#

```
        var aggregationExecutor = new ConcurrentAggregationExecutor();
```

The `ConcurrentAggregationExecutor` implementation:

C#

```
/// <summary>
/// Executor that aggregates the results from the concurrent agents.
/// </summary>
internal sealed class ConcurrentAggregationExecutor() :
    ReflectingExecutor<ConcurrentAggregationExecutor>
("ConcurrentAggregationExecutor"),
    IMessageHandler<ChatMessage>
{
    private readonly List<ChatMessage> _messages = [];

    /// <summary>
    /// Handles incoming messages from the agents and aggregates their responses.
    /// </summary>
    /// <param name="message">The message from the agent</param>
    /// <param name="context">Workflow context for accessing workflow services and
adding events</param>
    public async ValueTask HandleAsync(ChatMessage message, IWorkflowContext
context)
    {
        this._messages.Add(message);

        if (this._messages.Count == 2)
```

```
        {
            var formattedMessages = string.Join(Environment.NewLine,
                this._messages.Select(m => $"{m.AuthorName}: {m.Text}"));
            await context.YieldOutputAsync(formattedMessages);
        }
    }
}
```

Step 5: Build the Workflow

Connect the executors and agents using fan-out and fan-in edge patterns:

C#

```
        // Build the workflow by adding executors and connecting them
        var workflow = new WorkflowBuilder(startExecutor)
            .AddFanOutEdge(startExecutor, targets: [physicist, chemist])
            .AddFanInEdge(aggregationExecutor, sources: [physicist, chemist])
            .WithOutputFrom(aggregationExecutor)
            .Build();
```

Step 6: Execute the Workflow

Run the workflow and capture the streaming output:

C#

```
        // Execute the workflow in streaming mode
        StreamingRun run = await InProcessExecution.StreamAsync(workflow, "What is
temperature?");
        await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
        {
            if (evt is WorkflowOutputEvent output)
            {
                Console.WriteLine($"Workflow completed with
results:\n{output.Data}");
            }
        }
    }
}
```

How It Works

1. **Fan-Out**: The `ConcurrentStartExecutor` receives the input question and the fan-out edge sends it to both the Physicist and Chemist agents simultaneously
2. **Parallel Processing**: Both AI agents process the same question concurrently, each providing their expert perspective
3. **Fan-In**: The `ConcurrentAggregationExecutor` collects `ChatMessage` responses from both agents
4. **Aggregation**: Once both responses are received, the aggregator combines them into a formatted output

Key Concepts

- **Fan-Out Edges**: Use `AddFanOutEdge()` to distribute the same input to multiple executors or agents
- **Fan-In Edges**: Use `AddFanInEdge()` to collect results from multiple source executors
- **AI Agent Integration**: AI agents can be used directly as executors in workflows
- **ReflectingExecutor**: Base class for creating custom executors with automatic message handling
- **Streaming Execution**: Use `StreamAsync()` to get real-time updates as the workflow progresses

Complete Implementation

For the complete working implementation of this concurrent workflow with AI agents, see the Concurrent/Program.cs sample in the Agent Framework repository.

Next Steps

Learn about using agents in workflows

Agents in Workflows

10/01/2025

This tutorial demonstrates how to integrate AI agents into workflows using the Agent Framework. You'll learn to create workflows that leverage the power of specialized AI agents for content creation, review, and other collaborative tasks.

What You'll Build

You'll create a workflow that:

- Uses Azure Foundry Agent Service to create intelligent agents
- Implements a French translation agent that translates input to French
- Implements a Spanish translation agent that translates French to Spanish
- Implements an English translation agent that translates Spanish back to English
- Connects agents in a sequential workflow pipeline
- Streams real-time updates as agents process requests
- Demonstrates proper resource cleanup for Azure Foundry agents

Prerequisites

- .NET 9.0 or later
- Agent Framework installed via NuGet
- Azure Foundry project configured with proper environment variables
- Azure CLI authentication: `az login`

Step 1: Import Required Dependencies

Start by importing the necessary components for Azure Foundry agents and workflows:

C#

```
using System;
using System.Threading.Tasks;
using Azure.AI.Agents.Persistent;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Agents.AI.Workflows;
using Microsoft.Extensions.AI;
```

Step 2: Set Up Azure Foundry Client

Configure the Azure Foundry client with environment variables and authentication:

C#

```
public static class Program
{
    private static async Task Main()
    {
        // Set up the Azure Foundry client
        var endpoint =
Environment.GetEnvironmentVariable("AZURE_FOUNDRY_PROJECT_ENDPOINT")
            ?? throw new InvalidOperationException("AZURE_FOUNDRY_PROJECT_ENDPOINT
is not set.");
        var model =
Environment.GetEnvironmentVariable("AZURE_FOUNDRY_PROJECT_MODEL_ID") ?? "gpt-4o-
mini";
        var persistentAgentsClient = new PersistentAgentsClient(endpoint, new
AzureCliCredential());
```

Step 3: Create Specialized Azure Foundry Agents

Create three translation agents using the helper method:

C#

```
        // Create agents
        AIAgent frenchAgent = await GetTranslationAgentAsync("French",
persistentAgentsClient, model);
        AIAgent spanishAgent = await GetTranslationAgentAsync("Spanish",
persistentAgentsClient, model);
        AIAgent englishAgent = await GetTranslationAgentAsync("English",
persistentAgentsClient, model);
```

Step 4: Create Agent Factory Method

Implement a helper method to create Azure Foundry agents with specific instructions:

C#

```
    /// <summary>
    /// Creates a translation agent for the specified target language.
    /// </summary>
    /// <param name="targetLanguage">The target language for translation</param>
    /// <param name="persistentAgentsClient">The PersistentAgentsClient to create
the agent</param>
```

```
    /// <param name="model">The model to use for the agent</param>
    /// <returns>A ChatClientAgent configured for the specified language</returns>
    private static async Task<ChatClientAgent> GetTranslationAgentAsync(
        string targetLanguage,
        PersistentAgentsClient persistentAgentsClient,
        string model)
    {
        var agentMetadata = await
persistentAgentsClient.Administration.CreateAgentAsync(
            model: model,
            name: $"{targetLanguage} Translator",
            instructions: $"You are a translation assistant that translates the
provided text to {targetLanguage}.");

        return await
persistentAgentsClient.GetAIAgentAsync(agentMetadata.Value.Id);
    }
}
```

Step 5: Build the Workflow

Connect the agents in a sequential workflow using the WorkflowBuilder:

C#

```
        // Build the workflow by adding executors and connecting them
        var workflow = new WorkflowBuilder(frenchAgent)
            .AddEdge(frenchAgent, spanishAgent)
            .AddEdge(spanishAgent, englishAgent)
            .Build();
```

Step 6: Execute with Streaming

Run the workflow with streaming to observe real-time updates from both agents:

C#

```
        // Execute the workflow
        StreamingRun run = await InProcessExecution.StreamAsync(workflow, new
ChatMessage(ChatRole.User, "Hello World!"));

        // Must send the turn token to trigger the agents.
        // The agents are wrapped as executors. When they receive messages,
        // they will cache the messages and only start processing when they
receive a TurnToken.
        await run.TrySendMessageAsync(new TurnToken(emitEvents: true));
        await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
```

```
        {
            if (evt is AgentRunUpdateEvent executorComplete)
            {
                Console.WriteLine($"{executorComplete.ExecutorId}:
{executorComplete.Data}");
            }
        }
```

Step 7: Resource Cleanup

Properly clean up the Azure Foundry agents after use:

C#

```
        // Cleanup the agents created for the sample.
        await
persistentAgentsClient.Administration.DeleteAgentAsync(frenchAgent.Id);
        await
persistentAgentsClient.Administration.DeleteAgentAsync(spanishAgent.Id);
        await
persistentAgentsClient.Administration.DeleteAgentAsync(englishAgent.Id);
    }
```

How It Works

1. **Azure Foundry Client Setup**: Uses `PersistentAgentsClient` with Azure CLI credentials for authentication
2. **Agent Creation**: Creates persistent agents on Azure Foundry with specific instructions for translation
3. **Sequential Processing**: French agent translates input first, then Spanish agent, then English agent
4. **Turn Token Pattern**: Agents cache messages and only process when they receive a `TurnToken`
5. **Streaming Updates**: `AgentRunUpdateEvent` provides real-time token updates as agents generate responses
6. **Resource Management**: Proper cleanup of Azure Foundry agents using the Administration API

Key Concepts

- **Azure Foundry Agent Service**: Cloud-based AI agents with advanced reasoning capabilities

- **PersistentAgentsClient**: Client for creating and managing agents on Azure Foundry
- **AgentRunUpdateEvent**: Real-time streaming updates during agent execution
- **TurnToken**: Signal that triggers agent processing after message caching
- **Sequential Workflow**: Agents connected in a pipeline where output flows from one to the next

Complete Implementation

For the complete working implementation of this Azure Foundry agents workflow, see the FoundryAgent Program.cs sample in the Agent Framework repository.

Next Steps

Learn about branching in workflows

Create a Workflow with Branching Logic

10/01/2025

In this tutorial, you will learn how to create a workflow with branching logic using the Agent Framework. Branching logic allows your workflow to make decisions based on certain conditions, enabling more complex and dynamic behavior.

Conditional Edges

Conditional edges allow your workflow to make routing decisions based on the content or properties of messages flowing through the workflow. This enables dynamic branching where different execution paths are taken based on runtime conditions.

What You'll Build

You'll create an email processing workflow that demonstrates conditional routing:

- A spam detection agent that analyzes incoming emails and returns structured JSON
- Conditional edges that route emails to different handlers based on classification
- A legitimate email handler that drafts professional responses
- A spam handler that marks suspicious emails
- Shared state management to persist email data between workflow steps

Prerequisites

- .NET 9.0 or later
- Azure OpenAI deployment with structured output support
- Azure CLI authentication configured (`az login`)
- Basic understanding of C# and async programming

Setting Up the Environment

First, install the required packages for your .NET project:

Bash

```
dotnet add package Microsoft.Agents.AI.Workflows --prerelease
dotnet add package Microsoft.Agents.AI.Workflows.Reflection --prerelease
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Extensions.AI
dotnet add package Azure.Identity
```

Define Data Models

Start by defining the data structures that will flow through your workflow:

C#

```
using System.Text.Json.Serialization;

/// <summary>
/// Represents the result of spam detection.
/// </summary>
public sealed class DetectionResult
{
    [JsonPropertyName("is_spam")]
    public bool IsSpam { get; set; }

    [JsonPropertyName("reason")]
    public string Reason { get; set; } = string.Empty;

    // Email ID is generated by the executor, not the agent
    [JsonIgnore]
    public string EmailId { get; set; } = string.Empty;
}

/// <summary>
/// Represents an email.
/// </summary>
internal sealed class Email
{
    [JsonPropertyName("email_id")]
    public string EmailId { get; set; } = string.Empty;

    [JsonPropertyName("email_content")]
    public string EmailContent { get; set; } = string.Empty;
}

/// <summary>
/// Represents the response from the email assistant.
/// </summary>
public sealed class EmailResponse
{
    [JsonPropertyName("response")]
    public string Response { get; set; } = string.Empty;
}

/// <summary>
/// Constants for shared state scopes.
/// </summary>
internal static class EmailStateConstants
{
    public const string EmailStateScope = "EmailState";
}
```

Create Condition Functions

The condition function evaluates the spam detection result to determine which path the workflow should take:

C#

```
/// <summary>
/// Creates a condition for routing messages based on the expected spam detection
result.
/// </summary>
/// <param name="expectedResult">The expected spam detection result</param>
/// <returns>A function that evaluates whether a message meets the expected
result</returns>
private static Func<object?, bool> GetCondition(bool expectedResult) =>
    detectionResult => detectionResult is DetectionResult result && result.IsSpam
== expectedResult;
```

This condition function:

- Takes a `bool expectedResult` parameter (true for spam, false for non-spam)
- Returns a function that can be used as an edge condition
- Safely checks if the message is a `DetectionResult` and compares the `IsSpam` property

Create AI Agents

Set up the AI agents that will handle spam detection and email assistance:

C#

```
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;

/// <summary>
/// Creates a spam detection agent.
/// </summary>
/// <returns>A ChatClientAgent configured for spam detection</returns>
private static ChatClientAgent GetSpamDetectionAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are a spam
detection assistant that identifies spam emails.")
    {
        ChatOptions = new()
        {
            ResponseFormat =
ChatResponseFormat.ForJsonSchema(AIJsonUtilities.CreateJsonSchema(typeof(Detection
Result)))
        }
```

```
    });

/// <summary>
/// Creates an email assistant agent.
/// </summary>
/// <returns>A ChatClientAgent configured for email assistance</returns>
private static ChatClientAgent GetEmailAssistantAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are an email
assistant that helps users draft professional responses to emails.")
    {
        ChatOptions = new()
        {
            ResponseFormat =
ChatResponseFormat.ForJsonSchema(AIJsonUtilities.CreateJsonSchema(typeof(EmailResp
onse)))
        }
    });
```

Implement Executors

Create the workflow executors that handle different stages of email processing:

C#

```
using Microsoft.Agents.AI.Workflows;
using Microsoft.Agents.AI.Workflows.Reflection;
using System.Text.Json;

/// <summary>
/// Executor that detects spam using an AI agent.
/// </summary>
internal sealed class SpamDetectionExecutor :
ReflectingExecutor<SpamDetectionExecutor>, IMessageHandler<ChatMessage,
DetectionResult>
{
    private readonly AIAgent _spamDetectionAgent;

    public SpamDetectionExecutor(AIAgent spamDetectionAgent) :
base("SpamDetectionExecutor")
    {
        this._spamDetectionAgent = spamDetectionAgent;
    }

    public async ValueTask<DetectionResult> HandleAsync(ChatMessage message,
IWorkflowContext context)
    {
        // Generate a random email ID and store the email content to shared state
        var newEmail = new Email
        {
            EmailId = Guid.NewGuid().ToString("N"),
            EmailContent = message.Text
        };
```

```
        await context.QueueStateUpdateAsync(newEmail.EmailId, newEmail, scopeName:
EmailStateConstants.EmailStateScope);

        // Invoke the agent for spam detection
        var response = await this._spamDetectionAgent.RunAsync(message);
        var detectionResult = JsonSerializer.Deserialize<DetectionResult>
(response.Text);

        detectionResult!.EmailId = newEmail.EmailId;
        return detectionResult;
    }
}

/// <summary>
/// Executor that assists with email responses using an AI agent.
/// </summary>
internal sealed class EmailAssistantExecutor :
ReflectingExecutor<EmailAssistantExecutor>, IMessageHandler<DetectionResult,
EmailResponse>
{
    private readonly AIAgent _emailAssistantAgent;

    public EmailAssistantExecutor(AIAgent emailAssistantAgent) :
base("EmailAssistantExecutor")
    {
        this._emailAssistantAgent = emailAssistantAgent;
    }

    public async ValueTask<EmailResponse> HandleAsync(DetectionResult message,
IWorkflowContext context)
    {
        if (message.IsSpam)
        {
            throw new InvalidOperationException("This executor should only handle
non-spam messages.");
        }

        // Retrieve the email content from shared state
        var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope)
            ?? throw new InvalidOperationException("Email not found.");

        // Invoke the agent to draft a response
        var response = await
this._emailAssistantAgent.RunAsync(email.EmailContent);
        var emailResponse = JsonSerializer.Deserialize<EmailResponse>
(response.Text);

        return emailResponse!;
    }
}

/// <summary>
/// Executor that sends emails.
/// </summary>
```

```
internal sealed class SendEmailExecutor() : ReflectingExecutor<SendEmailExecutor>
("SendEmailExecutor"), IMessageHandler<EmailResponse>
{
    public async ValueTask HandleAsync(EmailResponse message, IWorkflowContext
context) =>
        await context.YieldOutputAsync($"Email sent: {message.Response}");
}

/// <summary>
/// Executor that handles spam messages.
/// </summary>
internal sealed class HandleSpamExecutor() :
ReflectingExecutor<HandleSpamExecutor>("HandleSpamExecutor"),
IMessageHandler<DetectionResult>
{
    public async ValueTask HandleAsync(DetectionResult message, IWorkflowContext
context)
    {
        if (message.IsSpam)
        {
            await context.YieldOutputAsync($"Email marked as spam:
{message.Reason}");
        }
        else
        {
            throw new InvalidOperationException("This executor should only handle
spam messages.");
        }
    }
}
```

Build the Workflow with Conditional Edges

Now create the main program that builds and executes the workflow:

C#

```
using Microsoft.Extensions.AI;

public static class Program
{
    private static async Task Main()
    {
        // Set up the Azure OpenAI client
        var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")
            ?? throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not
set.");
        var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
        var chatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
```

```
            .GetChatClient(deploymentName).AsIChatClient();

        // Create agents
        AIAgent spamDetectionAgent = GetSpamDetectionAgent(chatClient);
        AIAgent emailAssistantAgent = GetEmailAssistantAgent(chatClient);

        // Create executors
        var spamDetectionExecutor = new SpamDetectionExecutor(spamDetectionAgent);
        var emailAssistantExecutor = new
EmailAssistantExecutor(emailAssistantAgent);
        var sendEmailExecutor = new SendEmailExecutor();
        var handleSpamExecutor = new HandleSpamExecutor();

        // Build the workflow with conditional edges
        var workflow = new WorkflowBuilder(spamDetectionExecutor)
            // Non-spam path: route to email assistant when IsSpam = false
            .AddEdge(spamDetectionExecutor, emailAssistantExecutor, condition:
GetCondition(expectedResult: false))
            .AddEdge(emailAssistantExecutor, sendEmailExecutor)
            // Spam path: route to spam handler when IsSpam = true
            .AddEdge(spamDetectionExecutor, handleSpamExecutor, condition:
GetCondition(expectedResult: true))
            .WithOutputFrom(handleSpamExecutor, sendEmailExecutor)
            .Build();

        // Execute the workflow with sample spam email
        string emailContent = "Congratulations! You've won $1,000,000! Click here
to claim your prize now!";
        StreamingRun run = await InProcessExecution.StreamAsync(workflow, new
ChatMessage(ChatRole.User, emailContent));
        await run.TrySendMessageAsync(new TurnToken(emitEvents: true));

        await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
        {
            if (evt is WorkflowOutputEvent outputEvent)
            {
                Console.WriteLine($"{outputEvent}");
            }
        }
    }
}
```

How It Works

1. **Workflow Entry**: The workflow starts with `spamDetectionExecutor` receiving a `ChatMessage`.

2. **Spam Analysis**: The spam detection agent analyzes the email and returns a structured `DetectionResult` with `IsSpam` and `Reason` properties.

3. **Conditional Routing**: Based on the `IsSpam` value:
    - **If spam** (`IsSpam = true`): Routes to `HandleSpamExecutor` using `GetCondition(true)`
    - **If legitimate** (`IsSpam = false`): Routes to `EmailAssistantExecutor` using `GetCondition(false)`
4. **Response Generation**: For legitimate emails, the email assistant drafts a professional response.
5. **Final Output**: The workflow yields either a spam notice or sends the drafted email response.

Key Features of Conditional Edges

1. **Type-Safe Conditions**: The `GetCondition` method creates reusable condition functions that safely evaluate message content.
2. **Multiple Paths**: A single executor can have multiple outgoing edges with different conditions, enabling complex branching logic.
3. **Shared State**: Email data persists across executors using scoped state management, allowing downstream executors to access original content.
4. **Error Handling**: Executors validate their inputs and throw meaningful exceptions when receiving unexpected message types.
5. **Clean Architecture**: Each executor has a single responsibility, making the workflow maintainable and testable.

Running the Example

When you run this workflow with the sample spam email:

```
Email marked as spam: This email contains common spam indicators including
monetary prizes, urgency tactics, and suspicious links that are typical of
phishing attempts.
```

Try changing the email content to something legitimate:

```
C#
```

```
string emailContent = "Hi, I wanted to follow up on our meeting yesterday and get
your thoughts on the project proposal.";
```

The workflow will route to the email assistant and generate a professional response instead.

This conditional routing pattern forms the foundation for building sophisticated workflows that can handle complex decision trees and business logic.

Complete Implementation

For the complete working implementation, see this sample in the Agent Framework repository.

Switch-Case Edges

Building on Conditional Edges

The previous conditional edges example demonstrated two-way routing (spam vs. legitimate emails). However, many real-world scenarios require more sophisticated decision trees. Switch-case edges provide a cleaner, more maintainable solution when you need to route to multiple destinations based on different conditions.

What You'll Build with Switch-Case

You'll extend the email processing workflow to handle three decision paths:

- **NotSpam** → Email Assistant → Send Email
- **Spam** → Handle Spam Executor
- **Uncertain** → Handle Uncertain Executor (default case)

The key improvement is using the `SwitchBuilder` pattern instead of multiple individual conditional edges, making the workflow easier to understand and maintain as decision complexity grows.

Data Models for Switch-Case

Update your data models to support the three-way classification:

```
C#
```

```
/// <summary>
/// Represents the possible decisions for spam detection.
/// </summary>
public enum SpamDecision
{
    NotSpam,
    Spam,
    Uncertain
}

/// <summary>
/// Represents the result of spam detection with enhanced decision support.
/// </summary>
public sealed class DetectionResult
{
    [JsonPropertyName("spam_decision")]
    [JsonConverter(typeof(JsonStringEnumConverter))]
    public SpamDecision spamDecision { get; set; }

    [JsonPropertyName("reason")]
    public string Reason { get; set; } = string.Empty;

    // Email ID is generated by the executor, not the agent
    [JsonIgnore]
    public string EmailId { get; set; } = string.Empty;
}

/// <summary>
/// Represents an email stored in shared state.
/// </summary>
internal sealed class Email
{
    [JsonPropertyName("email_id")]
    public string EmailId { get; set; } = string.Empty;

    [JsonPropertyName("email_content")]
    public string EmailContent { get; set; } = string.Empty;
}

/// <summary>
/// Represents the response from the email assistant.
/// </summary>
public sealed class EmailResponse
{
    [JsonPropertyName("response")]
    public string Response { get; set; } = string.Empty;
}

/// <summary>
/// Constants for shared state scopes.
/// </summary>
internal static class EmailStateConstants
{
```

```
    public const string EmailStateScope = "EmailState";
}
```

Condition Factory for Switch-Case

Create a reusable condition factory that generates predicates for each spam decision:

C#

```
/// <summary>
/// Creates a condition for routing messages based on the expected spam detection
result.
/// </summary>
/// <param name="expectedDecision">The expected spam detection decision</param>
/// <returns>A function that evaluates whether a message meets the expected
result</returns>
private static Func<object?, bool> GetCondition(SpamDecision expectedDecision) =>
    detectionResult => detectionResult is DetectionResult result &&
result.spamDecision == expectedDecision;
```

This factory approach:

- **Reduces Code Duplication**: One function generates all condition predicates
- **Ensures Consistency**: All conditions follow the same pattern
- **Simplifies Maintenance**: Changes to condition logic happen in one place

Enhanced AI Agent

Update the spam detection agent to be less confident and return three-way classifications:

C#

```
/// <summary>
/// Creates a spam detection agent with enhanced uncertainty handling.
/// </summary>
/// <returns>A ChatClientAgent configured for three-way spam detection</returns>
private static ChatClientAgent GetSpamDetectionAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are a spam
detection assistant that identifies spam emails. Be less confident in your
assessments.")
    {
        ChatOptions = new()
        {
            ResponseFormat = ChatResponseFormat.ForJsonSchema<DetectionResult>()
        }
    });

/// <summary>
```

```
/// Creates an email assistant agent (unchanged from conditional edges example).
/// </summary>
/// <returns>A ChatClientAgent configured for email assistance</returns>
private static ChatClientAgent GetEmailAssistantAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are an email
assistant that helps users draft responses to emails with professionalism.")
    {
        ChatOptions = new()
        {
            ResponseFormat = ChatResponseFormat.ForJsonSchema<EmailResponse>()
        }
    });
```

Workflow Executors with Enhanced Routing

Implement executors that handle the three-way routing with shared state management:

C#

```
/// <summary>
/// Executor that detects spam using an AI agent with three-way classification.
/// </summary>
internal sealed class SpamDetectionExecutor :
ReflectingExecutor<SpamDetectionExecutor>, IMessageHandler<ChatMessage,
DetectionResult>
{
    private readonly AIAgent _spamDetectionAgent;

    public SpamDetectionExecutor(AIAgent spamDetectionAgent) :
base("SpamDetectionExecutor")
    {
        this._spamDetectionAgent = spamDetectionAgent;
    }

    public async ValueTask<DetectionResult> HandleAsync(ChatMessage message,
IWorkflowContext context)
    {
        // Generate a random email ID and store the email content in shared state
        var newEmail = new Email
        {
            EmailId = Guid.NewGuid().ToString("N"),
            EmailContent = message.Text
        };
        await context.QueueStateUpdateAsync(newEmail.EmailId, newEmail, scopeName:
EmailStateConstants.EmailStateScope);

        // Invoke the agent for enhanced spam detection
        var response = await this._spamDetectionAgent.RunAsync(message);
        var detectionResult = JsonSerializer.Deserialize<DetectionResult>
(response.Text);

        detectionResult!.EmailId = newEmail.EmailId;
```

```
        return detectionResult;
    }
}

/// <summary>
/// Executor that assists with email responses using an AI agent.
/// </summary>
internal sealed class EmailAssistantExecutor :
ReflectingExecutor<EmailAssistantExecutor>, IMessageHandler<DetectionResult,
EmailResponse>
{
    private readonly AIAgent _emailAssistantAgent;

    public EmailAssistantExecutor(AIAgent emailAssistantAgent) :
base("EmailAssistantExecutor")
    {
        this._emailAssistantAgent = emailAssistantAgent;
    }

    public async ValueTask<EmailResponse> HandleAsync(DetectionResult message,
IWorkflowContext context)
    {
        if (message.spamDecision == SpamDecision.Spam)
        {
            throw new InvalidOperationException("This executor should only handle
non-spam messages.");
        }

        // Retrieve the email content from shared state
        var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope);

        // Invoke the agent to draft a response
        var response = await
this._emailAssistantAgent.RunAsync(email!.EmailContent);
        var emailResponse = JsonSerializer.Deserialize<EmailResponse>
(response.Text);

        return emailResponse!;
    }
}

/// <summary>
/// Executor that sends emails.
/// </summary>
internal sealed class SendEmailExecutor() : ReflectingExecutor<SendEmailExecutor>
("SendEmailExecutor"), IMessageHandler<EmailResponse>
{
    public async ValueTask HandleAsync(EmailResponse message, IWorkflowContext
context) =>
        await context.YieldOutputAsync($"Email sent:
{message.Response}").ConfigureAwait(false);
}

/// <summary>
```

```
/// Executor that handles spam messages.
/// </summary>
internal sealed class HandleSpamExecutor() :
ReflectingExecutor<HandleSpamExecutor>("HandleSpamExecutor"),
IMessageHandler<DetectionResult>
{
    public async ValueTask HandleAsync(DetectionResult message, IWorkflowContext
context)
    {
        if (message.spamDecision == SpamDecision.Spam)
        {
            await context.YieldOutputAsync($"Email marked as spam:
{message.Reason}").ConfigureAwait(false);
        }
        else
        {
            throw new InvalidOperationException("This executor should only handle
spam messages.");
        }
    }
}

/// <summary>
/// Executor that handles uncertain emails requiring manual review.
/// </summary>
internal sealed class HandleUncertainExecutor() :
ReflectingExecutor<HandleUncertainExecutor>("HandleUncertainExecutor"),
IMessageHandler<DetectionResult>
{
    public async ValueTask HandleAsync(DetectionResult message, IWorkflowContext
context)
    {
        if (message.spamDecision == SpamDecision.Uncertain)
        {
            var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope);
            await context.YieldOutputAsync($"Email marked as uncertain:
{message.Reason}. Email content: {email?.EmailContent}");
        }
        else
        {
            throw new InvalidOperationException("This executor should only handle
uncertain spam decisions.");
        }
    }
}
```

Build Workflow with Switch-Case Pattern

Replace multiple conditional edges with the cleaner switch-case pattern:

```
C#
```

```
public static class Program
{
    private static async Task Main()
    {
        // Set up the Azure OpenAI client
        var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")
?? throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
        var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
        var chatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential()).GetChatClient(deploymentName).AsIChatClient();

        // Create agents
        AIAgent spamDetectionAgent = GetSpamDetectionAgent(chatClient);
        AIAgent emailAssistantAgent = GetEmailAssistantAgent(chatClient);

        // Create executors
        var spamDetectionExecutor = new SpamDetectionExecutor(spamDetectionAgent);
        var emailAssistantExecutor = new
EmailAssistantExecutor(emailAssistantAgent);
        var sendEmailExecutor = new SendEmailExecutor();
        var handleSpamExecutor = new HandleSpamExecutor();
        var handleUncertainExecutor = new HandleUncertainExecutor();

        // Build the workflow using switch-case for cleaner three-way routing
        WorkflowBuilder builder = new(spamDetectionExecutor);
        builder.AddSwitch(spamDetectionExecutor, switchBuilder =>
            switchBuilder
            .AddCase(
                GetCondition(expectedDecision: SpamDecision.NotSpam),
                emailAssistantExecutor
            )
            .AddCase(
                GetCondition(expectedDecision: SpamDecision.Spam),
                handleSpamExecutor
            )
            .WithDefault(
                handleUncertainExecutor
            )
        )
        // After the email assistant writes a response, it will be sent to the
send email executor
        .AddEdge(emailAssistantExecutor, sendEmailExecutor)
        .WithOutputFrom(handleSpamExecutor, sendEmailExecutor,
handleUncertainExecutor);

        var workflow = builder.Build();

        // Read an email from a text file (use ambiguous content for
demonstration)
        string email = Resources.Read("ambiguous_email.txt");

        // Execute the workflow
```

```
        StreamingRun run = await InProcessExecution.StreamAsync(workflow, new
ChatMessage(ChatRole.User, email));
        await run.TrySendMessageAsync(new TurnToken(emitEvents: true));
        await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
        {
            if (evt is WorkflowOutputEvent outputEvent)
            {
                Console.WriteLine($"{outputEvent}");
            }
        }
    }
}
```

Switch-Case Benefits

1. **Cleaner Syntax**: The `SwitchBuilder` provides a more readable alternative to multiple conditional edges
2. **Ordered Evaluation**: Cases are evaluated sequentially, stopping at the first match
3. **Guaranteed Routing**: The `WithDefault()` method ensures messages never get stuck
4. **Better Maintainability**: Adding new cases requires minimal changes to the workflow structure
5. **Type Safety**: Each executor validates its input to catch routing errors early

Pattern Comparison

Before (Conditional Edges):

C#

```
var workflow = new WorkflowBuilder(spamDetectionExecutor)
    .AddEdge(spamDetectionExecutor, emailAssistantExecutor, condition:
GetCondition(expectedResult: false))
    .AddEdge(spamDetectionExecutor, handleSpamExecutor, condition:
GetCondition(expectedResult: true))
    // No clean way to handle a third case
    .WithOutputFrom(handleSpamExecutor, sendEmailExecutor)
    .Build();
```

After (Switch-Case):

C#

```
WorkflowBuilder builder = new(spamDetectionExecutor);
builder.AddSwitch(spamDetectionExecutor, switchBuilder =>
    switchBuilder
    .AddCase(GetCondition(SpamDecision.NotSpam), emailAssistantExecutor)
```

```
    .AddCase(GetCondition(SpamDecision.Spam), handleSpamExecutor)
    .WithDefault(handleUncertainExecutor)  // Clean default case
)
// Continue building the rest of the workflow
```

The switch-case pattern scales much better as the number of routing decisions grows, and the default case provides a safety net for unexpected values.

Running the Example

When you run this workflow with ambiguous email content:

```
text
```

```
Email marked as uncertain: This email contains promotional language but may be
from a legitimate business contact, requiring human review for proper
classification.
```

Try changing the email content to something clearly spam or clearly legitimate to see the different routing paths in action.

Complete Implementation

For the complete working implementation, see this sample in the Agent Framework repository.

Multi-Selection Edges

Beyond Switch-Case: Multi-Selection Routing

While switch-case edges route messages to exactly one destination, real-world workflows often need to trigger multiple parallel operations based on data characteristics. **Partitioned edges** (implemented as fan-out edges with partitioners) enable sophisticated fan-out patterns where a single message can activate multiple downstream executors simultaneously.

Advanced Email Processing Workflow

Building on the switch-case example, you'll create an enhanced email processing system that demonstrates sophisticated routing logic:

- **Spam emails** → Single spam handler (like switch-case)

- **Legitimate emails** → **Always** trigger email assistant + **Conditionally** trigger summarizer for long emails
- **Uncertain emails** → Single uncertain handler (like switch-case)
- **Database persistence** → Triggered for both short emails and summarized long emails

This pattern enables parallel processing pipelines that adapt to content characteristics.

Data Models for Multi-Selection

Extend the data models to support email length analysis and summarization:

C#

```
/// <summary>
/// Represents the result of enhanced email analysis with additional metadata.
/// </summary>
public sealed class AnalysisResult
{
    [JsonPropertyName("spam_decision")]
    [JsonConverter(typeof(JsonStringEnumConverter))]
    public SpamDecision spamDecision { get; set; }

    [JsonPropertyName("reason")]
    public string Reason { get; set; } = string.Empty;

    // Additional properties for sophisticated routing
    [JsonIgnore]
    public int EmailLength { get; set; }

    [JsonIgnore]
    public string EmailSummary { get; set; } = string.Empty;

    [JsonIgnore]
    public string EmailId { get; set; } = string.Empty;
}

/// <summary>
/// Represents the response from the email assistant.
/// </summary>
public sealed class EmailResponse
{
    [JsonPropertyName("response")]
    public string Response { get; set; } = string.Empty;
}

/// <summary>
/// Represents the response from the email summary agent.
/// </summary>
public sealed class EmailSummary
{
    [JsonPropertyName("summary")]
```

```
    public string Summary { get; set; } = string.Empty;
}

/// <summary>
/// A custom workflow event for database operations.
/// </summary>
internal sealed class DatabaseEvent(string message) : WorkflowEvent(message) { }

/// <summary>
/// Constants for email processing thresholds.
/// </summary>
public static class EmailProcessingConstants
{
    public const int LongEmailThreshold = 100;
}
```

Partitioner Function: The Heart of Multi-Selection

The partitioner function determines which executors should receive each message:

C#

```
/// <summary>
/// Creates a partitioner for routing messages based on the analysis result.
/// </summary>
/// <returns>A function that takes an analysis result and returns the target
partitions.</returns>
private static Func<AnalysisResult?, int, IEnumerable<int>> GetPartitioner()
{
    return (analysisResult, targetCount) =>
    {
        if (analysisResult is not null)
        {
            if (analysisResult.spamDecision == SpamDecision.Spam)
            {
                return [0]; // Route only to spam handler (index 0)
            }
            else if (analysisResult.spamDecision == SpamDecision.NotSpam)
            {
                // Always route to email assistant (index 1)
                List<int> targets = [1];

                // Conditionally add summarizer for long emails (index 2)
                if (analysisResult.EmailLength >
EmailProcessingConstants.LongEmailThreshold)
                {
                    targets.Add(2);
                }

                return targets;
            }
            else // Uncertain
```

```
            {
                return [3]; // Route only to uncertain handler (index 3)
            }
        }
        throw new InvalidOperationException("Invalid analysis result.");
    };
}
```

Key Features of the Partitioner Function

1. **Dynamic Target Selection**: Returns a list of executor indices to activate
2. **Content-Aware Routing**: Makes decisions based on message properties like email length
3. **Parallel Processing**: Multiple targets can execute simultaneously
4. **Conditional Logic**: Complex branching based on multiple criteria

Enhanced Workflow Executors

Implement executors that handle the advanced analysis and routing:

C#

```
/// <summary>
/// Executor that analyzes emails using an AI agent with enhanced analysis.
/// </summary>
internal sealed class EmailAnalysisExecutor :
ReflectingExecutor<EmailAnalysisExecutor>, IMessageHandler<ChatMessage,
AnalysisResult>
{
    private readonly AIAgent _emailAnalysisAgent;

    public EmailAnalysisExecutor(AIAgent emailAnalysisAgent) :
base("EmailAnalysisExecutor")
    {
        this._emailAnalysisAgent = emailAnalysisAgent;
    }

    public async ValueTask<AnalysisResult> HandleAsync(ChatMessage message,
IWorkflowContext context)
    {
        // Generate a random email ID and store the email content
        var newEmail = new Email
        {
            EmailId = Guid.NewGuid().ToString("N"),
            EmailContent = message.Text
        };
        await context.QueueStateUpdateAsync(newEmail.EmailId, newEmail, scopeName:
EmailStateConstants.EmailStateScope);

        // Invoke the agent for enhanced analysis
```

```
        var response = await this._emailAnalysisAgent.RunAsync(message);
        var analysisResult = JsonSerializer.Deserialize<AnalysisResult>
(response.Text);

        // Enrich with metadata for routing decisions
        analysisResult!.EmailId = newEmail.EmailId;
        analysisResult.EmailLength = newEmail.EmailContent.Length;

        return analysisResult;
    }
}

/// <summary>
/// Executor that assists with email responses using an AI agent.
/// </summary>
internal sealed class EmailAssistantExecutor :
ReflectingExecutor<EmailAssistantExecutor>, IMessageHandler<AnalysisResult,
EmailResponse>
{
    private readonly AIAgent _emailAssistantAgent;

    public EmailAssistantExecutor(AIAgent emailAssistantAgent) :
base("EmailAssistantExecutor")
    {
        this._emailAssistantAgent = emailAssistantAgent;
    }

    public async ValueTask<EmailResponse> HandleAsync(AnalysisResult message,
IWorkflowContext context)
    {
        if (message.spamDecision == SpamDecision.Spam)
        {
            throw new InvalidOperationException("This executor should only handle
non-spam messages.");
        }

        // Retrieve the email content from shared state
        var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope);

        // Invoke the agent to draft a response
        var response = await
this._emailAssistantAgent.RunAsync(email!.EmailContent);
        var emailResponse = JsonSerializer.Deserialize<EmailResponse>
(response.Text);

        return emailResponse!;
    }
}

/// <summary>
/// Executor that summarizes emails using an AI agent for long emails.
/// </summary>
internal sealed class EmailSummaryExecutor :
ReflectingExecutor<EmailSummaryExecutor>, IMessageHandler<AnalysisResult,
```

```
AnalysisResult>
{
    private readonly AIAgent _emailSummaryAgent;

    public EmailSummaryExecutor(AIAgent emailSummaryAgent) :
base("EmailSummaryExecutor")
    {
        this._emailSummaryAgent = emailSummaryAgent;
    }

    public async ValueTask<AnalysisResult> HandleAsync(AnalysisResult message,
IWorkflowContext context)
    {
        // Read the email content from shared state
        var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope);

        // Generate summary for long emails
        var response = await
this._emailSummaryAgent.RunAsync(email!.EmailContent);
        var emailSummary = JsonSerializer.Deserialize<EmailSummary>
(response.Text);

        // Enrich the analysis result with the summary
        message.EmailSummary = emailSummary!.Summary;

        return message;
    }
}

/// <summary>
/// Executor that sends emails.
/// </summary>
internal sealed class SendEmailExecutor() : ReflectingExecutor<SendEmailExecutor>
("SendEmailExecutor"), IMessageHandler<EmailResponse>
{
    public async ValueTask HandleAsync(EmailResponse message, IWorkflowContext
context) =>
        await context.YieldOutputAsync($"Email sent: {message.Response}");
}

/// <summary>
/// Executor that handles spam messages.
/// </summary>
internal sealed class HandleSpamExecutor() :
ReflectingExecutor<HandleSpamExecutor>("HandleSpamExecutor"),
IMessageHandler<AnalysisResult>
{
    public async ValueTask HandleAsync(AnalysisResult message, IWorkflowContext
context)
    {
        if (message.spamDecision == SpamDecision.Spam)
        {
            await context.YieldOutputAsync($"Email marked as spam:
{message.Reason}");
```

```
        }
        else
        {
            throw new InvalidOperationException("This executor should only handle
spam messages.");
        }
    }
}

/// <summary>
/// Executor that handles uncertain messages requiring manual review.
/// </summary>
internal sealed class HandleUncertainExecutor() :
ReflectingExecutor<HandleUncertainExecutor>("HandleUncertainExecutor"),
IMessageHandler<AnalysisResult>
{
    public async ValueTask HandleAsync(AnalysisResult message, IWorkflowContext
context)
    {
        if (message.spamDecision == SpamDecision.Uncertain)
        {
            var email = await context.ReadStateAsync<Email>(message.EmailId,
scopeName: EmailStateConstants.EmailStateScope);
            await context.YieldOutputAsync($"Email marked as uncertain:
{message.Reason}. Email content: {email?.EmailContent}");
        }
        else
        {
            throw new InvalidOperationException("This executor should only handle
uncertain spam decisions.");
        }
    }
}

/// <summary>
/// Executor that handles database access with custom events.
/// </summary>
internal sealed class DatabaseAccessExecutor() :
ReflectingExecutor<DatabaseAccessExecutor>("DatabaseAccessExecutor"),
IMessageHandler<AnalysisResult>
{
    public async ValueTask HandleAsync(AnalysisResult message, IWorkflowContext
context)
    {
        // Simulate database operations
        await context.ReadStateAsync<Email>(message.EmailId, scopeName:
EmailStateConstants.EmailStateScope);
        await Task.Delay(100); // Simulate database access delay

        // Emit custom database event for monitoring
        await context.AddEventAsync(new DatabaseEvent($"Email {message.EmailId}
saved to database."));
    }
}
```

Enhanced AI Agents

Create agents for analysis, assistance, and summarization:

C#

```
/// <summary>
/// Create an enhanced email analysis agent.
/// </summary>
/// <returns>A ChatClientAgent configured for comprehensive email
analysis</returns>
private static ChatClientAgent GetEmailAnalysisAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are a spam
detection assistant that identifies spam emails.")
    {
        ChatOptions = new()
        {
            ResponseFormat = ChatResponseFormat.ForJsonSchema<AnalysisResult>()
        }
    });

/// <summary>
/// Creates an email assistant agent.
/// </summary>
/// <returns>A ChatClientAgent configured for email assistance</returns>
private static ChatClientAgent GetEmailAssistantAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are an email
assistant that helps users draft responses to emails with professionalism.")
    {
        ChatOptions = new()
        {
            ResponseFormat = ChatResponseFormat.ForJsonSchema<EmailResponse>()
        }
    });

/// <summary>
/// Creates an agent that summarizes emails.
/// </summary>
/// <returns>A ChatClientAgent configured for email summarization</returns>
private static ChatClientAgent GetEmailSummaryAgent(IChatClient chatClient) =>
    new(chatClient, new ChatClientAgentOptions(instructions: "You are an assistant
that helps users summarize emails.")
    {
        ChatOptions = new()
        {
            ResponseFormat = ChatResponseFormat.ForJsonSchema<EmailSummary>()
        }
    });
```

Multi-Selection Workflow Construction

Construct the workflow with sophisticated routing and parallel processing:

C#

```
public static class Program
{
    private const int LongEmailThreshold = 100;

    private static async Task Main()
    {
        // Set up the Azure OpenAI client
        var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")
?? throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
        var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
        var chatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential()).GetChatClient(deploymentName).AsIChatClient();

        // Create agents
        AIAgent emailAnalysisAgent = GetEmailAnalysisAgent(chatClient);
        AIAgent emailAssistantAgent = GetEmailAssistantAgent(chatClient);
        AIAgent emailSummaryAgent = GetEmailSummaryAgent(chatClient);

        // Create executors
        var emailAnalysisExecutor = new EmailAnalysisExecutor(emailAnalysisAgent);
        var emailAssistantExecutor = new
EmailAssistantExecutor(emailAssistantAgent);
        var emailSummaryExecutor = new EmailSummaryExecutor(emailSummaryAgent);
        var sendEmailExecutor = new SendEmailExecutor();
        var handleSpamExecutor = new HandleSpamExecutor();
        var handleUncertainExecutor = new HandleUncertainExecutor();
        var databaseAccessExecutor = new DatabaseAccessExecutor();

        // Build the workflow with multi-selection fan-out
        WorkflowBuilder builder = new(emailAnalysisExecutor);
        builder.AddFanOutEdge(
            emailAnalysisExecutor,
            targets: [
                handleSpamExecutor,        // Index 0: Spam handler
                emailAssistantExecutor,    // Index 1: Email assistant (always for
NotSpam)
                emailSummaryExecutor,      // Index 2: Summarizer (conditionally
for long NotSpam)
                handleUncertainExecutor,   // Index 3: Uncertain handler
            ],
            partitioner: GetPartitioner()
        )
        // Email assistant branch
        .AddEdge(emailAssistantExecutor, sendEmailExecutor)

        // Database persistence: conditional routing
        .AddEdge<AnalysisResult>(
            emailAnalysisExecutor,
```

```
            databaseAccessExecutor,
            condition: analysisResult => analysisResult?.EmailLength <=
LongEmailThreshold) // Short emails
        .AddEdge(emailSummaryExecutor, databaseAccessExecutor) // Long emails with
summary

        .WithOutputFrom(handleUncertainExecutor, handleSpamExecutor,
sendEmailExecutor);

        var workflow = builder.Build();

        // Read a moderately long email to trigger both assistant and summarizer
        string email = Resources.Read("email.txt");

        // Execute the workflow with custom event handling
        StreamingRun run = await InProcessExecution.StreamAsync(workflow, new
ChatMessage(ChatRole.User, email));
        await run.TrySendMessageAsync(new TurnToken(emitEvents: true));
        await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
        {
            if (evt is WorkflowOutputEvent outputEvent)
            {
                Console.WriteLine($"Output: {outputEvent}");
            }

            if (evt is DatabaseEvent databaseEvent)
            {
                Console.WriteLine($"Database: {databaseEvent}");
            }
        }
    }
}
```

Pattern Comparison: Multi-Selection vs. Switch-Case

Switch-Case Pattern (Previous):

C#

```
// One input → exactly one output
builder.AddSwitch(spamDetectionExecutor, switchBuilder =>
    switchBuilder
    .AddCase(GetCondition(SpamDecision.NotSpam), emailAssistantExecutor)
    .AddCase(GetCondition(SpamDecision.Spam), handleSpamExecutor)
    .WithDefault(handleUncertainExecutor)
)
```

Multi-Selection Pattern:

C#

```
// One input → one or more outputs (dynamic fan-out)
builder.AddFanOutEdge(
    emailAnalysisExecutor,
    targets: [handleSpamExecutor, emailAssistantExecutor, emailSummaryExecutor,
handleUncertainExecutor],
    partitioner: GetPartitioner() // Returns list of target indices
)
```

Key Advantages of Multi-Selection Edges

1. **Parallel Processing**: Multiple branches can execute simultaneously
2. **Conditional Fan-out**: Number of targets varies based on content
3. **Content-Aware Routing**: Decisions based on message properties, not just type
4. **Efficient Resource Usage**: Only necessary branches are activated
5. **Complex Business Logic**: Supports sophisticated routing scenarios

Running the Multi-Selection Example

When you run this workflow with a long email:

text

```
Output: Email sent: [Professional response generated by AI]
Database: Email abc123 saved to database.
```

When you run with a short email, the summarizer is skipped:

text

```
Output: Email sent: [Professional response generated by AI]
Database: Email def456 saved to database.
```

Real-World Use Cases

- **Email Systems**: Route to reply assistant + archive + analytics (conditionally)
- **Content Processing**: Trigger transcription + translation + analysis (based on content type)
- **Order Processing**: Route to fulfillment + billing + notifications (based on order properties)
- **Data Pipelines**: Trigger different analytics flows based on data characteristics

Multi-Selection Complete Implementation

For the complete working implementation, see this sample in the Agent Framework repository.

Next Steps

Learn about handling requests and responses in workflows

Handle Requests and Responses in Workflows

10/01/2025

This tutorial demonstrates how to handle requests and responses in workflows using the Agent Framework Workflows. You'll learn how to create interactive workflows that can pause execution to request input from external sources (like humans or other systems) and then resume once a response is provided.

In .NET, human-in-the-loop workflows use `InputPort` and external request handling to pause execution and gather user input. This pattern enables interactive workflows where the system can request information from external sources during execution.

Key Components

InputPort and External Requests

An `InputPort` acts as a bridge between the workflow and external input sources. When the workflow needs input, it generates a `RequestInfoEvent` that your application handles:

C#

```
// Create an InputPort for handling human input requests
InputPort numberInputPort = InputPort.Create<NumberSignal, int>("GuessNumber");
```

Signal Types

Define signal types to communicate different request types:

C#

```
/// <summary>
/// Signals used for communication between guesses and the JudgeExecutor.
/// </summary>
internal enum NumberSignal
{
    Init,     // Initial guess request
    Above,    // Previous guess was too high
    Below,    // Previous guess was too low
}
```

Workflow Executor

Create executors that process user input and provide feedback:

C#

```
/// <summary>
/// Executor that judges the guess and provides feedback.
/// </summary>
internal sealed class JudgeExecutor() : ReflectingExecutor<JudgeExecutor>
("Judge"), IMessageHandler<int>
{
    private readonly int _targetNumber;
    private int _tries;

    public JudgeExecutor(int targetNumber) : this()
    {
        _targetNumber = targetNumber;
    }

    public async ValueTask HandleAsync(int message, IWorkflowContext context)
    {
        _tries++;
        if (message == _targetNumber)
        {
            await context.YieldOutputAsync($"{_targetNumber} found in {_tries}
tries!")
                         .ConfigureAwait(false);
        }
        else if (message < _targetNumber)
        {
            await
context.SendMessageAsync(NumberSignal.Below).ConfigureAwait(false);
        }
        else
        {
            await
context.SendMessageAsync(NumberSignal.Above).ConfigureAwait(false);
        }
    }
}
```

Building the Workflow

Connect the InputPort and executor in a feedback loop:

C#

```
internal static ValueTask<Workflow<NumberSignal>> GetWorkflowAsync()
{
```

```
    // Create the executors
    InputPort numberInputPort = InputPort.Create<NumberSignal, int>
("GuessNumber");
    JudgeExecutor judgeExecutor = new(42);

    // Build the workflow by connecting executors in a loop
    return new WorkflowBuilder(numberInputPort)
        .AddEdge(numberInputPort, judgeExecutor)
        .AddEdge(judgeExecutor, numberInputPort)
        .WithOutputFrom(judgeExecutor)
        .BuildAsync<NumberSignal>();
}
```

Executing the Interactive Workflow

Handle external requests during workflow execution:

C#

```
private static async Task Main()
{
    // Create the workflow
    var workflow = await WorkflowHelper.GetWorkflowAsync().ConfigureAwait(false);

    // Execute the workflow
    StreamingRun handle = await InProcessExecution.StreamAsync(workflow,
NumberSignal.Init).ConfigureAwait(false);
    await foreach (WorkflowEvent evt in
handle.WatchStreamAsync().ConfigureAwait(false))
    {
        switch (evt)
        {
            case RequestInfoEvent requestInputEvt:
                // Handle human input request from the workflow
                ExternalResponse response =
HandleExternalRequest(requestInputEvt.Request);
                await handle.SendResponseAsync(response).ConfigureAwait(false);
                break;

            case WorkflowOutputEvent outputEvt:
                // The workflow has yielded output
                Console.WriteLine($"Workflow completed with result:
{outputEvt.Data}");
                return;
        }
    }
}
```

Request Handling

Process different types of input requests:

C#

```
private static ExternalResponse HandleExternalRequest(ExternalRequest request)
{
    if (request.DataIs<NumberSignal>())
    {
        switch (request.DataAs<NumberSignal>())
        {
            case NumberSignal.Init:
                int initialGuess = ReadIntegerFromConsole("Please provide your
initial guess: ");
                return request.CreateResponse(initialGuess);
            case NumberSignal.Above:
                int lowerGuess = ReadIntegerFromConsole("You previously guessed
too large. Please provide a new guess: ");
                return request.CreateResponse(lowerGuess);
            case NumberSignal.Below:
                int higherGuess = ReadIntegerFromConsole("You previously guessed
too small. Please provide a new guess: ");
                return request.CreateResponse(higherGuess);
        }
    }

    throw new NotSupportedException($"Request {request.PortInfo.RequestType} is
not supported");
}

private static int ReadIntegerFromConsole(string prompt)
{
    while (true)
    {
        Console.Write(prompt);
        string? input = Console.ReadLine();
        if (int.TryParse(input, out int value))
        {
            return value;
        }
        Console.WriteLine("Invalid input. Please enter a valid integer.");
    }
}
```

Implementation Concepts

RequestInfoEvent Flow

1. **Workflow Execution**: The workflow processes until it needs external input
2. **Request Generation**: InputPort generates a `RequestInfoEvent` with the request details

3. **External Handling**: Your application catches the event and gathers user input
4. **Response Submission**: Send an `ExternalResponse` back to continue the workflow
5. **Workflow Resumption**: The workflow continues processing with the provided input

Workflow Lifecycle

- **Streaming Execution**: Use `StreamAsync` to monitor events in real-time
- **Event Handling**: Process `RequestInfoEvent` for input requests and `WorkflowOutputEvent` for completion
- **Response Coordination**: Match responses to requests using the workflow's response handling mechanism

Implementation Flow

1. **Workflow Initialization**: The workflow starts by sending a `NumberSignal.Init` to the InputPort.

2. **Request Generation**: The InputPort generates a `RequestInfoEvent` requesting an initial guess from the user.

3. **Workflow Pause**: The workflow pauses and waits for external input while the application handles the request.

4. **Human Response**: The external application collects user input and sends an `ExternalResponse` back to the workflow.

5. **Processing and Feedback**: The `JudgeExecutor` processes the guess and either completes the workflow or sends a new signal (Above/Below) to request another guess.

6. **Loop Continuation**: The process repeats until the correct number is guessed.

Framework Benefits

- **Type Safety**: Strong typing ensures request-response contracts are maintained
- **Event-Driven**: Rich event system provides visibility into workflow execution
- **Pausable Execution**: Workflows can pause indefinitely while waiting for external input
- **State Management**: Workflow state is preserved across pause-resume cycles
- **Flexible Integration**: InputPorts can integrate with any external input source (UI, API, console, etc.)

Complete Sample

For the complete working implementation, see the Human-in-the-Loop Basic sample.

This pattern enables building sophisticated interactive applications where users can provide input at key decision points within automated workflows.

Next Steps

Learn about checkpointing and resuming workflows

Checkpointing and Resuming Workflows

10/01/2025

Checkpointing allows workflows to save their state at specific points and resume execution later, even after process restarts. This is crucial for long-running workflows, error recovery, and human-in-the-loop scenarios.

Key Components

CheckpointManager

The `CheckpointManager` provides checkpoint storage and retrieval functionality:

C#

```
using Microsoft.Agents.AI.Workflows;

// Use the default in-memory checkpoint manager
var checkpointManager = CheckpointManager.Default;

// Or create a custom checkpoint manager with JSON serialization
var checkpointManager = CheckpointManager.CreateJson(store, customOptions);
```

Enabling Checkpointing

Enable checkpointing when executing workflows using `InProcessExecution`:

C#

```
using Microsoft.Agents.AI.Workflows;

// Create workflow with checkpointing support
var workflow = await WorkflowHelper.GetWorkflowAsync();
var checkpointManager = CheckpointManager.Default;

// Execute with checkpointing enabled
Checkpointed<StreamingRun> checkpointedRun = await InProcessExecution
    .StreamAsync(workflow, NumberSignal.Init, checkpointManager);
```

State Persistence

Executor State

Executors can persist local state that survives checkpoints using the `ReflectingExecutor` base class:

C#

```
internal sealed class GuessNumberExecutor :
ReflectingExecutor<GuessNumberExecutor>, IMessageHandler<NumberSignal>
{
    private static readonly StateKey StateKey = new("GuessNumberExecutor.State");

    public int LowerBound { get; private set; }
    public int UpperBound { get; private set; }

    public async ValueTask HandleAsync(NumberSignal message, IWorkflowContext
context)
    {
        int guess = (LowerBound + UpperBound) / 2;
        await context.SendMessageAsync(guess);
    }

    /// <summary>
    /// Checkpoint the current state of the executor.
    /// This must be overridden to save any state that is needed to resume the
executor.
    /// </summary>
    protected override ValueTask OnCheckpointingAsync(IWorkflowContext context,
CancellationToken cancellationToken = default) =>
        context.QueueStateUpdateAsync(StateKey, (LowerBound, UpperBound));

    /// <summary>
    /// Restore the state of the executor from a checkpoint.
    /// This must be overridden to restore any state that was saved during
checkpointing.
    /// </summary>
    protected override async ValueTask OnCheckpointRestoredAsync(IWorkflowContext
context, CancellationToken cancellationToken = default)
    {
        var state = await context.ReadStateAsync<(int, int)>(StateKey);
        (LowerBound, UpperBound) = state;
    }
}
```

Automatic Checkpoint Creation

Checkpoints are automatically created at the end of each super step when a checkpoint manager is provided:

C#

```
var checkpoints = new List<CheckpointInfo>();

await foreach (WorkflowEvent evt in checkpointedRun.Run.WatchStreamAsync())
{
    switch (evt)
    {
        case SuperStepCompletedEvent superStepCompletedEvt:
            // Checkpoints are automatically created at super step boundaries
            CheckpointInfo? checkpoint =
superStepCompletedEvt.CompletionInfo!.Checkpoint;
            if (checkpoint is not null)
            {
                checkpoints.Add(checkpoint);
                Console.WriteLine($"Checkpoint created at step
{checkpoints.Count}.");
            }
            break;

        case WorkflowOutputEvent workflowOutputEvt:
            Console.WriteLine($"Workflow completed with result:
{workflowOutputEvt.Data}");
            break;
    }
}
```

Working with Checkpoints

Accessing Checkpoint Information

Access checkpoint metadata from completed runs:

C#

```
// Get all checkpoints from a checkpointed run
var allCheckpoints = checkpointedRun.Checkpoints;

// Get the latest checkpoint
var latestCheckpoint = checkpointedRun.LatestCheckpoint;

// Access checkpoint details
foreach (var checkpoint in checkpoints)
{
    Console.WriteLine($"Checkpoint ID: {checkpoint.CheckpointId}");
    Console.WriteLine($"Step Number: {checkpoint.StepNumber}");
    Console.WriteLine($"Parent ID: {checkpoint.Parent?.CheckpointId ?? "None"}");
}
```

Checkpoint Storage

Checkpoints are managed through the `CheckpointManager` interface:

C#

```
// Commit a checkpoint (usually done automatically)
CheckpointInfo checkpointInfo = await
checkpointManager.CommitCheckpointAsync(runId, checkpoint);

// Retrieve a checkpoint
Checkpoint restoredCheckpoint = await
checkpointManager.LookupCheckpointAsync(runId, checkpointInfo);
```

Resuming from Checkpoints

Streaming Resume

Resume execution from a checkpoint and stream events in real-time:

C#

```
// Resume from a specific checkpoint with streaming
CheckpointInfo savedCheckpoint = checkpoints[checkpointIndex];

Checkpointed<StreamingRun> resumedRun = await InProcessExecution
    .ResumeStreamAsync(workflow, savedCheckpoint, checkpointManager, runId);

await foreach (WorkflowEvent evt in resumedRun.Run.WatchStreamAsync())
{
    switch (evt)
    {
        case ExecutorCompletedEvent executorCompletedEvt:
            Console.WriteLine($"Executor {executorCompletedEvt.ExecutorId}
completed.");
            break;

        case WorkflowOutputEvent workflowOutputEvt:
            Console.WriteLine($"Workflow completed with result:
{workflowOutputEvt.Data}");
            return;
    }
}
```

Non-Streaming Resume

Resume and wait for completion:

```C#
// Resume from checkpoint without streaming
Checkpointed<Run> resumedRun = await InProcessExecution
    .ResumeAsync(workflow, savedCheckpoint, checkpointManager, runId);

// Wait for completion and get final result
var result = await resumedRun.Run.WaitForCompletionAsync();
```

In-Place Restoration

Restore a checkpoint directly to an existing run instance:

```C#
// Restore checkpoint to the same run instance
await checkpointedRun.RestoreCheckpointAsync(savedCheckpoint);

// Continue execution from the restored state
await foreach (WorkflowEvent evt in checkpointedRun.Run.WatchStreamAsync())
{
    // Handle events as normal
    if (evt is WorkflowOutputEvent outputEvt)
    {
        Console.WriteLine($"Resumed workflow result: {outputEvt.Data}");
        break;
    }
}
```

New Workflow Instance (Rehydration)

Create a new workflow instance from a checkpoint:

```C#
// Create a completely new workflow instance
var newWorkflow = await WorkflowHelper.GetWorkflowAsync();

// Resume with the new instance from a saved checkpoint
Checkpointed<StreamingRun> newCheckpointedRun = await InProcessExecution
    .ResumeStreamAsync(newWorkflow, savedCheckpoint, checkpointManager,
originalRunId);

await foreach (WorkflowEvent evt in newCheckpointedRun.Run.WatchStreamAsync())
{
    if (evt is WorkflowOutputEvent workflowOutputEvt)
```

```
    {
        Console.WriteLine($"Rehydrated workflow result:
{workflowOutputEvt.Data}");
        break;
    }
}
```

Human-in-the-Loop with Checkpointing

Combine checkpointing with human-in-the-loop workflows:

C#

```
var checkpoints = new List<CheckpointInfo>();

await foreach (WorkflowEvent evt in checkpointedRun.Run.WatchStreamAsync())
{
    switch (evt)
    {
        case RequestInfoEvent requestInputEvt:
            // Handle external requests
            ExternalResponse response =
HandleExternalRequest(requestInputEvt.Request);
            await checkpointedRun.Run.SendResponseAsync(response);
            break;

        case SuperStepCompletedEvent superStepCompletedEvt:
            // Save checkpoint after each interaction
            CheckpointInfo? checkpoint =
superStepCompletedEvt.CompletionInfo!.Checkpoint;
            if (checkpoint is not null)
            {
                checkpoints.Add(checkpoint);
                Console.WriteLine($"Checkpoint created after human interaction.");
            }
            break;

        case WorkflowOutputEvent workflowOutputEvt:
            Console.WriteLine($"Workflow completed: {workflowOutputEvt.Data}");
            return;
    }
}

// Later, resume from any checkpoint
if (checkpoints.Count > 0)
{
    var selectedCheckpoint = checkpoints[1]; // Select specific checkpoint
    await checkpointedRun.RestoreCheckpointAsync(selectedCheckpoint);

    // Continue from that point
    await foreach (WorkflowEvent evt in checkpointedRun.Run.WatchStreamAsync())
    {
```

```
        // Handle remaining workflow execution
    }
}
```

Complete Example Pattern

Here's a comprehensive checkpointing workflow pattern:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Agents.AI.Workflows;

public static class CheckpointingExample
{
    public static async Task RunAsync()
    {
        // Create workflow and checkpoint manager
        var workflow = await WorkflowHelper.GetWorkflowAsync();
        var checkpointManager = CheckpointManager.Default;
        var checkpoints = new List<CheckpointInfo>();

        Console.WriteLine("Starting workflow with checkpointing...");

        // Execute workflow with checkpointing
        Checkpointed<StreamingRun> checkpointedRun = await InProcessExecution
            .StreamAsync(workflow, NumberSignal.Init, checkpointManager);

        // Monitor execution and collect checkpoints
        await foreach (WorkflowEvent evt in
checkpointedRun.Run.WatchStreamAsync())
        {
            switch (evt)
            {
                case ExecutorCompletedEvent executorEvt:
                    Console.WriteLine($"Executor {executorEvt.ExecutorId}
completed.");
                    break;

                case SuperStepCompletedEvent superStepEvt:
                    var checkpoint = superStepEvt.CompletionInfo!.Checkpoint;
                    if (checkpoint is not null)
                    {
                        checkpoints.Add(checkpoint);
                        Console.WriteLine($"Checkpoint {checkpoints.Count}
created.");
                    }
                    break;

                case WorkflowOutputEvent outputEvt:
```

```
                    Console.WriteLine($"Workflow completed: {outputEvt.Data}");
                    goto FinishExecution;
                }
            }

            FinishExecution:
            Console.WriteLine($"Total checkpoints created: {checkpoints.Count}");

            // Demonstrate resuming from a checkpoint
            if (checkpoints.Count > 5)
            {
                var selectedCheckpoint = checkpoints[5];
                Console.WriteLine($"Resuming from checkpoint 6...");

                // Restore to same instance
                await checkpointedRun.RestoreCheckpointAsync(selectedCheckpoint);

                await foreach (WorkflowEvent evt in
checkpointedRun.Run.WatchStreamAsync())
                {
                    if (evt is WorkflowOutputEvent resumedOutputEvt)
                    {
                        Console.WriteLine($"Resumed workflow result:
{resumedOutputEvt.Data}");
                        break;
                    }
                }
            }

            // Demonstrate rehydration with new workflow instance
            if (checkpoints.Count > 3)
            {
                var newWorkflow = await WorkflowHelper.GetWorkflowAsync();
                var rehydrationCheckpoint = checkpoints[3];

                Console.WriteLine("Rehydrating from checkpoint 4 with new workflow
instance...");

                Checkpointed<StreamingRun> newRun = await InProcessExecution
                    .ResumeStreamAsync(newWorkflow, rehydrationCheckpoint,
checkpointManager, checkpointedRun.Run.RunId);

                await foreach (WorkflowEvent evt in newRun.Run.WatchStreamAsync())
                {
                    if (evt is WorkflowOutputEvent rehydratedOutputEvt)
                    {
                        Console.WriteLine($"Rehydrated workflow result:
{rehydratedOutputEvt.Data}");
                        break;
                    }
                }
            }
        }
    }
}
```

Key Benefits

- **Fault Tolerance**: Workflows can recover from failures by resuming from the last checkpoint
- **Long-Running Processes**: Break long workflows into manageable segments with automatic checkpoint boundaries
- **Human-in-the-Loop**: Pause for external input and resume later from saved state
- **Debugging**: Inspect workflow state at specific points and resume execution for testing
- **Portability**: Checkpoints can be restored to new workflow instances (rehydration)
- **Automatic Management**: Checkpoints are created automatically at super step boundaries

Running the Example

For the complete working implementation, see the CheckpointAndResume sample.

Next Steps

Learn about Workflow Visualization

Visualizing Workflows

10/01/2025

Overview

The Agent Framework provides powerful visualization capabilities for workflows through the `WorkflowViz` class. This allows you to generate visual diagrams of your workflow structure in multiple formats including Mermaid flowcharts, GraphViz DOT diagrams, and exported image files (SVG, PNG, PDF).

Getting Started with WorkflowViz

Basic Setup

Python

```python
from agent_framework import WorkflowBuilder, WorkflowViz

# Create your workflow
workflow = (
    WorkflowBuilder()
    .set_start_executor(start_executor)
    .add_edge(start_executor, end_executor)
    .build()
)

# Create visualization
viz = WorkflowViz(workflow)
```

Installation Requirements

For basic text output (Mermaid and DOT), no additional dependencies are needed. For image export:

Bash

```bash
# Install the viz extra
pip install agent-framework[viz]

# Install GraphViz binaries (required for image export)
# On Ubuntu/Debian:
sudo apt-get install graphviz
```

```
# On macOS:
brew install graphviz

# On Windows: Download from https://graphviz.org/download/
```

Visualization Formats

Mermaid Flowcharts

Generate Mermaid syntax for modern, web-friendly diagrams:

Python

```python
# Generate Mermaid flowchart
mermaid_content = viz.to_mermaid()
print("Mermaid flowchart:")
print(mermaid_content)

# Example output:
# flowchart TD
#   dispatcher["dispatcher (Start)"];
#   researcher["researcher"];
#   marketer["marketer"];
#   legal["legal"];
#   aggregator["aggregator"];
#   dispatcher --> researcher;
#   dispatcher --> marketer;
#   dispatcher --> legal;
#   researcher --> aggregator;
#   marketer --> aggregator;
#   legal --> aggregator;
```

GraphViz DOT Format

Generate DOT format for detailed graph representations:

Python

```python
# Generate DOT diagram
dot_content = viz.to_digraph()
print("DOT diagram:")
print(dot_content)

# Example output:
# digraph Workflow {
#   rankdir=TD;
#   node [shape=box, style=filled, fillcolor=lightblue];
```

```
#   "dispatcher" [fillcolor=lightgreen, label="dispatcher\n(Start)"];
#   "researcher" [label="researcher"];
#   "marketer" [label="marketer"];
#   ...
# }
```

Image Export

Supported Formats

Export workflows as high-quality images:

Python

```python
try:
    # Export as SVG (vector format, recommended)
    svg_file = viz.export(format="svg")
    print(f"SVG exported to: {svg_file}")

    # Export as PNG (raster format)
    png_file = viz.export(format="png")
    print(f"PNG exported to: {png_file}")

    # Export as PDF (vector format)
    pdf_file = viz.export(format="pdf")
    print(f"PDF exported to: {pdf_file}")

    # Export raw DOT file
    dot_file = viz.export(format="dot")
    print(f"DOT file exported to: {dot_file}")

except ImportError:
    print("Install 'viz' extra and GraphViz for image export:")
    print("pip install agent-framework[viz]")
    print("Also install GraphViz binaries for your platform")
```

Custom Filenames

Specify custom output filenames:

Python

```python
# Export with custom filename
svg_path = viz.export(format="svg", filename="my_workflow.svg")
png_path = viz.export(format="png", filename="workflow_diagram.png")

# Convenience methods
```

```
svg_path = viz.save_svg("workflow.svg")
png_path = viz.save_png("workflow.png")
pdf_path = viz.save_pdf("workflow.pdf")
```

Workflow Pattern Visualizations

Fan-out/Fan-in Patterns

Visualizations automatically handle complex routing patterns:

Python

```python
from agent_framework import (
    WorkflowBuilder, WorkflowViz, AgentExecutor,
    AgentExecutorRequest, AgentExecutorResponse
)

# Create agents
researcher = AgentExecutor(chat_client.create_agent(...), id="researcher")
marketer = AgentExecutor(chat_client.create_agent(...), id="marketer")
legal = AgentExecutor(chat_client.create_agent(...), id="legal")

# Build fan-out/fan-in workflow
workflow = (
    WorkflowBuilder()
    .set_start_executor(dispatcher)
    .add_fan_out_edges(dispatcher, [researcher, marketer, legal])  # Fan-out
    .add_fan_in_edges([researcher, marketer, legal], aggregator)   # Fan-in
    .build()
)

# Visualize
viz = WorkflowViz(workflow)
print(viz.to_mermaid())
```

Fan-in nodes are automatically rendered with special styling:

- **DOT format**: Ellipse shape with light golden background and "fan-in" label
- **Mermaid format**: Double circle nodes `((fan-in))` for clear identification

Conditional Edges

Conditional routing is visualized with distinct styling:

Python

```python
def spam_condition(content: str) -> bool:
    return "spam" in content.lower()

workflow = (
    WorkflowBuilder()
    .add_edge(classifier, spam_handler, condition=spam_condition)
    .add_edge(classifier, normal_processor)  # Unconditional edge
    .build()
)

viz = WorkflowViz(workflow)
print(viz.to_digraph())
```

Conditional edges appear as:

- **DOT format**: Dashed lines with "conditional" labels
- **Mermaid format**: Dotted arrows (`-.->`) with "conditional" labels

Sub-workflows

Nested workflows are visualized as clustered subgraphs:

Python

```python
from agent_framework import WorkflowExecutor

# Create sub-workflow
sub_workflow = WorkflowBuilder().add_edge(sub_exec1, sub_exec2).build()
sub_workflow_executor = WorkflowExecutor(sub_workflow, id="sub_workflow")

# Main workflow containing sub-workflow
main_workflow = (
    WorkflowBuilder()
    .add_edge(main_executor, sub_workflow_executor)
    .add_edge(sub_workflow_executor, final_executor)
    .build()
)

viz = WorkflowViz(main_workflow)
dot_content = viz.to_digraph()  # Shows nested clusters
mermaid_content = viz.to_mermaid()  # Shows subgraph structures
```

Complete Example

For a comprehensive example showing workflow visualization with fan-out/fan-in patterns, custom executors, and multiple export formats, see the Concurrent with Visualization sample.

The sample demonstrates:

- Expert agent workflow with researcher, marketer, and legal agents
- Custom dispatcher and aggregator executors
- Mermaid and DOT visualization generation
- SVG, PNG, and PDF export capabilities
- Integration with Azure OpenAI agents

Visualization Features

Node Styling

- **Start executors**: Green background with "(Start)" label
- **Regular executors**: Blue background with executor ID
- **Fan-in nodes**: Golden background with ellipse shape (DOT) or double circles (Mermaid)

Edge Styling

- **Normal edges**: Solid arrows
- **Conditional edges**: Dashed/dotted arrows with "conditional" labels
- **Fan-out/Fan-in**: Automatic routing through intermediate nodes

Layout Options

- **Top-down layout**: Clear hierarchical flow visualization
- **Subgraph clustering**: Nested workflows shown as grouped clusters
- **Automatic positioning**: GraphViz handles optimal node placement

Integration with Development Workflow

Documentation Generation

Python

```
# Generate documentation diagrams
workflow_viz = WorkflowViz(my_workflow)
doc_diagram = workflow_viz.save_svg("docs/workflow_architecture.svg")
```

Debugging and Analysis

Python

```python
# Analyze workflow structure
print("Workflow complexity analysis:")
dot_content = viz.to_digraph()
edge_count = dot_content.count(" -> ")
node_count = dot_content.count('[label=')
print(f"Nodes: {node_count}, Edges: {edge_count}")
```

CI/CD Integration

Python

```python
# Export diagrams for automated documentation
import os
if os.getenv("CI"):
    # Export for docs during CI build
    viz.save_svg("build/artifacts/workflow.svg")
    viz.export(format="dot", filename="build/artifacts/workflow.dot")
```

Best Practices

1. **Use descriptive executor IDs** - They become node labels in visualizations
2. **Export SVG for documentation** - Vector format scales well in docs
3. **Use Mermaid for web integration** - Copy-paste into Markdown/wiki systems
4. **Leverage fan-in/fan-out visualization** - Clearly shows parallelism patterns
5. **Include visualization in testing** - Verify workflow structure matches expectations

Running the Example

For the complete working implementation with visualization, see the Concurrent with Visualization sample .

Agent Framework User Guide

09/25/2025

Welcome to the Agent Framework User Guide. This guide provides comprehensive information for developers and solution architects working with the Agent Framework. Here, you'll find detailed explanations of agent concepts, configuration options, advanced features, and best practices for building robust, scalable agent-based applications. Whether you're just getting started or looking to deepen your expertise, this guide will help you understand how to leverage the full capabilities of the Agent Framework in your projects.

Microsoft Agent Framework Agent Types

10/01/2025

The Microsoft Agent Framework provides support for several types of agents to accommodate different use cases and requirements.

All agents are derived from a common base class, `AIAgent`, which provides a consistent interface for all agent types. This allows for building common, agent agnostic, higher level functionality such as multi-agent orchestrations.

> **Important**
>
> If you use the Microsoft Agent Framework to build applications that operate with third-party servers or agents, you do so at your own risk. We recommend reviewing all data being shared with third-party servers or agents and being cognizant of third-party practices for retention and location of data. It is your responsibility to manage whether your data will flow outside of your organization's Azure compliance and geographic boundaries and any related implications.

Simple agents based on inference services

The agent framework makes it easy to create simple agents based on many different inference services. Any inference service that provides an Microsoft.Extensions.AI.IChatClient implementation can be used to build these agents. The `Microsoft.Agents.AI.ChatClientAgent` is the agent class used to provide an agent for any `IChatClient` implementation.

These agents support a wide range of functionality out of the box:

1. Function calling
2. Multi-turn conversations with local chat history management or service provided chat history management
3. Custom service provided tools (e.g. MCP, Code Execution)
4. Structured output

To create one of these agents, simply construct a `ChatClientAgent` using the ChatClient implementation of your choice.

C#

```
using Microsoft.Agents.AI;
```

```
var agent = new ChatClientAgent(chatClient, instructions: "You are a helpful
assistant");
```

For many popular services, we also have helpers to make creating these agents even easier. See the documentation for each service, for more information:

Expand table

Underlying Inference Service	Description	Service Chat History storage supported	Custom Chat History storage supported
Azure AI Foundry Agent	An agent that uses the Azure AI Foundry Agents Service as its backend.	Yes	No
Azure OpenAI ChatCompletion	An agent that uses the Azure OpenAI ChatCompletion service.	No	Yes
Azure OpenAI Responses	An agent that uses the Azure OpenAI Responses service.	Yes	Yes
OpenAI ChatCompletion	An agent that uses the OpenAI ChatCompletion service.	No	Yes
OpenAI Responses	An agent that uses the OpenAI Responses service.	Yes	Yes
OpenAI Assistants	An agent that uses the OpenAI Assistants service.	Yes	No
Any other ChatClient	You can also use any other Microsoft.Extensions.AI.IChatClient implementation to create an agent.	Varies	Varies

Complex custom agents

It is also possible to create fully custom agents, that are not just wrappers around a ChatClient. The agent framework provides the `AIAgent` base type. This base type is the core abstraction for all agents, which when subclassed allows for complete control over the agent's behavior and capabilities.

See the documentation for Custom Agents for more information.

Proxies for remote agents

The agent framework provides out of the box `AIAgent` implementations for common service hosted agent protocols, such as A2A. This way you can easily connect to and use remote agents from your application.

See the documentation for each agent type, for more information:

Expand table

Protocol	Description
A2A	An agent that serves as a proxy to a remote agent via the A2A protocol.

Azure AI Foundry Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the Azure AI Foundry Agents service.

Getting Started

Add the Agents Azure AI NuGet package to your project.

PowerShell

```
dotnet add package Azure.Identity
dotnet add package Microsoft.Agents.AI.AzureAI --prerelease
```

Creating Azure AI Foundry Agents

As a first step you need to create a client to connect to the Azure AI Foundry Agents service.

C#

```
using System;
using Azure.AI.Agents.Persistent;
using Azure.Identity;
using Microsoft.Agents.AI;

var persistentAgentsClient = new PersistentAgentsClient(
    "https://<myresource>.services.ai.azure.com/api/projects/<myproject>",
    new AzureCliCredential());
```

To use the Azure AI Foundry Agents service, you need create an agent resource in the service. This can be done using either the Azure.AI.Agents.Persistent SDK or using Microsoft Agent Framework helpers.

Using the Persistent SDK

Create a persistent agent and retrieve it as an `AIAgent` using the `PersistentAgentsClient`.

C#

```
// Create a persistent agent
var agentMetadata = await persistentAgentsClient.Administration.CreateAgentAsync(
    model: "gpt-4o-mini",
```

```
    name: "Joker",
    instructions: "You are good at telling jokes.");

// Retrieve the agent that was just created as an AIAgent using its ID
AIAgent agent1 = await
persistentAgentsClient.GetAIAgentAsync(agentMetadata.Value.Id);
```

Using the Agent Framework helpers

You can also create and return an `AIAgent` in one step:

C#

```
AIAgent agent2 = await persistentAgentsClient.CreateAIAgentAsync(
    model: "gpt-4o-mini",
    name: "Joker",
    instructions: "You are good at telling jokes.");
```

Reusing Azure AI Foundry Agents

You can reuse existing Azure AI Foundry Agents by retrieving them using their IDs.

C#

```
AIAgent agent3 = await persistentAgentsClient.GetAIAgentAsync("<agent-id>");
```

Using the agent

The agent is a standard `AIAgent` and supports all standard `AIAgent` operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

OpenAI ChatCompletion Agents

Azure OpenAI ChatCompletion Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the Azure OpenAI ChatCompletion service.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
dotnet add package Azure.AI.OpenAI
dotnet add package Azure.Identity
```

Creating an Azure OpenAI ChatCompletion Agent

As a first step you need to create a client to connect to the Azure OpenAI service.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AzureOpenAIClient client = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential());
```

Azure OpenAI supports multiple services that all provide model calling capabilities. We need to pick the ChatCompletion service to create a ChatCompletion based agent.

C#

```
var chatCompletionClient = client.GetChatClient("gpt-4o-mini");
```

Finally, create the agent using the `CreateAIAgent` extension method on the `ChatCompletionClient`.

C#

```
AIAgent agent = chatCompletionClient.CreateAIAgent(
    instructions: "You are good at telling jokes.",
    name: "Joker");
// Invoke the agent and output the text result.
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate."));
```

Agent Features

Function Tools

You can provide custom function tools to Azure OpenAI ChatCompletion agents:

C#

```
using System;
using System.ComponentModel;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using OpenAI;

var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";

[Description("Get the weather for a given location.")]
static string GetWeather([Description("The location to get the weather for.")]
string location)
    => $"The weather in {location} is cloudy with a high of 15°C.";

// Create the chat client and agent, and provide the function tool to the agent.
AIAgent agent = new AzureOpenAIClient(
    new Uri(endpoint),
    new AzureCliCredential())
     .GetChatClient(deploymentName)
     .CreateAIAgent(instructions: "You are a helpful assistant", tools:
[AIFunctionFactory.Create(GetWeather)]);

// Non-streaming agent interaction with function tools.
Console.WriteLine(await agent.RunAsync("What is the weather like in Amsterdam?"));
```

Streaming Responses

Get responses as they are generated using streaming:

C#

```
AIAgent agent = chatCompletionClient.CreateAIAgent(
    instructions: "You are good at telling jokes.",
    name: "Joker");
// Invoke the agent with streaming support.
await foreach (var update in agent.RunStreamingAsync("Tell me a joke about a
pirate."))
{
    Console.Write(update);
}
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard `AIAgent` operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

OpenAI Response Agents

Azure OpenAI Responses Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the Azure OpenAI responses service.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
dotnet add package Azure.AI.OpenAI
dotnet add package Azure.Identity
```

Creating an Azure OpenAI Responses Agent

As a first step you need to create a client to connect to the Azure OpenAI service.

C#

```
using System;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.AI;
using OpenAI;

AzureOpenAIClient client = new AzureOpenAIClient(
    new Uri("https://<myresource>.openai.azure.com"),
    new AzureCliCredential());
```

Azure OpenAI supports multiple services that all provide model calling capabilities. We need to pick the Responses service to create a Responses based agent.

C#

```
var responseClient = client.GetOpenAIResponseClient("gpt-4o-mini");
```

Finally, create the agent using the `CreateAIAgent` extension method on the `ResponseClient`.

C#

```
AIAgent agent = responseClient.CreateAIAgent(
    instructions: "You are good at telling jokes.",
    name: "Joker");
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard `AIAgent` operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

OpenAI Chat Completion Agents

OpenAI ChatCompletion Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the OpenAI ChatCompletion service.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

Creating an OpenAI ChatCompletion Agent

As a first step you need to create a client to connect to the OpenAI service.

C#

```
using System;
using Microsoft.Agents.AI;
using OpenAI;

OpenAIClient client = new OpenAIClient("<your_api_key>");
```

OpenAI supports multiple services that all provide model calling capabilities. We need to pick the ChatCompletion service to create a ChatCompletion based agent.

C#

```
var chatCompletionClient = client.GetChatClient("gpt-4o-mini");
```

Finally, create the agent using the `CreateAIAgent` extension method on the `ChatCompletionClient`.

C#

```
AIAgent agent = chatCompletionClient.CreateAIAgent(
    instructions: "You are good at telling jokes.",
    name: "Joker");
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard `AIAgent` operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

Response Agents

OpenAI Responses Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the OpenAI responses service.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

Creating an OpenAI Responses Agent

As a first step you need to create a client to connect to the OpenAI service.

C#

```
using System;
using Microsoft.Agents.AI;
using OpenAI;

OpenAIClient client = new OpenAIClient("<your_api_key>");
```

OpenAI supports multiple services that all provide model calling capabilities. We need to pick the Responses service to create a Responses based agent.

C#

```
var responseClient = client.GetOpenAIResponseClient("gpt-4o-mini");
```

Finally, create the agent using the `CreateAIAgent` extension method on the `ResponseClient`.

C#

```
AIAgent agent = responseClient.CreateAIAgent(
    instructions: "You are good at telling jokes.",
    name: "Joker");
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard `AIAgent` operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

OpenAI Assistant Agents

OpenAI Assistants Agents

10/01/2025

The Microsoft Agent Framework supports creating agents that use the OpenAI Assistants service.

> ⚠ **Warning**
>
> The OpenAI Assistants API is deprecated and will be shut down. For more information see the **OpenAI documentation**.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.OpenAI --prerelease
```

Creating an OpenAI Assistants Agent

As a first step you need to create a client to connect to the OpenAI service.

C#

```
using System;
using Microsoft.Agents.AI;
using OpenAI;

OpenAIClient client = new OpenAIClient("<your_api_key>");
```

OpenAI supports multiple services that all provide model calling capabilities. We need to pick the Assistants service to create an Assistants based agent.

C#

```
var assistantClient = client.GetAssistantClient();
```

To use the OpenAI Assistants service, you need create an assistant resource in the service. This can be done using either the OpenAI SDK or using Microsoft Agent Framework helpers.

Using the OpenAI SDK

Create an assistant and retrieve it as an `AIAgent` using the client.

```C#
// Create a server-side assistant
var createResult = await assistantClient.CreateAssistantAsync(
    "gpt-4o-mini",
    new() { Name = "Joker", Instructions = "You are good at telling jokes." });

// Retrieve the assistant as an AIAgent
AIAgent agent1 = await assistantClient.GetAIAgentAsync(createResult.Value.Id);
```

Using the Agent Framework helpers

You can also create and return an `AIAgent` in one step:

```C#
AIAgent agent2 = await assistantClient.CreateAIAgentAsync(
    model: "gpt-4o-mini",
    name: "Joker",
    instructions: "You are good at telling jokes.");
```

Reusing OpenAI Assistants

You can reuse existing OpenAI Assistants by retrieving them using their IDs.

```C#
AIAgent agent3 = await assistantClient.GetAIAgentAsync("<agent-id>");
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard agent operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

Chat Client Agents

Agent based on any IChatClient

10/01/2025

The Microsoft Agent Framework supports creating agents for any inference service that provides a Microsoft.Extensions.AI.IChatClient implementation. This means that there is a very broad range of services that can be used to create agents, including open source models that can be run locally.

In this document, we will use Ollama as an example.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI --prerelease
```

You will also need to add the package for the specific `IChatClient` implementation you want to use. In this example, we will use OllamaSharp.

PowerShell

```
dotnet add package OllamaSharp
```

Creating a ChatClientAgent

To create an agent based on the `IChatClient` interface, you can use the `ChatClientAgent` class. The `ChatClientAgent` class takes `IChatClient` as a constructor parameter.

First, create an `OllamaApiClient` to access the Ollama service.

C#

```
using System;
using Microsoft.Agents.AI;
using OllamaSharp;

using OllamaApiClient chatClient = new(new Uri("http://localhost:11434"), "phi3");
```

The `OllamaApiClient` implements the `IChatClient` interface, so you can use it to create a `ChatClientAgent`.

C#

```
AIAgent agent = new ChatClientAgent(
    chatClient,
    instructions: "You are good at telling jokes.",
    name: "Joker");
```

> **ⓘ Important**
>
> To ensure that you get the most out of your agent, make sure to choose a service and model that is well-suited for conversational tasks and supports function calling.

Using the Agent

The agent is a standard `AIAgent` and supports all standard agent operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

Agent2Agent

A2A Agents

10/01/2025

The Microsoft Agent Framework supports using a remote agent that is exposed via the A2A protocol in your application using the same `AIAgent` abstraction as any other agent.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.A2A --prerelease
```

Creating an A2A Agent using the well known agent card location

First, let's look at a scenarios where we use the well known agent card location. We pass the root URI of the A2A agent host to the `A2ACardResolver` constructor and the resolver will look for the agent card at `https://your-a2a-agent-host/.well-known/agent-card.json`.

First, create an `A2ACardResolver` with the URI of the remote A2A agent host.

C#

```
using System;
using A2A;
using Microsoft.Agents.AI;
using Microsoft.Agents.AI.A2A;

A2ACardResolver agentCardResolver = new(new Uri("https://your-a2a-agent-host"));
```

Create an instance of the `AIAgent` for the remote A2A agent using the `GetAIAgentAsync` helper method.

C#

```
AIAgent agent = await agentCardResolver.GetAIAgentAsync();
```

Creating an A2A Agent using the Direct Configuration / Private Discovery mechanism

It is also possible to point directly at the agent URL if it's known to us. This can be useful for tightly coupled systems, private agents, or development purposes, where clients are directly configured with Agent Card information and agent URL."

In this case we construct an `A2AClient` directly with the URL of the agent.

```
C#
A2AClient a2aClient = new(new Uri("https://your-a2a-agent-host/echo"));
```

And then we can create an instance of the `AIAgent` using the `GetAIAgent` method.

```
C#
AIAgent agent = a2aClient.GetAIAgent();
```

Using the Agent

The agent is a standard `AIAgent` and supports all standard agent operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

Custom Agent

Custom Agents

10/01/2025

The Microsoft Agent Framework supports building custom agents by inheriting from the `AIAgent` class and implementing the required methods.

This document shows how to build a simple custom agent that parrots back user input in upper case. In most cases building your own agent will involve more complex logic and integration with an AI service.

Getting Started

Add the required NuGet packages to your project.

PowerShell

```
dotnet add package Microsoft.Agents.AI.Abstractions --prerelease
```

Creating a Custom Agent

The Agent Thread

To create a custom agent you also need a thread, which is used to keep track of the state of a single conversation, including message history, and any other state the agent needs to maintain.

To make it easy to get started, you can inherit from various base classes that implement common thread storage mechanisms.

1. `InMemoryAgentThread` - stores the chat history in memory and can be serialized to JSON.
2. `ServiceIdAgentThread` - doesn't store any chat history, but allows you to associate an id with the thread, under which the chat history can be stored externally.

For this example, we will use the `InMemoryAgentThread` as the base class for our custom thread.

C#

```
internal sealed class CustomAgentThread : InMemoryAgentThread
{
    internal CustomAgentThread() : base() { }
    internal CustomAgentThread(JsonElement serializedThreadState,
JsonSerializerOptions? jsonSerializerOptions = null)
```

```
        : base(serializedThreadState, jsonSerializerOptions) { }
}
```

The Agent class

Next, we want to create the agent class itself by inheriting from the `AIAgent` class.

C#

```
internal sealed class UpperCaseParrotAgent : AIAgent
{
}
```

Constructing threads

Threads are always created via two factory methods on the agent class. This allows for the agent to control how threads are created and deserialized. Agents can therefore attach any additional state or behaviors needed to the thread when constructed.

Two methods are required to be implemented:

C#

```
    public override AgentThread GetNewThread() => new CustomAgentThread();

    public override AgentThread DeserializeThread(JsonElement serializedThread,
JsonSerializerOptions? jsonSerializerOptions = null)
        => new CustomAgentThread(serializedThread, jsonSerializerOptions);
```

Core agent logic

The core logic of the agent, is to take any input messages, convert their text to upper case, and return them as response messages.

We want to add the following method to contain this logic. We are cloning the input messages, since various aspects of the input messages have to be modified to be valid response messages. E.g. the role has to be changed to `Assistant`.

C#

```
    private static IEnumerable<ChatMessage>
CloneAndToUpperCase(IEnumerable<ChatMessage> messages, string agentName) =>
messages.Select(x =>
        {
```

```
            var messageClone = x.Clone();
            messageClone.Role = ChatRole.Assistant;
            messageClone.MessageId = Guid.NewGuid().ToString();
            messageClone.AuthorName = agentName;
            messageClone.Contents = x.Contents.Select(c => c is TextContent tc ?
new TextContent(tc.Text.ToUpperInvariant())
            {
                AdditionalProperties = tc.AdditionalProperties,
                Annotations = tc.Annotations,
                RawRepresentation = tc.RawRepresentation
            } : c).ToList();
            return messageClone;
        });
```

Agent run methods

Finally we need to implement the two core methods that are used to run the agent. One for non-streaming and one for streaming.

For both methods, we need to ensure that a thread is provided, and if not we create a new thread. The thread can then be updated with the new messages by calling `NotifyThreadOfNewMessagesAsync`. If we don't do this, the user will not be able to have a multi-turn conversation with the agent and each run will be a fresh interaction.

C#

```
    public override async Task<AgentRunResponse> RunAsync(IEnumerable<ChatMessage>
messages, AgentThread? thread = null, AgentRunOptions? options = null,
CancellationToken cancellationToken = default)
    {
        thread ??= this.GetNewThread();
        List<ChatMessage> responseMessages = CloneAndToUpperCase(messages,
this.DisplayName).ToList();
        await NotifyThreadOfNewMessagesAsync(thread,
messages.Concat(responseMessages), cancellationToken);
        return new AgentRunResponse
        {
            AgentId = this.Id,
            ResponseId = Guid.NewGuid().ToString(),
            Messages = responseMessages
        };
    }

    public override async IAsyncEnumerable<AgentRunResponseUpdate>
RunStreamingAsync(IEnumerable<ChatMessage> messages, AgentThread? thread = null,
AgentRunOptions? options = null, [EnumeratorCancellation] CancellationToken
cancellationToken = default)
    {
        thread ??= this.GetNewThread();
        List<ChatMessage> responseMessages = CloneAndToUpperCase(messages,
```

```
this.DisplayName).ToList();
        await NotifyThreadOfNewMessagesAsync(thread,
messages.Concat(responseMessages), cancellationToken);
        foreach (var message in responseMessages)
        {
            yield return new AgentRunResponseUpdate
            {
                AgentId = this.Id,
                AuthorName = this.DisplayName,
                Role = ChatRole.Assistant,
                Contents = message.Contents,
                ResponseId = Guid.NewGuid().ToString(),
                MessageId = Guid.NewGuid().ToString()
            };
        }
    }
```

Using the Agent

If the `AIAgent` methods are all implemented correctly, the agent would be a standard `AIAgent` and support standard agent operations.

See the Agent getting started tutorials for more information on how to run and interact with agents.

Next steps

Running Agents

Running Agents

10/01/2025

The base Agent abstraction exposes various options for running the agent. Callers can choose to supply zero, one or many input messages. Callers can also choose between streaming and non-streaming. Let's dig into the different usage scenarios.

Streaming and non-streaming

The Microsoft Agent Framework supports both streaming and non-streaming methods for running an agent.

For non-streaming, use the `RunAsync` method.

C#

```
Console.WriteLine(await agent.RunAsync("What is the weather like in Amsterdam?"));
```

For streaming, use the `RunStreamingAsync` method.

C#

```
await foreach (var update in agent.RunStreamingAsync("What is the weather like in Amsterdam?"))
{
    Console.Write(update);
}
```

Agent run options

The base agent abstraction does allow passing an options object for each agent run, however the ability to customize a run at the abstraction level is quite limited. Agents can vary significantly and therefore there aren't really common customization options.

For cases where the caller knows the type of the agent they are working with, it is possible to pass type specific options to allow customizing the run.

For example, here the agent is a `ChatClientAgent` and it is possible to pass a `ChatClientAgentRunOptions` object that inherits from `AgentRunOptions`. This allows the caller to provide custom `ChatOptions` that are merged with any agent level options before being passed to the `IChatClient` that the `ChatClientAgent` is built on.

C#

```
var chatOptions = new ChatOptions() { Tools =
[AIFunctionFactory.Create(GetWeather)] };
Console.WriteLine(await agent.RunAsync("What is the weather like in Amsterdam?",
options: new ChatClientAgentRunOptions(chatOptions)));
```

Response types

Both streaming and non-streaming responses from agents contain all content produced by the agent. Content may include data that is not the result (i.e. the answer to the user question) from the agent. Examples of other data returned include function tool calls, results from function tool calls, reasoning text, status updates, and many more.

Since not all content returned is the result, it's important to look for specific content types when trying to isolate the result from the other content.

To extract the text result from a response, all `TextContent` items from all `ChatMessages` items need to be aggregated. To simplify this, we provide a `Text` property on all response types that aggregates all `TextContent`.

For the non-streaming case, everything is returned in one `AgentRunResponse` object. `AgentRunResponse` allows access to the produced messages via the `Messages` property.

C#

```
var response = await agent.RunAsync("What is the weather like in Amsterdam?");
Console.WriteLine(response.Text);
Console.WriteLine(response.Messages.Count);
```

For the streaming case, `AgentRunResponseUpdate` objects are streamed as they are produced. Each update may contain a part of the result from the agent, and also various other content items. Similar to the non-streaming case, it is possible to use the `Text` property to get the portion of the result contained in the update, and drill into the detail via the `Contents` property.

C#

```
await foreach (var update in agent.RunStreamingAsync("What is the weather like in
Amsterdam?"))
{
    Console.WriteLine(update.Text);
```

```
    Console.WriteLine(update.Contents.Count);
}
```

Message types

Input and output from agents are represented as messages. Messages are subdivided into content items.

The Microsoft Agent Framework uses the message and content types provided by the `Microsoft.Extensions.AI` abstractions. Messages are represented by the `ChatMessage` class and all content classes inherit from the base `AIContent` class.

Various `AIContent` subclasses exist that are used to represent different types of content. Some are provided as part of the base `Microsoft.Extensions.AI` abstractions, but providers can also add their own types, where needed.

Here are some popular types from `Microsoft.Extensions.AI`:

Expand table

Type	Description
TextContent	Textual content that can be both input, e.g. from a user or developer, and output from the agent. Typically contains the text result from an agent.
DataContent	Binary content that can be both input and output. Can be used to pass image, audio or video data to and from the agent (where supported).
UriContent	A url that typically points at hosted content such as an image, audio or video.
FunctionCallContent	A request by an inference service to invoke a function tool.
FunctionResultContent	The result of a function tool invocation.

Next steps

Agent Tools

Agent Tools

10/01/2025

Tooling support may vary considerably between different agent types. Some agents may allow developers to customize the agent at construction time by providing external function tools or by choosing to activate specific built-in tools that are supported by the agent. On the other hand, some custom agents may support no customization via providing external or activating built-in tools, if they already provide defined features that shouldn't be changed.

Therefore, the base abstraction does not provide any direct tooling support, however each agent can choose whether it accepts tooling customization at construction time.

Tooling support with ChatClientAgent

The `ChatClientAgent` is an agent class that can be used to build agentic capabilities on top of any inference service. It comes with support for:

1. Using your own function tools with the agent
2. Using built-in tools that the underlying service may support.

> **Tip**
>
> For more information on `ChatClientAgent` and information on supported services, see Simple agents based on inference services

Provide `AIFunction` instances during agent construction

There are various ways to construct a `ChatClientAgent`, e.g. directly or via factory helper methods on various service clients, but all support passing tools.

C#

```
// Sample function tool.
[Description("Get the weather for a given location.")]
static string GetWeather([Description("The location to get the weather for.")]
string location)
    => $"The weather in {location} is cloudy with a high of 15°C.";

// When calling the ChatClientAgent constructor.
new ChatClientAgent(
    chatClient,
    instructions: "You are a helpful assistant",
    tools: [AIFunctionFactory.Create(GetWeather)]);
```

```
// When using one of the helper factory methods.
openAIResponseClient.CreateAIAgent(
    instructions: "You are a helpful assistant",
    tools: [AIFunctionFactory.Create(GetWeather)]);
```

Provide `AIFunction` instances when running the agent

While the base `AIAgent` abstraction accepts `AgentRunOptions` on its run methods, subclasses of `AIAgent` can accept subclasses of `AgentRunOptions`. This allows specific agent implementations to accept agent specific per-run options.

The underlying `IChatClient` of the `ChatClientAgent` can be customized via the `ChatOptions` class for any invocation. The `ChatClientAgent` can accept a `ChatClientAgentRunOptions` which allows the caller to provide `ChatOptions` for the underlying `IChatClient.GetResponse` method. Where any option clashes with options provided to the agent at construction time, the per run options will take precedence.

Using this mechanism we can provide per-run tools.

C#

```
// Create the chat options class with the per-run tools.
var chatOptions = new ChatOptions()
{
    Tools = [AIFunctionFactory.Create(GetWeather)]
};
// Run the agent, with the per-run chat options.
await agent.RunAsync(
    "What is the weather like in Amsterdam?",
    options: new ChatClientAgentRunOptions(chatOptions));
```

> **Note**
>
> Not all agents support tool calling, so providing tools per run requires providing an agent specific options class.

Using built-in tools

Where the underlying service supports built-in tools, they can be provided using the same mechanisms as described above.

The IChatClient implementation for the underlying service should expose an `AITool` derived class that can be used to configure the built-in tool.

E.g, when creating an Azure AI Foundry Agent, you can provide a `CodeInterpreterToolDefinition` to enable the code interpreter tool that is built into the Azure AI Foundry service.

C#

```
var agent = await azureAgentClient.CreateAIAgentAsync(
    deploymentName,
    instructions: "You are a helpful assistant",
    tools: [new CodeInterpreterToolDefinition()]);
```

Next steps

Multi-turn Conversation

Microsoft Agent Framework Multi-Turn Conversations and Threading

10/01/2025

The Microsoft Agent Framework provides built-in support for managing multi-turn conversations with AI agents. This includes maintaining context across multiple interactions. Different agent types and underlying services that are used to build agents may support different threading types, and the agent framework abstracts these differences away, providing a consistent interface for developers.

For example, when using a ChatClientAgent based on a foundry agent, the conversation history is persisted in the service. While, when using a ChatClientAgent based on chat completion with gpt-4.1 the conversation history is in-memory and managed by the agent.

The differences between the underlying threading models are abstracted away via the `AgentThread` type.

AgentThread Creation

`AgentThread` instances can be created in two ways:

1. By calling `GetNewThread` on the agent.
2. By running the agent and not providing an `AgentThread`. In this case the agent will create a throwaway `AgentThread` with an underlying thread which will only be used for the duration of the run.

Some underlying threads may be persistently created in an underlying service, where the service requires this, e.g. Foundry Agents or OpenAI Responses. Any cleanup or deletion of these threads is the responsibility of the user.

C#

```
// Create a new thread.
AgentThread thread = agent.GetNewThread();
// Run the agent with the thread.
var response = await agent.RunAsync("Hello, how are you?", thread);

// Run an agent with a temporary thread.
response = await agent.RunAsync("Hello, how are you?");
```

AgentThread Storage

`AgentThread` instances can be serialized and stored for later use. This allows for the preservation of conversation context across different sessions or service calls.

For cases where the conversation history is stored in a service, the serialized `AgentThread` will contain an id of the thread in the service. For cases where the conversation history is managed in-memory, the serialized `AgentThread` will contain the messages themselves.

C#

```
// Create a new thread.
AgentThread thread = agent.GetNewThread();
// Run the agent with the thread.
var response = await agent.RunAsync("Hello, how are you?", thread);

// Serialize the thread for storage.
JsonElement serializedThread = await thread.SerializeAsync();
// Deserialize the thread state after loading from storage.
AgentThread resumedThread = await agent.DeserializeThreadAsync(serializedThread);

// Run the agent with the resumed thread.
var response = await agent.RunAsync("Hello, how are you?", resumedThread);
```

The Microsoft Agent Framework provides built-in support for managing multi-turn conversations with AI agents. This includes maintaining context across multiple interactions. Different agent types and underlying services that are used to build agents may support different threading types, and the Agent Framework abstracts these differences away, providing a consistent interface for developers.

For example, when using a `ChatAgent` based on a Foundry agent, the conversation history is persisted in the service. While when using a `ChatAgent` based on chat completion with gpt-4, the conversation history is in-memory and managed by the agent.

The differences between the underlying threading models are abstracted away via the `AgentThread` type.

Agent/AgentThread relationship

`AIAgent` instances are stateless and the same agent instance can be used with multiple `AgentThread` instances.

Not all agents support all thread types though. For example if you are using a `ChatClientAgent` with the responses service, `AgentThread` instances created by this agent, will not work with a `ChatClientAgent` using the Foundry Agent service. This is because these services both support

saving the conversation history in the service, and the `AgentThread` only has a reference to this service managed thread.

It is therefore considered unsafe to use an `AgentThread` instance that was created by one agent with a different agent instance, unless you are aware of the underlying threading model and its implications.

Threading support by service / protocol

Expand table

Service	Threading Support
Foundry Agents	Service managed persistent threads
OpenAI Responses	Service managed persistent threads OR in-memory threads
OpenAI ChatCompletion	In-memory threads
OpenAI Assistants	Service managed threads
A2A	Service managed threads

Next steps

Agent Middleware

Agent Middleware

10/01/2025

Middleware in the Agent Framework provides a powerful way to intercept, modify, and enhance agent interactions at various stages of execution. You can use middleware to implement cross-cutting concerns such as logging, security validation, error handling, and result transformation without modifying your core agent or function logic.

The Agent Framework can be customized using three different types of middleware:

1. Agent Run middleware: Allows interception of all agent runs, so that input and output can be inspected and/or modified as needed.
2. Function calling middleware: Allows interception of all function calls executed by the agent, so that input and output can be inspected and modified as needed.
3. `IChatClient` middleware: Allows interception of calls to an `IChatClient` implementation, where an agent is using `IChatClient` for inference calls, e.g. when using `ChatClientAgent`.

All the types of middleware are implemented via a function callback, and when multiple middleware instances of the same type are registered, they form a chain, where each middleware instance is expected to call the next in the chain, via a provided `next` `Func`.

Agent run and function calling middleware types can be registered on an agent, by using the agent builder with an existing agent object.

C#

```
var middlewareEnabledAgent = originalAgent
    .AsBuilder()
        .Use(CustomAgentRunMiddleware)
        .Use(CustomFunctionCallingMiddleware)
    .Build();
```

`IChatClient` middleware can be registered on an `IChatClient` before it is used with a `ChatClientAgent`, by using the chat client builder pattern.

C#

```
var chatClient = new AzureOpenAIClient(new
Uri("https://<myresource>.openai.azure.com"), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient();

var middlewareEnabledChatClient = chatClient
    .AsBuilder()
        .Use(getResponseFunc: CustomChatClientMiddleware,
```

```
getStreamingResponseFunc: null)
    .Build();

var agent = new ChatClientAgent(middlewareEnabledChatClient, instructions: "You
are a helpful assistant.");
```

`IChatClient` middleware can also be registered using a factory method when constructing an agent via one of the helper methods on SDK clients.

C#

```
var agent = new AzureOpenAIClient(new Uri(endpoint), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .CreateAIAgent("You are a helpful assistant.", clientFactory: (chatClient) =>
chatClient
        .AsBuilder()
            .Use(getResponseFunc: CustomChatClientMiddleware,
getStreamingResponseFunc: null)
        .Build());
```

Agent Run Middleware

Here is an example of agent run middleware, that can inspect and/or modify the input and output from the agent run.

C#

```
async Task<AgentRunResponse> CustomAgentRunMiddleware(
    IEnumerable<ChatMessage> messages,
    AgentThread? thread,
    AgentRunOptions? options,
    AIAgent innerAgent,
    CancellationToken cancellationToken)
{
    Console.WriteLine(messages.Count());
    var response = await innerAgent.RunAsync(messages, thread, options,
cancellationToken).ConfigureAwait(false);
    Console.WriteLine(response.Messages.Count);
    return response;
}
```

Function calling middleware

> ⓘ Note

Function calling middleware is currently only supported with an `AIAgent` that uses `Microsoft.Extensions.AI.FunctionInvokingChatClient`, e.g. `ChatClientAgent`.

Here is an example of function calling middleware, that can inspect and/or modify the function being called, and the result from the function call.

C#

```
async ValueTask<object?> CustomFunctionCallingMiddleware(
    AIAgent agent,
    FunctionInvocationContext context,
    Func<FunctionInvocationContext, CancellationToken, ValueTask<object?>> next,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"Function Name: {context!.Function.Name}");
    var result = await next(context, cancellationToken);
    Console.WriteLine($"Function Call Result: {result}");

    return result;
}
```

It is possible to terminate the function call loop with function calling middleware by setting the provided `FunctionInvocationContext.Terminate` to true. This will prevent the function calling loop from issuing a request to the inference service containing the function call results after function invocation. If there were more than one function available for invocation during this iteration, it may also prevent any remaining functions from being executed.

> ⚠ **Warning**
>
> Terminating the function call loop may result in your thread being left in an inconsistent state, e.g. containing function call content with no function result content. This may result in the thread being unusable for further runs.

IChatClient middleware

Here is an example of chat client middleware, that can inspect and/or modify the input and output for the request to the inference service that the chat client provides.

C#

```
async Task<ChatResponse> CustomChatClientMiddleware(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    IChatClient innerChatClient,
```

```
    CancellationToken cancellationToken)
{
    Console.WriteLine(messages.Count());
    var response = await innerChatClient.GetResponseAsync(messages, options,
cancellationToken);
    Console.WriteLine(response.Messages.Count);

    return response;
}
```

> [!NOTE]
> For more information about `IChatClient` middleware, see Custom IChatClient middleware in the Microsoft.Extensions.AI documentation.

Next steps

Agent Memory

Agent Memory

10/01/2025

Agent memory is a crucial capability that allows agents to maintain context across conversations, remember user preferences, and provide personalized experiences. The Agent Framework provides multiple memory mechanisms to suit different use cases, from simple in-memory storage to persistent databases and specialized memory services.

Memory Types

The Agent Framework supports several types of memory to accommodate different use cases, including managing chat history as part of short term memory and providing extension points for extracting, storing and injecting long term memories into agents.

Chat History (short term memory)

Various chat history storage options are supported by the Agent Framework. The available options vary by agent type and the underlying service(s) used to build the agent.

E.g. where an agent is built using a service that only supports storage of chat history in the service, the Agent Framework must respect what the service requires.

In-memory chat history storage

When using a service that does not support in-service storage of chat history, the Agent Framework will store chat history in-memory in the `AgentThread` object by default. In this case the full chat history stored in the thread object, plus any new messages, will be provided to the underlying service on each agent run.

E.g, when using OpenAI Chat Completion as the underlying service for agents, the following code will result in the thread object containing the chat history from the agent run.

C#

```
AIAgent agent = new OpenAIClient("<your_api_key>")
     .GetChatClient(modelName)
     .CreateAIAgent(JokerInstructions, JokerName);
AgentThread thread = agent.GetNewThread();
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate.", thread));
```

Where messages are stored in memory, it is possible to retrieve the list of messages from the thread and manipulate the mesages directly if required.

C#

```
IList<ChatMessage>? messages = thread.GetService<IList<ChatMessage>>();
```

> **Note**
>
> Retrieving messages from the `AgentThread` object in this way will only work if in-memory storage is being used.

Inference service chat history storage

When using a service that requires in-service storage of chat history, the Agent Framework will storage the id of the remote chat history in the `AgentThread` object.

E.g, when using OpenAI Responses with store=true as the underlying service for agents, the following code will result in the thread object containing the last response id returned by the service.

C#

```
AIAgent agent = new OpenAIClient("<your_api_key>")
     .GetOpenAIResponseClient(modelName)
     .CreateAIAgent(JokerInstructions, JokerName);
AgentThread thread = agent.GetNewThread();
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate.", thread));
```

> **Note**
>
> Some services, e.g. OpenAI Responses support either in-service storage of chat history (store=true), or providing the full chat history on each invocation (store=false). Therefore, depending on the mode that the service is used in, the Agent Framework will either default to storing the full chat history in memory, or storing an id reference to the service stored chat history.

3rd party chat history storage

When using a service that does not support in-service storage of chat history, the Agent Framework allows developers to replace the default in-memory storage of chat history with 3rd

party chat history storage. The developer is required to provide a subclass of the base abstract `ChatMessageStore` class.

The `ChatMessageStore` class defines the interface for storing and retrieving chat messages. Developers must implement the `AddMessagesAsync` and `GetMessagesAsync` methods to add messages to the remote store as they are generated, and retrieve messages from the remote store before invoking the underlying service.

The agent will use all messages returned by `GetMessagesAsync` when processing a user query. It is up to the implementer of `ChatMessageStore` to ensure that the size of the chat history does not exceed the context window of the underlying service.

When implementing a custom `ChatMessageStore` which stores chat history in a remote store, the chat history for that thread should be stored under a key that is unique to that thread. The `ChatMessageStore` implementation should generate this key and keep it in its state. `ChatMessageStore` has a `Serialize` method that can be overridden to serialize its state when the thread is serialized. The `ChatMessageStore` should also provide a constructor that takes a `JsonElement` as input to support deserialization of its state.

To supply a custom `ChatMessageStore` to a `ChatClientAgent`, you can use the `ChatMessageStoreFactory` option when creating the agent. Here is an example showing how to pass the custom implementation of `ChatMessageStore` to a `ChatClientAgent` that is based on Azure OpenAI Chat Completion.

C#

```
AIAgent agent = new AzureOpenAIClient(
    new Uri(endpoint),
    new AzureCliCredential())
     .GetChatClient(deploymentName)
     .CreateAIAgent(new ChatClientAgentOptions
     {
         Name = JokerName,
         Instructions = JokerInstructions,
         ChatMessageStoreFactory = ctx =>
         {
             // Create a new chat message store for this agent that stores the
messages in a custom store.
             // Each thread must get its own copy of the CustomMessageStore, since
the store
             // also contains the id that the thread is stored under.
             return new CustomMessageStore(vectorStore, ctx.SerializedState,
ctx.JsonSerializerOptions);
         }
     });
```

> **Tip**
>
> For a detailed example on how to create a custom message store, see the **Storing Chat History in 3rd Party Storage** tutorial.

Long term memory

The Agent Framework allows developers to provide custom components that can extract memories or provide memories to an agent.

To implement such a memory component, the developer needs to subclass the `AIContextProvider` abstract base class. This class has two core methods, `InvokingAsync` and `InvokedAsync`. When overridden, `InvokedAsync` allows developers to inspect all messages provided by users or generated by the agent. `InvokingAsync` allows developers to inject additional context for a specific agent run. System instructions, additional messages and additional functions can be provided.

> **Tip**
>
> For a detailed example on how to create a custom memory component, see the **Adding Memory to an Agent** tutorial.

AgentThread Serialization

It is important to be able to persist an `AgentThread` object between agent invocations. This allows for situations where a user may ask a question of the agent, and take a long time to ask follow up questions. This allows the `AgentThread` state to survive service or app restarts.

Even if the chat history is stored in a remote store, the `AgentThread` object still contains an id referencing the remote chat history. Losing the `AgentThread` state will therefore result in also losing the id of the remote chat history.

The `AgentThread` as well as any objects attached to it, all therefore provide the `SerializeAsync` method to serialize their state. The `AIAgent` also provides a `DeserializeThread` method that re-creates a thread from the serialized state. The `DeserializeThread` method re-creates the thread with the `ChatMessageStore` and `AIContextProvider` configured on the agent.

C#

```
// Serialize the thread state to a JsonElement, so it can be stored for later use.
JsonElement serializedThreadState = thread.Serialize();

// Re-create the thread from the JsonElement.
AgentThread resumedThread = AIAgent.DeserializeThread(serializedThreadState);
```

> ⚠ **Warning**
>
> Deserializing a thread with a different agent than that which originally created it, or with an agent that has a different configuration than the original agent, may result in errors or unexpected behavior.

Next steps

Agent Observability

Agent Observability

10/01/2025

Observability is a key aspect of building reliable and maintainable systems. Agent Framework provides built-in support for observability, allowing you to monitor the behavior of your agents.

This guide will walk you through the steps to enable observability with Agent Framework to help you understand how your agents are performing and diagnose any issues that may arise.

OpenTelemetry Integration

Agent Framework integrates with OpenTelemetry, and more specifically Agent Framework emits traces, logs, and metrics according to the OpenTelemetry GenAI Semantic Conventions.

Enable Observability

To enable observability for your chat client, you need to build the chat client as follows:

C#

```
// Using the Azure OpenAI client as an example
var instrumentedChatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient() // Converts a native OpenAI SDK ChatClient into a
Microsoft.Extensions.AI.IChatClient
    .AsBuilder()
    .UseOpenTelemetry(sourceName: "MyApplication", configure: (cfg) =>
cfg.EnableSensitiveData = true)    // Enable OpenTelemetry instrumentation with
sensitive data
    .Build();
```

To enable observability for your agent, you need to build the agent as follows:

C#

```
var agent = new ChatClientAgent(
    instrumentedChatClient,
    name: "OpenTelemetryDemoAgent",
    instructions: "You are a helpful assistant that provides concise and
informative responses.",
    tools: [AIFunctionFactory.Create(GetWeatherAsync)]
```

```
).WithOpenTelemetry(sourceName: "MyApplication", enableSensitiveData: true);    //
Enable OpenTelemetry instrumentation with sensitive data
```

> **Important**
>
> When you enable observability for your chat clients and agents, you may see duplicated information, especially when sensitive data is enabled. The chat context (including prompts and responses) that is captured by both the chat client and the agent will be included in both spans. Depending on your needs, you may choose to enable observability only on the chat client or only on the agent to avoid duplication. See the GenAI Semantic Conventions for more details on the attributes captured for LLM and Agents.

> **Note**
>
> Only enable sensitive data in development or testing environments, as it may expose user information in production logs and traces. Sensitive data includes prompts, responses, function call arguments, and results.

Configuration

Now that your chat client and agent are instrumented, you can configure the OpenTelemetry exporters to send the telemetry data to your desired backend.

Traces

To export traces to the desired backend, you can configure the OpenTelemetry SDK in your application startup code. For example, to export traces to an Azure Monitor resource:

C#

```
using Azure.Monitor.OpenTelemetry.Exporter;
using OpenTelemetry;
using OpenTelemetry.Trace;
using OpenTelemetry.Resources;
using System;

var SourceName = "MyApplication";

var applicationInsightsConnectionString =
Environment.GetEnvironmentVariable("APPLICATION_INSIGHTS_CONNECTION_STRING")
    ?? throw new InvalidOperationException("APPLICATION_INSIGHTS_CONNECTION_STRING
is not set.");
```

```
var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(ServiceName);

using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddSource(SourceName)
    .AddSource("*Microsoft.Extensions.AI") // Listen to the
Experimental.Microsoft.Extensions.AI source for chat client telemetry
    .AddSource("*Microsoft.Extensions.Agents*") // Listen to the
Experimental.Microsoft.Extensions.Agents source for agent telemetry
    .AddAzureMonitorTraceExporter(options => options.ConnectionString =
applicationInsightsConnectionString)
    .Build();
```

> **Tip**
>
> Depending on your backend, you can use different exporters, see the OpenTelemetry .NET documentation for more information. For local development, consider using the Aspire Dashboard.

Metrics

Similarly, to export metrics to the desired backend, you can configure the OpenTelemetry SDK in your application startup code. For example, to export metrics to an Azure Monitor resource:

C#

```
using Azure.Monitor.OpenTelemetry.Exporter;
using OpenTelemetry;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using System;

var applicationInsightsConnectionString =
Environment.GetEnvironmentVariable("APPLICATION_INSIGHTS_CONNECTION_STRING")
    ?? throw new InvalidOperationException("APPLICATION_INSIGHTS_CONNECTION_STRING
is not set.");

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService(ServiceName);

using var meterProvider = Sdk.CreateMeterProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddSource(SourceName)
    .AddMeter("*Microsoft.Agents.AI") // Agent Framework metrics
    .AddAzureMonitorMetricExporter(options => options.ConnectionString =
```

```
applicationInsightsConnectionString)
    .Build();
```

Logs

Logs are captured via the logging framework you are using, for example `Microsoft.Extensions.Logging`. To export logs to an Azure Monitor resource, you can configure the logging provider in your application startup code:

C#

```
using Azure.Monitor.OpenTelemetry.Exporter;
using Microsoft.Extensions.Logging;

var applicationInsightsConnectionString =
Environment.GetEnvironmentVariable("APPLICATION_INSIGHTS_CONNECTION_STRING")
    ?? throw new InvalidOperationException("APPLICATION_INSIGHTS_CONNECTION_STRING
is not set.");

using var loggerFactory = LoggerFactory.Create(builder =>
{
    // Add OpenTelemetry as a logging provider
    builder.AddOpenTelemetry(options =>
    {
        options.SetResourceBuilder(resourceBuilder);
        options.AddAzureMonitorLogExporter(options => options.ConnectionString =
applicationInsightsConnectionString);
        // Format log messages. This is default to false.
        options.IncludeFormattedMessage = true;
        options.IncludeScopes = true;
    })
    .SetMinimumLevel(LogLevel.Debug);
});

// Create a logger instance for your application
var logger = loggerFactory.CreateLogger<Program>();
```

Aspire Dashboard

Consider using the Aspire Dashboard as a quick way to visualize your traces and metrics during development. To Learn more, see Aspire Dashboard documentation. The Aspire Dashboard receives data via an OpenTelemetry Collector, which you can add to your tracer provider as follows:

C#

```
using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddSource(SourceName)
    .AddSource("*Microsoft.Extensions.AI") // Listen to the
Experimental.Microsoft.Extensions.AI source for chat client telemetry
    .AddSource("*Microsoft.Extensions.Agents*") // Listen to the
Experimental.Microsoft.Extensions.Agents source for agent telemetry
    .AddOtlpExporter(options => options.Endpoint = new
Uri("http://localhost:4317"))
    .Build();
```

Getting started

See a full example of an agent with OpenTelemetry enabled in the Agent Framework repository.

Next steps

Using MCP Tools

Model Context Protocol

09/25/2025

Model Context Protocol is an open standard that defines how applications provide tools and contextual data to large language models (LLMs). It enables consistent, scalable integration of external tools into model workflows.

You can extend the capabilities of your Agent Framework agents by connecting it to tools hosted on remote Model Context Protocol (MCP) servers.

Considerations for using third party Model Context Protocol servers

Your use of Model Context Protcol servers is subject to the terms between you and the service provider. When you connect to a non-Microsoft service, some of your data (such as prompt content) is passed to the non-Microsoft service, or your application might receive data from the non-Microsoft service. You're responsible for your use of non-Microsoft services and data, along with any charges associated with that use.

The remote MCP servers that you decide to use with the MCP tool described in this article were created by third parties, not Microsoft. Microsoft hasn't tested or verified these servers. Microsoft has no responsibility to you or others in relation to your use of any remote MCP servers.

We recommend that you carefully review and track what MCP servers you add to your Agent Framework based applications. We also recommend that you rely on servers hosted by trusted service providers themselves rather than proxies.

The MCP tool allows you to pass custom headers, such as authentication keys or schemas, that a remote MCP server might need. We recommend that you review all data that's shared with remote MCP servers and that you log the data for auditing purposes. Be cognizant of non-Microsoft practices for retention and location of data.

How it works

You can integrate multiple remote MCP servers by adding them as tools to your agent. Agent Framework makes it easy to convert an MCP tool to an AI tool that can be called by your agent.

The MCP tool supports custom headers, so you can connect to MCP servers by using the authentication schemas that they require or by passing other headers that the MCP servers require. **TODO You can specify headers only by including them in tool_resources at each run.**

In this way, you can put API keys, OAuth access tokens, or other credentials directly in your request. TODO

The most commonly used header is the authorization header. Headers that you pass in are available only for the current run and aren't persisted.

For more information on using MCP, see:

- Security Best Practices on the Model Context Protocol website.
- Understanding and mitigating security risks in MCP implementations in the Microsoft Security Community Blog.

Next steps

Using MCP tools with Agents | Using MCP tools with Foundry Agents

Using MCP tools with Agents

10/01/2025

The Microsoft Agent Framework supports integration with Model Context Protocol (MCP) servers, allowing your agents to access external tools and services. This guide shows how to connect to an MCP server and use its tools within your agent.

The .Net version of Agent Framework can be used together with the official MCP C# SDK to allow your agent to call MCP tools.

The following sample shows how to:

1. Set up and MCP server
2. Retrieve the list of available tools from the MCP Server
3. Convert the MCP tools to `AIFunction`'s so they can be added to an agent
4. Invoke the tools from an agent using function calling

Setting Up an MCP Client

First, create an MCP client that connects to your desired MCP server:

C#

```
// Create an MCPClient for the GitHub server
await using var mcpClient = await McpClientFactory.CreateAsync(new
StdioClientTransport(new()
{
    Name = "MCPServer",
    Command = "npx",
    Arguments = ["-y", "--verbose", "@modelcontextprotocol/server-github"],
}));
```

In this example:

- **Name**: A friendly name for your MCP server connection
- **Command**: The executable to run the MCP server (here using npx to run a Node.js package)
- **Arguments**: Command-line arguments passed to the MCP server

Retrieving Available Tools

Once connected, retrieve the list of tools available from the MCP server:

C#

```
// Retrieve the list of tools available on the GitHub server
var mcpTools = await mcpClient.ListToolsAsync().ConfigureAwait(false);
```

The `ListToolsAsync()` method returns a collection of tools that the MCP server exposes. These tools are automatically converted to AITool objects that can be used by your agent.

Creating an Agent with MCP Tools

Create your agent and provide the MCP tools during initialization:

C#

```
AIAgent agent = new AzureOpenAIClient(
    new Uri(endpoint),
    new AzureCliCredential())
     .GetChatClient(deploymentName)
     .CreateAIAgent(
         instructions: "You answer questions related to GitHub repositories
only.",
         tools: [.. mcpTools.Cast<AITool>()]);
```

Key points:

- **Instructions**: Provide clear instructions that align with the capabilities of your MCP tools
- **Tools**: Cast the MCP tools to `AITool` objects and spread them into the tools array
- The agent will automatically have access to all tools provided by the MCP server

Using the Agent

Once configured, your agent can automatically use the MCP tools to fulfill user requests:

C#

```
// Invoke the agent and output the text result
Console.WriteLine(await agent.RunAsync("Summarize the last four commits to the
microsoft/semantic-kernel repository?"));
```

The agent will:

1. Analyze the user's request
2. Determine which MCP tools are needed
3. Call the appropriate tools through the MCP server
4. Synthesize the results into a coherent response

Environment Configuration

Make sure to set up the required environment variables:

```C#
var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
    throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
```

Resource Management

Always properly dispose of MCP client resources:

```C#
await using var mcpClient = await McpClientFactory.CreateAsync(...);
```

Using `await using` ensures the MCP client connection is properly closed when it goes out of scope.

Common MCP Servers

Popular MCP servers include:

- `@modelcontextprotocol/server-github`: Access GitHub repositories and data
- `@modelcontextprotocol/server-filesystem`: File system operations
- `@modelcontextprotocol/server-sqlite`: SQLite database access

Each server provides different tools and capabilities that extend your agent's functionality. This integration allows your agents to seamlessly access external data and services while maintaining the security and standardization benefits of the Model Context Protocol.

The full source code and instructions to run this sample is available here.

Next steps

Using workflows as Agents

Using MCP tools with Foundry Agents

10/01/2025

You can extend the capabilities of your Azure AI Foundry agent by connecting it to tools hosted on remote Model Context Protocol (MCP) servers (bring your own MCP server endpoint).

How to use the Model Context Protocol tool

This section explains how to create an AI agent using Azure Foundry (Azure AI) with a hosted Model Context Protocol (MCP) server integration. The agent can utilize MCP tools that are managed and executed by the Azure Foundry service, allowing for secure and controlled access to external resources.

Key Features

- **Hosted MCP Server**: The MCP server is hosted and managed by Azure AI Foundry, eliminating the need to manage server infrastructure
- **Persistent Agents**: Agents are created and stored server-side, allowing for stateful conversations
- **Tool Approval Workflow**: Configurable approval mechanisms for MCP tool invocations

How It Works

1. Environment Setup

The sample requires two environment variables:

- `AZURE_FOUNDRY_PROJECT_ENDPOINT`: Your Azure AI Foundry project endpoint URL
- `AZURE_FOUNDRY_PROJECT_MODEL_ID`: The model deployment name (defaults to "gpt-4.1-mini")

C#

```
var endpoint =
Environment.GetEnvironmentVariable("AZURE_FOUNDRY_PROJECT_ENDPOINT")
    ?? throw new InvalidOperationException("AZURE_FOUNDRY_PROJECT_ENDPOINT is not
set.");
var model = Environment.GetEnvironmentVariable("AZURE_FOUNDRY_PROJECT_MODEL_ID")
?? "gpt-4.1-mini";
```

2. Agent Configuration

The agent is configured with specific instructions and metadata:

```
C#

const string AgentName = "MicrosoftLearnAgent";
const string AgentInstructions = "You answer questions by searching the Microsoft
Learn content only.";
```

This creates an agent specialized for answering questions using Microsoft Learn documentation.

3. MCP Tool Definition

The sample creates an MCP tool definition that points to a hosted MCP server:

```
C#

var mcpTool = new MCPToolDefinition(
    serverLabel: "microsoft_learn",
    serverUrl: "https://learn.microsoft.com/api/mcp");
mcpTool.AllowedTools.Add("microsoft_docs_search");
```

Key Components:

- **serverLabel**: A unique identifier for the MCP server instance
- **serverUrl**: The URL of the hosted MCP server
- **AllowedTools**: Specifies which tools from the MCP server the agent can use

4. Persistent Agent Creation

The agent is created server-side using the Azure AI Foundry Persistent Agents SDK:

```
C#

var persistentAgentsClient = new PersistentAgentsClient(endpoint, new
AzureCliCredential());

var agentMetadata = await persistentAgentsClient.Administration.CreateAgentAsync(
    model: model,
    name: AgentName,
    instructions: AgentInstructions,
    tools: [mcpTool]);
```

This creates a persistent agent that:

- Lives on the Azure AI Foundry service
- Has access to the specified MCP tools
- Can maintain conversation state across multiple interactions

5. Agent Retrieval and Execution

The created agent is retrieved as an `AIAgent` instance:

```
C#

AIAgent agent = await
persistentAgentsClient.GetAIAgentAsync(agentMetadata.Value.Id);
```

6. Tool Resource Configuration

The sample configures tool resources with approval settings:

```
C#

var runOptions = new ChatClientAgentRunOptions()
{
    ChatOptions = new()
    {
        RawRepresentationFactory = (_) => new ThreadAndRunOptions()
        {
            ToolResources = new MCPToolResource(serverLabel: "microsoft_learn")
            {
                RequireApproval = new MCPApproval("never"),
            }.ToToolResources()
        }
    }
};
```

Key Configuration:

- **MCPToolResource**: Links the MCP server instance to the agent execution
- **RequireApproval**: Controls when user approval is needed for tool invocations
  - `"never"`: Tools execute automatically without approval
  - `"always"`: All tool invocations require user approval
  - Custom approval rules can also be configured

7. Agent Execution

The agent is invoked with a question and executes using the configured MCP tools:

```C#
AgentThread thread = agent.GetNewThread();
var response = await agent.RunAsync(
    "Please summarize the Azure AI Agent documentation related to MCP Tool calling?",
    thread,
    runOptions);
Console.WriteLine(response);
```

8. Cleanup

The sample demonstrates proper resource cleanup:

```C#
await persistentAgentsClient.Administration.DeleteAgentAsync(agent.Id);
```

Next steps

Using workflows as Agents

Microsoft Agent Framework Workflows

09/25/2025

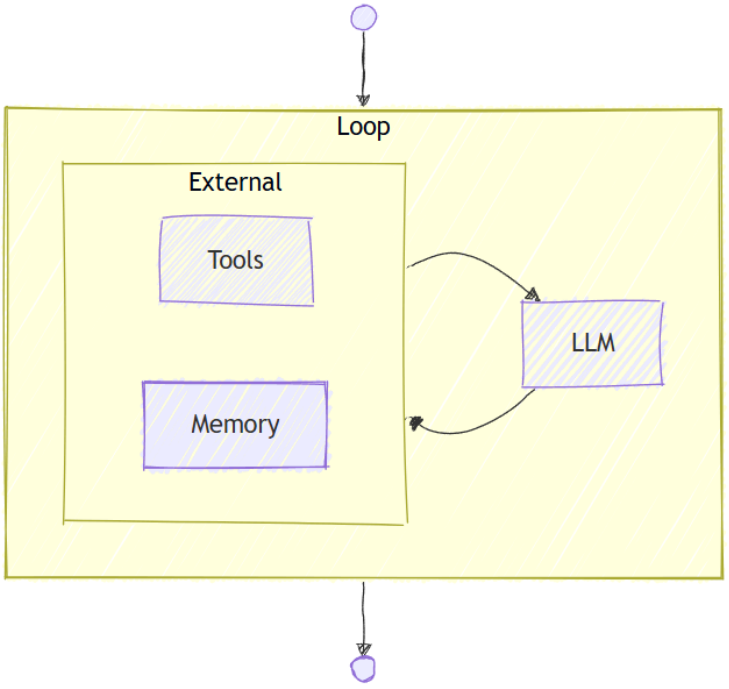
Overview

Microsoft Agent Framework Workflows empowers you to build intelligent automation systems that seamlessly blend AI agents with business processes. With its type-safe architecture and intuitive design, you can orchestrate complex workflows without getting bogged down in infrastructure complexity, allowing you to focus on your core business logic.

How is a Workflows different from an AI Agent?

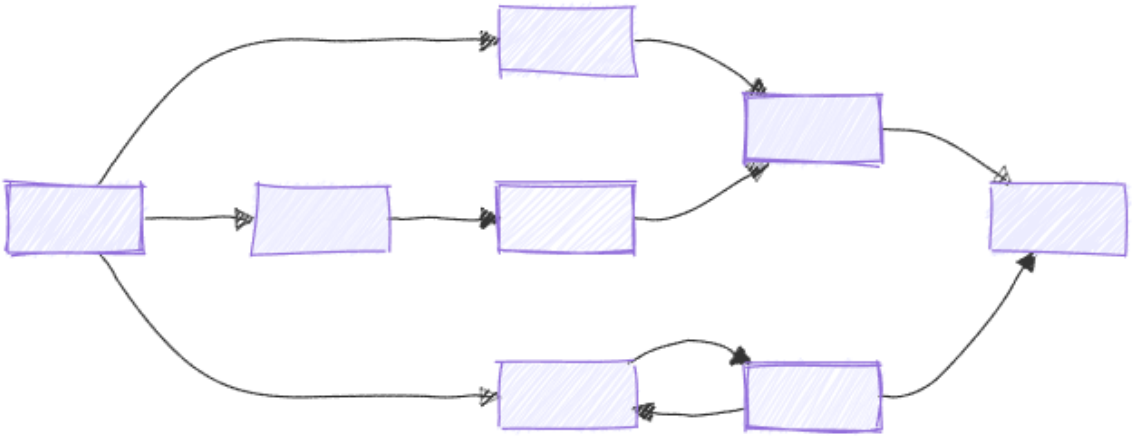
While an AI agent and a workflow can involve multiple steps to achieve a goal, they serve different purposes and operate at different levels of abstraction:

- **AI Agent**: An AI agent is typically driven by a large language model (LLM) and it has access to various tools to help it accomplish tasks. The steps an agent takes are dynamic and determined by the LLM based on the context of the conversation and the tools



  available.
- **Workflow**: A workflow, on the other hand, is a predefined sequence of operations that can include AI agents as components. Workflows are designed to handle complex business processes that may involve multiple agents, human interactions, and integrations with external systems. The flow of a workflow is explicitly defined, allowing

for more control over the execution path.



Key Features

- **Type Safety**: Strong typing ensures messages flow correctly between components, with comprehensive validation that prevents runtime errors.
- **Flexible Control Flow**: Graph-based architecture allows for intuitive modeling of complex workflows with `executors` and `edges`. Conditional routing, parallel processing, and dynamic execution paths are all supported.
- **External Integration**: Built-in request/response patterns for seamless integration with external APIs, and human-in-the-loop scenarios.
- **Checkpointing**: Save workflow states via checkpoints, enabling recovery and resumption of long-running processes on server sides.
- **Multi-Agent Orchestration**: Built-in patterns for coordinating multiple AI agents, including sequential, concurrent, hand-off, and magentic.

Core Concepts

- **Executors**: represent individual processing units within a workflow. They can be AI agents or custom logic components. They receive input messages, perform specific tasks, and produce output messages.
- **Edges**: define the connections between executors, determining the flow of messages. They can include conditions to control routing based on message contents.
- **Workflows**: are directed graphs composed of executors and edges. They define the overall process, starting from an initial executor and proceeding through various paths based on conditions and logic defined in the edges.

Getting Started

Begin your journey with Microsoft Agent Framework Workflows by exploring our getting started samples:

- C# Getting Started Sample
- Python Getting Started Sample

Next Steps

Dive deeper into the concepts and capabilities of Microsoft Agent Framework Workflows by continuing to the Workflows Concepts page.

Microsoft Agent Framework Workflows Core Concepts

09/25/2025

This page provides an overview of the core concepts and architecture of the Microsoft Agent Framework Workflow system. It covers the fundamental building blocks, execution model, and key features that enable developers to create robust, type-safe workflows.

Core Components

The workflow framework consists of four core layers that work together to create a flexible, type-safe execution environment:

- **Executors** and **Edges** form a directed graph representing the workflow structure
- **Workflows** orchestrate executor execution, message routing, and event streaming
- **Events** provide observability into the workflow execution

Next Steps

To dive deeper into each core component, explore the following sections:

- Executors
- Edges
- Workflows
- Events

Microsoft Agent Framework Workflows Core Concepts - Executors

09/26/2025

This document provides an in-depth look at the **Executors** component of the Microsoft Agent Framework Workflow system.

Overview

Executors are the fundamental building blocks that process messages in a workflow. They are autonomous processing units that receive typed messages, perform operations, and can produce output messages or events.

Executors implement the `IMessageHandler<TInput>` or `IMessageHandler<TInput, TOutput>` interfaces and inherit from the `ReflectingExecutor<T>` base class. Each executor has a unique identifier and can handle specific message types.

Basic Executor Structure

C#

```
using Microsoft.Agents.Workflows;
using Microsoft.Agents.Workflows.Reflection;

internal sealed class UppercaseExecutor() : ReflectingExecutor<UppercaseExecutor>
("UppercaseExecutor"),
    IMessageHandler<string, string>
{
    public async ValueTask<string> HandleAsync(string message, IWorkflowContext
context)
    {
        string result = message.ToUpperInvariant();
        return result; // Return value is automatically sent to connected
executors
    }
}
```

It is possible to send messages manually without returning a value:

C#

```
internal sealed class UppercaseExecutor() : ReflectingExecutor<UppercaseExecutor>
("UppercaseExecutor"),
    IMessageHandler<string>
```

```
{
    public async ValueTask HandleAsync(string message, IWorkflowContext context)
    {
        string result = message.ToUpperInvariant();
        await context.SendMessageAsync(result); // Manually send messages to
connected executors
    }
}
```

It is also possible to handle multiple input types by implementing multiple interfaces:

C#

```
internal sealed class SampleExecutor() : ReflectingExecutor<SampleExecutor>
("SampleExecutor"),
    IMessageHandler<string, string>, IMessageHandler<int, int>
{
    /// <summary>
    /// Converts input string to uppercase
    /// </summary>
    public async ValueTask<string> HandleAsync(string message, IWorkflowContext
context)
    {
        string result = message.ToUpperInvariant();
        return result;
    }

    /// <summary>
    /// Doubles the input integer
    /// </summary>
    public async ValueTask<int> HandleAsync(int message, IWorkflowContext context)
    {
        int result = message * 2;
        return result;
    }
}
```

Next Step

- Learn about Edges to understand how executors are connected in a workflow.

Microsoft Agent Framework Workflows Core Concepts - Edges

09/25/2025

This document provides an in-depth look at the **Edges** component of the Microsoft Agent Framework Workflow system.

Overview

Edges define how messages flow between executors with optional conditions. They represent the connections in the workflow graph and determine the data flow paths.

Types of Edges

The framework supports several edge patterns:

1. **Direct Edges**: Simple one-to-one connections between executors
2. **Conditional Edges**: Edges with conditions that determine when messages should flow
3. **Fan-out Edges**: One executor sending messages to multiple targets
4. **Fan-in Edges**: Multiple executors sending messages to a single target

Direct Edges

The simplest form of connection between two executors:

```
using Microsoft.Agents.Workflows;

WorkflowBuilder builder = new(sourceExecutor);
builder.AddEdge(sourceExecutor, targetExecutor);
```

Conditional Edges

Edges that only activate when certain conditions are met:

```
// Route based on message content
builder.AddEdge(
    source: spamDetector,
    target: emailProcessor,
    condition: result => result is SpamResult spam && !spam.IsSpam
);
```

```
builder.AddEdge(
    source: spamDetector,
    target: spamHandler,
    condition: result => result is SpamResult spam && spam.IsSpam
);
```

Switch-case Edges

Route messages to different executors based on conditions:

```
builder.AddSwitch(routerExecutor, switchBuilder =>
    switchBuilder
        .AddCase(
            message => message.Priority < Priority.Normal,
            executorA
        )
        .AddCase(
            message => message.Priority < Priority.High,
            executorB
        )
        .SetDefault(executorC)
);
```

Fan-out Edges

Distribute messages from one executor to multiple targets:

```
// Send to all targets
builder.AddFanOutEdge(splitterExecutor, targets: [worker1, worker2, worker3]);

// Send to specific targets based on partitioner function
builder.AddFanOutEdge(
    source: routerExecutor,
    partitioner: (message, targetCount) => message.Priority switch
    {
        Priority.High => [0], // Route to first worker only
        Priority.Normal => [1, 2], // Route to workers 2 and 3
        _ => Enumerable.Range(0, targetCount) // Route to all workers
    },
    targets: [highPriorityWorker, normalWorker1, normalWorker2]
);
```

Fan-in Edges

Collect messages from multiple sources into a single target:

```
// Aggregate results from multiple workers
builder.AddFanInEdge(aggregatorExecutor, sources: [worker1, worker2, worker3]);
```

Next Step

- Learn about Workflows to understand how to build and execute workflows.

Microsoft Agent Framework Workflows Core Concepts - Workflows

09/25/2025

This document provides an in-depth look at the **Workflows** component of the Microsoft Agent Framework Workflow system.

Overview

A Workflow ties everything together and manages execution. It's the orchestrator that coordinates executor execution, message routing, and event streaming.

Building Workflows

Workflows are constructed using the `WorkflowBuilder` class, which provides a fluent API for defining the workflow structure:

C#

```
// Create executors
using Microsoft.Agents.Workflows;

var processor = new DataProcessor();
var validator = new Validator();
var formatter = new Formatter();

// Build workflow
WorkflowBuilder builder = new(processor); // Set starting executor
builder.AddEdge(processor, validator);
builder.AddEdge(validator, formatter);
var workflow = builder.Build<string>(); // Specify input message type
```

Workflow Execution

Workflows support both streaming and non-streaming execution modes:

C#

```
using Microsoft.Agents.Workflows;

// Streaming execution - get events as they happen
StreamingRun run = await InProcessExecution.StreamAsync(workflow, inputMessage);
await foreach (WorkflowEvent evt in run.WatchStreamAsync())
```

```
{
    if (evt is ExecutorCompleteEvent executorComplete)
    {
        Console.WriteLine($"{executorComplete.ExecutorId}:
{executorComplete.Data}");
    }

    if (evt is WorkflowCompletedEvent completed)
    {
        Console.WriteLine($"Workflow completed: {completed.Data}");
    }
}

// Non-streaming execution - wait for completion
Run result = await InProcessExecution.RunAsync(workflow, inputMessage);
foreach (WorkflowEvent evt in result.NewEvents)
{
    if (evt is WorkflowCompletedEvent completed)
    {
        Console.WriteLine($"Final result: {completed.Data}");
    }
}
```

Workflow Validation

The framework performs comprehensive validation when building workflows:

- **Type Compatibility**: Ensures message types are compatible between connected executors
- **Graph Connectivity**: Verifies all executors are reachable from the start executor
- **Executor Binding**: Confirms all executors are properly bound and instantiated
- **Edge Validation**: Checks for duplicate edges and invalid connections

Execution Model

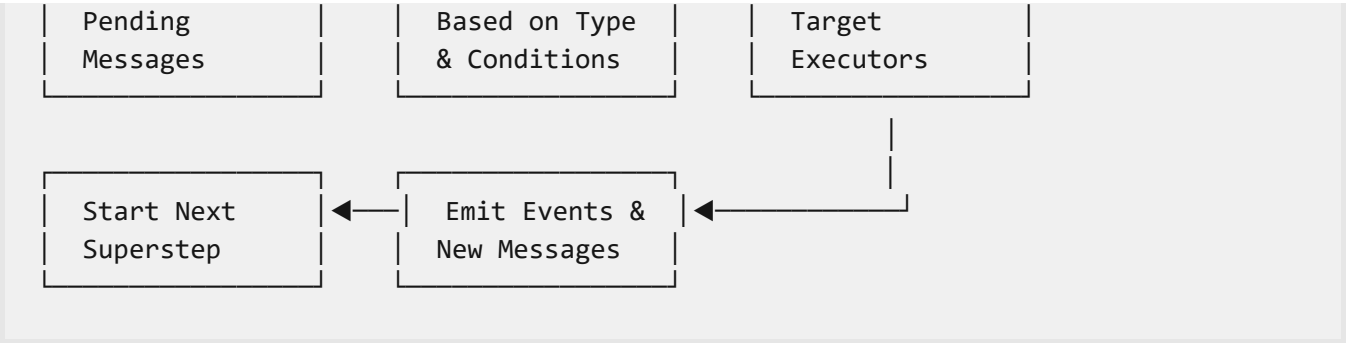
The framework uses a modified Pregel execution model with clear data flow semantics and superstep-based processing.

Pregel-Style Supersteps

Workflow execution is organized into discrete supersteps, where each superstep processes all available messages in parallel:

text

```
Superstep N:
┌──────────────────┐    ┌──────────────────┐    ┌──────────────────┐
│  Collect All     │──▶│  Route Messages  │──▶│  Execute All     │
```



Key Execution Characteristics

- **Superstep Isolation**: All executors in a superstep run concurrently without interfering with each other
- **Message Delivery**: Messages are delivered in parallel to all matching edges
- **Event Streaming**: Events are emitted in real-time as executors complete processing
- **Type Safety**: Runtime type validation ensures messages are routed to compatible handlers

Next Step

- Learn about events to understand how to monitor and observe workflow execution.

Microsoft Agent Framework Workflows Core Concepts - Events

09/26/2025

This document provides an in-depth look at the **Events** system of Workflows in the Microsoft Agent Framework.

Overview

There are built-in events that provide observability into the workflow execution.

Built-in Event Types

```
// Workflow lifecycle events
WorkflowStartedEvent    // Workflow execution begins
WorkflowCompletedEvent  // Workflow reaches completion
WorkflowErrorEvent      // Workflow encounters an error

// Executor events
ExecutorInvokeEvent     // Executor starts processing
ExecutorCompleteEvent   // Executor finishes processing
ExecutorFailureEvent    // Executor encounters an error

// Superstep events
SuperStepStartedEvent   // Superstep begins
SuperStepCompletedEvent // Superstep completes

// Request events
RequestInfoEvent        // A request is issued
```

Consuming Events

```
using Microsoft.Agents.Workflows;

await foreach (WorkflowEvent evt in run.WatchStreamAsync())
{
    switch (evt)
    {
        case ExecutorInvokeEvent invoke:
            Console.WriteLine($"Starting {invoke.ExecutorId}");
            break;

        case ExecutorCompleteEvent complete:
```

```
            Console.WriteLine($"Completed {complete.ExecutorId}:
{complete.Data}");
            break;

        case WorkflowCompletedEvent finished:
            Console.WriteLine($"Workflow finished: {finished.Data}");
            return;

        case WorkflowErrorEvent error:
            Console.WriteLine($"Workflow error: {error.Exception}");
            return;
    }
}
```

Custom Events

Users can define and emit custom events during workflow execution for enhanced observability.

```
using Microsoft.Agents.Workflows;
using Microsoft.Agents.Workflows.Reflection;

internal sealed class CustomEvent(string message) : WorkflowEvent(message) { }

internal sealed class CustomExecutor() : ReflectingExecutor<CustomExecutor>
("CustomExecutor"), IMessageHandler<string>
{
    public async ValueTask HandleAsync(string message, IWorkflowContext context)
    {
        await context.AddEventAsync(new CustomEvent($"Processing message:
{message}"));
        // Executor logic...
    }
}
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows Orchestrations

09/25/2025

Orchestrations are pre-built workflow patterns that allow developers to quickly create complex workflows by simply plugging in their own AI agents.

Why Multi-Agent?

Traditional single-agent systems are limited in their ability to handle complex, multi-faceted tasks. By orchestrating multiple agents, each with specialized skills or roles, we can create systems that are more robust, adaptive, and capable of solving real-world problems collaboratively.

Supported Orchestrations

Expand table

Pattern	Description	Typical Use Case
Concurrent	Broadcasts a task to all agents, collects results independently.	Parallel analysis, independent subtasks, ensemble decision making.
Sequential	Passes the result from one agent to the next in a defined order.	Step-by-step workflows, pipelines, multi-stage processing.
Handoff	Dynamically passes control between agents based on context or rules.	Dynamic workflows, escalation, fallback, or expert handoff scenarios.
Magentic	Inspired by MagenticOne.	Complex, generalist multi-agent collaboration.

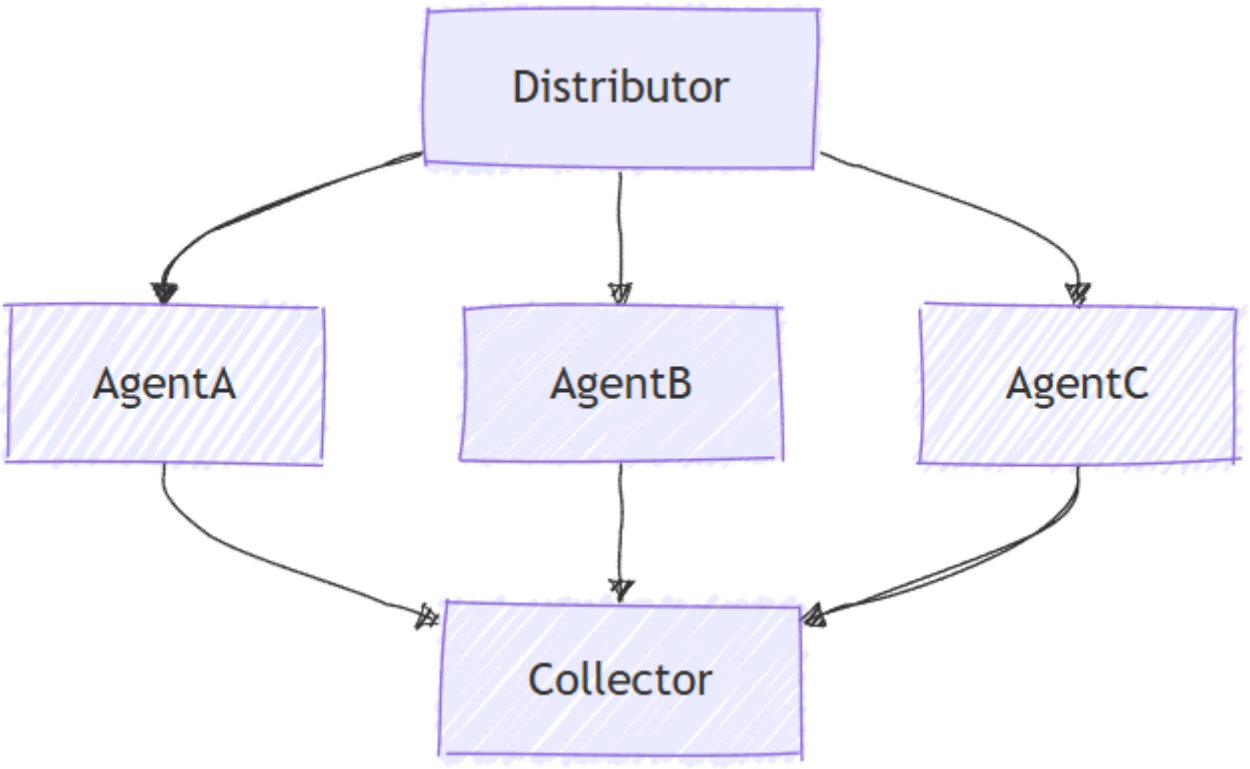
Next Steps

Explore the individual orchestration patterns to understand their unique features and how to use them effectively in your applications.

Microsoft Agent Framework Workflows Orchestrations - Concurrent

10/01/2025

Concurrent orchestration enables multiple agents to work on the same task in parallel. Each agent processes the input independently, and their results are collected and aggregated. This approach is well-suited for scenarios where diverse perspectives or solutions are valuable, such as brainstorming, ensemble reasoning, or voting systems.



What You'll Learn

- How to define multiple agents with different expertise
- How to orchestrate these agents to work concurrently on a single task
- How to collect and process the results

In concurrent orchestration, multiple agents work on the same task simultaneously and independently, providing diverse perspectives on the same input.

Set Up the Azure OpenAI Client

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.Workflows;
using Microsoft.Extensions.AI;
using Microsoft.Agents.AI;

// 1) Set up the Azure OpenAI client
var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
    throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
var client = new AzureOpenAIClient(new Uri(endpoint), new AzureCliCredential())
    .GetChatClient(deployment Name)
    .AsIChatClient();
```

Define Your Agents

Create multiple specialized agents that will work on the same task concurrently:

C#

```
// 2) Helper method to create translation agents
static ChatClientAgent GetTranslationAgent(string targetLanguage, IChatClient
chatClient) =>
    new(chatClient,
        $"You are a translation assistant who only responds in {targetLanguage}.
Respond to any " +
        $"input by outputting the name of the input language and then translating
the input to {targetLanguage}.");

// Create translation agents for concurrent processing
var translationAgents = (from lang in (string[])["French", "Spanish", "English"]
                         select GetTranslationAgent(lang, client));
```

Set Up the Concurrent Orchestration

Build the workflow using `AgentWorkflowBuilder` to run agents in parallel:

C#

```
// 3) Build concurrent workflow
```

```
var workflow = AgentWorkflowBuilder.BuildConcurrent(translationAgents);
```

Run the Concurrent Workflow and Collect Results

Execute the workflow and process events from all agents running simultaneously:

C#

```
// 4) Run the workflow
var messages = new List<ChatMessage> { new(ChatRole.User, "Hello, world!") };

StreamingRun run = await InProcessExecution.StreamAsync(workflow, messages);
await run.TrySendMessageAsync(new TurnToken(emitEvents: true));

List<ChatMessage> result = new();
await foreach (WorkflowEvent evt in run.WatchStreamAsync().ConfigureAwait(false))
{
    if (evt is AgentRunUpdateEvent e)
    {
        Console.WriteLine($"{e.ExecutorId}: {e.Data}");
    }
    else if (evt is WorkflowCompletedEvent completed)
    {
        result = (List<ChatMessage>)completed.Data!;
        break;
    }
}

// Display aggregated results from all agents
Console.WriteLine("===== Final Aggregated Results =====");
foreach (var message in result)
{
    Console.WriteLine($"{message.Role}: {message.Content}");
}
```

Sample Output

plaintext

```
French_Agent: English detected. Bonjour, le monde !
Spanish_Agent: English detected. ¡Hola, mundo!
English_Agent: English detected. Hello, world!

===== Final Aggregated Results =====
User: Hello, world!
Assistant: English detected. Bonjour, le monde !
Assistant: English detected. ¡Hola, mundo!
Assistant: English detected. Hello, world!
```

Key Concepts

- **Parallel Execution**: All agents process the input simultaneously and independently
- **AgentWorkflowBuilder.BuildConcurrent()**: Creates a concurrent workflow from a collection of agents
- **Automatic Aggregation**: Results from all agents are automatically collected into the final result
- **Event Streaming**: Real-time monitoring of agent progress through `AgentRunUpdateEvent`
- **Diverse Perspectives**: Each agent brings its unique expertise to the same problem

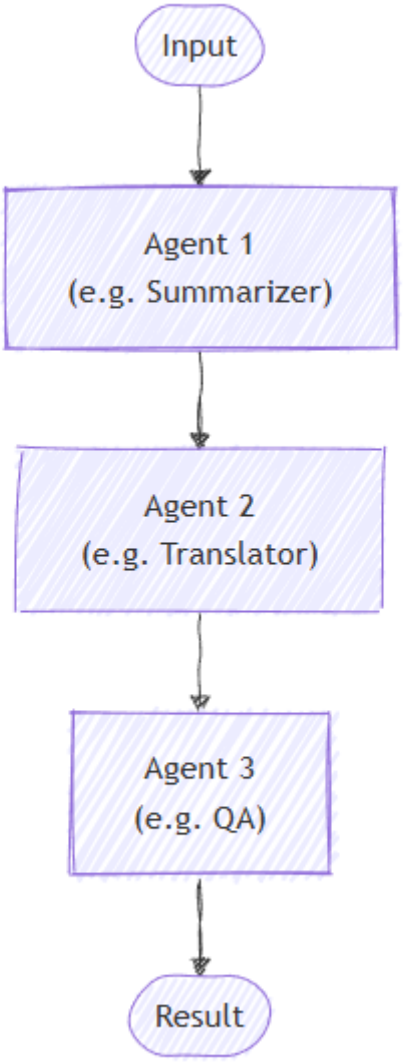
Next steps

Sequential Orchestration

Microsoft Agent Framework Workflows Orchestrations - Sequential

10/01/2025

In sequential orchestration, agents are organized in a pipeline. Each agent processes the task in turn, passing its output to the next agent in the sequence. This is ideal for workflows where each step builds upon the previous one, such as document review, data processing pipelines, or multi-stage reasoning.



What You'll Learn

- How to create a sequential pipeline of agents
- How to chain agents where each builds upon the previous output
- How to mix agents with custom executors for specialized tasks
- How to track the conversation flow through the pipeline

Define Your Agents

In sequential orchestration, agents are organized in a pipeline where each agent processes the task in turn, passing output to the next agent in the sequence.

Set Up the Azure OpenAI Client

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.Workflows;
using Microsoft.Extensions.AI;
using Microsoft.Agents.AI;

// 1) Set up the Azure OpenAI client
var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
    throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
var client = new AzureOpenAIClient(new Uri(endpoint), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient();
```

Create specialized agents that will work in sequence:

C#

```
// 2) Helper method to create translation agents
static ChatClientAgent GetTranslationAgent(string targetLanguage, IChatClient
chatClient) =>
    new(chatClient,
        $"You are a translation assistant who only responds in {targetLanguage}.
Respond to any " +
        $"input by outputting the name of the input language and then translating
the input to {targetLanguage}.");

// Create translation agents for sequential processing
var translationAgents = (from lang in (string[])["French", "Spanish", "English"]
                         select GetTranslationAgent(lang, client));
```

Set Up the Sequential Orchestration

Build the workflow using `AgentWorkflowBuilder`:

C#

```
// 3) Build sequential workflow
var workflow = AgentWorkflowBuilder.BuildSequential(translationAgents);
```

Run the Sequential Workflow

Execute the workflow and process the events:

C#

```
// 4) Run the workflow
var messages = new List<ChatMessage> { new(ChatRole.User, "Hello, world!") };

StreamingRun run = await InProcessExecution.StreamAsync(workflow, messages);
await run.TrySendMessageAsync(new TurnToken(emitEvents: true));

List<ChatMessage> result = new();
await foreach (WorkflowEvent evt in run.WatchStreamAsync().ConfigureAwait(false))
{
    if (evt is AgentRunUpdateEvent e)
    {
        Console.WriteLine($"{e.ExecutorId}: {e.Data}");
    }
    else if (evt is WorkflowCompletedEvent completed)
    {
        result = (List<ChatMessage>)completed.Data!;
        break;
    }
}

// Display final result
foreach (var message in result)
{
    Console.WriteLine($"{message.Role}: {message.Content}");
}
```

Sample Output

plaintext

```
French_Translation: User: Hello, world!
French_Translation: Assistant: English detected. Bonjour, le monde !
Spanish_Translation: Assistant: French detected. ¡Hola, mundo!
English_Translation: Assistant: Spanish detected. Hello, world!
```

Key Concepts

- **Sequential Processing**: Each agent processes the output of the previous agent in order
- **AgentWorkflowBuilder.BuildSequential()**: Creates a pipeline workflow from a collection of agents
- **ChatClientAgent**: Represents an agent backed by a chat client with specific instructions
- **StreamingRun**: Provides real-time execution with event streaming capabilities
- **Event Handling**: Monitor agent progress through `AgentRunUpdateEvent` and completion through `WorkflowCompletedEvent`

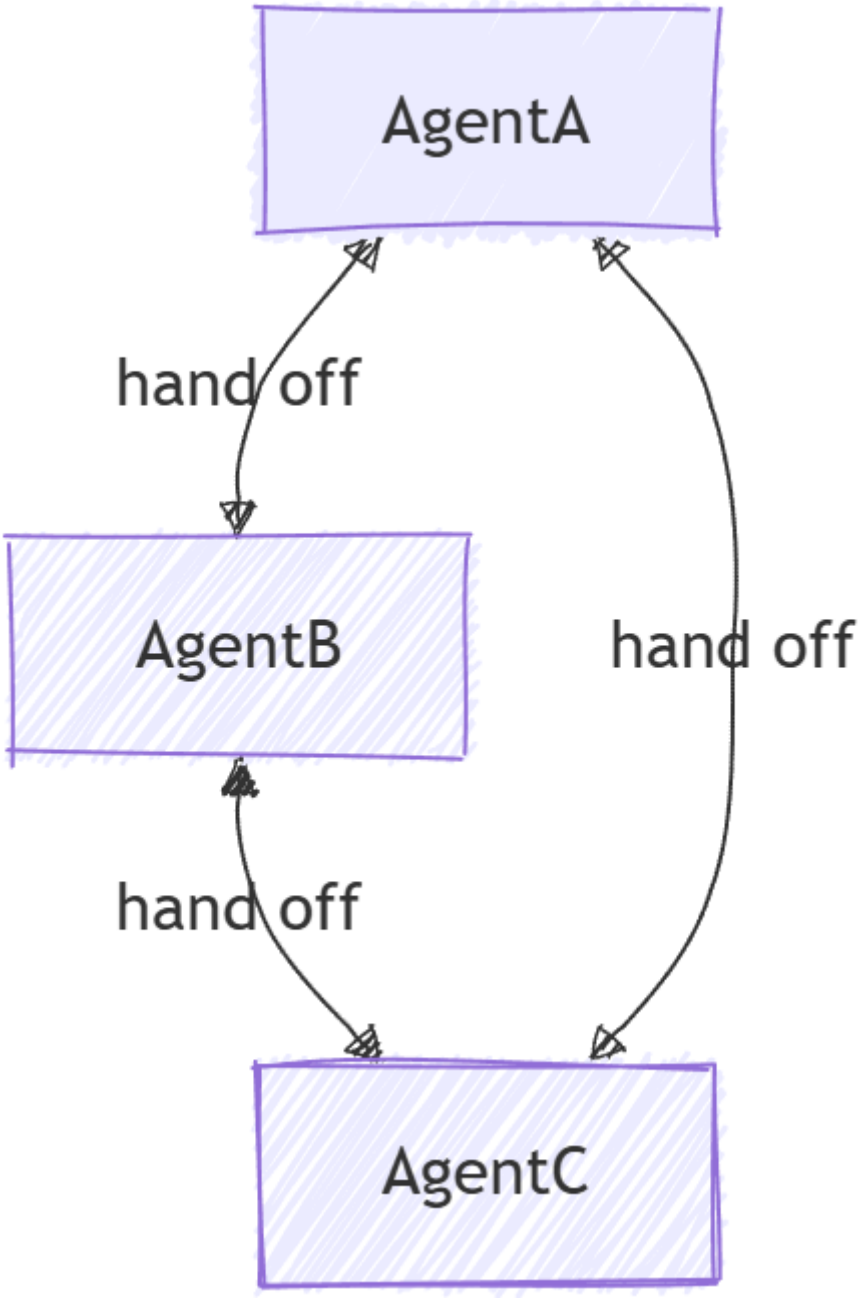
Next steps

Magentic Orchestration

Microsoft Agent Framework Workflows Orchestrations - Handoff

10/01/2025

Handoff orchestration allows agents to transfer control to one another based on the context or user request. Each agent can "handoff" the conversation to another agent with the appropriate expertise, ensuring that the right agent handles each part of the task. This is particularly useful in customer support, expert systems, or any scenario requiring dynamic delegation.



Differences Between Handoff and Agent-as-Tools

While agent-as-tools is commonly considered as a multi-agent pattern and it may look similar to handoff at first glance, there are fundamental differences between the two:

- **Control Flow**: In handoff orchestration, control is explicitly passed between agents based on defined rules. Each agent can decide to hand off the entire task to another agent. There is no central authority managing the workflow. In contrast, agent-as-tools involves a primary agent that delegates sub tasks to other agents and once the agent completes the sub task, control returns to the primary agent.
- **Task Ownership**: In handoff, the agent receiving the handoff takes full ownership of the task. In agent-as-tools, the primary agent retains overall responsibility for the task, while other agents are treated as tools to assist in specific subtasks.
- **Context Management**: In handoff orchestration, the conversation is handed off to another agent entirely. The receiving agent has full context of what has been done so far. In agent-as-tools, the primary agent manages the overall context and may provide only relevant information to the tool agents as needed.

What You'll Learn

- How to create specialized agents for different domains
- How to configure handoff rules between agents
- How to build interactive workflows with dynamic agent routing
- How to handle multi-turn conversations with agent switching

In handoff orchestration, agents can transfer control to one another based on context, allowing for dynamic routing and specialized expertise handling.

Set Up the Azure OpenAI Client

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Agents.Workflows;
using Microsoft.Extensions.AI;
using Microsoft.Agents.AI;

// 1) Set up the Azure OpenAI client
var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT") ??
    throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not set.");
var deploymentName =
Environment.GetEnvironmentVariable("AZURE_OPENAI_DEPLOYMENT_NAME") ?? "gpt-4o-
mini";
```

```
var client = new AzureOpenAIClient(new Uri(endpoint), new AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient();
```

Define Your Specialized Agents

Create domain-specific agents and a triage agent for routing:

C#

```
// 2) Create specialized agents
ChatClientAgent historyTutor = new(client,
    "You provide assistance with historical queries. Explain important events and
context clearly. Only respond about history.",
    "history_tutor",
    "Specialist agent for historical questions");

ChatClientAgent mathTutor = new(client,
    "You provide help with math problems. Explain your reasoning at each step and
include examples. Only respond about math.",
    "math_tutor",
    "Specialist agent for math questions");

ChatClientAgent triageAgent = new(client,
    "You determine which agent to use based on the user's homework question.
ALWAYS handoff to another agent.",
    "triage_agent",
    "Routes messages to the appropriate specialist agent");
```

Configure Handoff Rules

Define which agents can hand off to which other agents:

C#

```
// 3) Build handoff workflow with routing rules
var workflow = AgentWorkflowBuilder.StartHandoffWith(triageAgent)
    .WithHandoff(triageAgent, [mathTutor, historyTutor])  // Triage can route to
either specialist
    .WithHandoff(mathTutor, triageAgent)                // Math tutor can return
to triage
    .WithHandoff(historyTutor, triageAgent)             // History tutor can
return to triage
    .Build();
```

Run Interactive Handoff Workflow

Handle multi-turn conversations with dynamic agent switching:

C#

```
// 4) Process multi-turn conversations
List<ChatMessage> messages = new();

while (true)
{
    Console.Write("Q: ");
    string userInput = Console.ReadLine()!;
    messages.Add(new(ChatRole.User, userInput));

    // Execute workflow and process events
    StreamingRun run = await InProcessExecution.StreamAsync(workflow, messages);
    await run.TrySendMessageAsync(new TurnToken(emitEvents: true));

    List<ChatMessage> newMessages = new();
    await foreach (WorkflowEvent evt in
run.WatchStreamAsync().ConfigureAwait(false))
    {
        if (evt is AgentRunUpdateEvent e)
        {
            Console.WriteLine($"{e.ExecutorId}: {e.Data}");
        }
        else if (evt is WorkflowCompletedEvent completed)
        {
            newMessages = (List<ChatMessage>)completed.Data!;
            break;
        }
    }

    // Add new messages to conversation history
    messages.AddRange(newMessages.Skip(messages.Count));
}
```

Sample Interaction

plaintext

```
Q: What is the derivative of x^2?
triage_agent: This is a math question. I'll hand this off to the math tutor.
math_tutor: The derivative of x^2 is 2x. Using the power rule, we bring down the
exponent (2) and multiply it by the coefficient (1), then reduce the exponent by
1: d/dx(x^2) = 2x^(2-1) = 2x.

Q: Tell me about World War 2
triage_agent: This is a history question. I'll hand this off to the history tutor.
history_tutor: World War 2 was a global conflict from 1939 to 1945. It began when
Germany invaded Poland and involved most of the world's nations. Key events
included the Holocaust, Pearl Harbor attack, D-Day invasion, and ended with atomic
```

```
bombs on Japan.

Q: Can you help me with calculus integration?
triage_agent: This is another math question. I'll route this to the math tutor.
math_tutor: I'd be happy to help with calculus integration! Integration is the
reverse of differentiation. The basic power rule for integration is: ∫x^n dx =
x^(n+1)/(n+1) + C, where C is the constant of integration.
```

Key Concepts

- **Dynamic Routing**: Agents can decide which agent should handle the next interaction based on context
- **AgentWorkflowBuilder.StartHandoffWith()**: Defines the initial agent that starts the workflow
- **WithHandoff()**: Configures handoff rules between specific agents
- **Context Preservation**: Full conversation history is maintained across all handoffs
- **Multi-turn Support**: Supports ongoing conversations with seamless agent switching
- **Specialized Expertise**: Each agent focuses on their domain while collaborating through handoffs

Next steps

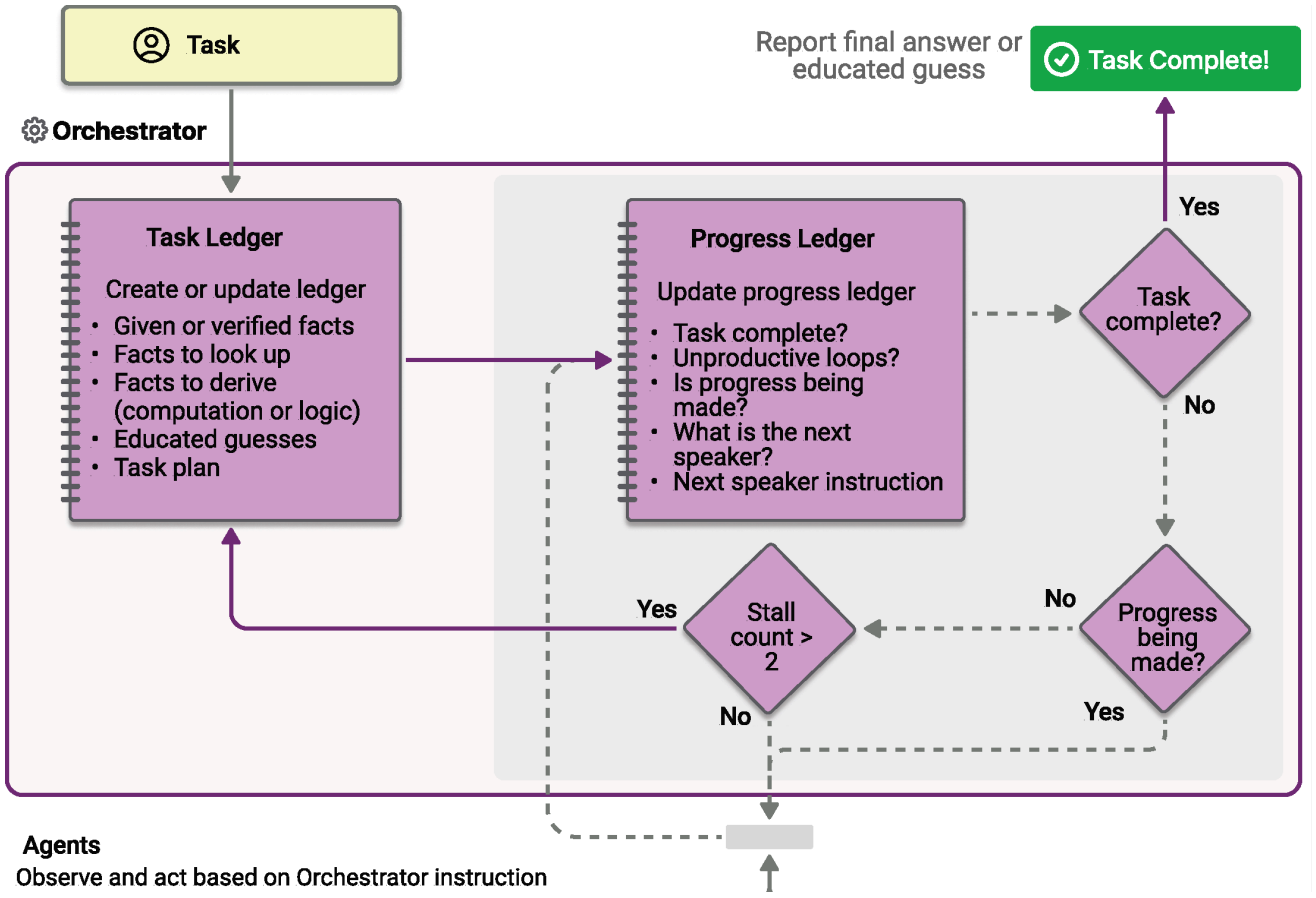
Magnetic Orchestration

Microsoft Agent Framework Workflows Orchestrations - Magentic

09/25/2025

Magentic orchestration is designed based on the Magentic-One system invented by AutoGen. It is a flexible, general-purpose multi-agent pattern designed for complex, open-ended tasks that require dynamic collaboration. In this pattern, a dedicated Magentic manager coordinates a team of specialized agents, selecting which agent should act next based on the evolving context, task progress, and agent capabilities.

The Magentic manager maintains a shared context, tracks progress, and adapts the workflow in real time. This enables the system to break down complex problems, delegate subtasks, and iteratively refine solutions through agent collaboration. The orchestration is especially well-suited for scenarios where the solution path is not known in advance and may require multiple rounds of reasoning, research, and computation.



What You'll Learn

- How to set up a Magentic manager to coordinate multiple specialized agents
- How to configure callbacks for streaming and event handling
- How to implement human-in-the-loop plan review

- How to track agent collaboration and progress through complex tasks

Define Your Specialized Agents

Coming soon...

Next steps

Handoff Orchestration

Microsoft Agent Framework Workflows - Working with Agents

10/01/2025

This page provides an overview of how to use **Agents** within the Microsoft Agent Framework Workflows.

Overview

To add intelligence to your workflows, you can leverage AI agents as part of your workflow execution. AI agents can be easily integrated into workflows, allowing you to create complex, intelligent solutions that were previously difficult to achieve.

Add an Agent Directly to a Workflow

You can add agents to your workflow via edges:

C#

```
using Microsoft.Agents.Workflows;
using Microsoft.Extensions.AI;
using Microsoft.Agents.AI;

// Create the agents first
AIAgent agentA = new ChatClientAgent(chatClient, instructions);
AIAgent agentB = new ChatClientAgent(chatClient, instructions);

// Build a workflow with the agents
WorkflowBuilder builder = new(agentA);
builder.AddEdge(agentA, agentB);
Workflow<ChatMessage> workflow = builder.Build<ChatMessage>();
```

Running the Workflow

Inside the workflow created above, the agents are actually wrapped inside an executor that handles the communication of the agent with other parts of the workflow. The executor can handle three message types:

- `ChatMessage`: A single chat message
- `List<ChatMessage>`: A list of chat messages
- `TurnToken`: A turn token that signals the start of a new turn

The executor doesn't trigger the agent to respond until it receives a `TurnToken`. Any messages received before the `TurnToken` are buffered and sent to the agent when the `TurnToken` is received.

C#

```
StreamingRun run = await InProcessExecution.StreamAsync(workflow, new
ChatMessage(ChatRole.User, "Hello World!"));
// Must send the turn token to trigger the agents. The agents are wrapped as
executors.
// When they receive messages, they will cache the messages and only start
processing
// when they receive a TurnToken. The turn token will be passed from one agent to
the next.
await run.TrySendMessageAsync(new TurnToken(emitEvents: true));
await foreach (WorkflowEvent evt in run.WatchStreamAsync().ConfigureAwait(false))
{
    // The agents will run in streaming mode and an AgentRunUpdateEvent
    // will be emitted as new chunks are generated.
    if (evt is AgentRunUpdateEvent agentRunUpdate)
    {
        Console.WriteLine($"{agentRunUpdate.ExecutorId}: {agentRunUpdate.Data}");
    }
}
```

Using a Custom Agent Executor

Sometimes you may want to customize how AI agents are integrated into a workflow. You can achieve this by creating a custom executor. This allows you to control:

- The invocation of the agent: streaming or non-streaming
- The message types the agent will handle, including custom message types
- The life cycle of the agent, including initialization and cleanup
- The usage of agent threads and other resources
- Additional events emitted during the agent's execution, including custom events
- Integration with other workflow features, such as shared states and requests/responses

C#

```
internal sealed class CustomAgentExecutor :
ReflectingExecutor<CustomAgentExecutor>, IMessageHandler<CustomInput,
CustomOutput>
{
    private readonly AIAgent _agent;

    /// <summary>
    /// Creates a new instance of the <see cref="CustomAgentExecutor"/> class.
    /// </summary>
```

```
    /// <param name="agent">The AI agent used for custom processing</param>
    public CustomAgentExecutor(AIAgent agent) : base("CustomAgentExecutor")
    {
        this._agent = agent;
    }

    public async ValueTask<CustomOutput> HandleAsync(CustomInput message,
IWorkflowContext context)
    {
        // Retrieve any shared states if needed
        var sharedState = await context.ReadStateAsync<SharedStateType>
("sharedStateId", scopeName: "SharedStateScope");

        // Render the input for the agent
        var agentInput = RenderInput(message, sharedState);

        // Invoke the agent
        // Assume the agent is configured with structured outputs with type
`CustomOutput`
        var response = await this._agent.RunAsync(agentInput);
        var customOutput = JsonSerializer.Deserialize<CustomOutput>
(response.Text);

        return customOutput;
    }
}
```

Next Steps

- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Using workflows as Agents

09/25/2025

This document provides an overview of how to use **Workflows as Agents** in the Microsoft Agent Framework.

Overview

Developers can turn a workflow into an Agent Framework Agent and interact with the workflow as if it were an agent. This feature enables the following scenarios:

- Integrate workflows with APIs that already support the Agent interface.
- Use a workflow to drive single agent interactions, which can create more powerful agents.
- Close the loop between agents and workflows, creating opportunities for advanced compositions.

Creating a Workflow Agent

Create a workflow of any complexity and then wrap it as an agent.

```
var workflowAgent = workflow.AsAgent(id: "workflow-agent", name: "Workflow
Agent");
var workflowAgentThread = workflowAgent.GetNewThread();
```

Using a Workflow Agent

Then use the workflow agent like any other Agent Framework agent.

```
await foreach (var update in workflowAgent.RunStreamingAsync(input,
workflowAgentThread).ConfigureAwait(false))
{
    Console.WriteLine(update);
}
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.

- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Request and Response

09/25/2025

This page provides an overview of how **Request and Response** handling works in the Microsoft Agent Framework Workflow system.

Overview

Executors in a workflow can send requests to outside of the workflow and wait for responses. This is useful for scenarios where an executor needs to interact with external systems, such as human-in-the-loop interactions, or any other asynchronous operations.

Enable Request and Response Handling in a Workflow

Requests and responses are handled via a special type called `InputPort`.

C#

```
// Create an input port that receives requests of type CustomRequestType and
responses of type CustomResponseType.
var inputPort = InputPort.Create<CustomRequestType, CustomResponseType>("input-
port");
```

Add the input port to a workflow.

C#

```
var executorA = new SomeExecutor();
var workflow = new WorkflowBuilder(inputPort)
    .AddEdge(inputPort, executorA)
    .AddEdge(executorA, inputPort)
    .Build<CustomRequestType>();
```

Now, because in the workflow we have `executorA` connected to the `inputPort` in both directions, `executorA` needs to be able to send requests and receive responses via the `inputPort`. Here is what we need to do in `SomeExecutor` to send a request and receive a response.

C#

```
internal sealed class SomeExecutor() : ReflectingExecutor<SomeExecutor>
("SomeExecutor"), IMessageHandler<CustomResponseType>
{
    public async ValueTask HandleAsync(CustomResponseType message,
IWorkflowContext context)
    {
        // Process the response...
        ...
        // Send a request
        await context.SendMessageAsync(new
CustomRequestType(...)).ConfigureAwait(false);
    }
}
```

Alternatively, `SomeExecutor` can separate the request sending and response handling into two handlers.

C#

```
internal sealed class SomeExecutor() : ReflectingExecutor<SomeExecutor>
("SomeExecutor"), IMessageHandler<CustomResponseType>,
IMessageHandler<OtherDataType>
{
    public async ValueTask HandleAsync(CustomResponseType message,
IWorkflowContext context)
    {
        // Process the response...
        ...
    }

    public async ValueTask HandleAsync(OtherDataType message, IWorkflowContext
context)
    {
        // Process the message...
        ...
        // Send a request
        await context.SendMessageAsync(new
CustomRequestType(...)).ConfigureAwait(false);
    }
}
```

Handling Requests and Responses

An `InputPort` emits a `RequestInfoEvent` when it receives a request. You can subscribe to these events to handle incoming requests from the workflow. When you receive a response from an external system, send it back to the workflow using the response mechanism. The framework automatically routes the response to the executor that sent the original request.

C#

```
StreamingRun handle = await InProcessExecution.StreamAsync(workflow,
input).ConfigureAwait(false);
await foreach (WorkflowEvent evt in
handle.WatchStreamAsync().ConfigureAwait(false))
{
    switch (evt)
    {
        case RequestInfoEvent requestInputEvt:
            // Handle `RequestInfoEvent` from the workflow
            ExternalResponse response =
requestInputEvt.Request.CreateResponse<CustomResponseType>(...);
            await handle.SendResponseAsync(response).ConfigureAwait(false);
            break;

        case WorkflowCompletedEvent workflowCompleteEvt:
            // The workflow has completed successfully
            Console.WriteLine($"Workflow completed with result:
{workflowCompleteEvt.Data}");
            return;
    }
}
```

Checkpoints and Requests

To learn more about checkpoints, please refer to this page.

When a checkpoint is created, pending requests are also saved as part of the checkpoint state. When you restore from a checkpoint, any pending requests will be re-emitted, allowing the workflow to continue processing from where it left off.

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Shared States

09/25/2025

This document provides an overview of **Shared States** in the Microsoft Agent Framework Workflow system.

Overview

Shared States allow multiple executors within a workflow to access and modify common data. This feature is essential for scenarios where different parts of the workflow need to share information where direct message passing is not feasible or efficient.

Writing to Shared States

```
using Microsoft.Agents.Workflows;
using Microsoft.Agents.Workflows.Reflection;

internal sealed class FileReadExecutor() : ReflectingExecutor<FileReadExecutor>
("FileReadExecutor"), IMessageHandler<string, string>
{
    /// <summary>
    /// Reads a file and stores its content in a shared state.
    /// </summary>
    /// <param name="message">The path to the embedded resource file.</param>
    /// <param name="context">The workflow context for accessing shared states.
</param>
    /// <returns>The ID of the shared state where the file content is stored.
</returns>
    public async ValueTask<string> HandleAsync(string message, IWorkflowContext
context)
    {
        // Read file content from embedded resource
        string fileContent = File.ReadAllText(message);
        // Store file content in a shared state for access by other executors
        string fileID = Guid.NewGuid().ToString();
        await context.QueueStateUpdateAsync<string>(fileID, fileContent,
scopeName: "FileContent");

        return fileID;
    }
}
```

Accessing Shared States

```
using Microsoft.Agents.Workflows;
using Microsoft.Agents.Workflows.Reflection;

internal sealed class WordCountingExecutor() :
ReflectingExecutor<WordCountingExecutor>("WordCountingExecutor"),
IMessageHandler<string, int>
{
    /// <summary>
    /// Counts the number of words in the file content stored in a shared state.
    /// </summary>
    /// <param name="message">The ID of the shared state containing the file
content.</param>
    /// <param name="context">The workflow context for accessing shared states.
</param>
    /// <returns>The number of words in the file content.</returns>
    public async ValueTask<int> HandleAsync(string message, IWorkflowContext
context)
    {
        // Retrieve the file content from the shared state
        var fileContent = await context.ReadStateAsync<string>(message, scopeName:
"FileContent")
            ?? throw new InvalidOperationException("File content state not
found");

        return fileContent.Split([' ', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Checkpoints

10/01/2025

This page provides an overview of **Checkpoints** in the Microsoft Agent Framework Workflow system.

Overview

Checkpoints allow you to save the state of a workflow at specific points during its execution, and resume from those points later. This feature is particularly useful for the following scenarios:

- Long-running workflows where you want to avoid losing progress in case of failures.
- Long-running workflows where you want to pause and resume execution at a later time.
- Workflows that require periodic state saving for auditing or compliance purposes.
- Workflows that need to be migrated across different environments or instances.

When Are Checkpoints Created?

Remember that workflows are executed in **supersteps**, as documented in the core concepts. Checkpoints are created at the end of each superstep, after all executors in that superstep have completed their execution. A checkpoint captures the entire state of the workflow, including:

- The current state of all executors
- All pending messages in the workflow for the next superstep
- Pending requests and responses
- Shared states

Capturing Checkpoints

To enable check pointing, a `CheckpointManager` needs to be provided when creating a workflow run. A checkpoint then can be accessed via a `SuperStepCompletedEvent`.

C#

```
using Microsoft.Agents.Workflows;

// Create a checkpoint manager to manage checkpoints
var checkpointManager = new CheckpointManager();
```

```
// List to store checkpoint info for later use
var checkpoints = new List<CheckpointInfo>();

// Run the workflow with checkpointing enabled
Checkpointed<StreamingRun> checkpointedRun = await InProcessExecution
    .StreamAsync(workflow, input, checkpointManager)
    .ConfigureAwait(false);
await foreach (WorkflowEvent evt in
checkpointedRun.Run.WatchStreamAsync().ConfigureAwait(false))
{
    if (evt is SuperStepCompletedEvent superStepCompletedEvt)
    {
        // Access the checkpoint and store it
        CheckpointInfo? checkpoint =
superStepCompletedEvt.CompletionInfo!.Checkpoint;
        if (checkpoint != null)
        {
            checkpoints.Add(checkpoint);
        }
    }
}
```

Resuming from Checkpoints

You can resume a workflow from a specific checkpoint directly on the same run.

C#

```
// Assume we want to resume from the 6th checkpoint
CheckpointInfo savedCheckpoint = checkpoints[5];
// Note that we are restoring the state directly to the same run instance.
await checkpointedRun.RestoreCheckpointAsync(savedCheckpoint,
CancellationToken.None).ConfigureAwait(false);
await foreach (WorkflowEvent evt in
checkpointedRun.Run.WatchStreamAsync().ConfigureAwait(false))
{
    if (evt is WorkflowCompletedEvent workflowCompletedEvt)
    {
        Console.WriteLine($"Workflow completed with result:
{workflowCompletedEvt.Data}");
    }
}
```

Rehydrating from Checkpoints

Or you can rehydrate a workflow from a checkpoint into a new run instance.

C#

```
// Assume we want to resume from the 6th checkpoint
CheckpointInfo savedCheckpoint = checkpoints[5];
Checkpointed<StreamingRun> newCheckpointedRun = await InProcessExecution
    .ResumeStreamAsync(newWorkflow, savedCheckpoint, checkpointManager)
    .ConfigureAwait(false);
await foreach (WorkflowEvent evt in
newCheckpointedRun.Run.WatchStreamAsync().ConfigureAwait(false))
{
    if (evt is WorkflowCompletedEvent workflowCompletedEvt)
    {
        Console.WriteLine($"Workflow completed with result:
{workflowCompletedEvt.Data}");
    }
}
```

Save Executor States

To ensure that the state of an executor is captured in a checkpoint, the executor must override the `OnCheckpointingAsync` method and save its state to the workflow context.

C#

```
using Microsoft.Agents.Workflows;
using Microsoft.Agents.Workflows.Reflection;

internal sealed class CustomExecutor() : ReflectingExecutor<CustomExecutor>
("CustomExecutor"), IMessageHandler<string>
{
    private const string StateKey = "CustomExecutorState";

    private List<string> messages = new();

    public async ValueTask HandleAsync(string message, IWorkflowContext context)
    {
        this.messages.Add(message);
        // Executor logic...
    }

    protected override ValueTask OnCheckpointingAsync(IWorkflowContext context,
CancellationToken cancellation = default)
    {
        return context.QueueStateUpdateAsync(StateKey, this.messages);
    }
}
```

Also, to ensure the state is correctly restored when resuming from a checkpoint, the executor must override the `OnCheckpointRestoredAsync` method and load its state from the workflow context.

C#

```
protected override async ValueTask OnCheckpointRestoredAsync(IWorkflowContext
context, CancellationToken cancellation = default)
{
    this.messages = await context.ReadStateAsync<List<string>>
(StateKey).ConfigureAwait(false);
}
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.

Microsoft Agent Framework Workflows - Observability

10/01/2025

Observability provides insights into the internal state and behavior of workflows during execution. This includes logging, metrics, and tracing capabilities that help monitor and debug workflows.

Aside from the standard GenAI telemetry, Agent Framework Workflows emits additional spans, logs, and metrics to provide deeper insights into workflow execution. These observability features help developers understand the flow of messages, the performance of executors, and any errors that may occur.

Enable Observability

Observability is enabled framework-wide by setting the `ENABLE_OTEL=true` environment variable or calling `setup_observability()` at the beginning of your application.

env

```
# This is not required if you run `setup_observability()` in your code
ENABLE_OTEL=true
# Sensitive data (e.g., message content) will be included in logs and traces if
this is set to true
ENABLE_SENSITIVE_DATA=true
```

Python

```python
from agent_framework.observability import setup_observability

setup_observability(enable_sensitive_data=True)
```

Workflow Spans

Expand table

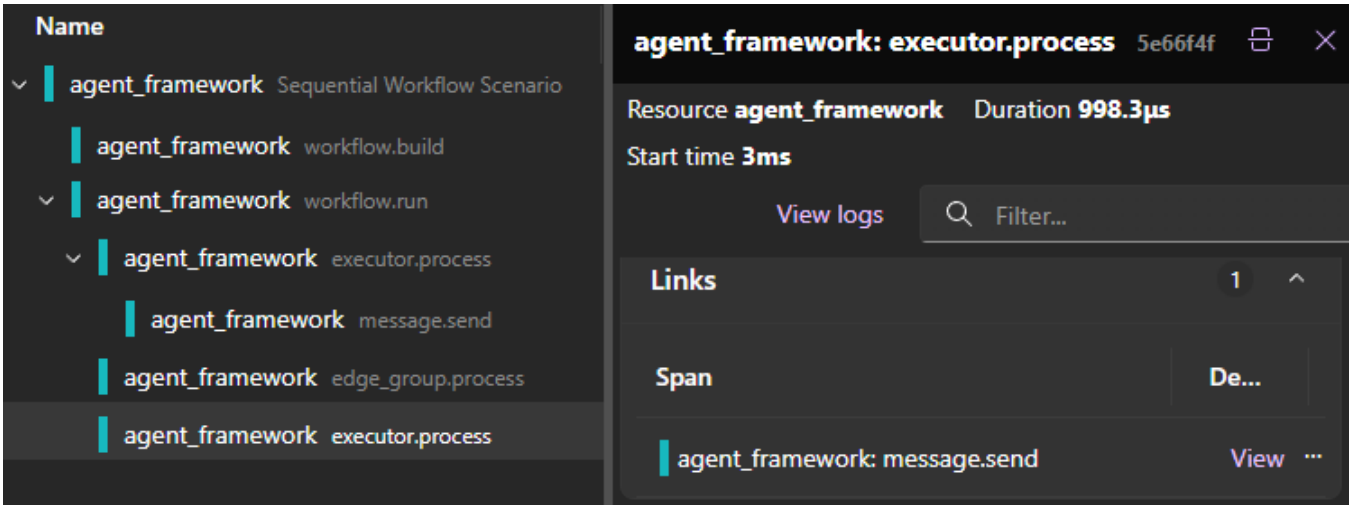
Span Name	Description
`workflow.build`	For each workflow build
`workflow.run`	For each workflow execution

Span Name	Description
`message.send`	For each message sent to an executor
`executor.process`	For each executor processing a message
`edge_group.process`	For each edge group processing a message

Links between Spans

When an executor sends a message to another executor, the `message.send` span is created as a child of the `executor.process` span. However, the `executor.process` span of the target executor will not be a child of the `message.send` span because the execution is not nested. Instead, the `executor.process` span of the target executor is linked to the `message.send` span of the source executor. This creates a traceable path through the workflow execution.

For example:



Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Visualization

10/01/2025

Sometimes a workflow that has multiple executors and complex interactions can be hard to understand from just reading the code. Visualization can help you see the structure of the workflow more clearly, so that you can verify that it has the intended design.

Workflow visualization is done via a `WorkflowViz` object that can be instantiated with a `Workflow` object. The `WorkflowViz` object can then generate visualizations in different formats, such as Graphviz DOT format or Mermaid diagram format.

> **Tip**
>
> To export visualization images you also need to install GraphViz.

Creating a `WorkflowViz` object is straightforward:

Python

```python
from agent_framework import WorkflowBuilder, WorkflowViz

# Create a workflow with a fan-out and fan-in pattern
workflow = (
    WorkflowBuilder()
    .set_start_executor(dispatcher)
    .add_fan_out_edges(dispatcher, [researcher, marketer, legal])
    .add_fan_in_edges([researcher, marketer, legal], aggregator)
    .build()
)

viz = WorkflowViz(workflow)
```

Then, you can create visualizations in different formats:

Python

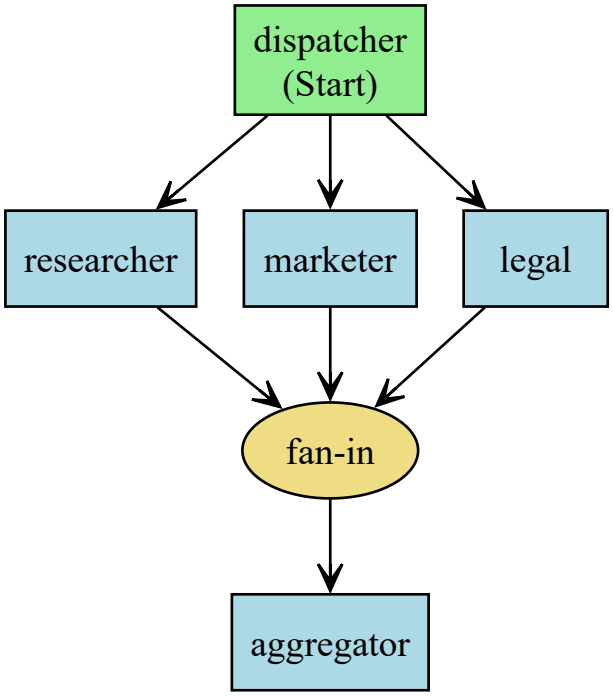
```python
# Mermaid diagram
print(viz.to_mermaid())
# DiGraph string
print(viz.to_digraph())
# Export to a file
print(viz.export(format="svg"))
```

The exported diagram will look similar to the following for the example workflow:

mermaid

```
flowchart TD
  dispatcher["dispatcher (Start)"];
  researcher["researcher"];
  marketer["marketer"];
  legal["legal"];
  aggregator["aggregator"];
  fan_in__aggregator__e3a4ff58((fan-in))
  legal --> fan_in__aggregator__e3a4ff58;
  marketer --> fan_in__aggregator__e3a4ff58;
  researcher --> fan_in__aggregator__e3a4ff58;
  fan_in__aggregator__e3a4ff58 --> aggregator;
  dispatcher --> researcher;
  dispatcher --> marketer;
  dispatcher --> legal;
```

or in Graphviz DOT format:



AutoGen to Microsoft Agent Framework Migration Guide

10/01/2025

A comprehensive guide for migrating from AutoGen to the Microsoft Agent Framework Python SDK.

Table of Contents

Background

AutoGen is a framework for building AI agents and multi-agent systems using large language models (LLMs). It started as a research project at Microsoft Research and pioneered several concepts in multi-agent orchestration, such as GroupChat and event-driven agent runtime. The project has been a fruitful collaboration of the open-source community and many important features came from external contributors.

Microsoft Agent Framework is a new multi-language SDK for building AI agents and workflows using LLMs. It represents a significant evolution of the ideas pioneered in AutoGen and incorporates lessons learned from real-world usage. It is developed by the core AutoGen team and Semantic Kernel team at Microsoft, and is designed to be a new foundation for building AI applications going forward.

What follows is a practical migration path: we'll start by grounding on what stays the same and what changes at a glance, then cover model client setup, single-agent features, and finally multi-agent orchestration with concrete code side-by-side. Along the way, links to runnable samples in the Agent Framework repo help you validate each step.

Key Similarities and Differences

What Stays the Same

The foundations are familiar. You still create agents around a model client, provide instructions, and attach tools. Both libraries support function-style tools, token streaming, multimodal content, and async I/O.

Python

```
# Both frameworks follow similar patterns
# AutoGen
agent = AssistantAgent(name="assistant", model_client=client, tools=[my_tool])
result = await agent.run(task="Help me with this task")

# Agent Framework
agent = ChatAgent(name="assistant", chat_client=client, tools=[my_tool])
result = await agent.run("Help me with this task")
```

Key Differences

1. Orchestration style: AutoGen pairs an event-driven core with a high-level `Team`. Agent Framework centers on a typed, graph-based `Workflow` that routes data along edges and activates executors when inputs are ready.
2. Tools: AutoGen wraps functions with `FunctionTool`. Agent Framework uses `@ai_function`, infers schemas automatically, and adds hosted tools such as a code interpreter and web search.
3. Agent behavior: `AssistantAgent` is single-turn unless you increase `max_tool_iterations`. `ChatAgent` is multi-turn by default and keeps invoking tools until it can return a final answer.
4. Runtime: AutoGen offers embedded and experimental distributed runtimes. Agent Framework focuses on single-process composition today; distributed execution is planned.

Model Client Creation and Configuration

Both frameworks provide model clients for major AI providers, with similar but not identical APIs.

Expand table

Feature	AutoGen	Agent Framework
OpenAI Client	`OpenAIChatCompletionClient`	`OpenAIChatClient`
OpenAI Responses Client	❌ Not available	`OpenAIResponsesClient`
Azure OpenAI	`AzureOpenAIChatCompletionClient`	`AzureOpenAIChatClient`
Azure OpenAI Responses	❌ Not available	`AzureOpenAIResponsesClient`

Feature	AutoGen	Agent Framework
Azure AI	`AzureAIChatCompletionClient`	`AzureAIAgentClient`
Anthropic	`AnthropicChatCompletionClient`	🚧 Planned
Ollama	`OllamaChatCompletionClient`	🚧 Planned
Caching	`ChatCompletionCache` wrapper	🚧 Planned

AutoGen Model Clients

Python

```python
from autogen_ext.models.openai import OpenAIChatCompletionClient,
AzureOpenAIChatCompletionClient

# OpenAI
client = OpenAIChatCompletionClient(
    model="gpt-5",
    api_key="your-key"
)

# Azure OpenAI
client = AzureOpenAIChatCompletionClient(
    azure_endpoint="https://your-endpoint.openai.azure.com/",
    azure_deployment="gpt-5",
    api_version="2024-12-01",
    api_key="your-key"
)
```

Agent Framework ChatClients

Python

```python
from agent_framework.openai import OpenAIChatClient
from agent_framework.azure import AzureOpenAIChatClient

# OpenAI (reads API key from environment)
client = OpenAIChatClient(model_id="gpt-5")

# Azure OpenAI (uses environment or default credentials; see samples for auth
options)
client = AzureOpenAIChatClient(model_id="gpt-5")
```

For detailed examples, see:

- OpenAI Chat Client - Basic OpenAI client setup

- Azure OpenAI Chat Client - Azure OpenAI with authentication
- Azure AI Client - Azure AI agent integration

Responses API Support (Agent Framework Exclusive)

Agent Framework's `AzureOpenAIResponsesClient` and `OpenAIResponsesClient` provide specialized support for reasoning models and structured responses not available in AutoGen:

Python

```python
from agent_framework.azure import AzureOpenAIResponsesClient
from agent_framework.openai import OpenAIResponsesClient

# Azure OpenAI with Responses API
azure_responses_client = AzureOpenAIResponsesClient(model_id="gpt-5")

# OpenAI with Responses API
openai_responses_client = OpenAIResponsesClient(model_id="gpt-5")
```

For Responses API examples, see:

- Azure Responses Client Basic - Azure OpenAI with responses
- OpenAI Responses Client Basic - OpenAI responses integration

Single-Agent Feature Mapping

This section maps single-agent features between AutoGen and Agent Framework. With a client in place, create an agent, attach tools, and choose between non-streaming and streaming execution.

Basic Agent Creation and Execution

Once you have a model client configured, the next step is creating agents. Both frameworks provide similar agent abstractions, but with different default behaviors and configuration options.

AutoGen AssistantAgent

Python

```python
from autogen_agentchat.agents import AssistantAgent

agent = AssistantAgent(
    name="assistant",
```

```
    model_client=client,
    system_message="You are a helpful assistant.",
    tools=[my_tool],
    max_tool_iterations=1  # Single-turn by default
)

# Execution
result = await agent.run(task="What's the weather?")
```

Agent Framework ChatAgent

Python

```python
from agent_framework import ChatAgent, ai_function
from agent_framework.openai import OpenAIChatClient

# Create simple tools for the example
@ai_function
def get_weather(location: str) -> str:
    """Get weather for a location."""
    return f"Weather in {location}: sunny"

@ai_function
def get_time() -> str:
    """Get current time."""
    return "Current time: 2:30 PM"

# Create client
client = OpenAIChatClient(model_id="gpt-5")

async def example():
    # Direct creation
    agent = ChatAgent(
        name="assistant",
        chat_client=client,
        instructions="You are a helpful assistant.",
        tools=[get_weather]  # Multi-turn by default
    )

    # Factory method (more convenient)
    agent = client.create_agent(
        name="assistant",
        instructions="You are a helpful assistant.",
        tools=[get_weather]
    )

    # Execution with runtime tool configuration
    result = await agent.run(
        "What's the weather?",
        tools=[get_time],  # Can add tools at runtime
```

```
        tool_choice="auto"
    )
```

Key Differences:

- **Default behavior**: `ChatAgent` automatically iterates through tool calls, while `AssistantAgent` requires explicit `max_tool_iterations` setting
- **Runtime configuration**: `ChatAgent.run()` accepts `tools` and `tool_choice` parameters for per-invocation customization
- **Factory methods**: Agent Framework provides convenient factory methods directly from chat clients
- **State management**: `ChatAgent` is stateless and doesn't maintain conversation history between invocations, unlike `AssistantAgent` which maintains conversation history as part of its state

Managing Conversation State with AgentThread

To continue conversations with `ChatAgent`, use `AgentThread` to manage conversation history:

Python

```
# Assume we have an agent from previous examples
async def conversation_example():
    # Create a new thread that will be reused
    thread = agent.get_new_thread()

    # First interaction - thread is empty
    result1 = await agent.run("What's 2+2?", thread=thread)
    print(result1.text)  # "4"

    # Continue conversation - thread contains previous messages
    result2 = await agent.run("What about that number times 10?", thread=thread)
    print(result2.text)  # "40" (understands "that number" refers to 4)

    # AgentThread can use external storage, similar to ChatCompletionContext in
AutoGen
```

Stateless by default: quick demo

Python

```
# Without a thread (two independent invocations)
r1 = await agent.run("What's 2+2?")
print(r1.text)  # e.g., "4"

r2 = await agent.run("What about that number times 10?")
print(r2.text)  # Likely ambiguous without prior context; may not be "40"
```

```python
# With a thread (shared context across calls)
thread = agent.get_new_thread()
print((await agent.run("What's 2+2?", thread=thread)).text)  # "4"
print((await agent.run("What about that number times 10?", thread=thread)).text)
# "40"
```

For thread management examples, see:

- Azure AI with Thread - Conversation state management
- OpenAI Chat Client with Thread - Thread usage patterns
- Redis-backed Threads - Persisting conversation state externally

OpenAI Assistant Agent Equivalence

Both frameworks provide OpenAI Assistant API integration:

Python

```python
# AutoGen OpenAIAssistantAgent
from autogen_ext.agents.openai import OpenAIAssistantAgent
```

Python

```python
# Agent Framework has OpenAI Assistants support via OpenAIAssistantsClient
from agent_framework.openai import OpenAIAssistantsClient
```

For OpenAI Assistant examples, see:

- OpenAI Assistants Basic - Basic assistant setup
- OpenAI Assistants with Function Tools - Custom tools integration
- Azure OpenAI Assistants Basic - Azure assistant setup
- OpenAI Assistants with Thread - Thread management

Streaming Support

Both frameworks stream tokens in real time—from clients and from agents—to keep UIs responsive.

AutoGen Streaming

Python

```python
# Model client streaming
async for chunk in client.create_stream(messages):
    if isinstance(chunk, str):
        print(chunk, end="")

# Agent streaming
async for event in agent.run_stream(task="Hello"):
    if isinstance(event, ModelClientStreamingChunkEvent):
        print(event.content, end="")
    elif isinstance(event, TaskResult):
        print("Final result received")
```

Agent Framework Streaming

Python

```python
# Assume we have client, agent, and tools from previous examples
async def streaming_example():
    # Chat client streaming
    async for chunk in client.get_streaming_response("Hello", tools=tools):
        if chunk.text:
            print(chunk.text, end="")

    # Agent streaming
    async for chunk in agent.run_stream("Hello"):
        if chunk.text:
            print(chunk.text, end="", flush=True)
```

Tip: In Agent Framework, both clients and agents yield the same update shape; you can read `chunk.text` in either case.

Message Types and Creation

Understanding how messages work is crucial for effective agent communication. Both frameworks provide different approaches to message creation and handling, with AutoGen using separate message classes and Agent Framework using a unified message system.

AutoGen Message Types

Python

```python
from autogen_agentchat.messages import TextMessage, MultiModalMessage
from autogen_core.models import UserMessage

# Text message
text_msg = TextMessage(content="Hello", source="user")
```

```
# Multi-modal message
multi_modal_msg = MultiModalMessage(
    content=["Describe this image", image_data],
    source="user"
)

# Convert to model format for use with model clients
user_message = text_msg.to_model_message()
```

Agent Framework Message Types

Python

```
from agent_framework import ChatMessage, TextContent, DataContent, UriContent,
Role
import base64

# Text message
text_msg = ChatMessage(role=Role.USER, text="Hello")

# Supply real image bytes, or use a data: URI/URL via UriContent
image_bytes = b"<your_image_bytes>"
image_b64 = base64.b64encode(image_bytes).decode()
image_uri = f"{image_b64}"

# Multi-modal message with mixed content
multi_modal_msg = ChatMessage(
    role=Role.USER,
    contents=[
        TextContent(text="Describe this image"),
        DataContent(uri=image_uri, media_type="image/jpeg")
    ]
)
```

Key Differences:

- AutoGen uses separate message classes (`TextMessage`, `MultiModalMessage`) with a `source` field
- Agent Framework uses a unified `ChatMessage` with typed content objects and a `role` field
- Agent Framework messages use `Role` enum (USER, ASSISTANT, SYSTEM, TOOL) instead of string sources

Tool Creation and Integration

Tools extend agent capabilities beyond text generation. The frameworks take different approaches to tool creation, with Agent Framework providing more automated schema

generation.

AutoGen FunctionTool

Python

```python
from autogen_core.tools import FunctionTool

async def get_weather(location: str) -> str:
    """Get weather for a location."""
    return f"Weather in {location}: sunny"

# Manual tool creation
tool = FunctionTool(
    func=get_weather,
    description="Get weather information"
)

# Use with agent
agent = AssistantAgent(name="assistant", model_client=client, tools=[tool])
```

Agent Framework @ai_function

Python

```python
from agent_framework import ai_function
from typing import Annotated
from pydantic import Field

@ai_function
def get_weather(
    location: Annotated[str, Field(description="The location to get weather for")]
) -> str:
    """Get weather for a location."""
    return f"Weather in {location}: sunny"

# Direct use with agent (automatic conversion)
agent = ChatAgent(name="assistant", chat_client=client, tools=[get_weather])
```

For detailed examples, see:

- OpenAI Chat Agent Basic - Simple OpenAI chat agent
- OpenAI with Function Tools - Agent with custom tools
- Azure OpenAI Basic - Azure OpenAI agent setup

Hosted Tools (Agent Framework Exclusive)

Agent Framework provides hosted tools that are not available in AutoGen:

Python

```python
from agent_framework import ChatAgent, HostedCodeInterpreterTool,
HostedWebSearchTool
from agent_framework.azure import AzureOpenAIChatClient

# Azure OpenAI client with a model that supports hosted tools
client = AzureOpenAIChatClient(model_id="gpt-5")

# Code execution tool
code_tool = HostedCodeInterpreterTool()

# Web search tool
search_tool = HostedWebSearchTool()

agent = ChatAgent(
    name="researcher",
    chat_client=client,
    tools=[code_tool, search_tool]
)
```

For detailed examples, see:

- Azure AI with Code Interpreter - Code execution tool
- Azure AI with Multiple Tools - Multiple hosted tools
- OpenAI with Web Search - Web search integration

Requirements and caveats:

- Hosted tools are only available on models/accounts that support them. Verify entitlements and model support for your provider before enabling these tools.
- Configuration differs by provider; follow the prerequisites in each sample for setup and permissions.
- Not every model supports every hosted tool (e.g., web search vs code interpreter). Choose a compatible model in your environment.

Note: AutoGen supports local code execution tools, but this feature is planned for future Agent Framework versions.

Key Difference: Agent Framework handles tool iteration automatically at the agent level. Unlike AutoGen's `max_tool_iterations` parameter, Agent Framework agents continue tool execution until completion by default, with built-in safety mechanisms to prevent infinite loops.

MCP Server Support

For advanced tool integration, both frameworks support Model Context Protocol (MCP), enabling agents to interact with external services and data sources. Agent Framework provides more comprehensive built-in support.

AutoGen MCP Support

AutoGen has basic MCP support through extensions (specific implementation details vary by version).

Agent Framework MCP Support

Python

```python
from agent_framework import ChatAgent, MCPStdioTool, MCPStreamableHTTPTool,
MCPWebsocketTool
from agent_framework.openai import OpenAIChatClient

# Create client for the example
client = OpenAIChatClient(model_id="gpt-5")

# Stdio MCP server
mcp_tool = MCPStdioTool(
    name="filesystem",
    command="uvx mcp-server-filesystem",
    args=["/allowed/directory"]
)

# HTTP streaming MCP
http_mcp = MCPStreamableHTTPTool(
    name="http_mcp",
    url="http://localhost:8000/sse"
)

# WebSocket MCP
ws_mcp = MCPWebsocketTool(
    name="websocket_mcp",
    url="ws://localhost:8000/ws"
)

agent = ChatAgent(name="assistant", chat_client=client, tools=[mcp_tool])
```

For MCP examples, see:

- OpenAI with Local MCP - Using MCPStreamableHTTPTool with OpenAI
- OpenAI with Hosted MCP - Using hosted MCP services
- Azure AI with Local MCP - Using MCP with Azure AI
- Azure AI with Hosted MCP - Using hosted MCP with Azure AI

Agent-as-a-Tool Pattern

One powerful pattern is using agents themselves as tools, enabling hierarchical agent architectures. Both frameworks support this pattern with different implementations.

AutoGen AgentTool

Python

```python
from autogen_agentchat.tools import AgentTool

# Create specialized agent
writer = AssistantAgent(
    name="writer",
    model_client=client,
    system_message="You are a creative writer."
)

# Wrap as tool
writer_tool = AgentTool(agent=writer)

# Use in coordinator (requires disabling parallel tool calls)
coordinator_client = OpenAIChatCompletionClient(
    model="gpt-5",
    parallel_tool_calls=False
)
coordinator = AssistantAgent(
    name="coordinator",
    model_client=coordinator_client,
    tools=[writer_tool]
)
```

Agent Framework as_tool()

Python

```python
from agent_framework import ChatAgent

# Assume we have client from previous examples
# Create specialized agent
writer = ChatAgent(
    name="writer",
    chat_client=client,
    instructions="You are a creative writer."
)

# Convert to tool
writer_tool = writer.as_tool(
    name="creative_writer",
```

```
    description="Generate creative content",
    arg_name="request",
    arg_description="What to write"
)

# Use in coordinator
coordinator = ChatAgent(
    name="coordinator",
    chat_client=client,
    tools=[writer_tool]
)
```

Explicit migration note: In AutoGen, set `parallel_tool_calls=False` on the coordinator's model client when wrapping agents as tools to avoid concurrency issues when invoking the same agent instance. In Agent Framework, `as_tool()` does not require disabling parallel tool calls as agents are stateless by default.

Middleware (Agent Framework Feature)

Agent Framework introduces middleware capabilities that AutoGen lacks. Middleware enables powerful cross-cutting concerns like logging, security, and performance monitoring.

Python

```python
from agent_framework import ChatAgent, AgentRunContext, FunctionInvocationContext
from typing import Callable, Awaitable

# Assume we have client from previous examples
async def logging_middleware(
    context: AgentRunContext,
    next: Callable[[AgentRunContext], Awaitable[None]]
) -> None:
    print(f"Agent {context.agent.name} starting")
    await next(context)
    print(f"Agent {context.agent.name} completed")

async def security_middleware(
    context: FunctionInvocationContext,
    next: Callable[[FunctionInvocationContext], Awaitable[None]]
) -> None:
    if "password" in str(context.arguments):
        print("Blocking function call with sensitive data")
        return  # Don't call next()
    await next(context)

agent = ChatAgent(
    name="secure_agent",
    chat_client=client,
```

```
    middleware=[logging_middleware, security_middleware]
)
```

Benefits:

- **Security**: Input validation and content filtering
- **Observability**: Logging, metrics, and tracing
- **Performance**: Caching and rate limiting
- **Error handling**: Graceful degradation and retry logic

For detailed middleware examples, see:

- Function-based Middleware - Simple function middleware
- Class-based Middleware - Object-oriented middleware
- Exception Handling Middleware - Error handling patterns
- Shared State Middleware - State management across agents

Custom Agents

Sometimes you don't want a model-backed agent at all—you want a deterministic or API-backed agent with custom logic. Both frameworks support building custom agents, but the patterns differ.

AutoGen: Subclass BaseChatAgent

Python

```python
from typing import Sequence
from autogen_agentchat.agents import BaseChatAgent
from autogen_agentchat.base import Response
from autogen_agentchat.messages import BaseChatMessage, TextMessage, StopMessage
from autogen_core import CancellationToken

class StaticAgent(BaseChatAgent):
    def __init__(self, name: str = "static", description: str = "Static
responder") -> None:
        super().__init__(name, description)

    @property
    def produced_message_types(self) -> Sequence[type[BaseChatMessage]]:  # Which
message types this agent produces
        return (TextMessage,)

    async def on_messages(self, messages: Sequence[BaseChatMessage],
cancellation_token: CancellationToken) -> Response:
        # Always return a static response
```

```
        return Response(chat_message=TextMessage(content="Hello from AutoGen
custom agent", source=self.name))
```

Notes:

- Implement `on_messages(...)` and return a `Response` with a chat message.
- Optionally implement `on_reset(...)` to clear internal state between runs.

Agent Framework: Extend BaseAgent (thread-aware)

Python

```python
from collections.abc import AsyncIterable
from typing import Any
from agent_framework import (
    AgentRunResponse,
    AgentRunResponseUpdate,
    AgentThread,
    BaseAgent,
    ChatMessage,
    Role,
    TextContent,
)

class StaticAgent(BaseAgent):
    async def run(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None = None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AgentRunResponse:
        # Build a static reply
        reply = ChatMessage(role=Role.ASSISTANT, contents=[TextContent(text="Hello
from AF custom agent")])

        # Persist conversation to the provided AgentThread (if any)
        if thread is not None:
            normalized = self._normalize_messages(messages)
            await self._notify_thread_of_new_messages(thread, normalized, reply)

        return AgentRunResponse(messages=[reply])

    async def run_stream(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None = None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AsyncIterable[AgentRunResponseUpdate]:
        # Stream the same static response in a single chunk for simplicity
```

```
        yield AgentRunResponseUpdate(contents=[TextContent(text="Hello from AF
custom agent")], role=Role.ASSISTANT)

        # Notify thread of input and the complete response once streaming ends
        if thread is not None:
            reply = ChatMessage(role=Role.ASSISTANT, contents=
[TextContent(text="Hello from AF custom agent")])
            normalized = self._normalize_messages(messages)
            await self._notify_thread_of_new_messages(thread, normalized, reply)
```

Notes:

- `AgentThread` maintains conversation state externally; use `agent.get_new_thread()` and pass it to `run`/`run_stream`.
- Call `self._notify_thread_of_new_messages(thread, input_messages, response_messages)` so the thread has both sides of the exchange.
- See the full sample: Custom Agent

Next, let's look at multi-agent orchestration—the area where the frameworks differ most.

Multi-Agent Feature Mapping

Programming Model Overview

The multi-agent programming models represent the most significant difference between the two frameworks.

AutoGen's Dual Model Approach

AutoGen provides two programming models:

1. **`autogen-core`**: Low-level, event-driven programming with `RoutedAgent` and message subscriptions
2. **`Team` abstraction**: High-level, run-centric model built on top of `autogen-core`

Python

```
# Low-level autogen-core (complex)
class MyAgent(RoutedAgent):
    @message_handler
    async def handle_message(self, message: TextMessage, ctx: MessageContext) ->
None:
        # Handle specific message types
```

```
        pass

# High-level Team (easier but limited)
team = RoundRobinGroupChat(
    participants=[agent1, agent2],
    termination_condition=StopAfterNMessages(5)
)
result = await team.run(task="Collaborate on this task")
```

Challenges:

- Low-level model is too complex for most users
- High-level model can become limiting for complex behaviors
- Bridging between the two models adds implementation complexity

Agent Framework's Unified Workflow Model

Agent Framework provides a single `Workflow` abstraction that combines the best of both approaches:

Python

```
from agent_framework import WorkflowBuilder, executor, WorkflowContext
from typing_extensions import Never

# Assume we have agent1 and agent2 from previous examples
@executor(id="agent1")
async def agent1_executor(input_msg: str, ctx: WorkflowContext[str]) -> None:
    response = await agent1.run(input_msg)
    await ctx.send_message(response.text)

@executor(id="agent2")
async def agent2_executor(input_msg: str, ctx: WorkflowContext[Never, str]) ->
None:
    response = await agent2.run(input_msg)
    await ctx.yield_output(response.text)  # Final output

# Build typed data flow graph
workflow = (WorkflowBuilder()
            .add_edge(agent1_executor, agent2_executor)
            .set_start_executor(agent1_executor)
            .build())

# Example usage (would be in async context)
# result = await workflow.run("Initial input")
```

For detailed workflow examples, see:

- Workflow Basics - Introduction to executors and edges

- Agents in Workflow - Integrating agents in workflows
- Workflow Streaming - Real-time workflow execution

Benefits:

- **Unified model**: Single abstraction for all complexity levels
- **Type safety**: Strongly typed inputs and outputs
- **Graph visualization**: Clear data flow representation
- **Flexible composition**: Mix agents, functions, and sub-workflows

Workflow vs GraphFlow

The Agent Framework's `Workflow` abstraction is inspired by AutoGen's experimental `GraphFlow` feature, but represents a significant evolution in design philosophy:

- **GraphFlow**: Control-flow based where edges are transitions and messages are broadcast to all agents; transitions are conditioned on broadcasted message content
- **Workflow**: Data-flow based where messages are routed through specific edges and executors are activated by edges, with support for concurrent execution.

Visual Overview

The diagram below contrasts AutoGen's control-flow GraphFlow (left) with Agent Framework's data-flow Workflow (right). GraphFlow models agents as nodes with conditional transitions and broadcasts. Workflow models executors (agents, functions, or sub-workflows) connected by typed edges; it also supports request/response pauses and checkpointing.

mermaid

```
flowchart LR

  subgraph AutoGenGraphFlow
    direction TB
    U[User / Task] --> A[Agent A]
    A -->|success| B[Agent B]
    A -->|retry| C[Agent C]
    A -. broadcast .- B
    A -. broadcast .- C
  end

  subgraph AgentFrameworkWorkflow
    direction TB
    I[Input] --> E1[Executor 1]
    E1 -->|"str"| E2[Executor 2]
    E1 -->|"image"| E3[Executor 3]
    E3 -->|"str"| E2
    E2 --> OUT[(Final Output)]
```

```
  end

  R[Request / Response Gate]
  E2 -. request .-> R
  R -. resume .-> E2

  CP[Checkpoint]
  E1 -. save .-> CP
  CP -. load .-> E1
```

In practice:

- GraphFlow uses agents as nodes and broadcasts messages; edges represent conditional transitions.
- Workflow routes typed messages along edges. Nodes (executors) can be agents, pure functions, or sub-workflows.
- Request/response lets a workflow pause for external input; checkpointing persists progress and enables resume.

Code Comparison

1) Sequential + Conditional

Python

```python
# AutoGen GraphFlow (fluent builder) — writer → reviewer → editor (conditional)
from autogen_agentchat.agents import AssistantAgent
from autogen_agentchat.teams import DiGraphBuilder, GraphFlow

writer = AssistantAgent(name="writer", description="Writes a draft",
model_client=client)
reviewer = AssistantAgent(name="reviewer", description="Reviews the draft",
model_client=client)
editor = AssistantAgent(name="editor", description="Finalizes the draft",
model_client=client)

graph = (
    DiGraphBuilder()
    .add_node(writer).add_node(reviewer).add_node(editor)
    .add_edge(writer, reviewer)  # always
    .add_edge(reviewer, editor, condition=lambda msg: "approve" in
msg.to_model_text())
    .set_entry_point(writer)
).build()

team = GraphFlow(participants=[writer, reviewer, editor], graph=graph)
result = await team.run(task="Draft a short paragraph about solar power")
```

Python

```python
# Agent Framework Workflow — sequential executors with conditional logic
from agent_framework import WorkflowBuilder, executor, WorkflowContext
from typing_extensions import Never

@executor(id="writer")
async def writer_exec(task: str, ctx: WorkflowContext[str]) -> None:
    await ctx.send_message(f"Draft: {task}")

@executor(id="reviewer")
async def reviewer_exec(draft: str, ctx: WorkflowContext[str]) -> None:
    decision = "approve" if "solar" in draft.lower() else "revise"
    await ctx.send_message(f"{decision}:{draft}")

@executor(id="editor")
async def editor_exec(msg: str, ctx: WorkflowContext[Never, str]) -> None:
    if msg.startswith("approve:"):
        await ctx.yield_output(msg.split(":", 1)[1])
    else:
        await ctx.yield_output("Needs revision")

workflow_seq = (
    WorkflowBuilder()
    .add_edge(writer_exec, reviewer_exec)
    .add_edge(reviewer_exec, editor_exec)
    .set_start_executor(writer_exec)
    .build()
)
```

2) Fan-out + Join (ALL vs ANY)

Python

```python
# AutoGen GraphFlow — A → (B, C) → D with ALL/ANY join
from autogen_agentchat.teams import DiGraphBuilder, GraphFlow
A, B, C, D = agent_a, agent_b, agent_c, agent_d

# ALL (default): D runs after both B and C
g_all = (
    DiGraphBuilder()
    .add_node(A).add_node(B).add_node(C).add_node(D)
    .add_edge(A, B).add_edge(A, C)
    .add_edge(B, D).add_edge(C, D)
    .set_entry_point(A)
).build()

# ANY: D runs when either B or C completes
g_any = (
    DiGraphBuilder()
    .add_node(A).add_node(B).add_node(C).add_node(D)
    .add_edge(A, B).add_edge(A, C)
```

```
    .add_edge(B, D, activation_group="join_d", activation_condition="any")
    .add_edge(C, D, activation_group="join_d", activation_condition="any")
    .set_entry_point(A)
).build()
```

Python

```python
# Agent Framework Workflow — A → (B, C) → aggregator (ALL vs ANY)
from agent_framework import WorkflowBuilder, executor, WorkflowContext
from typing_extensions import Never

@executor(id="A")
async def start(task: str, ctx: WorkflowContext[str]) -> None:
    await ctx.send_message(f"B:{task}", target_id="B")
    await ctx.send_message(f"C:{task}", target_id="C")

@executor(id="B")
async def branch_b(text: str, ctx: WorkflowContext[str]) -> None:
    await ctx.send_message(f"B_done:{text}")

@executor(id="C")
async def branch_c(text: str, ctx: WorkflowContext[str]) -> None:
    await ctx.send_message(f"C_done:{text}")

@executor(id="join_any")
async def join_any(msg: str, ctx: WorkflowContext[Never, str]) -> None:
    await ctx.yield_output(f"First: {msg}")  # ANY join (first arrival)

@executor(id="join_all")
async def join_all(msg: str, ctx: WorkflowContext[str, str]) -> None:
    state = await ctx.get_state() or {"items": []}
    state["items"].append(msg)
    await ctx.set_state(state)
    if len(state["items"]) >= 2:
        await ctx.yield_output(" | ".join(state["items"]))  # ALL join

wf_any = (
    WorkflowBuilder()
    .add_edge(start, branch_b).add_edge(start, branch_c)
    .add_edge(branch_b, join_any).add_edge(branch_c, join_any)
    .set_start_executor(start)
    .build()
)

wf_all = (
    WorkflowBuilder()
    .add_edge(start, branch_b).add_edge(start, branch_c)
    .add_edge(branch_b, join_all).add_edge(branch_c, join_all)
    .set_start_executor(start)
    .build()
)
```

3) Targeted Routing (no broadcast)

Python

```python
from agent_framework import WorkflowBuilder, executor, WorkflowContext
from typing_extensions import Never

@executor(id="ingest")
async def ingest(task: str, ctx: WorkflowContext[str]) -> None:
    # Route selectively using target_id
    if task.startswith("image:"):
        await ctx.send_message(task.removeprefix("image:"), target_id="vision")
    else:
        await ctx.send_message(task, target_id="writer")

@executor(id="writer")
async def write(text: str, ctx: WorkflowContext[Never, str]) -> None:
    await ctx.yield_output(f"Draft: {text}")

@executor(id="vision")
async def caption(image_ref: str, ctx: WorkflowContext[Never, str]) -> None:
    await ctx.yield_output(f"Caption: {image_ref}")

workflow = (
    WorkflowBuilder()
    .add_edge(ingest, write)
    .add_edge(ingest, caption)
    .set_start_executor(ingest)
    .build()
)

# Example usage (async):
# await workflow.run("Summarize the benefits of solar power")
# await workflow.run("image:https://example.com/panel.jpg")
```

What to notice:

- GraphFlow broadcasts messages and uses conditional transitions. Join behavior is configured via target-side `activation` and per-edge `activation_group` / `activation_condition` (e.g., group both edges into `join_d` with `activation_condition="any"`).
- Workflow routes data explicitly; use `target_id` to select downstream executors. Join behavior lives in the receiving executor (e.g., yield on first input vs wait for all), or via orchestration builders/aggregators.
- Executors in Workflow are free-form: wrap a `ChatAgent`, a function, or a sub-workflow and mix them within the same graph.

Key Differences

The table below summarizes the fundamental differences between AutoGen's GraphFlow and Agent Framework's Workflow:

Expand table

Aspect	AutoGen GraphFlow	Agent Framework Workflow
Flow Type	Control flow (edges are transitions)	Data flow (edges route messages)
Node Types	Agents only	Agents, functions, sub-workflows
Activation	Message broadcast	Edge-based activation
Type Safety	Limited	Strong typing throughout
Composability	Limited	Highly composable

Nesting Patterns

AutoGen Team Nesting

Python

```
# Inner team
inner_team = RoundRobinGroupChat(
    participants=[specialist1, specialist2],
    termination_condition=StopAfterNMessages(3)
)

# Outer team with nested team as participant
outer_team = RoundRobinGroupChat(
    participants=[coordinator, inner_team, reviewer],  # Team as participant
    termination_condition=StopAfterNMessages(10)
)

# Messages are broadcasted to all participants including nested team
result = await outer_team.run("Complex task requiring collaboration")
```

AutoGen nesting characteristics:

- Nested team receives all messages from outer team
- Nested team messages are broadcast to all outer team participants
- Shared message context across all levels

Agent Framework Workflow Nesting

Python

```python
from agent_framework import WorkflowExecutor, WorkflowBuilder

# Assume we have executors from previous examples
# specialist1_executor, specialist2_executor, coordinator_executor,
reviewer_executor

# Create sub-workflow
sub_workflow = (WorkflowBuilder()
                .add_edge(specialist1_executor, specialist2_executor)
                .set_start_executor(specialist1_executor)
                .build())

# Wrap as executor
sub_workflow_executor = WorkflowExecutor(
    workflow=sub_workflow,
    id="sub_process"
)

# Use in parent workflow
parent_workflow = (WorkflowBuilder()
                   .add_edge(coordinator_executor, sub_workflow_executor)
                   .add_edge(sub_workflow_executor, reviewer_executor)
                   .set_start_executor(coordinator_executor)
                   .build())
```

Agent Framework nesting characteristics:

- Isolated input/output through `WorkflowExecutor`
- No message broadcasting - data flows through specific connections
- Independent state management for each workflow level

Group Chat Patterns

Group chat patterns enable multiple agents to collaborate on complex tasks. Here's how common patterns translate between frameworks.

RoundRobinGroupChat Pattern

AutoGen Implementation:

Python

```python
from autogen_agentchat.teams import RoundRobinGroupChat
from autogen_agentchat.conditions import StopAfterNMessages

team = RoundRobinGroupChat(
    participants=[agent1, agent2, agent3],
```

```
    termination_condition=StopAfterNMessages(10)
)
result = await team.run("Discuss this topic")
```

Agent Framework Implementation:

Python

```python
from agent_framework import SequentialBuilder, WorkflowOutputEvent

# Assume we have agent1, agent2, agent3 from previous examples
# Sequential workflow through participants
workflow = SequentialBuilder().participants([agent1, agent2, agent3]).build()

# Example usage (would be in async context)
async def sequential_example():
    # Each agent appends to shared conversation
    async for event in workflow.run_stream("Discuss this topic"):
        if isinstance(event, WorkflowOutputEvent):
            conversation_history = event.data  # list[ChatMessage]
```

For detailed orchestration examples, see:

- Sequential Agents - Round-robin style agent execution
- Sequential Custom Executors - Custom executor patterns

For concurrent execution patterns, Agent Framework also provides:

Python

```python
from agent_framework import ConcurrentBuilder, WorkflowOutputEvent

# Assume we have agent1, agent2, agent3 from previous examples
# Concurrent workflow for parallel processing
workflow = (ConcurrentBuilder()
            .participants([agent1, agent2, agent3])
            .build())

# Example usage (would be in async context)
async def concurrent_example():
    # All agents process the input concurrently
    async for event in workflow.run_stream("Process this in parallel"):
        if isinstance(event, WorkflowOutputEvent):
            results = event.data  # Combined results from all agents
```

For concurrent execution examples, see:

- Concurrent Agents - Parallel agent execution
- Concurrent Custom Executors - Custom parallel patterns

- Concurrent with Custom Aggregator - Result aggregation patterns

MagenticOneGroupChat Pattern

AutoGen Implementation:

Python

```python
from autogen_agentchat.teams import MagenticOneGroupChat

team = MagenticOneGroupChat(
    participants=[researcher, coder, executor],
    model_client=coordinator_client,
    termination_condition=StopAfterNMessages(20)
)
result = await team.run("Complex research and analysis task")
```

Agent Framework Implementation:

Python

```python
from agent_framework import (
    MagenticBuilder, MagenticCallbackMode, WorkflowOutputEvent,
    MagenticCallbackEvent, MagenticOrchestratorMessageEvent,
MagenticAgentDeltaEvent
)

# Assume we have researcher, coder, and coordinator_client from previous examples
async def on_event(event: MagenticCallbackEvent) -> None:
    if isinstance(event, MagenticOrchestratorMessageEvent):
        print(f"[ORCHESTRATOR]: {event.message.text}")
    elif isinstance(event, MagenticAgentDeltaEvent):
        print(f"[{event.agent_id}]: {event.text}", end="")

workflow = (MagenticBuilder()
            .participants(researcher=researcher, coder=coder)
            .on_event(on_event, mode=MagenticCallbackMode.STREAMING)
            .with_standard_manager(
                chat_client=coordinator_client,
                max_round_count=20,
                max_stall_count=3,
                max_reset_count=2
            )
            .build())

# Example usage (would be in async context)
async def magentic_example():
    async for event in workflow.run_stream("Complex research task"):
        if isinstance(event, WorkflowOutputEvent):
            final_result = event.data
```

Agent Framework Customization Options:

The Magentic workflow provides extensive customization options:

- **Manager configuration**: Custom orchestrator models and prompts
- **Round limits**: `max_round_count`, `max_stall_count`, `max_reset_count`
- **Event callbacks**: Real-time streaming with granular event filtering
- **Agent specialization**: Custom instructions and tools per agent
- **Callback modes**: `STREAMING` for real-time updates or `BATCH` for final results
- **Human-in-the-loop planning**: Custom planner functions for interactive workflows

Python

```python
# Advanced customization example with human-in-the-loop
from agent_framework.openai import OpenAIChatClient
from agent_framework import MagenticBuilder, MagenticCallbackMode,
MagenticPlannerContext

# Assume we have researcher_agent, coder_agent, analyst_agent,
detailed_event_handler
# and get_human_input function defined elsewhere

async def custom_planner(context: MagenticPlannerContext) -> str:
    """Custom planner with human input for critical decisions."""
    if context.round_count > 5:
        # Request human input for complex decisions
        return await get_human_input(f"Next action for: {context.current_state}")
    return "Continue with automated planning"

workflow = (MagenticBuilder()
            .participants(
                researcher=researcher_agent,
                coder=coder_agent,
                analyst=analyst_agent
            )
            .with_standard_manager(
                chat_client=OpenAIChatClient(model_id="gpt-5"),
                max_round_count=15,      # Limit total rounds
                max_stall_count=2,       # Prevent infinite loops
                max_reset_count=1,       # Allow one reset on failure
                orchestrator_prompt="Custom orchestration instructions..."
            )
            .with_planner(custom_planner)  # Human-in-the-loop planning
            .on_event(detailed_event_handler, mode=MagenticCallbackMode.STREAMING)
            .build())
```

For detailed Magentic examples, see:

- Basic Magentic Workflow - Standard orchestrated multi-agent workflow
- Magentic with Checkpointing - Persistent orchestrated workflows

- Magentic Human Plan Update - Human-in-the-loop planning

Future Patterns

The Agent Framework roadmap includes several AutoGen patterns currently in development:

- **Swarm pattern**: Handoff-based agent coordination
- **SelectorGroupChat**: LLM-driven speaker selection

Human-in-the-Loop with Request Response

A key new feature in Agent Framework's `Workflow` is the concept of **request and response**, which allows workflows to pause execution and wait for external input before continuing. This capability is not present in AutoGen's `Team` abstraction and enables sophisticated human-in-the-loop patterns.

AutoGen Limitations

AutoGen's `Team` abstraction runs continuously once started and doesn't provide built-in mechanisms to pause execution for human input. Any human-in-the-loop functionality requires custom implementations outside the framework.

Agent Framework RequestInfoExecutor

Agent Framework provides `RequestInfoExecutor` - a workflow-native bridge that pauses the graph at a request for information, emits a `RequestInfoEvent` with a typed payload, and resumes execution only after the application supplies a matching `RequestResponse`.

Python

```python
from agent_framework import (
    RequestInfoExecutor, RequestInfoEvent, RequestInfoMessage,
    RequestResponse, WorkflowBuilder, WorkflowContext, executor
)
from dataclasses import dataclass
from typing_extensions import Never

# Assume we have agent_executor defined elsewhere

# Define typed request payload
@dataclass
class ApprovalRequest(RequestInfoMessage):
    """Request human approval for agent output."""
    content: str = ""
```

```
    agent_name: str = ""

# Workflow executor that requests human approval
@executor(id="reviewer")
async def approval_executor(
    agent_response: str,
    ctx: WorkflowContext[ApprovalRequest]
) -> None:
    # Request human input with structured data
    approval_request = ApprovalRequest(
        content=agent_response,
        agent_name="writer_agent"
    )
    await ctx.send_message(approval_request)

# Human feedback handler
@executor(id="processor")
async def process_approval(
    feedback: RequestResponse[ApprovalRequest, str],
    ctx: WorkflowContext[Never, str]
) -> None:
    decision = feedback.data.strip().lower()
    original_content = feedback.original_request.content

    if decision == "approved":
        await ctx.yield_output(f"APPROVED: {original_content}")
    else:
        await ctx.yield_output(f"REVISION NEEDED: {decision}")

# Build workflow with human-in-the-loop
hitl_executor = RequestInfoExecutor(id="request_approval")

workflow = (WorkflowBuilder()
            .add_edge(agent_executor, approval_executor)
            .add_edge(approval_executor, hitl_executor)
            .add_edge(hitl_executor, process_approval)
            .set_start_executor(agent_executor)
            .build())
```

Running Human-in-the-Loop Workflows

Agent Framework provides streaming APIs to handle the pause-resume cycle:

Python

```
from agent_framework import RequestInfoEvent, WorkflowOutputEvent

# Assume we have workflow defined from previous examples
async def run_with_human_input():
    pending_responses = None
    completed = False
```

```
    while not completed:
        # First iteration uses run_stream, subsequent use send_responses_streaming
        stream = (
            workflow.send_responses_streaming(pending_responses)
            if pending_responses
            else workflow.run_stream("initial input")
        )

        events = [event async for event in stream]
        pending_responses = None

        # Collect human requests and outputs
        for event in events:
            if isinstance(event, RequestInfoEvent):
                # Display request to human and collect response
                request_data = event.data  # ApprovalRequest instance
                print(f"Review needed: {request_data.content}")

                human_response = input("Enter 'approved' or revision notes: ")
                pending_responses = {event.request_id: human_response}

            elif isinstance(event, WorkflowOutputEvent):
                print(f"Final result: {event.data}")
                completed = True
```

For human-in-the-loop workflow examples, see:

- Guessing Game with Human Input - Interactive workflow with user feedback
- Workflow as Agent with Human Input - Nested workflows with human interaction

Checkpointing and Resuming Workflows

Another key advantage of Agent Framework's `Workflow` over AutoGen's `Team` abstraction is built-in support for checkpointing and resuming execution. This enables workflows to be paused, persisted, and resumed later from any checkpoint, providing fault tolerance and enabling long-running or asynchronous workflows.

AutoGen Limitations

AutoGen's `Team` abstraction does not provide built-in checkpointing capabilities. Any persistence or recovery mechanisms must be implemented externally, often requiring complex state management and serialization logic.

Agent Framework Checkpointing

Agent Framework provides comprehensive checkpointing through `FileCheckpointStorage` and the `with_checkpointing()` method on `WorkflowBuilder`. Checkpoints capture:

- **Executor state**: Local state for each executor using `ctx.set_state()`
- **Shared state**: Cross-executor state using `ctx.set_shared_state()`
- **Message queues**: Pending messages between executors
- **Workflow position**: Current execution progress and next steps

Python

```python
from agent_framework import (
    FileCheckpointStorage, WorkflowBuilder, WorkflowContext,
    Executor, handler
)
from typing_extensions import Never

class ProcessingExecutor(Executor):
    @handler
    async def process(self, data: str, ctx: WorkflowContext[str]) -> None:
        # Process the data
        result = f"Processed: {data.upper()}"
        print(f"Processing: '{data}' -> '{result}'")

        # Persist executor-local state
        prev_state = await ctx.get_state() or {}
        count = prev_state.get("count", 0) + 1
        await ctx.set_state({
            "count": count,
            "last_input": data,
            "last_output": result
        })

        # Persist shared state for other executors
        await ctx.set_shared_state("original_input", data)
        await ctx.set_shared_state("processed_output", result)

        await ctx.send_message(result)

class FinalizeExecutor(Executor):
    @handler
    async def finalize(self, data: str, ctx: WorkflowContext[Never, str]) -> None:
        result = f"Final: {data}"
        await ctx.yield_output(result)

# Configure checkpoint storage
checkpoint_storage = FileCheckpointStorage(storage_path="./checkpoints")
processing_executor = ProcessingExecutor(id="processing")
finalize_executor = FinalizeExecutor(id="finalize")

# Build workflow with checkpointing enabled
workflow = (WorkflowBuilder()
            .add_edge(processing_executor, finalize_executor)
```

```
            .set_start_executor(processing_executor)
            .with_checkpointing(checkpoint_storage=checkpoint_storage)  # Enable
checkpointing
            .build())

# Example usage (would be in async context)
async def checkpoint_example():
    # Run workflow - checkpoints are created automatically
    async for event in workflow.run_stream("input data"):
        print(f"Event: {event}")
```

Resuming from Checkpoints

Agent Framework provides APIs to list, inspect, and resume from specific checkpoints:

Python

```
from agent_framework import (
    RequestInfoExecutor, FileCheckpointStorage, WorkflowBuilder,
    Executor, WorkflowContext, handler
)
from typing_extensions import Never

class UpperCaseExecutor(Executor):
    @handler
    async def process(self, text: str, ctx: WorkflowContext[str]) -> None:
        result = text.upper()
        await ctx.send_message(result)

class ReverseExecutor(Executor):
    @handler
    async def process(self, text: str, ctx: WorkflowContext[Never, str]) -> None:
        result = text[::-1]
        await ctx.yield_output(result)

def create_workflow(checkpoint_storage: FileCheckpointStorage):
    """Create a workflow with two executors and checkpointing."""
    upper_executor = UpperCaseExecutor(id="upper")
    reverse_executor = ReverseExecutor(id="reverse")

    return (WorkflowBuilder()
            .add_edge(upper_executor, reverse_executor)
            .set_start_executor(upper_executor)
            .with_checkpointing(checkpoint_storage=checkpoint_storage)
            .build())

# Assume we have checkpoint_storage from previous examples
checkpoint_storage = FileCheckpointStorage(storage_path="./checkpoints")

async def checkpoint_resume_example():
    # List available checkpoints
    checkpoints = await checkpoint_storage.list_checkpoints()
```

```
    # Display checkpoint information
    for checkpoint in checkpoints:
        summary = RequestInfoExecutor.checkpoint_summary(checkpoint)
        print(f"Checkpoint {summary.checkpoint_id}: iteration=
{summary.iteration_count}")
        print(f"  Shared state: {checkpoint.shared_state}")
        print(f"  Executor states: {list(checkpoint.executor_states.keys())}")

    # Resume from a specific checkpoint
    if checkpoints:
        chosen_checkpoint_id = checkpoints[0].checkpoint_id

        # Create new workflow instance and resume
        new_workflow = create_workflow(checkpoint_storage)
        async for event in new_workflow.run_stream_from_checkpoint(
            chosen_checkpoint_id,
            checkpoint_storage=checkpoint_storage
        ):
            print(f"Resumed event: {event}")
```

Advanced Checkpointing Features

Checkpoint with Human-in-the-Loop Integration:

Checkpointing works seamlessly with human-in-the-loop workflows, allowing workflows to be paused for human input and resumed later:

Python

```
# Assume we have workflow, checkpoint_id, and checkpoint_storage from previous
examples
async def resume_with_responses_example():
    # Resume with pre-supplied human responses
    responses = {"request_id_123": "approved"}

    async for event in workflow.run_stream_from_checkpoint(
        checkpoint_id,
        checkpoint_storage=checkpoint_storage,
        responses=responses  # Pre-supply human responses
    ):
        print(f"Event: {event}")
```

Key Benefits

Compared to AutoGen, Agent Framework's checkpointing provides:

- **Automatic persistence**: No manual state management required

- **Granular recovery**: Resume from any superstep boundary
- **State isolation**: Separate executor-local and shared state
- **Human-in-the-loop integration**: Seamless pause-resume with human input
- **Fault tolerance**: Robust recovery from failures or interruptions

Practical Examples

For comprehensive checkpointing examples, see:

- Checkpoint with Resume - Basic checkpointing and interactive resume
- Checkpoint with Human-in-the-Loop - Persistent workflows with human approval gates
- Sub-workflow Checkpoint - Checkpointing nested workflows
- Magentic Checkpoint - Checkpointing orchestrated multi-agent workflows

---

Observability

Both AutoGen and Agent Framework provide observability capabilities, but with different approaches and features.

AutoGen Observability

AutoGen has native support for OpenTelemetry with instrumentation for:

- **Runtime tracing**: `SingleThreadedAgentRuntime` and `GrpcWorkerAgentRuntime`
- **Tool execution**: `BaseTool` with `execute_tool` spans following GenAI semantic conventions
- **Agent operations**: `BaseChatAgent` with `create_agent` and `invoke_agent` spans

Python

```python
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from autogen_core import SingleThreadedAgentRuntime

# Configure OpenTelemetry
tracer_provider = TracerProvider()
trace.set_tracer_provider(tracer_provider)

# Pass to runtime
runtime = SingleThreadedAgentRuntime(tracer_provider=tracer_provider)
```

Agent Framework Observability

Agent Framework provides comprehensive observability through multiple approaches:

- **Zero-code setup**: Automatic instrumentation via environment variables
- **Manual configuration**: Programmatic setup with custom parameters
- **Rich telemetry**: Agents, workflows, and tool execution tracking
- **Console output**: Built-in console logging and visualization

Python

```python
from agent_framework import ChatAgent
from agent_framework.observability import setup_observability
from agent_framework.openai import OpenAIChatClient

# Zero-code setup via environment variables
# Set ENABLE_OTEL=true
# Set OTLP_ENDPOINT=http://localhost:4317

# Or manual setup
setup_observability(
    otlp_endpoint="http://localhost:4317"
)

# Create client for the example
client = OpenAIChatClient(model_id="gpt-5")

async def observability_example():
    # Observability is automatically applied to all agents and workflows
    agent = ChatAgent(name="assistant", chat_client=client)
    result = await agent.run("Hello")  # Automatically traced
```

Key Differences:

- **Setup complexity**: Agent Framework offers simpler zero-code setup options
- **Scope**: Agent Framework provides broader coverage including workflow-level observability
- **Visualization**: Agent Framework includes built-in console output and development UI
- **Configuration**: Agent Framework offers more flexible configuration options

For detailed observability examples, see:

- Zero-code Setup - Environment variable configuration
- Manual Setup - Programmatic configuration
- Agent Observability - Single agent telemetry
- Workflow Observability - Multi-agent workflow tracing

Conclusion

This migration guide provides a comprehensive mapping between AutoGen and Microsoft Agent Framework, covering everything from basic agent creation to complex multi-agent workflows. Key takeaways for migration:

- **Single-agent migration** is straightforward, with similar APIs and enhanced capabilities in Agent Framework
- **Multi-agent patterns** require rethinking your approach from event-driven to data-flow based architectures, but if you already familiar with GraphFlow, the transition will be easier
- **Agent Framework offers** additional features like middleware, hosted tools, and typed workflows

For additional examples and detailed implementation guidance, refer to the Agent Framework samples directory.

Additional Sample Categories

The Agent Framework provides samples across several other important areas:

- **Threads**: Thread samples - Managing conversation state and context
- **Multimodal Input**: Multimodal samples - Working with images and other media types
- **Context Providers**: Context Provider samples - External context integration patterns

Next steps

Quickstart Guide

Semantic Kernel to Agent Framework Migration Guide

10/01/2025

Benefits of Microsoft Agent Framework compared to Semantic Kernel Agent Framework

- **Simplified API**: Reduced complexity and boilerplate code
- **Better Performance**: Optimized object creation and memory usage
- **Unified Interface**: Consistent patterns across different AI providers
- **Enhanced Developer Experience**: More intuitive and discoverable APIs

Key differences

Here is a summary of the key differences between the Semantic Kernel Agent Framework and the Microsoft Agent Framework to help you migrate your code.

1. Namespace Updates

Semantic Kernel

C#

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Agents;
```

Agent Framework

Agent Framework namespaces are under `Microsoft.Agents.AI`. Agent Framework uses the core AI message and content types from `Microsoft.Extensions.AI` for communication between components.

C#

```
using Microsoft.Extensions.AI;
using Microsoft.Agents.AI;
```

2. Agent Creation Simplification

Semantic Kernel

Every agent in Semantic Kernel depends on a `Kernel` instance and will have an empty `Kernel` if not provided.

C#

```
 Kernel kernel = Kernel
    .AddOpenAIChatClient(modelId, apiKey)
    .Build();

 ChatCompletionAgent agent = new() { Instructions = ParrotInstructions, Kernel =
kernel };
```

Azure AI Foundry requires an agent resource to be created in the cloud before creating a local agent class that uses it.

C#

```
PersistentAgentsClient azureAgentClient =
AzureAIAgent.CreateAgentsClient(azureEndpoint, new AzureCliCredential());

PersistentAgent definition = await
azureAgentClient.Administration.CreateAgentAsync(
    deploymentName,
    instructions: ParrotInstructions);

AzureAIAgent agent = new(definition, azureAgentClient);
```

Agent Framework

Agent creation in Agent Framework is made simpler with extensions provided by all main providers.

C#

```
AIAgent openAIAgent = chatClient.CreateAIAgent(instructions: ParrotInstructions);
AIAgent azureFoundryAgent = await
persistentAgentsClient.CreateAIAgentAsync(instructions: ParrotInstructions);
AIAgent openAIAssistantAgent = await
assistantClient.CreateAIAgentAsync(instructions: ParrotInstructions);
```

Additionally for hosted agent providers you can also use the `GetAIAgent` to retrieve an agent from an existing hosted agent.

C#

```
AIAgent azureFoundryAgent = await persistentAgentsClient.GetAIAgentAsync(agentId);
```

3. Agent Thread Creation

Semantic Kernel

The caller has to know the thread type and create it manually.

C#

```
// Create a thread for the agent conversation.
AgentThread thread = new OpenAIAssistantAgentThread(this.AssistantClient);
AgentThread thread = new AzureAIAgentThread(this.Client);
AgentThread thread = new OpenAIResponseAgentThread(this.Client);
```

Agent Framework

The agent is responsible for creating the thread.

C#

```
// New
AgentThread thread = agent.GetNewThread();
```

4. Hosted Agent Thread Cleanup

This case applies exclusively to a few AI providers that still provide hosted threads.

Semantic Kernel

Threads have a `self` deletion method

i.e: OpenAI Assistants Provider

C#

```
await thread.DeleteAsync();
```

Agent Framework

> **Note**
>
> OpenAI Responses introduced a new conversation model that simplifies how conversations are handled. This simplifies hosted thread management compared to the now deprecated OpenAI Assistants model. For more information see the OpenAI Assistants migration guide.

Agent Framework doesn't have a thread deletion API in the `AgentThread` type as not all providers support hosted threads or thread deletion and this will become more common as more providers shift to responses based architectures.

If you require thread deletion and the provider allows this, the caller **should** keep track of the created threads and delete them later when necessary via the provider's sdk.

i.e: OpenAI Assistants Provider

C#

```
await assistantClient.DeleteThreadAsync(thread.ConversationId);
```

5. Tool Registration

Semantic Kernel

In semantic kernel to expose a function as a tool you must:

1. Decorate the function with a `[KernelFunction]` attribute.
2. Have a `Plugin` class or use the `KernelPluginFactory` to wrap the function.
3. Have a `Kernel` to add your plugin to.
4. Pass the `Kernel` to the agent.

C#

```
KernelFunction function = KernelFunctionFactory.CreateFromMethod(GetWeather);
KernelPlugin plugin = KernelPluginFactory.CreateFromFunctions("KernelPluginName",
[function]);
Kernel kernel = ... // Create kernel
```

```
kernel.Plugins.Add(plugin);

ChatCompletionAgent agent = new() { Kernel = kernel, ... };
```

Agent Framework

In agent framework in a single call you can register tools directly in the agent creation process.

```
C#

AIAgent agent = chatClient.CreateAIAgent(tools:
[AIFunctionFactory.Create(GetWeather)]);
```

6. Agent Non-Streaming Invocation

Key differences can be seen in the method names from `Invoke` to `Run`, return types and parameters `AgentRunOptions`.

Semantic Kernel

The Non-Streaming uses a streaming pattern `IAsyncEnumerable<AgentResponseItem<ChatMessageContent>>` for returning multiple agent messages.

```
C#

await foreach (AgentResponseItem<ChatMessageContent> result in
agent.InvokeAsync(userInput, thread, agentOptions))
{
    Console.WriteLine(result.Message);
}
```

Agent Framework

The Non-Streaming returns a single `AgentRunResponse` with the agent response that can contain multiple messages. The text result of the run is available in `AgentRunResponse.Text` or `AgentRunResponse.ToString()`. All messages created as part of the response is returned in the `AgentRunResponse.Messages` list. This may include tool call messages, function results, reasoning updates and final results.

```
C#
```

```
AgentRunResponse agentResponse = await agent.RunAsync(userInput, thread);
```

7. Agent Streaming Invocation

Key differences in the method names from `Invoke` to `Run`, return types and parameters `AgentRunOptions`.

Semantic Kernel

C#

```
await foreach (StreamingChatMessageContent update in
agent.InvokeStreamingAsync(userInput, thread))
{
    Console.Write(update);
}
```

Agent Framework

Similar streaming API pattern with the key difference being that it returns `AgentRunResponseUpdate` objects including more agent related information per update.

All updates produced by any service underlying the AIAgent is returned. The textual result of the agent is available by concatenating the `AgentRunResponse.Text` values.

C#

```
await foreach (AgentRunResponseUpdate update in agent.RunStreamingAsync(userInput,
thread))
{
    Console.Write(update); // Update is ToString() friendly
}
```

8. Tool Function Signatures

Problem: SK plugin methods need `[KernelFunction]` attributes

C#

```
public class MenuPlugin
{
    [KernelFunction] // Required for SK
```

```
    public static MenuItem[] GetMenu() => ...;
}
```

Solution: AF can use methods directly without attributes

C#

```
public class MenuTools
{
    [Description("Get menu items")] // Optional description
    public static MenuItem[] GetMenu() => ...;
}
```

9. Options Configuration

Problem: Complex options setup in SK

C#

```
OpenAIPromptExecutionSettings settings = new() { MaxTokens = 1000 };
AgentInvokeOptions options = new() { KernelArguments = new(settings) };
```

Solution: Simplified options in AF

C#

```
ChatClientAgentRunOptions options = new(new() { MaxOutputTokens = 1000 });
```

> ⓘ **Important**
>
> This example shows passing implementation specific options to a `ChatClientAgent`. Not all `AIAgents` support `ChatClientAgentRunOptions`. `ChatClientAgent` is provided to build agents based on underlying inference services, and therefore supports inference options like `MaxOutputTokens`.

10. Dependency Injection

Semantic Kernel

A `Kernel` registration is required in the service container to be able to create an agent as every agent abstractions needs to be initialized with a `Kernel` property.

Semantic Kernel uses the `Agent` type as the base abstraction class for agents.

```
C#
```

```
services.AddKernel().AddProvider(...);
serviceContainer.AddKeyedSingleton<SemanticKernel.Agents.Agent>(
    TutorName,
    (sp, key) =>
        new ChatCompletionAgent()
        {
            // Passing the kernel is required
            Kernel = sp.GetRequiredService<Kernel>(),
        });
```

Agent Framework

The Agent framework provides the `AIAgent` type as the base abstraction class.

```
C#
```

```
services.AddKeyedSingleton<AIAgent>(() => client.CreateAIAgent(...));
```

11. Agent Type Consolidation

Semantic Kernel

Semantic kernel provides specific agent classes for various services, e.g.

- `ChatCompletionAgent` for use with chat-completion-based inference services.
- `OpenAIAssistantAgent` for use with the OpenAI Assistants service.
- `AzureAIAgent` for use with the Azure AI Foundry Agents service.

Agent Framework

The agent framework supports all the above mentioned services via a single agent type, `ChatClientAgent`.

`ChatClientAgent` can be used to build agents using any underlying service that provides an SDK implementing the `Microsoft.Extensions.AI.IChatClient` interface.

Next steps

Quickstart Guide

Semantic Kernel to Agent Framework Migration Samples

10/01/2025

See the Agent Framework repository for detailed per agent type code samples showing the the Agent Framework equivalent code for Semantic Kernel features.