A2A Agents

10/02/2025

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 Python packages to your project.

Bash

```
pip install agent-framework-a2a
```

Creating an A2A Agent

First, let's look at a scenario where we use the well known agent card location. We pass the base URL 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.json`.

First, create an `A2ACardResolver` with the URL of the remote A2A agent host.

Python

```
import httpx
from a2a.client import A2ACardResolver

# Create httpx client for HTTP communication
async with httpx.AsyncClient(timeout=60.0) as http_client:
    resolver = A2ACardResolver(httpx_client=http_client,
base_url="https://your-a2a-agent-host")
```

Get the agent card and create an instance of the `A2AAgent` for the remote A2A agent.

Python

```
from agent_framework.a2a import A2AAgent

# Get agent card from the well-known location
agent_card = await resolver.get_agent_card(relative_card_path="/.well-
known/agent.json")

# Create A2A agent instance
agent = A2AAgent(
```

```
    name=agent_card.name,
    description=agent_card.description,
    agent_card=agent_card,
    url="https://your-a2a-agent-host"
)
```

Creating an A2A Agent using URL

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 `A2AAgent` directly with the URL of the agent.

Python

```python
from agent_framework.a2a import A2AAgent

# Create A2A agent with direct URL configuration
agent = A2AAgent(
    name="My A2A Agent",
    description="A directly configured A2A agent",
    url="https://your-a2a-agent-host/echo"
)
```

Using the Agent

The A2A agent 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

Adding Memory to an Agent

10/02/2025

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

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

Prerequisites

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

Creating a ContextProvider

`ContextProvider` is an abstract class that you can inherit from, and which can be associated with an `AgentThread` for a `ChatAgent`. 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 `ContextProvider` class has two methods that you can override to run custom logic before and after the agent invokes the underlying inference service:

- `invoking` - called before the agent invokes the underlying inference service. You can provide additional context to the agent by returning a `Context` 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.
- `invoked` - 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

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

The `ContextProvider` 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 serialization for the `ContextProvider` class. You also need to provide a constructor that can restore state from serialized data when resuming a thread.

Sample ContextProvider 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.

Python

```python
from pydantic import BaseModel

class UserInfo(BaseModel):
    name: str | None = None
    age: int | None = None
```

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

1. It uses a chat client 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 no 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.

Python

```python
from agent_framework import ContextProvider, Context, InvokedContext,
InvokingContext, ChatAgent, ChatClientProtocol
```

```python
class UserInfoMemory(ContextProvider):
    def __init__(self, chat_client: ChatClientProtocol, user_info: UserInfo
| None = None, **kwargs: Any):
        """Create the memory.

        If you pass in kwargs, they will be attempted to be used to create
a UserInfo object.
        """

        self._chat_client = chat_client
        if user_info:
            self.user_info = user_info
        elif kwargs:
            self.user_info = UserInfo.model_validate(kwargs)
        else:
            self.user_info = UserInfo()

    async def invoked(
        self,
        request_messages: ChatMessage | Sequence[ChatMessage],
        response_messages: ChatMessage | Sequence[ChatMessage] | None =
None,
        invoke_exception: Exception | None = None,
        **kwargs: Any,
    ) -> None:
        """Extract user information from messages after each agent call."""
        # Check if we need to extract user info from user messages
        user_messages = [msg for msg in request_messages if hasattr(msg,
"role") and msg.role.value == "user"]

        if (self.user_info.name is None or self.user_info.age is None) and
user_messages:
            try:
                # Use the chat client to extract structured information
                result = await self._chat_client.get_response(
                    messages=request_messages,
                    chat_options=ChatOptions(
                        instructions="Extract the user's name and age from
the message if present. If not present return nulls.",
                        response_format=UserInfo,
                    ),
                )

                # Update user info with extracted data
                if result.value:
                    if self.user_info.name is None and result.value.name:
                        self.user_info.name = result.value.name
                    if self.user_info.age is None and result.value.age:
                        self.user_info.age = result.value.age

            except Exception:
                pass  # Failed to extract, continue without updating

    async def invoking(self, messages: ChatMessage |
```

```
MutableSequence[ChatMessage], **kwargs: Any) -> Context:
        """Provide user information context before each agent call."""
        instructions: list[str] = []

        if self.user_info.name is None:
            instructions.append(
                "Ask the user for their name and politely decline to answer
any questions until they provide it."
            )
        else:
            instructions.append(f"The user's name is
{self.user_info.name}.")

        if self.user_info.age is None:
            instructions.append(
                "Ask the user for their age and politely decline to answer
any questions until they provide it."
            )
        else:
            instructions.append(f"The user's age is {self.user_info.age}.")

        # Return context with additional instructions
        return Context(instructions=" ".join(instructions))

    def serialize(self) -> str:
        """Serialize the user info for thread persistence."""
        return self.user_info.model_dump_json()
```

Using the ContextProvider with an agent

To use the custom `ContextProvider`, you need to provide the instantiated `ContextProvider` when creating the agent.

When creating a `ChatAgent` you can provide the `context_providers` parameter to attach the memory component to the agent.

Python

```
import asyncio
from agent_framework import ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with AzureCliCredential() as credential:
        chat_client = AzureAIAgentClient(async_credential=credential)

        # Create the memory provider
        memory_provider = UserInfoMemory(chat_client)
```

```
        # Create the agent with memory
        async with ChatAgent(
            chat_client=chat_client,
            instructions="You are a friendly assistant. Always address the
user by their name.",
            context_providers=memory_provider,
        ) as agent:
            # Create a new thread for the conversation
            thread = agent.get_new_thread()

            print(await agent.run("Hello, what is the square root of 9?",
thread=thread))
            print(await agent.run("My name is Ruaidhrí", thread=thread))
            print(await agent.run("I am 20 years old", thread=thread))

            # Access the memory component via the thread's contex-
t_providers attribute and inspect the memories
            user_info_memory = thread.context_provider.providers[0]
            if user_info_memory:
                print()
                print(f"MEMORY - User Name: {user_info_memory.user_info.-
name}")
                print(f"MEMORY - User Age:
{user_info_memory.user_info.age}")


if __name__ == "__main__":
    asyncio.run(main())
```

Next steps

Create a simple workflow

Adding Middleware to Agents

10/02/2025

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.

Step 1: Create a Simple Agent

First, let's create a basic agent:

Python

```
import asyncio
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    credential = AzureCliCredential()

    async with
AzureAIAgentClient(async_credential=credential).create_agent(
        name="GreetingAgent",
        instructions="You are a friendly greeting assistant.",
    ) as agent:
        result = await agent.run("Hello!")
        print(result.text)

if __name__ == "__main__":
    asyncio.run(main())
```

Step 2: Create Your Middleware

Create a simple logging middleware to see when your agent runs:

Python

```
from agent_framework import AgentRunContext

async def logging_agent_middleware(
    context: AgentRunContext,
    next: Callable[[AgentRunContext], Awaitable[None]],
) -> None:
    """Simple middleware that logs agent execution."""
    print("Agent starting...")

    # Continue to agent execution
    await next(context)
```

```
    print("Agent finished!")
```

Step 3: Add Middleware to Your Agent

Add the middleware when creating your agent:

Python

```
async def main():
    credential = AzureCliCredential()

    async with
AzureAIAgentClient(async_credential=credential).create_agent(
        name="GreetingAgent",
        instructions="You are a friendly greeting assistant.",
        middleware=logging_agent_middleware,  # Add your middleware here
    ) as agent:
        result = await agent.run("Hello!")
        print(result.text)
```

Step 4: Create Function Middleware

If your agent uses functions, you can intercept function calls:

Python

```
from agent_framework import FunctionInvocationContext

def get_time():
    """Get the current time."""
    from datetime import datetime
    return datetime.now().strftime("%H:%M:%S")

async def logging_function_middleware(
    context: FunctionInvocationContext,
    next: Callable[[FunctionInvocationContext], Awaitable[None]],
) -> None:
    """Middleware that logs function calls."""
    print(f"Calling function: {context.function.name}")

    await next(context)

    print(f"Function result: {context.result}")

# Add both the function and middleware to your agent
async with AzureAIAgentClient(async_credential=credential).create_agent(
    name="TimeAgent",
```

```
    instructions="You can tell the current time.",
    tools=[get_time],
    middleware=[logging_function_middleware],
) as agent:
    result = await agent.run("What time is it?")
```

Step 5: Use Run-Level Middleware

You can also add middleware for specific runs:

Python

```
# Use middleware for this specific run only
result = await agent.run(
    "This is important!",
    middleware=[logging_function_middleware]
)
```

What's Next?

For more advanced scenarios, check out the Agent Middleware User Guide which covers:

- Different types of middleware (agent, function, chat)
- Class-based middleware for complex scenarios
- Middleware termination and result overrides
- Advanced middleware patterns and best practices

Agent based on any IChatClient

10/02/2025

The Microsoft Agent Framework supports creating agents for any inference service that provides a chat client implementation compatible with the `ChatClientProtocol`. 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.

Getting Started

Add the required Python packages to your project.

Bash

```
pip install agent-framework
```

You may also need to add packages for specific chat client implementations you want to use:

Bash

```
# For Azure AI
pip install agent-framework-azure-ai

# For custom implementations
# Install any required dependencies for your custom client
```

Built-in Chat Clients

The framework provides several built-in chat client implementations:

OpenAI Chat Client

Python

```
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIChatClient

# Create agent using OpenAI
agent = ChatAgent(
    chat_client=OpenAIChatClient(model_id="gpt-4o"),
    instructions="You are a helpful assistant.",
```

```
    name="OpenAI Assistant"
)
```

Azure OpenAI Chat Client

Python

```python
from agent_framework import ChatAgent
from agent_framework.azure import AzureOpenAIChatClient

# Create agent using Azure OpenAI
agent = ChatAgent(
    chat_client=AzureOpenAIChatClient(
        model_id="gpt-4o",
        endpoint="https://your-resource.openai.azure.com/",
        api_key="your-api-key"
    ),
    instructions="You are a helpful assistant.",
    name="Azure OpenAI Assistant"
)
```

Azure AI Agent Client

Python

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

# Create agent using Azure AI
async with AzureCliCredential() as credential:
    agent = ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        instructions="You are a helpful assistant.",
        name="Azure AI Assistant"
    )
```

> **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 if you plan to use tools.

Using the Agent

The agent 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

Agent Framework Tutorials

10/02/2025

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.

Agent Framework User Guide

10/02/2025

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.

Agent Memory

10/02/2025

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.

In-Memory Storage (Default)

The simplest form of memory where conversation history is stored in memory during the application runtime. This is the default behavior and requires no additional configuration.

Python

```python
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIChatClient

# Default behavior - uses in-memory storage
agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant."
)

# Conversation history is maintained in memory for this thread
thread = agent.get_new_thread()

response = await agent.run("Hello, my name is Alice", thread=thread)
```

Persistent Message Stores

For applications that need to persist conversation history across sessions, the framework provides `ChatMessageStore` implementations:

Built-in ChatMessageStore

The default in-memory implementation that can be serialized:

Python

```python
from agent_framework import ChatMessageStore

# Create a custom message store
def create_message_store():
    return ChatMessageStore()

agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant.",
    chat_message_store_factory=create_message_store
)
```

Redis Message Store

For production applications requiring persistent storage:

Python

```python
from agent_framework.redis import RedisChatMessageStore

def create_redis_store():
    return RedisChatMessageStore(
        redis_url="redis://localhost:6379",
        thread_id="user_session_123",
        max_messages=100  # Keep last 100 messages
    )

agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant.",
    chat_message_store_factory=create_redis_store
)
```

Custom Message Store

You can implement your own storage backend by implementing the `ChatMessageStoreProtocol`:

Python

```python
from agent_framework import ChatMessage, ChatMessageStoreProtocol
from typing import Any
from collections.abc import Sequence
```

```python
class DatabaseMessageStore(ChatMessageStoreProtocol):
    def __init__(self, connection_string: str):
        self.connection_string = connection_string
        self._messages: list[ChatMessage] = []

    async def add_messages(self, messages: Sequence[ChatMessage]) -> None:
        """Add messages to database."""
        # Implement database insertion logic
        self._messages.extend(messages)

    async def list_messages(self) -> list[ChatMessage]:
        """Retrieve messages from database."""
        # Implement database query logic
        return self._messages

    async def serialize(self, **kwargs: Any) -> Any:
        """Serialize store state for persistence."""
        return {"connection_string": self.connection_string}

    async def update_from_state(self, serialized_store_state: Any,
**kwargs: Any) -> None:
        """Update store from serialized state."""
        if serialized_store_state:
            self.connection_string = serialized_store_state["connec-
tion_string"]
```

> **Tip**
>
> For a detailed example on how to create a custom message store, see the Storing Chat History in 3rd Party Storage tutorial.

Context Providers (Dynamic Memory)

Context providers enable sophisticated memory patterns by injecting relevant context before each agent invocation:

Basic Context Provider

Python

```python
from agent_framework import ContextProvider, Context, ChatMessage
from collections.abc import MutableSequence
from typing import Any

class UserPreferencesMemory(ContextProvider):
    def __init__(self):
        self.preferences = {}
```

```
    async def invoking(self, messages: ChatMessage |
MutableSequence[ChatMessage], **kwargs: Any) -> Context:
        """Provide user preferences before each invocation."""
        if self.preferences:
            preferences_text = ", ".join([f"{k}: {v}" for k, v in self-
.preferences.items()])
            instructions = f"User preferences: {preferences_text}"
            return Context(instructions=instructions)
        return Context()

    async def invoked(
        self,
        request_messages: ChatMessage | Sequence[ChatMessage],
        response_messages: ChatMessage | Sequence[ChatMessage] | None =
None,
        invoke_exception: Exception | None = None,
        **kwargs: Any,
    ) -> None:
        """Extract and store user preferences from the conversation."""
        # Implement preference extraction logic
        pass
```

> **Tip**
>
> For a detailed example on how to create a custom memory component, see the Adding Memory to an Agent tutorial.

External Memory Services

The framework supports integration with specialized memory services like Mem0:

Python

```
from agent_framework.mem0 import Mem0Provider

# Using Mem0 for advanced memory capabilities
memory_provider = Mem0Provider(
    api_key="your-mem0-api-key",
    user_id="user_123",
    application_id="my_app"
)

agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant with memory.",
    context_providers=memory_provider
)
```

Thread Serialization and Persistence

The framework supports serializing entire thread states for persistence across application restarts:

Python

```python
import json

# Create agent and thread
agent = ChatAgent(chat_client=OpenAIChatClient())
thread = agent.get_new_thread()

# Have conversation
await agent.run("Hello, my name is Alice", thread=thread)

# Serialize thread state
serialized_thread = await thread.serialize()
# Save to file/database
with open("thread_state.json", "w") as f:
    json.dump(serialized_thread, f)

# Later, restore the thread
with open("thread_state.json", "r") as f:
    thread_data = json.load(f)

restored_thread = await agent.deserialize_thread(thread_data)
# Continue conversation with full context
await agent.run("What's my name?", thread=restored_thread)
```

Next steps

Agent Observability

Agent Middleware

10/02/2025

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.

Function-Based Middleware

Function-based middleware is the simplest way to implement middleware using async functions. This approach is ideal for stateless operations and provides a lightweight solution for common middleware scenarios.

Agent Middleware

Agent middleware intercepts and modifies agent run execution. It uses the `AgentRunContext` which contains:

- `agent`: The agent being invoked
- `messages`: List of chat messages in the conversation
- `is_streaming`: Boolean indicating if the response is streaming
- `metadata`: Dictionary for storing additional data between middleware
- `result`: The agent's response (can be modified)
- `terminate`: Flag to stop further processing
- `kwargs`: Additional keyword arguments passed to the agent run method

The `next` callable continues the middleware chain or executes the agent if it's the last middleware.

Here's a simple logging example with logic before and after `next` callable:

Python

```
async def logging_agent_middleware(
    context: AgentRunContext,
    next: Callable[[AgentRunContext], Awaitable[None]],
) -> None:
    """Agent middleware that logs execution timing."""
    # Pre-processing: Log before agent execution
    print("[Agent] Starting execution")

    # Continue to next middleware or agent execution
```

```
    await next(context)

    # Post-processing: Log after agent execution
    print("[Agent] Execution completed")
```

Function Middleware

Function middleware intercepts function calls within agents. It uses the `FunctionInvocationContext` which contains:

- `function`: The function being invoked
- `arguments`: The validated arguments for the function
- `metadata`: Dictionary for storing additional data between middleware
- `result`: The function's return value (can be modified)
- `terminate`: Flag to stop further processing
- `kwargs`: Additional keyword arguments passed to the chat method that invoked this function

The `next` callable continues to the next middleware or executes the actual function.

Here's a simple logging example with logic before and after `next` callable:

Python

```
async def logging_function_middleware(
    context: FunctionInvocationContext,
    next: Callable[[FunctionInvocationContext], Awaitable[None]],
) -> None:
    """Function middleware that logs function execution."""
    # Pre-processing: Log before function execution
    print(f"[Function] Calling {context.function.name}")

    # Continue to next middleware or function execution
    await next(context)

    # Post-processing: Log after function execution
    print(f"[Function] {context.function.name} completed")
```

Chat Middleware

Chat middleware intercepts chat requests sent to AI models. It uses the `ChatContext` which contains:

- `chat_client`: The chat client being invoked

- `messages`: List of messages being sent to the AI service
- `chat_options`: The options for the chat request
- `is_streaming`: Boolean indicating if this is a streaming invocation
- `metadata`: Dictionary for storing additional data between middleware
- `result`: The chat response from the AI (can be modified)
- `terminate`: Flag to stop further processing
- `kwargs`: Additional keyword arguments passed to the chat client

The `next` callable continues to the next middleware or sends the request to the AI service.

Here's a simple logging example with logic before and after `next` callable:

Python

```python
async def logging_chat_middleware(
    context: ChatContext,
    next: Callable[[ChatContext], Awaitable[None]],
) -> None:
    """Chat middleware that logs AI interactions."""
    # Pre-processing: Log before AI call
    print(f"[Chat] Sending {len(context.messages)} messages to AI")

    # Continue to next middleware or AI service
    await next(context)

    # Post-processing: Log after AI response
    print("[Chat] AI response received")
```

Function Middleware Decorators

Decorators provide explicit middleware type declaration without requiring type annotations. They're helpful when:

- You don't use type annotations
- You need explicit middleware type declaration
- You want to prevent type mismatches

Python

```python
from agent_framework import agent_middleware, function_middleware,
chat_middleware

@agent_middleware  # Explicitly marks as agent middleware
async def simple_agent_middleware(context, next):
    """Agent middleware with decorator - types are inferred."""
    print("Before agent execution")
```

```
    await next(context)
    print("After agent execution")

@function_middleware  # Explicitly marks as function middleware
async def simple_function_middleware(context, next):
    """Function middleware with decorator - types are inferred."""
    print(f"Calling function: {context.function.name}")
    await next(context)
    print("Function call completed")

@chat_middleware  # Explicitly marks as chat middleware
async def simple_chat_middleware(context, next):
    """Chat middleware with decorator - types are inferred."""
    print(f"Processing {len(context.messages)} chat messages")
    await next(context)
    print("Chat processing completed")
```

Class-Based Middleware

Class-based middleware is useful for stateful operations or complex logic that benefits from object-oriented design patterns.

Agent Middleware Class

Class-based agent middleware uses a `process` method that has the same signature and behavior as function-based middleware. The `process` method receives the same `context` and `next` parameters and is invoked in exactly the same way.

Python

```
from agent_framework import AgentMiddleware, AgentRunContext

class LoggingAgentMiddleware(AgentMiddleware):
    """Agent middleware that logs execution."""

    async def process(
        self,
        context: AgentRunContext,
        next: Callable[[AgentRunContext], Awaitable[None]],
    ) -> None:
        # Pre-processing: Log before agent execution
        print("[Agent Class] Starting execution")

        # Continue to next middleware or agent execution
        await next(context)

        # Post-processing: Log after agent execution
        print("[Agent Class] Execution completed")
```

Function Middleware Class

Class-based function middleware also uses a `process` method with the same signature and behavior as function-based middleware. The method receives the same `context` and `next` parameters.

Python

```python
from agent_framework import FunctionMiddleware, FunctionInvocationContext

class LoggingFunctionMiddleware(FunctionMiddleware):
    """Function middleware that logs function execution."""

    async def process(
        self,
        context: FunctionInvocationContext,
        next: Callable[[FunctionInvocationContext], Awaitable[None]],
    ) -> None:
        # Pre-processing: Log before function execution
        print(f"[Function Class] Calling {context.function.name}")

        # Continue to next middleware or function execution
        await next(context)

        # Post-processing: Log after function execution
        print(f"[Function Class] {context.function.name} completed")
```

Chat Middleware Class

Class-based chat middleware follows the same pattern with a `process` method that has identical signature and behavior to function-based chat middleware.

Python

```python
from agent_framework import ChatMiddleware, ChatContext

class LoggingChatMiddleware(ChatMiddleware):
    """Chat middleware that logs AI interactions."""

    async def process(
        self,
        context: ChatContext,
        next: Callable[[ChatContext], Awaitable[None]],
    ) -> None:
        # Pre-processing: Log before AI call
        print(f"[Chat Class] Sending {len(context.messages)} messages to
AI")

        # Continue to next middleware or AI service
```

```
        await next(context)

        # Post-processing: Log after AI response
        print("[Chat Class] AI response received")
```

Middleware Registration

Middleware can be registered at two levels with different scopes and behaviors.

Agent-Level vs Run-Level Middleware

Python

```
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

# Agent-level middleware: Applied to ALL runs of the agent
async with AzureAIAgentClient(async_credential=credential).create_agent(
    name="WeatherAgent",
    instructions="You are a helpful weather assistant.",
    tools=get_weather,
    middleware=[
        SecurityAgentMiddleware(),  # Applies to all runs
        TimingFunctionMiddleware(),  # Applies to all runs
    ],
) as agent:

    # This run uses agent-level middleware only
    result1 = await agent.run("What's the weather in Seattle?")

    # This run uses agent-level + run-level middleware
    result2 = await agent.run(
        "What's the weather in Portland?",
        middleware=[  # Run-level middleware (this run only)
            logging_chat_middleware,
        ]
    )

    # This run uses agent-level middleware only (no run-level)
    result3 = await agent.run("What's the weather in Vancouver?")
```

Key Differences:

- **Agent-level**: Persistent across all runs, configured once when creating the agent
- **Run-level**: Applied only to specific runs, allows per-request customization
- **Execution Order**: Agent middleware (outermost) → Run middleware (innermost) → Agent execution

Middleware Termination

Middleware can terminate execution early using `context.terminate`. This is useful for security checks, rate limiting, or validation failures.

Python

```python
async def blocking_middleware(
    context: AgentRunContext,
    next: Callable[[AgentRunContext], Awaitable[None]],
) -> None:
    """Middleware that blocks execution based on conditions."""
    # Check for blocked content
    last_message = context.messages[-1] if context.messages else None
    if last_message and last_message.text:
        if "blocked" in last_message.text.lower():
            print("Request blocked by middleware")
            context.terminate = True
            return

    # If no issues, continue normally
    await next(context)
```

What termination means:

- Setting `context.terminate = True` signals that processing should stop
- You can provide a custom result before terminating to give users feedback
- The agent execution is completely skipped when middleware terminates

Middleware Result Override

Middleware can override results in both non-streaming and streaming scenarios, allowing you to modify or completely replace agent responses.

The result type in `context.result` depends on whether the agent invocation is streaming or non-streaming:

- **Non-streaming**: `context.result` contains an `AgentRunResponse` with the complete response
- **Streaming**: `context.result` contains an async generator that yields `AgentRunResponseUpdate` chunks

You can use `context.is_streaming` to differentiate between these scenarios and handle result overrides appropriately.

Python

```
async def weather_override_middleware(
    context: AgentRunContext,
    next: Callable[[AgentRunContext], Awaitable[None]]
) -> None:
    """Middleware that overrides weather results for both streaming and
non-streaming."""

    # Execute the original agent logic
    await next(context)

    # Override results if present
    if context.result is not None:
        custom_message_parts = [
            "Weather Override: ",
            "Perfect weather everywhere today! ",
            "22°C with gentle breezes. ",
            "Great day for outdoor activities!"
        ]

        if context.is_streaming:
            # Streaming override
            async def override_stream() ->
AsyncIterable[AgentRunResponseUpdate]:
                for chunk in custom_message_parts:
                    yield AgentRunResponseUpdate(contents=
[TextContent(text=chunk)])

            context.result = override_stream()
        else:
            # Non-streaming override
            custom_message = "".join(custom_message_parts)
            context.result = AgentRunResponse(
                messages=[ChatMessage(role=Role.ASSISTANT, text=custom_mes-
sage)]
            )
```

This middleware approach allows you to implement sophisticated response transformation, content filtering, result enhancement, and streaming customization while keeping your agent logic clean and focused.

Next steps

Agent Memory

Agent Observability

10/02/2025

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 in your python application, you do not need to install anything extra, by default the following package are installed:

text

```
"opentelemetry-api",
"opentelemetry-sdk",
"azure-monitor-opentelemetry",
"azure-monitor-opentelemetry-exporter",
"opentelemetry-exporter-otlp-proto-grpc",
"opentelemetry-semantic-conventions-ai",
```

The easiest way to enable observability is to setup using the environment variables below. After those have been set, all you need to do is call at the start of your program:

Python

```
from agent_framework.observability import setup_observability

setup_observability()
```

This will take the environment variables into account and setup observability accordingly, it will set the global tracer provider and meter provider, so you can start using it right away, for instance to create custom spans or metrics:

Python

```python
from agent_framework.observability import get_tracer, get_meter

tracer = get_tracer()
meter = get_meter()
with tracer.start_as_current_span("my_custom_span"):
    # do something
    pass
counter = meter.create_counter("my_custom_counter")
counter.add(1, {"key": "value"})
```

Those are wrappers of the OpenTelemetry API, that will return a tracer or meter from the global provider, but with `agent_framework` set as the instrumentation library name, unless you override the name.

For `otlp_endpoints`, these will be created as a OTLPExporter, one each for span, metrics and logs. The `connection_string` for Application Insights will be used to create an AzureMonitorTraceExporter, AzureMonitorMetricExporter and AzureMonitorLogExporter.

Environment variables

The easiest way to enable observability for your application is to set the following environment variables:

- ENABLE_OTEL Default is `false`, set to `true` to enable OpenTelemetry This is needed for the basic setup, but also to visualize the workflows.
- ENABLE_SENSITIVE_DATA Default is `false`, set to `true` to enable logging of sensitive data, such as prompts, responses, function call arguments and results. This is needed if you want to see the actual prompts and responses in your traces. Be careful with this setting, as it may expose sensitive data in your logs.
- OTLP_ENDPOINT Default is `None`, set to your host for otel, often: `http://localhost:4317` This can be used for any compliant OTLP endpoint, such as OpenTelemetry Collector , Aspire Dashboard or any other OTLP compliant endpoint.
- APPLICATIONINSIGHTS_CONNECTION_STRING Default is `None`, set to your Application Insights connection string to export to Azure Monitor. You can find the connection string in the Azure portal, in the "Overview" section of your Application Insights resource.
- VS_CODE_EXTENSION_PORT Default is `4317`, set to the port the AI Toolkit or AzureAI Foundry VS Code extension is running on.

Programmatic setup

If you prefer to set up observability programmatically, you can do so by calling the `setup_observability` function with the desired configuration options:

Python

```python
from agent_framework.observability import setup_observability

setup_observability(
    enable_sensitive_data=True,
    otlp_endpoint="http://localhost:4317",
    applicationinsights_connection_string="InstrumentationKey=your_instru-
mentation_key",
    vs_code_extension_port=4317
)
```

This will take the provided configuration options and set up observability accordingly. It will assume you mean to enable the tracing, so `enable_otel` is implicitly set to `True`. If you also have endpoints or connection strings set via environment variables, those will also be created, and we check if there is no doubling.

Custom exporters

If you want to have different exporters, then the standard ones above, or if you want to customize the setup further, you can do so by creating your own tracer provider and meter provider, and then passing those to the `setup_observability` function, for example:

Python

```python
from agent_framework.observability import setup_observability
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter

custom_span_exporter = OTLPSpanExporter(endpoint="http://localhost:4317",
timeout=5, compression=Compression.Gzip)

setup_observability(exporters=[custom_span_exporter])
```

Azure AI Foundry setup

Azure AI Foundry has built-in support for tracing, with a really great visualization for your spans.

When you have a Azure AI Foundry project setup with a Application Insights resource, you can do the following:

Python

```python
from agent_framework.azure import AzureAIAgentClient
from azure.identity import AzureCliCredential

agent_client = AzureAIAgentClient(credential=AzureCliCredential(), projec-
t_endpoint="https://<your-project>.foundry.azure.com")

await agent_client.setup_azure_ai_observability()
```

This is a convenience method, that will use the project client, to get the Application Insights connection string, and then call `setup_observability` with that connection string.

Spans and metrics

Once everything is setup, you will start seeing spans and metrics being created automatically for you, the spans are:

- `invoke_agent <agent_name>`: This is the top level span for each agent invocation, it will contain all other spans as children.
- `chat <model_name>`: This span is created when the agent calls the underlying chat model, it will contain the prompt and response as attributes, if `enable_sensitive_data` is set to `True`.
- `execute_tool <function_name>`: This span is created when the agent calls a function tool, it will contain the function arguments and result as attributes, if `enable_sensitive_data` is set to `True`.

The metrics that are created are:

- For the chat client and `chat` operations:
  - `gen_ai.client.operation.duration` (histogram): This metric measures the duration of each operation, in seconds.
  - `gen_ai.client.token.usage` (histogram): This metric measures the token usage, in number of tokens.
- For function invocation during the `execute_tool` operations:
  - `agent_framework.function.invocation.duration` (histogram): This metric measures the duration of each function execution, in seconds.

Example trace output

When you run an agent with observability enabled, you'll see trace data similar to the following console output:

text


```
{
    "name": "invoke_agent Joker",
    "context": {
        "trace_id": "0xf2258b51421fe9cf4c0bd428c87b1ae4",
        "span_id": "0x2cad6fc139dcf01d",
        "trace_state": "[]"
    },
    "kind": "SpanKind.CLIENT",
    "parent_id": null,
    "start_time": "2025-09-25T11:00:48.663688Z",
    "end_time": "2025-09-25T11:00:57.271389Z",
    "status": {
        "status_code": "UNSET"
    },
    "attributes": {
        "gen_ai.operation.name": "invoke_agent",
        "gen_ai.system": "openai",
        "gen_ai.agent.id": "Joker",
        "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
    }
}
```


This trace shows:

- **Trace and span identifiers**: For correlating related operations
- **Timing information**: When the operation started and ended
- **Agent metadata**: Agent ID, name, and instructions
- **Model information**: The AI system used (OpenAI) and response ID
- **Token usage**: Input and output token counts for cost tracking

Getting started

We have a number of samples in our repository that demonstrate these capabilities, see the observability samples folder on Github. That includes samples for using zero-code telemetry as well.

Next steps

Using MCP Tools

Agent Tools

10/02/2025

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.

Tooling support with ChatAgent

The `ChatAgent` 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
3. Using hosted tools like web search and MCP (Model Context Protocol) servers

Provide function tools during agent construction

There are various ways to construct a `ChatAgent`, either directly or via factory helper methods on various service clients. All approaches support passing tools at construction time.

Python

```python
from typing import Annotated
from pydantic import Field
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIChatClient

# Sample function tool
def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is cloudy with a high of 15°C."

# When creating a ChatAgent directly
agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant",
    tools=[get_weather]  # Tools provided at construction
)

# When using factory helper methods
```

```
agent = OpenAIChatClient().create_agent(
    instructions="You are a helpful assistant",
    tools=[get_weather]
)
```

The agent will automatically use these tools whenever they're needed to answer user queries:

Python

```
result = await agent.run("What's the weather like in Amsterdam?")
print(result.text)  # The agent will call get_weather() function
```

Provide function tools when running the agent

Python agents support providing tools on a per-run basis using the `tools` parameter in both `run()` and `run_stream()` methods. When both agent-level and run-level tools are provided, they are combined, with run-level tools taking precedence.

Python

```
# Agent created without tools
agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant"
    # No tools defined here
)

# Provide tools for specific runs
result1 = await agent.run(
    "What's the weather in Seattle?",
    tools=[get_weather]  # Tool provided for this run only
)

# Use different tools for different runs
result2 = await agent.run(
    "What's the current time?",
    tools=[get_time]  # Different tool for this query
)

# Provide multiple tools for a single run
result3 = await agent.run(
    "What's the weather and time in Chicago?",
    tools=[get_weather, get_time]  # Multiple tools
)
```

This also works with streaming:

Python

```
async for update in agent.run_stream(
    "Tell me about the weather",
    tools=[get_weather]
):
    if update.text:
        print(update.text, end="", flush=True)
```

Using built-in and hosted tools

The Python Agent Framework supports various built-in and hosted tools that extend agent capabilities:

Web Search Tool

Python

```
from agent_framework import HostedWebSearchTool

agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant with web search
capabilities",
    tools=[
        HostedWebSearchTool(
            additional_properties={
                "user_location": {
                    "city": "Seattle",
                    "country": "US"
                }
            }
        )
    ]
)

result = await agent.run("What are the latest news about AI?")
```

MCP (Model Context Protocol) Tools

Python

```
from agent_framework import HostedMCPTool

agent = ChatAgent(
    chat_client=AzureAIAgentClient(async_credential=credential),
    instructions="You are a documentation assistant",
    tools=[
        HostedMCPTool(
```

```
            name="Microsoft Learn MCP",
            url="https://learn.microsoft.com/api/mcp"
        )
    ]
)

result = await agent.run("How do I create an Azure storage account?")
```

File Search Tool

Python

```
from agent_framework import HostedFileSearchTool, HostedVectorStoreContent

agent = ChatAgent(
    chat_client=AzureAIAgentClient(async_credential=credential),
    instructions="You are a document search assistant",
    tools=[
        HostedFileSearchTool(
            inputs=[
                HostedVectorStoreContent(vector_store_id="vs_123")
            ],
            max_results=10
        )
    ]
)

result = await agent.run("Find information about quarterly reports")
```

Code Interpreter Tool

Python

```
from agent_framework import HostedCodeInterpreterTool

agent = ChatAgent(
    chat_client=AzureAIAgentClient(async_credential=credential),
    instructions="You are a data analysis assistant",
    tools=[HostedCodeInterpreterTool()]
)

result = await agent.run("Analyze this dataset and create a visualization")
```

Mixing agent-level and run-level tools

You can combine tools defined at the agent level with tools provided at runtime:

Python

```python
# Agent with base tools
agent = ChatAgent(
    chat_client=OpenAIChatClient(),
    instructions="You are a helpful assistant",
    tools=[get_time]  # Base tool available for all runs
)

# This run has access to both get_time (agent-level) and get_weather (run-level)
result = await agent.run(
    "What's the weather and time in New York?",
    tools=[get_weather]  # Additional tool for this run
)
```

> **Note**
>
> Tool support varies by service provider. Some services like Azure AI support hosted tools natively, while others may require different approaches. Always check your service provider's documentation for specific tool capabilities.

Next steps

Multi-turn Conversation

Agents in Workflows

10/02/2025

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 AI Agent Service to create intelligent agents
- Implements a Writer agent that creates content based on prompts
- Implements a Reviewer agent that provides feedback on the content
- Connects agents in a sequential workflow pipeline
- Streams real-time updates as agents process requests
- Demonstrates proper async context management for Azure AI clients

Prerequisites

- Python 3.10 or later
- Agent Framework installed: `pip install agent-framework-azure-ai`
- Azure AI Agent Service configured with proper environment variables
- Azure CLI authentication: `az login`

Step 1: Import Required Dependencies

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

Python

```
import asyncio
from collections.abc import Awaitable, Callable
from contextlib import AsyncExitStack
from typing import Any

from agent_framework import AgentRunUpdateEvent, WorkflowBuilder,
WorkflowOutputEvent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential
```

Step 2: Create Azure AI Agent Factory

Create a helper function to manage Azure AI agent creation with proper async context handling:

Python

```
async def create_azure_ai_agent() -> tuple[Callable[..., Awaitable[Any]],
Callable[[], Awaitable[None]]]:
    """Helper method to create an Azure AI agent factory and a close func-
tion.

    This makes sure the async context managers are properly handled.
    """
    stack = AsyncExitStack()
    cred = await stack.enter_async_context(AzureCliCredential())

    client = await stack.enter_async_context(AzureAIAgentClient(async_cre-
dential=cred))

    async def agent(**kwargs: Any) -> Any:
        return await
stack.enter_async_context(client.create_agent(**kwargs))

    async def close() -> None:
        await stack.aclose()

    return agent, close
```

Step 3: Create Specialized Azure AI Agents

Create two specialized agents for content creation and review:

Python

```
async def main() -> None:
    agent, close = await create_azure_ai_agent()
    try:
        # Create a Writer agent that generates content
        writer = await agent(
            name="Writer",
            instructions=(
                "You are an excellent content writer. You create new con-
tent and edit contents based on the feedback."
            ),
        )

        # Create a Reviewer agent that provides feedback
        reviewer = await agent(
            name="Reviewer",
```

```
            instructions=(
                "You are an excellent content reviewer. "
                "Provide actionable feedback to the writer about the pro-
vided content. "
                "Provide the feedback in the most concise manner possible."
            ),
        )
```

Step 4: Build the Workflow

Connect the agents in a sequential workflow using the fluent builder:

Python

```
        # Build the workflow with agents as executors
        workflow =
WorkflowBuilder().set_start_executor(writer).add_edge(writer,
reviewer).build()
```

Step 5: Execute with Streaming

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

Python

```
        last_executor_id: str | None = None

        events = workflow.run_stream("Create a slogan for a new electric
SUV that is affordable and fun to drive.")
        async for event in events:
            if isinstance(event, AgentRunUpdateEvent):
                # Handle streaming updates from agents
                eid = event.executor_id
                if eid != last_executor_id:
                    if last_executor_id is not None:
                        print()
                    print(f"{eid}:", end=" ", flush=True)
                    last_executor_id = eid
                print(event.data, end="", flush=True)
            elif isinstance(event, WorkflowOutputEvent):
                print("\n===== Final output =====")
                print(event.data)
    finally:
        await close()
```

Step 6: Complete Main Function

Wrap everything in the main function with proper async execution:

Python

```python
if __name__ == "__main__":
    asyncio.run(main())
```

How It Works

1. **Azure AI Client Setup**: Uses `AzureAIAgentClient` with Azure CLI credentials for authentication
2. **Agent Factory Pattern**: Creates a factory function that manages async context lifecycle for multiple agents
3. **Sequential Processing**: Writer agent generates content first, then passes it to the Reviewer agent
4. **Streaming Updates**: `AgentRunUpdateEvent` provides real-time token updates as agents generate responses
5. **Context Management**: Proper cleanup of Azure AI resources using `AsyncExitStack`

Key Concepts

- **Azure AI Agent Service**: Cloud-based AI agents with advanced reasoning capabilities
- **AgentRunUpdateEvent**: Real-time streaming updates during agent execution
- **AsyncExitStack**: Proper async context management for multiple resources
- **Agent Factory Pattern**: Reusable agent creation with shared client configuration
- **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 AI agents workflow, see the azure_ai_agents_streaming.py sample in the Agent Framework repository.

Next Steps

Learn about branching in workflows

Azure AI Foundry Agents

10/02/2025

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

Configuration

Environment Variables

Before using Azure AI Foundry Agents, you need to set up these environment variables:

Bash

```
export AZURE_AI_PROJECT_ENDPOINT="https://<your-project>.ser-
vices.ai.azure.com/api/projects/<project-id>"
export AZURE_AI_MODEL_DEPLOYMENT_NAME="gpt-4o-mini"
```

Alternatively, you can provide these values directly in your code.

Installation

Add the Agent Framework Azure AI package to your project:

Bash

```
pip install agent-framework[azure-ai]
```

Getting Started

Authentication

Azure AI Foundry Agents use Azure credentials for authentication. The simplest approach is to use `AzureCliCredential` after running `az login`:

Python

```
from azure.identity.aio import AzureCliCredential
```

```
async with AzureCliCredential() as credential:
    # Use credential with Azure AI agent client
```

Creating Azure AI Foundry Agents

Basic Agent Creation

The simplest way to create an agent is using the `AzureAIAgentClient` with environment variables:

Python

```python
import asyncio
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential).create_agent(
            name="HelperAgent",
            instructions="You are a helpful assistant."
        ) as agent,
    ):
        result = await agent.run("Hello!")
        print(result.text)

asyncio.run(main())
```

Explicit Configuration

You can also provide configuration explicitly instead of using environment variables:

Python

```python
import asyncio
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(
            project_endpoint="https://<your-
project>.services.ai.azure.com/api/projects/<project-id>",
            model_deployment_name="gpt-4o-mini",
            async_credential=credential,
```

```
            agent_name="HelperAgent"
        ).create_agent(
            instructions="You are a helpful assistant."
        ) as agent,
    ):
        result = await agent.run("Hello!")
        print(result.text)

asyncio.run(main())
```

Using Existing Azure AI Foundry Agents

Using an Existing Agent by ID

If you have an existing agent in Azure AI Foundry, you can use it by providing its ID:

Python

```
import asyncio
from agent_framework import ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        ChatAgent(
            chat_client=AzureAIAgentClient(
                async_credential=credential,
                agent_id="<existing-agent-id>"
            ),
            instructions="You are a helpful assistant."
        ) as agent,
    ):
        result = await agent.run("Hello!")
        print(result.text)

asyncio.run(main())
```

Creating and Managing Persistent Agents

For more control over agent lifecycle, you can create persistent agents using the Azure AI Projects client:

Python

```python
import asyncio
import os
from agent_framework import ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.ai.projects.aio import AIProjectClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        AIProjectClient(
            endpoint=os.environ["AZURE_AI_PROJECT_ENDPOINT"],
            credential=credential
        ) as project_client,
    ):
        # Create a persistent agent
        created_agent = await project_client.agents.create_agent(
            model=os.environ["AZURE_AI_MODEL_DEPLOYMENT_NAME"],
            name="PersistentAgent",
            instructions="You are a helpful assistant."
        )

        try:
            # Use the agent
            async with ChatAgent(
                chat_client=AzureAIAgentClient(
                    project_client=project_client,
                    agent_id=created_agent.id
                ),
                instructions="You are a helpful assistant."
            ) as agent:
                result = await agent.run("Hello!")
                print(result.text)
        finally:
            # Clean up the agent
            await project_client.agents.delete_agent(created_agent.id)

asyncio.run(main())
```

Agent Features

Function Tools

You can provide custom function tools to Azure AI Foundry agents:

Python

```python
import asyncio
from typing import Annotated
```

```python
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with a high of 25°C."

async def main():
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential).create_agent(
            name="WeatherAgent",
            instructions="You are a helpful weather assistant.",
            tools=get_weather
        ) as agent,
    ):
        result = await agent.run("What's the weather like in Seattle?")
        print(result.text)

asyncio.run(main())
```

Code Interpreter

Azure AI Foundry agents support code execution through the hosted code interpreter:

Python

```python
import asyncio
from agent_framework import HostedCodeInterpreterTool
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential).create_agent(
            name="CodingAgent",
            instructions="You are a helpful assistant that can write and
execute Python code.",
            tools=HostedCodeInterpreterTool()
        ) as agent,
    ):
        result = await agent.run("Calculate the factorial of 20 using
Python code.")
        print(result.text)

asyncio.run(main())
```

Streaming Responses

Get responses as they are generated using streaming:

Python

```python
import asyncio
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def main():
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential).create_agent(
            name="StreamingAgent",
            instructions="You are a helpful assistant."
        ) as agent,
    ):
        print("Agent: ", end="", flush=True)
        async for chunk in agent.run_stream("Tell me a short story"):
            if chunk.text:
                print(chunk.text, end="", flush=True)
        print()

asyncio.run(main())
```

Using the Agent

The agent is a standard `BaseAgent` 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

OpenAI ChatCompletion Agents

Azure OpenAI ChatCompletion Agents

10/02/2025

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

Configuration

Environment Variables

Before using Azure OpenAI ChatCompletion agents, you need to set up these environment variables:

```bash
export AZURE_OPENAI_ENDPOINT="https://<myresource>.openai.azure.com"
export AZURE_OPENAI_CHAT_DEPLOYMENT_NAME="gpt-4o-mini"
```

Optionally, you can also set:

```bash
export AZURE_OPENAI_API_VERSION="2024-10-21"  # Default API version
export AZURE_OPENAI_API_KEY="<your-api-key>"  # If not using Azure CLI authentication
```

Installation

Add the Agent Framework package to your project:

```bash
pip install agent-framework
```

Getting Started

Authentication

Azure OpenAI agents use Azure credentials for authentication. The simplest approach is to use `AzureCliCredential` after running `az login`:

Python

```
from azure.identity import AzureCliCredential

credential = AzureCliCredential()
```

Creating an Azure OpenAI ChatCompletion Agent

Basic Agent Creation

The simplest way to create an agent is using the `AzureOpenAIChatClient` with environment variables:

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

async def main():
    agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
        instructions="You are good at telling jokes.",
        name="Joker"
    )

    result = await agent.run("Tell me a joke about a pirate.")
    print(result.text)

asyncio.run(main())
```

Explicit Configuration

You can also provide configuration explicitly instead of using environment variables:

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

async def main():
    agent = AzureOpenAIChatClient(
```

```
        endpoint="https://<myresource>.openai.azure.com",
        deployment_name="gpt-4o-mini",
        credential=AzureCliCredential()
    ).create_agent(
        instructions="You are good at telling jokes.",
        name="Joker"
    )

    result = await agent.run("Tell me a joke about a pirate.")
    print(result.text)

asyncio.run(main())
```

Agent Features

Function Tools

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

Python

```
import asyncio
from typing import Annotated
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with a high of 25°C."

async def main():
    agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
        instructions="You are a helpful weather assistant.",
        tools=get_weather
    )

    result = await agent.run("What's the weather like in Seattle?")
    print(result.text)

asyncio.run(main())
```

Streaming Responses

Get responses as they are generated using streaming:

Python

```python
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

async def main():
    agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
        instructions="You are a helpful assistant."
    )

    print("Agent: ", end="", flush=True)
    async for chunk in agent.run_stream("Tell me a short story about a ro-
bot"):
        if chunk.text:
            print(chunk.text, end="", flush=True)
    print()

asyncio.run(main())
```

Using the Agent

The agent is a standard `BaseAgent` 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

OpenAI Response Agents

Azure OpenAI Responses Agents

10/02/2025

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

Configuration

Environment Variables

Before using Azure OpenAI Responses agents, you need to set up these environment variables:

```Bash
export AZURE_OPENAI_ENDPOINT="https://<myresource>.openai.azure.com"
export AZURE_OPENAI_RESPONSES_DEPLOYMENT_NAME="gpt-4o-mini"
```

Optionally, you can also set:

```Bash
export AZURE_OPENAI_API_VERSION="preview"  # Required for Responses API
export AZURE_OPENAI_API_KEY="<your-api-key>"  # If not using Azure CLI
authentication
```

Installation

Add the Agent Framework package to your project:

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

Getting Started

Authentication

Azure OpenAI Responses agents use Azure credentials for authentication. The simplest approach is to use `AzureCliCredential` after running `az login`:

Python

```
from azure.identity import AzureCliCredential

credential = AzureCliCredential()
```

Creating an Azure OpenAI Responses Agent

Basic Agent Creation

The simplest way to create an agent is using the `AzureOpenAIResponsesClient` with environment variables:

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIResponsesClient
from azure.identity import AzureCliCredential

async def main():
    agent =
AzureOpenAIResponsesClient(credential=AzureCliCredential()).create_agent(
        instructions="You are good at telling jokes.",
        name="Joker"
    )

    result = await agent.run("Tell me a joke about a pirate.")
    print(result.text)

asyncio.run(main())
```

Explicit Configuration

You can also provide configuration explicitly instead of using environment variables:

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIResponsesClient
from azure.identity import AzureCliCredential

async def main():
    agent = AzureOpenAIResponsesClient(
        endpoint="https://<myresource>.openai.azure.com",
        deployment_name="gpt-4o-mini",
        api_version="preview",
```

```
        credential=AzureCliCredential()
    ).create_agent(
        instructions="You are good at telling jokes.",
        name="Joker"
    )

    result = await agent.run("Tell me a joke about a pirate.")
    print(result.text)

asyncio.run(main())
```

Agent Features

Function Tools

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

Python

```
import asyncio
from typing import Annotated
from agent_framework.azure import AzureOpenAIResponsesClient
from azure.identity import AzureCliCredential
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with a high of 25°C."

async def main():
    agent =
AzureOpenAIResponsesClient(credential=AzureCliCredential()).create_agent(
        instructions="You are a helpful weather assistant.",
        tools=get_weather
    )

    result = await agent.run("What's the weather like in Seattle?")
    print(result.text)

asyncio.run(main())
```

Code Interpreter

Azure OpenAI Responses agents support code execution through the hosted code interpreter:

Python

```
import asyncio
from agent_framework import ChatAgent, HostedCodeInterpreterTool
from agent_framework.azure import AzureOpenAIResponsesClient
from azure.identity import AzureCliCredential

async def main():
    async with ChatAgent(
        chat_client=AzureOpenAIResponsesClient(credential=AzureCliCreden-
tial()),
        instructions="You are a helpful assistant that can write and exe-
cute Python code.",
        tools=HostedCodeInterpreterTool()
    ) as agent:
        result = await agent.run("Calculate the factorial of 20 using
Python code.")
        print(result.text)

asyncio.run(main())
```

Streaming Responses

Get responses as they are generated using streaming:

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIResponsesClient
from azure.identity import AzureCliCredential

async def main():
    agent =
AzureOpenAIResponsesClient(credential=AzureCliCredential()).create_agent(
        instructions="You are a helpful assistant."
    )

    print("Agent: ", end="", flush=True)
    async for chunk in agent.run_stream("Tell me a short story about a ro-
bot"):
        if chunk.text:
            print(chunk.text, end="", flush=True)
    print()

asyncio.run(main())
```

Using the Agent

The agent is a standard `BaseAgent` 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

OpenAI Chat Completion Agents

Checkpointing and Resuming Workflows

10/02/2025

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

FileCheckpointStorage

The `FileCheckpointStorage` class provides persistent checkpoint storage using JSON files:

Python

```python
from agent_framework import FileCheckpointStorage
from pathlib import Path

# Initialize checkpoint storage
checkpoint_storage = FileCheckpointStorage(storage_path="./checkpoints")
```

Enabling Checkpointing

Enable checkpointing when building your workflow:

Python

```python
from agent_framework import WorkflowBuilder

workflow = (
    WorkflowBuilder(max_iterations=5)
    .add_edge(executor1, executor2)
    .set_start_executor(executor1)
    .with_checkpointing(checkpoint_storage=checkpoint_storage)  # Enable checkpointing
    .build()
)
```

State Persistence

Executor State

Executors can persist local state that survives checkpoints:

Python

```python
from agent_framework import Executor, WorkflowContext, handler

class UpperCaseExecutor(Executor):
    @handler
    async def to_upper_case(self, text: str, ctx: WorkflowContext[str]) ->
None:
        result = text.upper()

        # Persist executor-local state for checkpoints
        prev = await ctx.get_state() or {}
        count = int(prev.get("count", 0)) + 1
        await ctx.set_state({
            "count": count,
            "last_input": text,
            "last_output": result,
        })

        # Send result to next executor
        await ctx.send_message(result)
```

Shared State

Use shared state for data that multiple executors need to access:

Python

```python
class ProcessorExecutor(Executor):
    @handler
    async def process(self, text: str, ctx: WorkflowContext[str]) -> None:
        # Write to shared state for cross-executor visibility
        await ctx.set_shared_state("original_input", text)
        await ctx.set_shared_state("processed_output", text.upper())

        await ctx.send_message(text.upper())
```

Working with Checkpoints

Listing Checkpoints

Retrieve and inspect available checkpoints:

Python

```python
# List all checkpoints
all_checkpoints = await checkpoint_storage.list_checkpoints()

# List checkpoints for a specific workflow
workflow_checkpoints = await checkpoint_storage.list_checkpoints(work-
flow_id="my-workflow")

# Sort by creation time
sorted_checkpoints = sorted(all_checkpoints, key=lambda cp: cp.timestamp)
```

Checkpoint Information

Access checkpoint metadata and state:

Python

```python
from agent_framework import RequestInfoExecutor

for checkpoint in checkpoints:
    # Get human-readable summary
    summary = RequestInfoExecutor.checkpoint_summary(checkpoint)

    print(f"Checkpoint: {summary.checkpoint_id}")
    print(f"Iteration: {summary.iteration_count}")
    print(f"Status: {summary.status}")
    print(f"Messages: {len(checkpoint.messages)}")
    print(f"Shared State: {checkpoint.shared_state}")
    print(f"Executor States: {list(checkpoint.executor_states.keys())}")
```

Resuming from Checkpoints

Streaming Resume

Resume execution and stream events in real-time:

Python

```python
# Resume from a specific checkpoint
async for event in workflow.run_stream_from_checkpoint(
    checkpoint_id="checkpoint-id",
    checkpoint_storage=checkpoint_storage
):
    print(f"Resumed Event: {event}")

    if isinstance(event, WorkflowOutputEvent):
```

```
        print(f"Final Result: {event.data}")
        break
```

Non-Streaming Resume

Resume and get all results at once:

Python

```python
# Resume and wait for completion
result = await workflow.run_from_checkpoint(
    checkpoint_id="checkpoint-id",
    checkpoint_storage=checkpoint_storage
)

# Access final outputs
outputs = result.get_outputs()
print(f"Final outputs: {outputs}")
```

Resume with Responses

For workflows with pending requests, provide responses during resume:

Python

```python
# Resume with pre-supplied responses for RequestInfoExecutor
responses = {
    "request-id-1": "user response data",
    "request-id-2": "another response"
}

async for event in workflow.run_stream_from_checkpoint(
    checkpoint_id="checkpoint-id",
    checkpoint_storage=checkpoint_storage,
    responses=responses  # Inject responses during resume
):
    print(f"Event: {event}")
```

Interactive Checkpoint Selection

Build user-friendly checkpoint selection:

Python

```python
async def select_and_resume_checkpoint(workflow, storage):
    # Get available checkpoints
```

```
    checkpoints = await storage.list_checkpoints()
    if not checkpoints:
        print("No checkpoints available")
        return

    # Sort and display options
    sorted_cps = sorted(checkpoints, key=lambda cp: cp.timestamp)
    print("Available checkpoints:")
    for i, cp in enumerate(sorted_cps):
        summary = RequestInfoExecutor.checkpoint_summary(cp)
        print(f"[{i}] {summary.checkpoint_id[:8]}... iter={summary.itera-
tion_count}")

    # Get user selection
    try:
        idx = int(input("Enter checkpoint index: "))
        selected = sorted_cps[idx]

        # Resume from selected checkpoint
        print(f"Resuming from checkpoint: {selected.checkpoint_id}")
        async for event in workflow.run_stream_from_checkpoint(
            selected.checkpoint_id,
            checkpoint_storage=storage
        ):
            print(f"Event: {event}")

    except (ValueError, IndexError):
        print("Invalid selection")
```

Complete Example Pattern

Here's a typical checkpointing workflow pattern:

Python

```
import asyncio
from pathlib import Path
from agent_framework import (
    WorkflowBuilder, FileCheckpointStorage,
    WorkflowOutputEvent, RequestInfoExecutor
)

async def main():
    # Setup checkpoint storage
    checkpoint_dir = Path("./checkpoints")
    checkpoint_dir.mkdir(exist_ok=True)
    storage = FileCheckpointStorage(checkpoint_dir)

    # Build workflow with checkpointing
    workflow = (
        WorkflowBuilder()
        .add_edge(executor1, executor2)
```

```
        .set_start_executor(executor1)
        .with_checkpointing(storage)
        .build()
    )

    # Initial run
    print("Running workflow...")
    async for event in workflow.run_stream("input data"):
        print(f"Event: {event}")

    # List and inspect checkpoints
    checkpoints = await storage.list_checkpoints()
    for cp in sorted(checkpoints, key=lambda c: c.timestamp):
        summary = RequestInfoExecutor.checkpoint_summary(cp)
        print(f"Checkpoint: {summary.checkpoint_id[:8]}... iter={summa-
ry.iteration_count}")

    # Resume from a checkpoint
    if checkpoints:
        latest = max(checkpoints, key=lambda cp: cp.timestamp)
        print(f"Resuming from: {latest.checkpoint_id}")

        async for event in workflow.run_stream_from_checkpoint(lat-
est.checkpoint_id):
            print(f"Resumed: {event}")

if __name__ == "__main__":
    asyncio.run(main())
```

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 checkpoint boundaries
- **Human-in-the-Loop**: Pause for human input and resume later with responses
- **Debugging**: Inspect workflow state at specific points and resume execution for testing
- **Resource Management**: Stop and restart workflows based on resource availability

Running the Example

For the complete working implementation, see the Checkpoint with Resume sample .

Next Steps

Learn about Workflow Visualization

Create a Simple Concurrent Workflow

10/02/2025

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.

In the Python implementation, you'll build a concurrent workflow that processes data through multiple parallel executors and aggregates results of different types. This example demonstrates how the framework handles mixed result types from concurrent processing.

What You'll Build

You'll create a workflow that:

- Takes a list of numbers as input
- Distributes the list to two parallel executors (one calculating average, one calculating sum)
- Aggregates the different result types (float and int) into a final output
- Demonstrates how the framework handles different result types from concurrent executors

Prerequisites

- Python 3.10 or later
- Agent Framework Core installed: `pip install agent-framework-core`

Step 1: Import Required Dependencies

Start by importing the necessary components from the Agent Framework:

Python

```
import asyncio
import random

from agent_framework import Executor, WorkflowBuilder, WorkflowContext,
WorkflowOutputEvent, handler
from typing_extensions import Never
```

Step 2: Create the Dispatcher Executor

The dispatcher is responsible for distributing the initial input to multiple parallel executors:

Python

```
class Dispatcher(Executor):
    """
    The sole purpose of this executor is to dispatch the input of the work-
flow to
    other executors.
    """

    @handler
    async def handle(self, numbers: list[int], ctx:
WorkflowContext[list[int]]):
        if not numbers:
            raise RuntimeError("Input must be a valid list of integers.")

        await ctx.send_message(numbers)
```

Step 3: Create Parallel Processing Executors

Create two executors that will process the data concurrently:

Python

```
class Average(Executor):
    """Calculate the average of a list of integers."""

    @handler
    async def handle(self, numbers: list[int], ctx:
WorkflowContext[float]):
        average: float = sum(numbers) / len(numbers)
        await ctx.send_message(average)


class Sum(Executor):
    """Calculate the sum of a list of integers."""

    @handler
    async def handle(self, numbers: list[int], ctx: WorkflowContext[int]):
        total: int = sum(numbers)
        await ctx.send_message(total)
```

Step 4: Create the Aggregator Executor

The aggregator collects results from the parallel executors and yields the final output:

Python

```
class Aggregator(Executor):
    """Aggregate the results from the different tasks and yield the final
output."""

    @handler
    async def handle(self, results: list[int | float], ctx:
WorkflowContext[Never, list[int | float]]):
        """Receive the results from the source executors.

        The framework will automatically collect messages from the source
executors
        and deliver them as a list.

        Args:
            results (list[int | float]): execution results from upstream
executors.
                The type annotation must be a list of union types that the
upstream
                executors will produce.
            ctx (WorkflowContext[Never, list[int | float]]): A workflow
context that can yield the final output.
        """
        await ctx.yield_output(results)
```

Step 5: Build the Workflow

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

Python

```
async def main() -> None:
    # 1) Create the executors
    dispatcher = Dispatcher(id="dispatcher")
    average = Average(id="average")
    summation = Sum(id="summation")
    aggregator = Aggregator(id="aggregator")

    # 2) Build a simple fan out and fan in workflow
    workflow = (
        WorkflowBuilder()
        .set_start_executor(dispatcher)
        .add_fan_out_edges(dispatcher, [average, summation])
        .add_fan_in_edges([average, summation], aggregator)
        .build()
    )
```

Step 6: Run the Workflow

Execute the workflow with sample data and capture the output:

Python

```
    # 3) Run the workflow
    output: list[int | float] | None = None
    async for event in workflow.run_stream([random.randint(1, 100) for _
in range(10)]):
        if isinstance(event, WorkflowOutputEvent):
            output = event.data

    if output is not None:
        print(output)

if __name__ == "__main__":
    asyncio.run(main())
```

How It Works

1. **Fan-Out**: The `Dispatcher` receives the input list and sends it to both the `Average` and `Sum` executors simultaneously
2. **Parallel Processing**: Both executors process the same input concurrently, producing different result types:
   - `Average` executor produces a `float` result
   - `Sum` executor produces an `int` result
3. **Fan-In**: The `Aggregator` receives results from both executors as a list containing both types
4. **Type Handling**: The framework automatically handles the different result types using union types (`int | float`)

Key Concepts

- **Fan-Out Edges**: Use `add_fan_out_edges()` to send the same input to multiple executors
- **Fan-In Edges**: Use `add_fan_in_edges()` to collect results from multiple source executors
- **Union Types**: Handle different result types using type annotations like `list[int | float]`
- **Concurrent Execution**: Multiple executors process data simultaneously, improving performance

Complete Implementation

For the complete working implementation of this concurrent workflow, see the aggregate_results_of_different_types.py sample in the Agent Framework repository.

Next Steps

Learn about using agents in workflows

Create a Simple Sequential Workflow

10/02/2025

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

The workflow demonstrates core concepts like:

- Using the `@executor` decorator to create workflow nodes
- Connecting executors with `WorkflowBuilder`
- Passing data between steps with `ctx.send_message()`
- Yielding final output with `ctx.yield_output()`
- Streaming events for real-time observability

Prerequisites

- Python 3.10 or later
- Agent Framework Core Python package installed: `pip install agent-framework-core`
- No external AI services required for this basic example

Step-by-Step Implementation

Let's build the sequential workflow step by step.

Step 1: Import Required Modules

First, import the necessary modules from the Agent Framework:

Python

```
import asyncio
from typing_extensions import Never
from agent_framework import WorkflowBuilder, WorkflowContext,
WorkflowOutputEvent, executor
```

Step 2: Create the First Executor

Create an executor that converts text to uppercase using the `@executor` decorator:

Python

```
@executor(id="upper_case_executor")
async def to_upper_case(text: str, ctx: WorkflowContext[str]) -> None:
    """Transform the input to uppercase and forward it to the next step."""
    result = text.upper()

    # Send the intermediate result to the next executor
    await ctx.send_message(result)
```

Key Points:

- The `@executor` decorator registers this function as a workflow node
- `WorkflowContext[str]` indicates this executor sends a string downstream
- `ctx.send_message()` passes data to the next step

Step 3: Create the Second Executor

Create an executor that reverses the text and yields the final output:

Python

```
@executor(id="reverse_text_executor")
async def reverse_text(text: str, ctx: WorkflowContext[Never, str]) ->
None:
    """Reverse the input and yield the workflow output."""
    result = text[::-1]

    # Yield the final output for this workflow run
    await ctx.yield_output(result)
```

Key Points:

- `WorkflowContext[Never, str]` indicates this is a terminal executor
- `ctx.yield_output()` provides the final workflow result

- The workflow completes when it becomes idle

Step 4: Build the Workflow

Connect the executors using `WorkflowBuilder`:

Python

```
workflow = (
    WorkflowBuilder()
    .add_edge(to_upper_case, reverse_text)
    .set_start_executor(to_upper_case)
    .build()
)
```

Key Points:

- `add_edge()` creates directed connections between executors
- `set_start_executor()` defines the entry point
- `build()` finalizes the workflow

Step 5: Run the Workflow with Streaming

Execute the workflow and observe events in real-time:

Python

```
async def main():
    # Run the workflow and stream events
    async for event in workflow.run_stream("hello world"):
        print(f"Event: {event}")
        if isinstance(event, WorkflowOutputEvent):
            print(f"Workflow completed with result: {event.data}")

if __name__ == "__main__":
    asyncio.run(main())
```

Step 6: Understanding the Output

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

text

```
Event: ExecutorInvokedEvent(executor_id=upper_case_executor)
Event: ExecutorCompletedEvent(executor_id=upper_case_executor)
```

```
Event: ExecutorInvokedEvent(executor_id=reverse_text_executor)
Event: ExecutorCompletedEvent(executor_id=reverse_text_executor)
Event: WorkflowOutputEvent(data='DLROW OLLEH', source_executor_id=reverse_-
text_executor)
Workflow completed with result: DLROW OLLEH
```

Key Concepts Explained

Workflow Context Types

The `WorkflowContext` generic type defines what data flows between executors:

- `WorkflowContext[str]` - Sends a string to the next executor
- `WorkflowContext[Never, str]` - Terminal executor that yields workflow output of type string

Event Types

During streaming execution, you'll observe these event types:

- `ExecutorInvokedEvent` - When an executor starts processing
- `ExecutorCompletedEvent` - When an executor finishes processing
- `WorkflowOutputEvent` - Contains the final workflow result

Python Workflow Builder Pattern

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

- **add_edge()**: Creates directed connections between executors
- **set_start_executor()**: Defines the workflow entry point
- **build()**: Finalizes and returns an immutable workflow object

Running the Example

1. Combine all the code snippets from the steps above into a single Python file
2. Save it as `sequential_workflow.py`
3. Run with: `python sequential_workflow.py`

The workflow will process the input "hello world" through both executors and display the streaming events.

Complete Example

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

This sample includes:

- Full implementation with all imports and documentation
- Additional comments explaining the workflow concepts
- Sample output showing the expected results

Next Steps

Learn about creating a simple concurrent workflow

Create a Workflow with Branching Logic

10/02/2025

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
- 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

Prerequisites

- Python 3.10 or later
- Agent Framework installed: `pip install agent-framework-core`
- Azure OpenAI service configured with proper environment variables
- Azure CLI authentication: `az login`

Step 1: Import Required Dependencies

Start by importing the necessary components for conditional workflows:

Python

```python
import asyncio
import os
from dataclasses import dataclass
from typing import Any, Literal
from uuid import uuid4

from typing_extensions import Never
```

```
from agent_framework import (
    AgentExecutor,
    AgentExecutorRequest,
    AgentExecutorResponse,
    ChatMessage,
    Role,
    WorkflowBuilder,
    WorkflowContext,
    executor,
    Case,
    Default,
)
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential
from pydantic import BaseModel
```

Step 2: Define Data Models

Create Pydantic models for structured data exchange between workflow components:

Python

```
class DetectionResult(BaseModel):
    """Represents the result of spam detection."""
    # is_spam drives the routing decision taken by edge conditions
    is_spam: bool
    # Human readable rationale from the detector
    reason: str
    # The agent must include the original email so downstream agents can
operate without reloading content
    email_content: str


class EmailResponse(BaseModel):
    """Represents the response from the email assistant."""
    # The drafted reply that a user could copy or send
    response: str
```

Step 3: Create Condition Functions

Define condition functions that will determine routing decisions:

Python

```
def get_condition(expected_result: bool):
    """Create a condition callable that routes based on
DetectionResult.is_spam."""
```

```
    # The returned function will be used as an edge predicate.
    # It receives whatever the upstream executor produced.
    def condition(message: Any) -> bool:
        # Defensive guard. If a non AgentExecutorResponse appears, let the
edge pass to avoid dead ends.
        if not isinstance(message, AgentExecutorResponse):
            return True

        try:
            # Prefer parsing a structured DetectionResult from the agent
JSON text.
            # Using model_validate_json ensures type safety and raises if
the shape is wrong.
            detection =
DetectionResult.model_validate_json(message.agent_run_response.text)
            # Route only when the spam flag matches the expected path.
            return detection.is_spam == expected_result
        except Exception:
            # Fail closed on parse errors so we do not accidentally route
to the wrong path.
            # Returning False prevents this edge from activating.
            return False

    return condition
```

Step 4: Create Handler Executors

Define executors to handle different routing outcomes:

Python

```
@executor(id="send_email")
async def handle_email_response(response: AgentExecutorResponse, ctx:
WorkflowContext[Never, str]) -> None:
    """Handle legitimate emails by drafting a professional response."""
    # Downstream of the email assistant. Parse a validated EmailResponse
and yield the workflow output.
    email_response =
EmailResponse.model_validate_json(response.agent_run_response.text)
    await ctx.yield_output(f"Email sent:\n{email_response.response}")


@executor(id="handle_spam")
async def handle_spam_classifier_response(response: AgentExecutorResponse,
ctx: WorkflowContext[Never, str]) -> None:
    """Handle spam emails by marking them appropriately."""
    # Spam path. Confirm the DetectionResult and yield the workflow output.
Guard against accidental non spam input.
    detection =
DetectionResult.model_validate_json(response.agent_run_response.text)
    if detection.is_spam:
        await ctx.yield_output(f"Email marked as spam: {detection.reason}")
```

```
    else:
        # This indicates the routing predicate and executor contract are
out of sync.
        raise RuntimeError("This executor should only handle spam
messages.")


@executor(id="to_email_assistant_request")
async def to_email_assistant_request(
    response: AgentExecutorResponse, ctx:
WorkflowContext[AgentExecutorRequest]
) -> None:
    """Transform spam detection response into a request for the email as-
sistant."""
    # Parse the detection result and extract the email content for the as-
sistant
    detection =
DetectionResult.model_validate_json(response.agent_run_response.text)

    # Create a new request for the email assistant with the original email
content
    request = AgentExecutorRequest(
        messages=[ChatMessage(Role.USER, text=detection.email_content)],
        should_respond=True
    )
    await ctx.send_message(request)
```

Step 5: Create AI Agents

Set up the Azure OpenAI agents with structured output formatting:

Python

```
async def main() -> None:
    # Create agents
    # AzureCliCredential uses your current az login. This avoids embedding
secrets in code.
    chat_client = AzureOpenAIChatClient(credential=AzureCliCredential())

    # Agent 1. Classifies spam and returns a DetectionResult object.
    # response_format enforces that the LLM returns parsable JSON for the
Pydantic model.
    spam_detection_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are a spam detection assistant that identifies spam
emails. "
                "Always return JSON with fields is_spam (bool), reason
(string), and email_content (string). "
                "Include the original email content in email_content."
            ),
            response_format=DetectionResult,
```

```
        ),
        id="spam_detection_agent",
    )

    # Agent 2. Drafts a professional reply. Also uses structured JSON out-
put for reliability.
    email_assistant_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are an email assistant that helps users draft profes-
sional responses to emails. "
                "Your input may be a JSON object that includes 'email_con-
tent'; base your reply on that content. "
                "Return JSON with a single field 'response' containing the
drafted reply."
            ),
            response_format=EmailResponse,
        ),
        id="email_assistant_agent",
    )
```

Step 6: Build the Conditional Workflow

Create a workflow with conditional edges that route based on spam detection results:

Python

```
    # Build the workflow graph.
    # Start at the spam detector.
    # If not spam, hop to a transformer that creates a new
AgentExecutorRequest,
    # then call the email assistant, then finalize.
    # If spam, go directly to the spam handler and finalize.
    workflow = (
        WorkflowBuilder()
        .set_start_executor(spam_detection_agent)
        # Not spam path: transform response -> request for assistant -> as-
sistant -> send email
        .add_edge(spam_detection_agent, to_email_assistant_request, condi-
tion=get_condition(False))
        .add_edge(to_email_assistant_request, email_assistant_agent)
        .add_edge(email_assistant_agent, handle_email_response)
        # Spam path: send to spam handler
        .add_edge(spam_detection_agent, handle_spam_classifier_response,
condition=get_condition(True))
        .build()
    )
```

Step 7: Execute the Workflow

Run the workflow with sample email content:

Python

```
    # Read Email content from the sample resource file.
    # This keeps the sample deterministic since the model sees the same
email every run.
    email_path = os.path.join(os.path.dirname(os.path.dirname(os.path.real-
path(__file__))), "resources", "email.txt")

    with open(email_path) as email_file:  # noqa: ASYNC230
        email = email_file.read()

    # Execute the workflow. Since the start is an AgentExecutor, pass an
AgentExecutorRequest.
    # The workflow completes when it becomes idle (no more work to do).
    request = AgentExecutorRequest(messages=[ChatMessage(Role.USER,
text=email)], should_respond=True)
    events = await workflow.run(request)
    outputs = events.get_outputs()
    if outputs:
        print(f"Workflow output: {outputs[0]}")


if __name__ == "__main__":
    asyncio.run(main())
```

How Conditional Edges Work

1. **Condition Functions**: The `get_condition()` function creates a predicate that examines the message content and returns `True` or `False` to determine if the edge should be traversed.

2. **Message Inspection**: Conditions can inspect any aspect of the message, including structured data from agent responses parsed with Pydantic models.

3. **Defensive Programming**: The condition function includes error handling to prevent routing failures when parsing structured data.

4. **Dynamic Routing**: Based on the spam detection result, emails are automatically routed to either the email assistant (for legitimate emails) or the spam handler (for suspicious emails).

Key Concepts

- **Edge Conditions**: Boolean predicates that determine whether an edge should be traversed

- **Structured Outputs**: Using Pydantic models with `response_format` ensures reliable data parsing
- **Defensive Routing**: Condition functions handle edge cases to prevent workflow dead-ends
- **Message Transformation**: Executors can transform message types between workflow steps

Complete Implementation

For the complete working implementation, see the edge_condition.py 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 Next

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

- **NotSpam** → Email Assistant → Send Email
- **Spam** → Mark as Spam
- **Uncertain** → Flag for Manual Review (default case)

The key improvement is using a single switch-case edge group instead of multiple individual conditional edges, making the workflow easier to understand and maintain as decision complexity grows.

Enhanced Data Models

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

Python

```python
from typing import Literal
```

```python
class DetectionResultAgent(BaseModel):
    """Structured output returned by the spam detection agent."""

    # The agent classifies the email into one of three categories
    spam_decision: Literal["NotSpam", "Spam", "Uncertain"]
    reason: str

class EmailResponse(BaseModel):
    """Structured output returned by the email assistant agent."""

    response: str

@dataclass
class DetectionResult:
    """Internal typed payload used for routing and downstream handling."""

    spam_decision: str
    reason: str
    email_id: str

@dataclass
class Email:
    """In memory record of the email content stored in shared state."""

    email_id: str
    email_content: str
```

Switch-Case Condition Factory

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

Python

```python
def get_case(expected_decision: str):
    """Factory that returns a predicate matching a specific spam_decision
value."""

    def condition(message: Any) -> bool:
        # Only match when the upstream payload is a DetectionResult with
the expected decision
        return isinstance(message, DetectionResult) and message.spam_deci-
sion == expected_decision

    return condition
```

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

Workflow Executors with Shared State

Implement executors that use shared state to avoid passing large email content through every workflow step:

Python

```
EMAIL_STATE_PREFIX = "email:"
CURRENT_EMAIL_ID_KEY = "current_email_id"

@executor(id="store_email")
async def store_email(email_text: str, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
    """Store email content once and pass around a lightweight ID
reference."""

    # Persist the raw email content in shared state
    new_email = Email(email_id=str(uuid4()), email_content=email_text)
    await ctx.set_shared_state(f"{EMAIL_STATE_PREFIX}{new_email.email_id}",
new_email)
    await ctx.set_shared_state(CURRENT_EMAIL_ID_KEY, new_email.email_id)

    # Forward email to spam detection agent
    await ctx.send_message(
        AgentExecutorRequest(messages=[ChatMessage(Role.USER, text=new_e-
mail.email_content)], should_respond=True)
    )

@executor(id="to_detection_result")
async def to_detection_result(response: AgentExecutorResponse, ctx:
WorkflowContext[DetectionResult]) -> None:
    """Transform agent response into a typed DetectionResult with email
ID."""

    # Parse the agent's structured JSON output
    parsed =
DetectionResultAgent.model_validate_json(response.agent_run_response.text)
    email_id: str = await ctx.get_shared_state(CURRENT_EMAIL_ID_KEY)

    # Create typed message for switch-case routing
    await ctx.send_message(DetectionResult(
        spam_decision=parsed.spam_decision,
        reason=parsed.reason,
        email_id=email_id
    ))

@executor(id="submit_to_email_assistant")
async def submit_to_email_assistant(detection: DetectionResult, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
    """Handle NotSpam emails by forwarding to the email assistant."""

    # Guard against misrouting
    if detection.spam_decision != "NotSpam":
```

```
        raise RuntimeError("This executor should only handle NotSpam mes-
sages.")

    # Retrieve original email content from shared state
    email: Email = await ctx.get_shared_state(f"{EMAIL_STATE_PREFIX}{detec-
tion.email_id}")
    await ctx.send_message(
        AgentExecutorRequest(messages=[ChatMessage(Role.USER, text=email.e-
mail_content)], should_respond=True)
    )

@executor(id="finalize_and_send")
async def finalize_and_send(response: AgentExecutorResponse, ctx:
WorkflowContext[Never, str]) -> None:
    """Parse email assistant response and yield final output."""

    parsed =
EmailResponse.model_validate_json(response.agent_run_response.text)
    await ctx.yield_output(f"Email sent: {parsed.response}")

@executor(id="handle_spam")
async def handle_spam(detection: DetectionResult, ctx:
WorkflowContext[Never, str]) -> None:
    """Handle confirmed spam emails."""

    if detection.spam_decision == "Spam":
        await ctx.yield_output(f"Email marked as spam: {detection.reason}")
    else:
        raise RuntimeError("This executor should only handle Spam
messages.")

@executor(id="handle_uncertain")
async def handle_uncertain(detection: DetectionResult, ctx:
WorkflowContext[Never, str]) -> None:
    """Handle uncertain classifications that need manual review."""

    if detection.spam_decision == "Uncertain":
        # Include original content for human review
        email: Email | None = await ctx.get_shared_state(f"{EMAIL_S-
TATE_PREFIX}{detection.email_id}")
        await ctx.yield_output(
            f"Email marked as uncertain: {detection.reason}. Email content:
{getattr(email, 'email_content', '')}"
        )
    else:
        raise RuntimeError("This executor should only handle Uncertain
messages.")
```

Create Enhanced AI Agent

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

Python

```
async def main():
    chat_client = AzureOpenAIChatClient(credential=AzureCliCredential())

    # Enhanced spam detection agent with three-way classification
    spam_detection_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are a spam detection assistant that identifies spam
emails. "
                "Be less confident in your assessments. "
                "Always return JSON with fields 'spam_decision' (one of
NotSpam, Spam, Uncertain) "
                "and 'reason' (string)."
            ),
            response_format=DetectionResultAgent,
        ),
        id="spam_detection_agent",
    )

    # Email assistant remains the same
    email_assistant_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are an email assistant that helps users draft respons-
es to emails with professionalism."
            ),
            response_format=EmailResponse,
        ),
        id="email_assistant_agent",
    )
```

Build Workflow with Switch-Case Edge Group

Replace multiple conditional edges with a single switch-case group:

Python

```
    # Build workflow using switch-case for cleaner three-way routing
    workflow = (
        WorkflowBuilder()
        .set_start_executor(store_email)
        .add_edge(store_email, spam_detection_agent)
        .add_edge(spam_detection_agent, to_detection_result)
        .add_switch_case_edge_group(
            to_detection_result,
            [
                # Explicit cases for specific decisions
                Case(condition=get_case("NotSpam"), target=submit_to_e-
mail_assistant),
```

```
                Case(condition=get_case("Spam"), target=handle_spam),
                # Default case catches anything that doesn't match above
                Default(target=handle_uncertain),
            ],
        )
        .add_edge(submit_to_email_assistant, email_assistant_agent)
        .add_edge(email_assistant_agent, finalize_and_send)
        .build()
    )
```

Execute and Test

Run the workflow with ambiguous email content that demonstrates the three-way routing:

Python

```
    # Use ambiguous email content that might trigger uncertain classifica-
tion
    email = (
        "Hey there, I noticed you might be interested in our latest offer—
no pressure, but it expires soon. "
        "Let me know if you'd like more details."
    )

    # Execute and display results
    events = await workflow.run(email)
    outputs = events.get_outputs()
    if outputs:
        for output in outputs:
            print(f"Workflow output: {output}")
```

Key Advantages of Switch-Case Edges

1. **Cleaner Syntax**: One edge group instead of multiple conditional edges
2. **Ordered Evaluation**: Cases are evaluated sequentially, stopping at the first match
3. **Guaranteed Routing**: The default case ensures messages never get stuck
4. **Better Maintainability**: Adding new cases requires minimal changes
5. **Type Safety**: Each executor validates its input to catch routing errors

Comparison: Conditional vs. Switch-Case

Before (Conditional Edges):

Python

```
.add_edge(detector, handler_a, condition=lambda x: x.result == "A")
.add_edge(detector, handler_b, condition=lambda x: x.result == "B")
.add_edge(detector, handler_c, condition=lambda x: x.result == "C")
```

After (Switch-Case):

Python

```
.add_switch_case_edge_group(
    detector,
    [
        Case(condition=lambda x: x.result == "A", target=handler_a),
        Case(condition=lambda x: x.result == "B", target=handler_b),
        Default(target=handler_c),  # Catches everything else
    ],
)
```

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.

Switch-Case Sample Code

For the complete working implementation, see the switch_case_edge_group.py 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 multi-selection edge groups) 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.

Enhanced Data Models for Multi-Selection

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

Python

```python
class AnalysisResultAgent(BaseModel):
    """Enhanced structured output from email analysis agent."""

    spam_decision: Literal["NotSpam", "Spam", "Uncertain"]
    reason: str

class EmailResponse(BaseModel):
    """Response from email assistant."""

    response: str

class EmailSummaryModel(BaseModel):
    """Summary generated by email summary agent."""

    summary: str

@dataclass
class AnalysisResult:
    """Internal analysis result with email metadata for routing
decisions."""

    spam_decision: str
    reason: str
    email_length: int  # Used for conditional routing
    email_summary: str  # Populated by summary agent
    email_id: str

@dataclass
class Email:
    """Email content stored in shared state."""

    email_id: str
    email_content: str

# Custom event for database operations
class DatabaseEvent(WorkflowEvent):
    """Custom event for tracking database operations."""
    pass
```

Selection Function: The Heart of Multi-Selection

The selection function determines which executors should receive each message:

Python

```
LONG_EMAIL_THRESHOLD = 100

def select_targets(analysis: AnalysisResult, target_ids: list[str]) ->
list[str]:
    """Intelligent routing based on spam decision and email
characteristics."""

    # Target order: [handle_spam, submit_to_email_assistant, summarize_e-
mail, handle_uncertain]
    handle_spam_id, submit_to_email_assistant_id, summarize_email_id, han-
dle_uncertain_id = target_ids

    if analysis.spam_decision == "Spam":
        # Route only to spam handler
        return [handle_spam_id]

    elif analysis.spam_decision == "NotSpam":
        # Always route to email assistant
        targets = [submit_to_email_assistant_id]

        # Conditionally add summarizer for long emails
        if analysis.email_length > LONG_EMAIL_THRESHOLD:
            targets.append(summarize_email_id)

        return targets

    else:  # Uncertain
        # Route only to uncertain handler
        return [handle_uncertain_id]
```

Key Features of Selection Functions

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

Multi-Selection Workflow Executors

Implement executors that handle the enhanced analysis and routing:

Python

```
EMAIL_STATE_PREFIX = "email:"
CURRENT_EMAIL_ID_KEY = "current_email_id"

@executor(id="store_email")
async def store_email(email_text: str, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
    """Store email and initiate analysis."""

    new_email = Email(email_id=str(uuid4()), email_content=email_text)
    await ctx.set_shared_state(f"{EMAIL_STATE_PREFIX}{new_email.email_id}",
new_email)
    await ctx.set_shared_state(CURRENT_EMAIL_ID_KEY, new_email.email_id)

    await ctx.send_message(
        AgentExecutorRequest(messages=[ChatMessage(Role.USER, text=new_e-
mail.email_content)], should_respond=True)
    )

@executor(id="to_analysis_result")
async def to_analysis_result(response: AgentExecutorResponse, ctx:
WorkflowContext[AnalysisResult]) -> None:
    """Transform agent response into enriched analysis result."""

    parsed =
AnalysisResultAgent.model_validate_json(response.agent_run_response.text)
    email_id: str = await ctx.get_shared_state(CURRENT_EMAIL_ID_KEY)
    email: Email = await ctx.get_shared_state(f"{EMAIL_STATE_PREFIX}
{email_id}")

    # Create enriched analysis result with email length for routing deci-
sions
    await ctx.send_message(
        AnalysisResult(
            spam_decision=parsed.spam_decision,
            reason=parsed.reason,
            email_length=len(email.email_content),  # Key for conditional
routing
            email_summary="",
            email_id=email_id,
        )
    )

@executor(id="submit_to_email_assistant")
async def submit_to_email_assistant(analysis: AnalysisResult, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
    """Handle legitimate emails by forwarding to email assistant."""

    if analysis.spam_decision != "NotSpam":
        raise RuntimeError("This executor should only handle NotSpam mes-
sages.")

    email: Email = await ctx.get_shared_state(f"{EMAIL_STATE_PREFIX}{analy-
sis.email_id}")
    await ctx.send_message(
```

```
        AgentExecutorRequest(messages=[ChatMessage(Role.USER, text=email.e-
mail_content)], should_respond=True)
    )

@executor(id="finalize_and_send")
async def finalize_and_send(response: AgentExecutorResponse, ctx:
WorkflowContext[Never, str]) -> None:
    """Final step for email assistant branch."""

    parsed =
EmailResponse.model_validate_json(response.agent_run_response.text)
    await ctx.yield_output(f"Email sent: {parsed.response}")

@executor(id="summarize_email")
async def summarize_email(analysis: AnalysisResult, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
    """Generate summary for long emails (parallel branch)."""

    # Only called for long NotSpam emails by selection function
    email: Email = await ctx.get_shared_state(f"{EMAIL_STATE_PREFIX}{analy-
sis.email_id}")
    await ctx.send_message(
        AgentExecutorRequest(messages=[ChatMessage(Role.USER, text=email.e-
mail_content)], should_respond=True)
    )

@executor(id="merge_summary")
async def merge_summary(response: AgentExecutorResponse, ctx:
WorkflowContext[AnalysisResult]) -> None:
    """Merge summary back into analysis result for database persistence."""

    summary =
EmailSummaryModel.model_validate_json(response.agent_run_response.text)
    email_id: str = await ctx.get_shared_state(CURRENT_EMAIL_ID_KEY)
    email: Email = await ctx.get_shared_state(f"{EMAIL_STATE_PREFIX}
{email_id}")

    # Create analysis result with summary for database storage
    await ctx.send_message(
        AnalysisResult(
            spam_decision="NotSpam",
            reason="",
            email_length=len(email.email_content),
            email_summary=summary.summary,  # Now includes summary
            email_id=email_id,
        )
    )

@executor(id="handle_spam")
async def handle_spam(analysis: AnalysisResult, ctx: WorkflowContext[Never,
str]) -> None:
    """Handle spam emails (single target like switch-case)."""

    if analysis.spam_decision == "Spam":
        await ctx.yield_output(f"Email marked as spam: {analysis.reason}")
```

```
    else:
        raise RuntimeError("This executor should only handle Spam
messages.")

@executor(id="handle_uncertain")
async def handle_uncertain(analysis: AnalysisResult, ctx:
WorkflowContext[Never, str]) -> None:
    """Handle uncertain emails (single target like switch-case)."""

    if analysis.spam_decision == "Uncertain":
        email: Email | None = await ctx.get_shared_state(f"{EMAIL_S-
TATE_PREFIX}{analysis.email_id}")
        await ctx.yield_output(
            f"Email marked as uncertain: {analysis.reason}. Email content:
{getattr(email, 'email_content', '')}"
        )
    else:
        raise RuntimeError("This executor should only handle Uncertain mes-
sages.")

@executor(id="database_access")
async def database_access(analysis: AnalysisResult, ctx:
WorkflowContext[Never, str]) -> None:
    """Simulate database persistence with custom events."""

    await asyncio.sleep(0.05)  # Simulate DB operation
    await ctx.add_event(DatabaseEvent(f"Email {analysis.email_id} saved to
database."))
```

Enhanced AI Agents

Create agents for analysis, assistance, and summarization:

Python

```
async def main() -> None:
    chat_client = AzureOpenAIChatClient(credential=AzureCliCredential())

    # Enhanced analysis agent
    email_analysis_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are a spam detection assistant that identifies spam
emails. "
                "Always return JSON with fields 'spam_decision' (one of
NotSpam, Spam, Uncertain) "
                "and 'reason' (string)."
            ),
            response_format=AnalysisResultAgent,
        ),
        id="email_analysis_agent",
    )
```

```
    # Email assistant (same as before)
    email_assistant_agent = AgentExecutor(
        chat_client.create_agent(
            instructions=(
                "You are an email assistant that helps users draft respons-
es to emails with professionalism."
            ),
            response_format=EmailResponse,
        ),
        id="email_assistant_agent",
    )

    # New: Email summary agent for long emails
    email_summary_agent = AgentExecutor(
        chat_client.create_agent(
            instructions="You are an assistant that helps users summarize
emails.",
            response_format=EmailSummaryModel,
        ),
        id="email_summary_agent",
    )
```

Build Multi-Selection Workflow

Construct the workflow with sophisticated routing and parallel processing:

Python

```
    workflow = (
        WorkflowBuilder()
        .set_start_executor(store_email)
        .add_edge(store_email, email_analysis_agent)
        .add_edge(email_analysis_agent, to_analysis_result)

        # Multi-selection edge group: intelligent fan-out based on content
        .add_multi_selection_edge_group(
            to_analysis_result,
            [handle_spam, submit_to_email_assistant, summarize_email, han-
dle_uncertain],
            selection_func=select_targets,
        )

        # Email assistant branch (always for NotSpam)
        .add_edge(submit_to_email_assistant, email_assistant_agent)
        .add_edge(email_assistant_agent, finalize_and_send)

        # Summary branch (only for long NotSpam emails)
        .add_edge(summarize_email, email_summary_agent)
        .add_edge(email_summary_agent, merge_summary)

        # Database persistence: conditional routing
```

```
        .add_edge(to_analysis_result, database_access,
                  condition=lambda r: r.email_length <=
LONG_EMAIL_THRESHOLD)  # Short emails
        .add_edge(merge_summary, database_access)  # Long emails with sum-
mary

        .build()
    )
```

Execution with Event Streaming

Run the workflow and observe parallel execution through custom events:

Python

```
    # Use a moderately long email to trigger both assistant and summarizer
    email = """
    Hello team, here are the updates for this week:

    1. Project Alpha is on track and we should have the first milestone
completed by Friday.
    2. The client presentation has been scheduled for next Tuesday at 2 PM.
    3. Please review the Q4 budget allocation and provide feedback by
Wednesday.

    Let me know if you have any questions or concerns.

    Best regards,
    Alex
    """

    # Stream events to see parallel execution
    async for event in workflow.run_stream(email):
        if isinstance(event, DatabaseEvent):
            print(f"Database: {event}")
        elif isinstance(event, WorkflowOutputEvent):
            print(f"Output: {event.data}")
```

Multi-Selection vs. Switch-Case Comparison

Switch-Case Pattern (Previous):

Python

```
# One input → exactly one output
.add_switch_case_edge_group(
    source,
    [
        Case(condition=lambda x: x.result == "A", target=handler_a),
```

```
        Case(condition=lambda x: x.result == "B", target=handler_b),
        Default(target=handler_c),
    ],
)
```

Multi-Selection Pattern:

Python

```
# One input → one or more outputs (dynamic fan-out)
.add_multi_selection_edge_group(
    source,
    [handler_a, handler_b, handler_c, handler_d],
    selection_func=intelligent_router,  # Returns list of target IDs
)
```

C# Multi-Selection Benefits

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

C# Real-World Applications

- **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 Sample Code

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

Next Steps

Learn about handling requests and responses in workflows

Create and run an agent with Agent Framework

10/02/2025

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:

- Python 3.10 or later
- 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.

> **Important**
>
> For this tutorial we are using Azure OpenAI for the Chat Completion service, but you can use any inference service that is compatible with the Agent Framework's chat client protocol.

Installing Python packages

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

Bash

```
pip install agent-framework
```

Creating the agent

- First we create a chat client for communicating with Azure OpenAI, where we use the same login as was used when authenticating with the Azure CLI in the Prerequisites step.
- Then we create the agent, providing instructions and a name for the agent.

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    instructions="You are good at telling jokes.",
    name="Joker"
)
```

Running the agent

To run the agent, call the `run` method on the agent instance, providing the user input. The agent will return a response object, and accessing the `.text` property provides the text result from the agent.

Python

```
async def main():
    result = await agent.run("Tell me a joke about a pirate.")
    print(result.text)

asyncio.run(main())
```

Running the agent with streaming

To run the agent with streaming, call the `run_stream` method on the agent instance, providing the user input. The agent will stream a list of update objects, and accessing the `.text` property on each update object provides the part of the text result contained in that update.

Python

```
async def main():
    async for update in agent.run_stream("Tell me a joke about a pirate."):
        if update.text:
            print(update.text, end="", flush=True)
```

```
    print()  # New line after streaming is complete

asyncio.run(main())
```

Running the agent with a ChatMessage

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

Python

```
from agent_framework import ChatMessage, TextContent, UriContent, Role

message = ChatMessage(
    role=Role.USER,
    contents=[
        TextContent(text="Tell me a joke about this image?"),
        UriContent(uri="https://samplesite.org/clown.jpg", media_type="im-
age/jpeg")
    ]
)

async def main():
    result = await agent.run(message)
    print(result.text)

asyncio.run(main())
```

Next steps

Using images with an agent

Custom Agents

10/02/2025

The Microsoft Agent Framework supports building custom agents by inheriting from the `BaseAgent` class and implementing the required methods.

This document shows how to build a simple custom agent that echoes back user input with a prefix. In most cases building your own agent will involve more complex logic and integration with an AI service.

Getting Started

Add the required Python packages to your project.

Bash

```
pip install agent-framework-core
```

Creating a Custom Agent

The Agent Protocol

The framework provides the `AgentProtocol` protocol that defines the interface all agents must implement. Custom agents can either implement this protocol directly or extend the `BaseAgent` class for convenience.

Python

```
from agent_framework import AgentProtocol, AgentRunResponse,
AgentRunResponseUpdate, AgentThread, ChatMessage
from collections.abc import AsyncIterable
from typing import Any

class MyCustomAgent(AgentProtocol):
    """A custom agent that implements the AgentProtocol directly."""

    @property
    def id(self) -> str:
        """Returns the ID of the agent."""
        ...

    async def run(
        self,
```

```
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AgentRunResponse:
        """Execute the agent and return a complete response."""
        ...

    def run_stream(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AsyncIterable[AgentRunResponseUpdate]:
        """Execute the agent and yield streaming response updates."""
        ...
```

Using BaseAgent

The recommended approach is to extend the `BaseAgent` class, which provides common functionality and simplifies implementation:

Python

```
from agent_framework import (
    BaseAgent,
    AgentRunResponse,
    AgentRunResponseUpdate,
    AgentThread,
    ChatMessage,
    Role,
    TextContent,
)
from collections.abc import AsyncIterable
from typing import Any


class EchoAgent(BaseAgent):
    """A simple custom agent that echoes user messages with a prefix."""

    echo_prefix: str = "Echo: "

    def __init__(
        self,
        *,
        name: str | None = None,
        description: str | None = None,
        echo_prefix: str = "Echo: ",
```

```
        **kwargs: Any,
    ) -> None:
        """Initialize the EchoAgent.

        Args:
            name: The name of the agent.
            description: The description of the agent.
            echo_prefix: The prefix to add to echoed messages.
            **kwargs: Additional keyword arguments passed to BaseAgent.
        """
        super().__init__(
            name=name,
            description=description,
            echo_prefix=echo_prefix,
            **kwargs,
        )

    async def run(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AgentRunResponse:
        """Execute the agent and return a complete response.

        Args:
            messages: The message(s) to process.
            thread: The conversation thread (optional).
            **kwargs: Additional keyword arguments.

        Returns:
            An AgentRunResponse containing the agent's reply.
        """
        # Normalize input messages to a list
        normalized_messages = self._normalize_messages(messages)

        if not normalized_messages:
            response_message = ChatMessage(
                role=Role.ASSISTANT,
                contents=[TextContent(text="Hello! I'm a custom echo agent.
Send me a message and I'll echo it back.")],
            )
        else:
            # For simplicity, echo the last user message
            last_message = normalized_messages[-1]
            if last_message.text:
                echo_text = f"{self.echo_prefix}{last_message.text}"
            else:
                echo_text = f"{self.echo_prefix}[Non-text message
received]"

            response_message = ChatMessage(role=Role.ASSISTANT, contents=
[TextContent(text=echo_text)])
```

```
        # Notify the thread of new messages if provided
        if thread is not None:
            await self._notify_thread_of_new_messages(thread, normal-
ized_messages, response_message)

        return AgentRunResponse(messages=[response_message])

    async def run_stream(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AsyncIterable[AgentRunResponseUpdate]:
        """Execute the agent and yield streaming response updates.

        Args:
            messages: The message(s) to process.
            thread: The conversation thread (optional).
            **kwargs: Additional keyword arguments.

        Yields:
            AgentRunResponseUpdate objects containing chunks of the re-
sponse.
        """
        # Normalize input messages to a list
        normalized_messages = self._normalize_messages(messages)

        if not normalized_messages:
            response_text = "Hello! I'm a custom echo agent. Send me a mes-
sage and I'll echo it back."
        else:
            # For simplicity, echo the last user message
            last_message = normalized_messages[-1]
            if last_message.text:
                response_text = f"{self.echo_prefix}{last_message.text}"
            else:
                response_text = f"{self.echo_prefix}[Non-text message re-
ceived]"

        # Simulate streaming by yielding the response word by word
        words = response_text.split()
        for i, word in enumerate(words):
            # Add space before word except for the first one
            chunk_text = f" {word}" if i > 0 else word

            yield AgentRunResponseUpdate(
                contents=[TextContent(text=chunk_text)],
                role=Role.ASSISTANT,
            )

            # Small delay to simulate streaming
            await asyncio.sleep(0.1)
```

```
        # Notify the thread of the complete response if provided
        if thread is not None:
            complete_response = ChatMessage(role=Role.ASSISTANT, contents=
[TextContent(text=response_text)])
            await self._notify_thread_of_new_messages(thread, normal-
ized_messages, complete_response)
```

Using the Agent

If agent methods are all implemented correctly, the agent would support all standard agent operations.

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

Next steps

Running Agents

Enabling observability for Agents

10/02/2025

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.

Prerequisites

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

Installing packages

To use the Agent Framework with Azure OpenAI, you need to install the following packages. The agent framework automatically includes all necessary OpenTelemetry dependencies:

Bash

```
pip install agent-framework
```

The following OpenTelemetry packages are included by default:

text

```
opentelemetry-api
opentelemetry-sdk
azure-monitor-opentelemetry
azure-monitor-opentelemetry-exporter
opentelemetry-exporter-otlp-proto-grpc
opentelemetry-semantic-conventions-ai
```

Enable OpenTelemetry in your app

The agent framework provides a convenient `setup_observability` function that configures OpenTelemetry with sensible defaults. By default, it exports to the console if no specific exporter is configured.

Python

```
import asyncio
from agent_framework.observability import setup_observability
```

```
# Enable agent framework telemetry with console output (default behavior)
setup_observability(enable_sensitive_data=True)
```

Understanding `setup_observability` parameters

The `setup_observability` function accepts the following parameters to customize your observability configuration:

- **`enable_otel`** (bool, optional): Enables OpenTelemetry tracing and metrics. Default is `False` when using environment variables only, but is assumed `True` when calling `setup_observability()` programmatically. When using environment variables, set `ENABLE_OTEL=true`.

- **`enable_sensitive_data`** (bool, optional): Controls whether sensitive data like prompts, responses, function call arguments, and results are included in traces. Default is `False`. Set to `True` to see actual prompts and responses in your traces. **Warning**: Be careful with this setting as it may expose sensitive data in your logs. Can also be set via `ENABLE_SENSITIVE_DATA=true` environment variable.

- **`otlp_endpoint`** (str, optional): The OTLP endpoint URL for exporting telemetry data. Default is `None`. Commonly set to `http://localhost:4317`. This creates an OTLPExporter for spans, metrics, and logs. Can be used with any OTLP-compliant endpoint such as OpenTelemetry Collector , Aspire Dashboard, or other OTLP endpoints. Can also be set via `OTLP_ENDPOINT` environment variable.

- **`applicationinsights_connection_string`** (str, optional): Azure Application Insights connection string for exporting to Azure Monitor. Default is `None`. Creates AzureMonitorTraceExporter, AzureMonitorMetricExporter, and AzureMonitorLogExporter. You can find this connection string in the Azure portal under the "Overview" section of your Application Insights resource. Can also be set via `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable.

- **`vs_code_extension_port`** (int, optional): Port number for the AI Toolkit or Azure AI Foundry VS Code extension. Default is `4317`. Allows integration with VS Code extensions for local development and debugging. Can also be set via `VS_CODE_EXTENSION_PORT` environment variable.

- **`exporters`** (list, optional): Custom list of OpenTelemetry exporters for advanced scenarios. Default is `None`. Allows you to provide your own configured exporters when the standard options don't meet your needs.

Setup options

You can configure observability in three ways:

1. Environment variables (simplest approach):

```bash
export ENABLE_OTEL=true
export ENABLE_SENSITIVE_DATA=true
export OTLP_ENDPOINT=http://localhost:4317
```

Then in your code:

```python
from agent_framework.observability import setup_observability

setup_observability()  # Reads from environment variables
```

2. Programmatic configuration:

```python
from agent_framework.observability import setup_observability

setup_observability(
    enable_sensitive_data=True,
    otlp_endpoint="http://localhost:4317",
    applicationinsights_connection_string="InstrumentationKey=your_key"
)
```

3. Custom exporters (for advanced scenarios):

```python
from agent_framework.observability import setup_observability
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter
from opentelemetry.sdk.trace.export import ConsoleSpanExporter

custom_exporters = [
    OTLPSpanExporter(endpoint="http://localhost:4317"),
    ConsoleSpanExporter()
]

setup_observability(exporters=custom_exporters, enable_sensitive_data=True)
```

The `setup_observability` function sets the global tracer provider and meter provider, allowing you to create custom spans and metrics:

Python

```python
from agent_framework.observability import get_tracer, get_meter

tracer = get_tracer()
meter = get_meter()

with tracer.start_as_current_span("my_custom_span"):
    # Your code here
    pass

counter = meter.create_counter("my_custom_counter")
counter.add(1, {"key": "value"})
```

Create and run the agent

Create an agent using the agent framework. The observability will be automatically enabled for the agent once `setup_observability` has been called.

Python

```python
from agent_framework import ChatAgent
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

# Create the agent - telemetry is automatically enabled
agent = ChatAgent(
    chat_client=AzureOpenAIChatClient(
        credential=AzureCliCredential(),
        model="gpt-4o-mini"
    ),
    name="Joker",
    instructions="You are good at telling jokes."
)

# Run the agent
result = await agent.run("Tell me a joke about a pirate.")
print(result.text)
```

The console exporter will show trace data on the console similar to the following:

text

```
{
    "name": "invoke_agent Joker",
```

```
    "context": {
        "trace_id": "0xf2258b51421fe9cf4c0bd428c87b1ae4",
        "span_id": "0x2cad6fc139dcf01d",
        "trace_state": "[]"
    },
    "kind": "SpanKind.CLIENT",
    "parent_id": null,
    "start_time": "2025-09-25T11:00:48.663688Z",
    "end_time": "2025-09-25T11:00:57.271389Z",
    "status": {
        "status_code": "UNSET"
    },
    "attributes": {
        "gen_ai.operation.name": "invoke_agent",
        "gen_ai.system": "openai",
        "gen_ai.agent.id": "Joker",
        "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
    }
}
```


Followed by the text response from the agent:

text

```
Why did the pirate go to school?

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

Understanding the telemetry output

Once observability is enabled, the agent framework automatically creates the following spans:

- **`invoke_agent <agent_name>`**: The top-level span for each agent invocation. Contains all other spans as children and includes metadata like agent ID, name, and instructions.
- **`chat <model_name>`**: Created when the agent calls the underlying chat model. Includes the prompt and response as attributes when `enable_sensitive_data` is `True`, along with token usage information.
- **`execute_tool <function_name>`**: Created when the agent calls a function tool. Contains function arguments and results as attributes when `enable_sensitive_data` is `True`.

The following metrics are also collected:

For chat operations:

- `gen_ai.client.operation.duration` (histogram): Duration of each operation in seconds
- `gen_ai.client.token.usage` (histogram): Token usage in number of tokens

For function invocations:

- `agent_framework.function.invocation.duration` (histogram): Duration of each function execution in seconds

Azure AI Foundry integration

If you're using Azure AI Foundry, there's a convenient method for automatic setup:

Python

```python
from agent_framework.azure import AzureAIAgentClient
from azure.identity import AzureCliCredential

agent_client = AzureAIAgentClient(
    credential=AzureCliCredential(),
    project_endpoint="https://<your-project>.foundry.azure.com"
)

# Automatically configures observability with Application Insights
await agent_client.setup_azure_ai_observability()
```

This method retrieves the Application Insights connection string from your Azure AI Foundry project and calls `setup_observability` automatically.

Next steps

For more advanced observability scenarios and examples, see the Agent Observability user guide and the observability samples in the GitHub repository.

Persisting conversations

Exposing an agent as an MCP tool

10/02/2025

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

Handle Requests and Responses in Workflows

10/02/2025

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.

What You'll Build

You'll create an interactive number guessing game workflow that demonstrates request-response patterns:

- An AI agent that makes intelligent guesses
- A `RequestInfoExecutor` that pauses the workflow to request human input
- A turn manager that coordinates between the agent and human interactions
- Interactive console input/output for real-time feedback

Prerequisites

- Python 3.10 or later
- Azure OpenAI deployment configured
- Azure CLI authentication configured (`az login`)
- Basic understanding of Python async programming

Key Concepts

RequestInfoExecutor

`RequestInfoExecutor` is a specialized workflow component that:

- Pauses workflow execution to request external information
- Emits a `RequestInfoEvent` with typed payload
- Resumes execution after receiving a correlated response
- Preserves request-response correlation via unique request IDs

Request-Response Flow

1. Workflow sends a `RequestInfoMessage` to `RequestInfoExecutor`
2. `RequestInfoExecutor` emits a `RequestInfoEvent`
3. External system (human, API, etc.) processes the request
4. Response is sent back via `send_responses_streaming()`
5. Workflow resumes with the response data

Setting Up the Environment

First, install the required packages:

Bash

```
pip install agent-framework-core
pip install azure-identity
pip install pydantic
```

Define Request and Response Models

Start by defining the data structures for request-response communication:

Python

```
import asyncio
from dataclasses import dataclass
from pydantic import BaseModel

from agent_framework import (
    AgentExecutor,
    AgentExecutorRequest,
    AgentExecutorResponse,
    ChatMessage,
    Executor,
    RequestInfoEvent,
    RequestInfoExecutor,
    RequestInfoMessage,
    RequestResponse,
    Role,
    WorkflowBuilder,
    WorkflowContext,
    WorkflowOutputEvent,
    WorkflowRunState,
    WorkflowStatusEvent,
    handler,
)
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

@dataclass
```

```python
class HumanFeedbackRequest(RequestInfoMessage):
    """Request message for human feedback in the guessing game."""
    prompt: str = ""
    guess: int | None = None

class GuessOutput(BaseModel):
    """Structured output from the AI agent with response_format
enforcement."""
    guess: int
```

The `HumanFeedbackRequest` inherits from `RequestInfoMessage`, which provides:

- Strong typing for request payloads
- Forward-compatible validation
- Clear correlation semantics with responses
- Contextual fields (like the previous guess) for rich UI prompts

Create the Turn Manager

The turn manager coordinates the flow between the AI agent and human:

Python

```python
class TurnManager(Executor):
    """Coordinates turns between the AI agent and human player.

    Responsibilities:
    - Start the game by requesting the agent's first guess
    - Process agent responses and request human feedback
    - Handle human feedback and continue the game or finish
    """

    def __init__(self, id: str | None = None):
        super().__init__(id=id or "turn_manager")

    @handler
    async def start(self, _: str, ctx:
WorkflowContext[AgentExecutorRequest]) -> None:
        """Start the game by asking the agent for an initial guess."""
        user = ChatMessage(Role.USER, text="Start by making your first
guess.")
        await ctx.send_message(AgentExecutorRequest(messages=[user],
should_respond=True))

    @handler
    async def on_agent_response(
        self,
        result: AgentExecutorResponse,
        ctx: WorkflowContext[HumanFeedbackRequest],
    ) -> None:
```

```
        """Handle the agent's guess and request human guidance."""
        # Parse structured model output (defensive default if agent didn't
reply)
        text = result.agent_run_response.text or ""
        last_guess = GuessOutput.model_validate_json(text).guess if text
else None

        # Craft a clear human prompt that defines higher/lower relative to
agent's guess
        prompt = (
            f"The agent guessed: {last_guess if last_guess is not None
else text}. "
            "Type one of: higher (your number is higher than this guess), "
            "lower (your number is lower than this guess), correct, or
exit."
        )
        await ctx.send_message(HumanFeedbackRequest(prompt=prompt,
guess=last_guess))

    @handler
    async def on_human_feedback(
        self,
        feedback: RequestResponse[HumanFeedbackRequest, str],
        ctx: WorkflowContext[AgentExecutorRequest, str],
    ) -> None:
        """Continue the game or finish based on human feedback."""
        reply = (feedback.data or "").strip().lower()
        # Use the correlated request's guess to avoid extra state reads
        last_guess = getattr(feedback.original_request, "guess", None)

        if reply == "correct":
            await ctx.yield_output(f"Guessed correctly: {last_guess}")
            return

        # Provide feedback to the agent for the next guess
        user_msg = ChatMessage(
            Role.USER,
            text=f'Feedback: {reply}. Return ONLY a JSON object matching
the schema {{"guess": <int 1..10>}}.',
        )
        await ctx.send_message(AgentExecutorRequest(messages=[user_msg],
should_respond=True))
```

Build the Workflow

Create the main workflow that connects all components:

Python

```
async def main() -> None:
    # Create the chat agent with structured output enforcement
    chat_client = AzureOpenAIChatClient(credential=AzureCliCredential())
```

```
    agent = chat_client.create_agent(
        instructions=(
            "You guess a number between 1 and 10. "
            "If the user says 'higher' or 'lower', adjust your next guess.
"
            'You MUST return ONLY a JSON object exactly matching this
schema: {"guess": <integer 1..10>}. '
            "No explanations or additional text."
        ),
        response_format=GuessOutput,
    )

    # Create workflow components
    turn_manager = TurnManager(id="turn_manager")
    agent_exec = AgentExecutor(agent=agent, id="agent")
    request_info_executor = RequestInfoExecutor(id="request_info")

    # Build the workflow graph
    workflow = (
        WorkflowBuilder()
        .set_start_executor(turn_manager)
        .add_edge(turn_manager, agent_exec)  # Ask agent to make/adjust a
guess
        .add_edge(agent_exec, turn_manager)  # Agent's response goes back
to coordinator
        .add_edge(turn_manager, request_info_executor)  # Ask human for
guidance
        .add_edge(request_info_executor, turn_manager)  # Feed human guid-
ance back to coordinator
        .build()
    )

    # Execute the interactive workflow
    await run_interactive_workflow(workflow)

async def run_interactive_workflow(workflow):
    """Run the workflow with human-in-the-loop interaction."""
    pending_responses: dict[str, str] | None = None
    completed = False
    workflow_output: str | None = None

    print("🎯 Number Guessing Game")
    print("Think of a number between 1 and 10, and I'll try to guess it!")
    print("-" * 50)

    while not completed:
        # First iteration uses run_stream("start")
        # Subsequent iterations use send_responses_streaming with pending
responses
        stream = (
            workflow.send_responses_streaming(pending_responses)
            if pending_responses
            else workflow.run_stream("start")
        )
```

```
        # Collect events for this turn
        events = [event async for event in stream]
        pending_responses = None

        # Process events to collect requests and detect completion
        requests: list[tuple[str, str]] = []  # (request_id, prompt)
        for event in events:
            if isinstance(event, RequestInfoEvent) and isinstance(event.da-
ta, HumanFeedbackRequest):
                # RequestInfoEvent for our HumanFeedbackRequest
                requests.append((event.request_id, event.data.prompt))
            elif isinstance(event, WorkflowOutputEvent):
                # Capture workflow output when yielded
                workflow_output = str(event.data)
                completed = True

        # Check workflow status
        pending_status = any(
            isinstance(e, WorkflowStatusEvent) and e.state ==
WorkflowRunState.IN_PROGRESS_PENDING_REQUESTS
            for e in events
        )
        idle_with_requests = any(
            isinstance(e, WorkflowStatusEvent) and e.state ==
WorkflowRunState.IDLE_WITH_PENDING_REQUESTS
            for e in events
        )

        if pending_status:
            print("🔄 State: IN_PROGRESS_PENDING_REQUESTS (requests out-
standing)")
        if idle_with_requests:
            print("⏸️  State: IDLE_WITH_PENDING_REQUESTS (awaiting human
input)")

        # Handle human requests if any
        if requests and not completed:
            responses: dict[str, str] = {}
            for req_id, prompt in requests:
                print(f"\n🤖 {prompt}")
                answer = input("👤 Enter higher/lower/correct/exit: ").low-
er()

                if answer == "exit":
                    print("👋 Exiting...")
                    return
                responses[req_id] = answer
            pending_responses = responses

    # Show final result
    print(f"\n🎉 {workflow_output}")
```

Running the Example

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

How It Works

1. **Workflow Initialization**: The workflow starts with the `TurnManager` requesting an initial guess from the AI agent.
2. **Agent Response**: The AI agent makes a guess and returns structured JSON, which flows back to the `TurnManager`.
3. **Human Request**: The `TurnManager` processes the agent's guess and sends a `HumanFeedbackRequest` to the `RequestInfoExecutor`.
4. **Workflow Pause**: The `RequestInfoExecutor` emits a `RequestInfoEvent` and the workflow pauses, waiting for human input.
5. **Human Response**: The external application collects human input and sends responses back using `send_responses_streaming()`.
6. **Resume and Continue**: The workflow resumes, the `TurnManager` processes the human feedback, and either ends the game or sends another request to the agent.

Key Benefits

- **Structured Communication**: Type-safe request and response models prevent runtime errors
- **Correlation**: Request IDs ensure responses are matched to the correct requests
- **Pausable Execution**: Workflows can pause indefinitely while waiting for external input
- **State Preservation**: Workflow state is maintained across pause-resume cycles
- **Event-Driven**: Rich event system provides visibility into workflow status and transitions

This pattern enables building sophisticated interactive applications where AI agents and humans collaborate seamlessly within structured workflows.

Next Steps

Learn about checkpointing and resuming workflows

Microsoft Agent Framework

10/02/2025

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
```

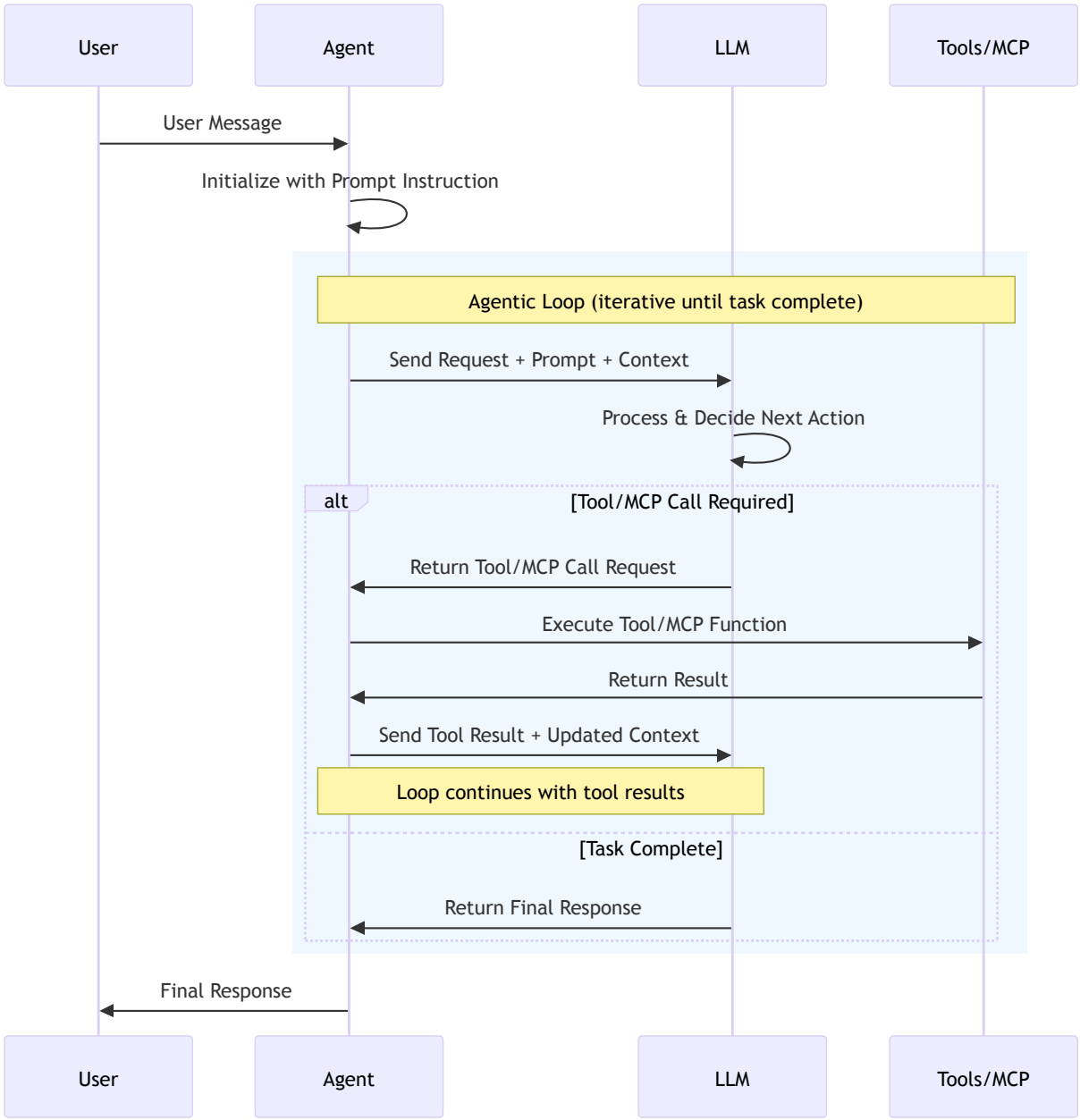
.NET:

```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.

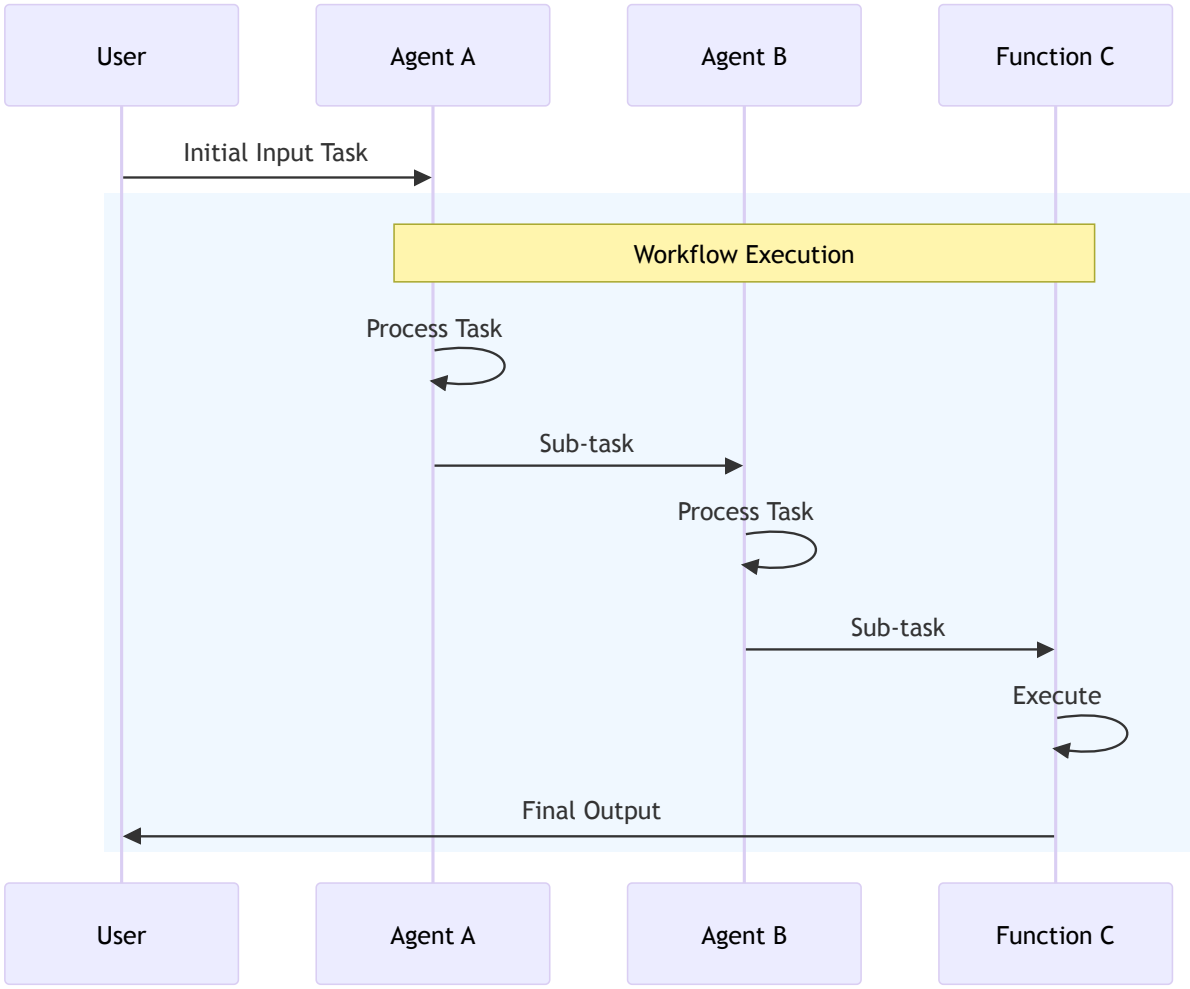
In these cases, consider using workflows instead.

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

Using MCP tools with Foundry Agents

10/02/2025

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

Python Azure AI Foundry MCP Integration

Azure AI Foundry provides seamless integration with Model Context Protocol (MCP) servers through the Python Agent Framework. The service manages the MCP server hosting and execution, eliminating infrastructure management while providing secure, controlled access to external tools.

Environment Setup

Configure your Azure AI Foundry project credentials through environment variables:

Python

```
import os
from azure.identity.aio import AzureCliCredential
from agent_framework.azure import AzureAIAgentClient
```

```
# Required environment variables
os.environ["AZURE_AI_PROJECT_ENDPOINT"] = "https://<your-project>.ser-
vices.ai.azure.com/api/projects/<project-id>"
os.environ["AZURE_AI_MODEL_DEPLOYMENT_NAME"] = "gpt-4o-mini"  # Optional,
defaults to this
```

Basic MCP Integration

Create an Azure AI Foundry agent with hosted MCP tools:

Python

```
import asyncio
from agent_framework import HostedMCPTool
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def basic_foundry_mcp_example():
    """Basic example of Azure AI Foundry agent with hosted MCP tools."""
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential) as chat_client,
    ):
        # Enable Azure AI observability (optional but recommended)
        await chat_client.setup_azure_ai_observability()

        # Create agent with hosted MCP tool
        agent = chat_client.create_agent(
            name="MicrosoftLearnAgent",
            instructions="You answer questions by searching Microsoft Learn
content only.",
            tools=HostedMCPTool(
                name="Microsoft Learn MCP",
                url="https://learn.microsoft.com/api/mcp",
            ),
        )

        # Simple query without approval workflow
        result = await agent.run(
            "Please summarize the Azure AI Agent documentation related to
MCP tool calling?"
        )
        print(result)

if __name__ == "__main__":
    asyncio.run(basic_foundry_mcp_example())
```

Multi-Tool MCP Configuration

Use multiple hosted MCP tools with a single agent:

Python

```
async def multi_tool_mcp_example():
    """Example using multiple hosted MCP tools."""
    async with (
        AzureCliCredential() as credential,
        AzureAIAgentClient(async_credential=credential) as chat_client,
    ):
        await chat_client.setup_azure_ai_observability()

        # Create agent with multiple MCP tools
        agent = chat_client.create_agent(
            name="MultiToolAgent",
            instructions="You can search documentation and access GitHub
repositories.",
            tools=[
                HostedMCPTool(
                    name="Microsoft Learn MCP",
                    url="https://learn.microsoft.com/api/mcp",
                    approval_mode="never_require",  # Auto-approve documen-
tation searches
                ),
                HostedMCPTool(
                    name="GitHub MCP",
                    url="https://api.github.com/mcp",
                    approval_mode="always_require",  # Require approval for
GitHub operations
                    headers={"Authorization": "Bearer github-token"},
                ),
            ],
        )

        result = await agent.run(
            "Find Azure documentation and also check the latest commits in
microsoft/semantic-kernel"
        )
        print(result)

if __name__ == "__main__":
    asyncio.run(multi_tool_mcp_example())
```

The Python Agent Framework provides seamless integration with Azure AI Foundry's hosted MCP capabilities, enabling secure and scalable access to external tools while maintaining the flexibility and control needed for production applications.

Next steps

Using workflows as Agents

Microsoft Agent Framework Agent Types

10/02/2025

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 a chat client implementation can be used to build these agents.

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
5. Streaming responses

To create one of these agents, simply construct a `ChatAgent` using the chat client implementation of your choice.

Python

```
from agent_framework import ChatAgent
from agent_framework.azure import AzureAIAgentClient
```

```
from azure.identity.aio import DefaultAzureCredential

async with (
    DefaultAzureCredential() as credential,
    ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        instructions="You are a helpful assistant"
    ) as agent
):
    response = await agent.run("Hello!")
```

Alternatively, you can use the convenience method on the chat client:

Python

```
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import DefaultAzureCredential

async with DefaultAzureCredential() as credential:
    agent = AzureAIAgentClient(async_credential=credential).create_agent(
        instructions="You are a helpful assistant"
    )
```

For detailed examples, see the agent-specific documentation sections below.

Supported Agent Types

Expand table

Underlying Inference Service	Description	Service Chat History storage supported	Custom Chat History storage supported
Azure AI Agent	An agent that uses the Azure AI Agents Service as its backend.	Yes	No
Azure OpenAI Chat Completion	An agent that uses the Azure OpenAI Chat Completion service.	No	Yes
Azure OpenAI Responses	An agent that uses the Azure OpenAI Responses service.	Yes	Yes
OpenAI Chat Completion	An agent that uses the OpenAI Chat Completion service.	No	Yes
OpenAI Responses	An agent that uses the OpenAI Responses service.	Yes	Yes

Underlying Inference Service	Description	Service Chat History storage supported	Custom Chat History storage supported
OpenAI Assistants	An agent that uses the OpenAI Assistants service.	Yes	No
Any other ChatClient	You can also use any other chat client implementation to create an agent.	Varies	Varies

Function Tools

You can provide function tools to agents for enhanced capabilities:

Python

```python
from typing import Annotated
from pydantic import Field
from azure.identity.aio import DefaultAzureCredential
from agent_framework.azure import AzureAIAgentClient

def get_weather(location: Annotated[str, Field(description="The location to
get the weather for.")]) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with a high of 25°C."

async with (
    DefaultAzureCredential() as credential,
    AzureAIAgentClient(async_credential=credential).create_agent(
        instructions="You are a helpful weather assistant.",
        tools=get_weather
    ) as agent
):
    response = await agent.run("What's the weather in Seattle?")
```

For complete examples with function tools, see:

- Azure AI with function tools
- Azure OpenAI with function tools
- OpenAI with function tools

Streaming Responses

Agents support both regular and streaming responses:

Python

```
# Regular response (wait for complete result)
response = await agent.run("What's the weather like in Seattle?")
print(response.text)

# Streaming response (get results as they are generated)
async for chunk in agent.run_stream("What's the weather like in
Portland?"):
    if chunk.text:
        print(chunk.text, end="", flush=True)
```

For streaming examples, see:

- Azure AI streaming examples
- Azure OpenAI streaming examples
- OpenAI streaming examples

Code Interpreter Tools

Azure AI agents support hosted code interpreter tools for executing Python code:

Python

```
from agent_framework import ChatAgent, HostedCodeInterpreterTool
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import DefaultAzureCredential

async with (
    DefaultAzureCredential() as credential,
    ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        instructions="You are a helpful assistant that can execute Python
code.",
        tools=HostedCodeInterpreterTool()
    ) as agent
):
    response = await agent.run("Calculate the factorial of 100 using
Python")
```

For code interpreter examples, see:

- Azure AI with code interpreter
- Azure OpenAI Assistants with code interpreter
- OpenAI Assistants with code interpreter

Custom agents

It is also possible to create fully custom agents that are not just wrappers around a chat client. Agent Framework provides the `AgentProtocol` protocol and `BaseAgent` base class, which when implemented/subclassed allows for complete control over the agent's behavior and capabilities.

Python

```
from agent_framework import BaseAgent, AgentRunResponse,
AgentRunResponseUpdate, AgentThread, ChatMessage
from collections.abc import AsyncIterable

class CustomAgent(BaseAgent):
    async def run(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AgentRunResponse:
        # Custom agent implementation
        pass

    def run_stream(
        self,
        messages: str | ChatMessage | list[str] | list[ChatMessage] | None
= None,
        *,
        thread: AgentThread | None = None,
        **kwargs: Any,
    ) -> AsyncIterable[AgentRunResponseUpdate]:
        # Custom streaming implementation
        pass
```

Microsoft Agent Framework Multi-Turn Conversations and Threading

10/02/2025

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.

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.

AgentThread Creation

`AgentThread` instances can be created in two ways:

1. By calling `get_new_thread()` 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. Azure AI Agents or OpenAI Responses. Any cleanup or deletion of these threads is the responsibility of the user.

Python

```python
# Create a new thread.
thread = agent.get_new_thread()
# Run the agent with the thread.
response = await agent.run("Hello, how are you?", thread=thread)

# Run an agent with a temporary thread.
response = await agent.run("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.

Python

```python
# Create a new thread.
thread = agent.get_new_thread()
# Run the agent with the thread.
response = await agent.run("Hello, how are you?", thread=thread)

# Serialize the thread for storage.
serialized_thread = await thread.serialize()
# Deserialize the thread state after loading from storage.
resumed_thread = await agent.deserialize_thread(serialized_thread)

# Run the agent with the resumed thread.
response = await agent.run("Hello, how are you?", thread=resumed_thread)
```

Custom Message Stores

For in-memory threads, you can provide a custom message store implementation to control how messages are stored and retrieved:

Python

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

class CustomStore(ChatMessageStore):
    # Implement custom storage logic here
    pass

# You can also provide a custom message store factory when creating the
agent
def custom_message_store_factory():
    return CustomStore()  # or your custom implementation

async with AzureCliCredential() as credential:
    agent = ChatAgent(
        chat_client=AzureAIAgentClient(async_credential=credential),
        instructions="You are a helpful assistant",
        chat_message_store_factory=custom_message_store_factory
    )
    # Or let the agent create one automatically
    thread = agent.get_new_thread()
    # thread.message_store is not a instance of CustomStore
```

Agent/AgentThread relationship

`Agents` 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 `ChatAgent` with the OpenAI Responses service and `store=True`, `AgentThread` instances used by this agent, will not work with a `ChatAgent` using the Azure AI Agent service, because the thread_ids are not compatible.

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.

Practical Multi-Turn Example

Here's a complete example showing how to maintain context across multiple interactions:

Python

```
from agent_framework import ChatAgent
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def multi_turn_example():
    async with (
        AzureCliCredential() as credential,
        ChatAgent(
            chat_client=AzureAIAgentClient(async_credential=credential),
            instructions="You are a helpful assistant"
        ) as agent
    ):
        # Create a thread for persistent conversation
        thread = agent.get_new_thread()

        # First interaction
        response1 = await agent.run("My name is Alice", thread=thread)
        print(f"Agent: {response1.text}")

        # Second interaction - agent remembers the name
        response2 = await agent.run("What's my name?", thread=thread)
        print(f"Agent: {response2.text}")  # Should mention "Alice"

        # Serialize thread for storage
        serialized = await thread.serialize()

        # Later, deserialize and continue conversation
        new_thread = await agent.deserialize_thread(serialized)
        response3 = await agent.run("What did we talk about?",
thread=new_thread)
        print(f"Agent: {response3.text}")  # Should remember previous
context
```

Next steps

Agent Middleware

Model Context Protocol

10/02/2025

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

Multi-turn conversations with an agent

10/02/2025

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 `get_new_thread()` method on the agent instance.

Python

```
thread = agent.get_new_thread()
```

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

Python

```
async def main():
    result1 = await agent.run("Tell me a joke about a pirate.",
thread=thread)
    print(result1.text)
```

```
    result2 = await agent.run("Now add some emojis to the joke and tell it
in the voice of a pirate's parrot.", thread=thread)
    print(result2.text)

asyncio.run(main())
```

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 agent will determine how conversation history is stored. E.g. when using a Chat Completion 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.

Python

```
async def main():
    thread1 = agent.get_new_thread()
    thread2 = agent.get_new_thread()

    result1 = await agent.run("Tell me a joke about a pirate.",
thread=thread1)
    print(result1.text)

    result2 = await agent.run("Tell me a joke about a robot.",
thread=thread2)
    print(result2.text)

    result3 = await agent.run("Now add some emojis to the joke and tell it
in the voice of a pirate's parrot.", thread=thread1)
    print(result3.text)

    result4 = await agent.run("Now add some emojis to the joke and tell it
in the voice of a robot.", thread=thread2)
    print(result4.text)
```

```
asyncio.run(main())
```

Next steps

Using function tools with an agent

OpenAI Assistants Agents

10/02/2025

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 .

Prerequisites

Install the Microsoft Agent Framework package.

Bash

```
pip install agent-framework
```

Configuration

Environment Variables

Set up the required environment variables for OpenAI authentication:

Bash

```
# Required for OpenAI API access
OPENAI_API_KEY="your-openai-api-key"
OPENAI_CHAT_MODEL_ID="gpt-4o-mini"  # or your preferred model
```

Alternatively, you can use a `.env` file in your project root:

env

```
OPENAI_API_KEY=your-openai-api-key
OPENAI_CHAT_MODEL_ID=gpt-4o-mini
```

Getting Started

Import the required classes from the Agent Framework:

Python

```python
import asyncio
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIAssistantsClient
```

Creating an OpenAI Assistants Agent

Basic Agent Creation

The simplest way to create an agent is by using the `OpenAIAssistantsClient` which automatically creates and manages assistants:

Python

```python
async def basic_example():
    # Create an agent with automatic assistant creation and cleanup
    async with OpenAIAssistantsClient().create_agent(
        instructions="You are a helpful assistant.",
        name="MyAssistant"
    ) as agent:
        result = await agent.run("Hello, how are you?")
        print(result.text)
```

Using Explicit Configuration

You can provide explicit configuration instead of relying on environment variables:

Python

```python
async def explicit_config_example():
    async with OpenAIAssistantsClient(
        ai_model_id="gpt-4o-mini",
        api_key="your-api-key-here",
    ).create_agent(
        instructions="You are a helpful assistant.",
    ) as agent:
        result = await agent.run("What's the weather like?")
        print(result.text)
```

Using an Existing Assistant

You can reuse existing OpenAI assistants by providing their IDs:

Python

```python
from openai import AsyncOpenAI

async def existing_assistant_example():
    # Create OpenAI client directly
    client = AsyncOpenAI()

    # Create or get an existing assistant
    assistant = await client.beta.assistants.create(
        model="gpt-4o-mini",
        name="WeatherAssistant",
        instructions="You are a weather forecasting assistant."
    )

    try:
        # Use the existing assistant with Agent Framework
        async with ChatAgent(
            chat_client=OpenAIAssistantsClient(
                async_client=client,
                assistant_id=assistant.id
            ),
            instructions="You are a helpful weather agent.",
        ) as agent:
            result = await agent.run("What's the weather like in Seattle?")
            print(result.text)
    finally:
        # Clean up the assistant
        await client.beta.assistants.delete(assistant.id)
```

Agent Features

Function Tools

You can equip your assistant with custom functions:

Python

```python
from typing import Annotated
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")]
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is sunny with 25°C."
```

```python
async def tools_example():
    async with ChatAgent(
        chat_client=OpenAIAssistantsClient(),
        instructions="You are a helpful weather assistant.",
        tools=get_weather,  # Provide tools to the agent
    ) as agent:
        result = await agent.run("What's the weather like in Tokyo?")
        print(result.text)
```

Code Interpreter

Enable your assistant to execute Python code:

Python

```python
from agent_framework import HostedCodeInterpreterTool

async def code_interpreter_example():
    async with ChatAgent(
        chat_client=OpenAIAssistantsClient(),
        instructions="You are a helpful assistant that can write and exe-
cute Python code.",
        tools=HostedCodeInterpreterTool(),
    ) as agent:
        result = await agent.run("Calculate the factorial of 100 using
Python code.")
        print(result.text)
```

Streaming Responses

Get responses as they are generated for better user experience:

Python

```python
async def streaming_example():
    async with OpenAIAssistantsClient().create_agent(
        instructions="You are a helpful assistant.",
    ) as agent:
        print("Assistant: ", end="", flush=True)
        async for chunk in agent.run_stream("Tell me a story about AI."):
            if chunk.text:
                print(chunk.text, end="", flush=True)
        print()  # New line after streaming is complete
```

Using the Agent

The agent is a standard `BaseAgent` 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

OpenAI ChatCompletion Agents

10/02/2025

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

Prerequisites

Install the Microsoft Agent Framework package.

Bash

```
pip install agent-framework
```

Configuration

Environment Variables

Set up the required environment variables for OpenAI authentication:

Bash

```
# Required for OpenAI API access
OPENAI_API_KEY="your-openai-api-key"
OPENAI_CHAT_MODEL_ID="gpt-4o-mini"  # or your preferred model
```

Alternatively, you can use a `.env` file in your project root:

env

```
OPENAI_API_KEY=your-openai-api-key
OPENAI_CHAT_MODEL_ID=gpt-4o-mini
```

Getting Started

Import the required classes from the Agent Framework:

Python

```python
import asyncio
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIChatClient
```

Creating an OpenAI ChatCompletion Agent

Basic Agent Creation

The simplest way to create a chat completion agent:

Python

```python
async def basic_example():
    # Create an agent using OpenAI ChatCompletion
    agent = OpenAIChatClient().create_agent(
        name="HelpfulAssistant",
        instructions="You are a helpful assistant.",
    )

    result = await agent.run("Hello, how can you help me?")
    print(result.text)
```

Using Explicit Configuration

You can provide explicit configuration instead of relying on environment variables:

Python

```python
async def explicit_config_example():
    agent = OpenAIChatClient(
        ai_model_id="gpt-4o-mini",
        api_key="your-api-key-here",
    ).create_agent(
        instructions="You are a helpful assistant.",
    )

    result = await agent.run("What can you do?")
    print(result.text)
```

Agent Features

Function Tools

Equip your agent with custom functions:

Python

```python
from typing import Annotated
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get weather for")]
) -> str:
    """Get the weather for a given location."""
    # Your weather API implementation here
    return f"The weather in {location} is sunny with 25°C."

async def tools_example():
    agent = ChatAgent(
        chat_client=OpenAIChatClient(),
        instructions="You are a helpful weather assistant.",
        tools=get_weather,  # Add tools to the agent
    )

    result = await agent.run("What's the weather like in Tokyo?")
    print(result.text)
```

Streaming Responses

Get responses as they are generated for better user experience:

Python

```python
async def streaming_example():
    agent = OpenAIChatClient().create_agent(
        name="StoryTeller",
        instructions="You are a creative storyteller.",
    )

    print("Assistant: ", end="", flush=True)
    async for chunk in agent.run_stream("Tell me a short story about AI."):
        if chunk.text:
            print(chunk.text, end="", flush=True)
    print()  # New line after streaming
```

Using the Agent

The agent is a standard `BaseAgent` 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

Response Agents

OpenAI Responses Agents

10/02/2025

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

Prerequisites

Install the Microsoft Agent Framework package.

Bash

```
pip install agent-framework
```

Configuration

Environment Variables

Set up the required environment variables for OpenAI authentication:

Bash

```
# Required for OpenAI API access
OPENAI_API_KEY="your-openai-api-key"
OPENAI_RESPONSES_MODEL_ID="gpt-4o"  # or your preferred Responses-compati-
ble model
```

Alternatively, you can use a `.env` file in your project root:

env

```
OPENAI_API_KEY=your-openai-api-key
OPENAI_RESPONSES_MODEL_ID=gpt-4o
```

Getting Started

Import the required classes from the Agent Framework:

Python

```
import asyncio
from agent_framework.openai import OpenAIResponsesClient
```

Creating an OpenAI Responses Agent

Basic Agent Creation

The simplest way to create a responses agent:

Python

```
async def basic_example():
    # Create an agent using OpenAI Responses
    agent = OpenAIResponsesClient().create_agent(
        name="WeatherBot",
        instructions="You are a helpful weather assistant.",
    )

    result = await agent.run("What's a good way to check the weather?")
    print(result.text)
```

Using Explicit Configuration

You can provide explicit configuration instead of relying on environment variables:

Python

```
async def explicit_config_example():
    agent = OpenAIResponsesClient(
        ai_model_id="gpt-4o",
        api_key="your-api-key-here",
    ).create_agent(
        instructions="You are a helpful assistant.",
    )

    result = await agent.run("Tell me about AI.")
    print(result.text)
```

Basic Usage Patterns

Streaming Responses

Get responses as they are generated for better user experience:

Python

```
async def streaming_example():
    agent = OpenAIResponsesClient().create_agent(
        instructions="You are a creative storyteller.",
    )

    print("Assistant: ", end="", flush=True)
    async for chunk in agent.run_stream("Tell me a short story about AI."):
        if chunk.text:
            print(chunk.text, end="", flush=True)
    print()  # New line after streaming
```

Agent Features

Reasoning Models

Use advanced reasoning capabilities with models like GPT-5:

Python

```
from agent_framework import HostedCodeInterpreterTool, TextContent,
TextReasoningContent

async def reasoning_example():
    agent = OpenAIResponsesClient(ai_model_id="gpt-5").create_agent(
        name="MathTutor",
        instructions="You are a personal math tutor. When asked a math
question, "
                     "write and run code to answer the question.",
        tools=HostedCodeInterpreterTool(),
        reasoning={"effort": "high", "summary": "detailed"},
    )

    print("Assistant: ", end="", flush=True)
    async for chunk in agent.run_stream("Solve: 3x + 11 = 14"):
        if chunk.contents:
            for content in chunk.contents:
                if isinstance(content, TextReasoningContent):
                    # Reasoning content in gray text
                    print(f"\033[97m{content.text}\033[0m", end="",
flush=True)
                elif isinstance(content, TextContent):
                    print(content.text, end="", flush=True)
    print()
```

Structured Output

Get responses in structured formats:

Python

```
from pydantic import BaseModel
from agent_framework import AgentRunResponse

class CityInfo(BaseModel):
    """A structured output for city information."""
    city: str
    description: str

async def structured_output_example():
    agent = OpenAIResponsesClient().create_agent(
        name="CityExpert",
        instructions="You describe cities in a structured format.",
    )

    # Non-streaming structured output
    result = await agent.run("Tell me about Paris, France",
response_format=CityInfo)

    if result.value:
        city_data = result.value
        print(f"City: {city_data.city}")
        print(f"Description: {city_data.description}")

    # Streaming structured output
    structured_result = await
AgentRunResponse.from_agent_response_generator(
        agent.run_stream("Tell me about Tokyo, Japan",
response_format=CityInfo),
        output_format_type=CityInfo,
    )

    if structured_result.value:
        tokyo_data = structured_result.value
        print(f"City: {tokyo_data.city}")
        print(f"Description: {tokyo_data.description}")
```

Function Tools

Equip your agent with custom functions:

Python

```
from typing import Annotated
from pydantic import Field
```

```
def get_weather(
    location: Annotated[str, Field(description="The location to get weather
for")]
) -> str:
    """Get the weather for a given location."""
    # Your weather API implementation here
    return f"The weather in {location} is sunny with 25°C."

async def tools_example():
    agent = OpenAIResponsesClient().create_agent(
        instructions="You are a helpful weather assistant.",
        tools=get_weather,
    )

    result = await agent.run("What's the weather like in Tokyo?")
    print(result.text)
```

Image Generation

Generate images using the Responses API:

Python

```
from agent_framework import DataContent, UriContent

async def image_generation_example():
    agent = OpenAIResponsesClient().create_agent(
        instructions="You are a helpful AI that can generate images.",
        tools=[{
            "type": "image_generation",
            "size": "1024x1024",
            "quality": "low",
        }],
    )

    result = await agent.run("Generate an image of a sunset over the
ocean.")

    # Check for generated images in the response
    for content in result.contents:
        if isinstance(content, (DataContent, UriContent)):
            print(f"Image generated: {content.uri}")
```

Code Interpreter

Enable your assistant to execute Python code:

Python

```python
from agent_framework import HostedCodeInterpreterTool

async def code_interpreter_example():
    agent = OpenAIResponsesClient().create_agent(
        instructions="You are a helpful assistant that can write and execute Python code.",
        tools=HostedCodeInterpreterTool(),
    )

    result = await agent.run("Calculate the factorial of 100 using Python code.")
    print(result.text)
```

Using the Agent

The agent is a standard `BaseAgent` 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

OpenAI Assistant Agents

Persisting and Resuming Agent Conversations

10/02/2025

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 Python 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.

Python

```
from azure.identity import AzureCliCredential
from agent_framework import ChatAgent
from agent_framework.azure import AzureOpenAIChatClient

agent = ChatAgent(
    chat_client=AzureOpenAIChatClient(
        endpoint="https://<myresource>.openai.azure.com",
        credential=AzureCliCredential(),
        ai_model_id="gpt-4o-mini"
    ),
    name="Assistant",
    instructions="You are a helpful assistant."
)

thread = agent.get_new_thread()
```

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

Python

```
# Run the agent and append the exchange to the thread
response = await agent.run("Tell me a short pirate joke.", thread=thread)
```

```
print(response.text)
```

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

Python

```
import json
import tempfile
import os

# Serialize the thread state
serialized_thread = await thread.serialize()
serialized_json = json.dumps(serialized_thread)

# Example: save to a local file (replace with DB or blob storage in production)
temp_dir = tempfile.gettempdir()
file_path = os.path.join(temp_dir, "agent_thread.json")
with open(file_path, "w") as f:
    f.write(serialized_json)
```

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.

Python

```
# Read persisted JSON
with open(file_path, "r") as f:
    loaded_json = f.read()

reloaded_data = json.loads(loaded_json)

# Deserialize the thread into an AgentThread tied to the same agent type
resumed_thread = await agent.deserialize_thread(reloaded_data)
```

Use the resumed thread to continue the conversation.

Python

```
# Continue the conversation with resumed thread
response = await agent.run("Now tell that joke in the voice of a pirate.",
thread=resumed_thread)
print(response.text)
```

Next steps

Third Party chat history storage

Producing Structured Output with Agents

10/02/2025

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. The `ChatAgent` supports structured output when used with compatible chat clients.

Prerequisites

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

Creating the agent with structured output

The `ChatAgent` is built on top of any chat client implementation that supports structured output. The `ChatAgent` uses the `response_format` parameter to specify the desired output schema.

When creating or running the agent, we can provide a Pydantic model that defines the structure of the expected output.

Various response formats are supported based on the underlying chat client capabilities.

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 Pydantic model schema.

First, define a Pydantic model that represents the structure of the output you want from the agent:

Python

```python
from pydantic import BaseModel

class PersonInfo(BaseModel):
    """Information about a person."""
    name: str | None = None
```

```
    age: int | None = None
    occupation: str | None = None
```

Now we can create an agent using the Azure OpenAI Chat Client:

Python

```
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

# Create the agent using Azure OpenAI Chat Client
agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    name="HelpfulAssistant",
    instructions="You are a helpful assistant that extracts person informa-
tion from text."
)
```

Now we can run the agent with some textual information and specify the structured output format using the `response_format` parameter:

Python

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

The agent response will contain the structured output in the `value` property, which can be accessed directly as a Pydantic model instance:

Python

```
if response.value:
    person_info = response.value
    print(f"Name: {person_info.name}, Age: {person_info.age}, Occupation:
{person_info.occupation}")
else:
    print("No structured data found in response")
```

When streaming, the agent response is streamed as a series of updates. To get the structured output, we need to collect all the updates and then access the final response value:

Python

```python
from agent_framework import AgentRunResponse

# Get structured response from streaming agent using
AgentRunResponse.from_agent_response_generator
# This method collects all streaming updates and combines them into a sin-
gle AgentRunResponse
final_response = await AgentRunResponse.from_agent_response_generator(
    agent.run_stream(query, response_format=PersonInfo),
    output_format_type=PersonInfo,
)

if final_response.value:
    person_info = final_response.value
    print(f"Name: {person_info.name}, Age: {person_info.age}, Occupation:
{person_info.occupation}")
```

Next steps

Using an agent as a function tool

Microsoft Agent Framework Quick Start

10/02/2025

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:

- Python 3.10 or later
- An Azure AI project with a deployed model (e.g., `gpt-4o-mini`)
- Azure CLI installed and authenticated (`az login`)

Note: This demo uses Azure CLI credentials for authentication. Make sure you're logged in with `az login` and have access to the Azure AI project. For more information, see the Azure CLI documentation.

Running a Basic Agent Sample

This sample demonstrates how to create and use a simple AI agent with Azure AI as the backend. It will create a basic agent using `ChatAgent` with `AzureAIAgentClient` and custom instructions.

Make sure to set the following environment variables:

- `AZURE_AI_PROJECT_ENDPOINT`: Your Azure AI project endpoint
- `AZURE_AI_MODEL_DEPLOYMENT_NAME`: The name of your model deployment

Sample Code

Python

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

async def main():
    async with (
        AzureCliCredential() as credential,
        ChatAgent(
            chat_client=AzureAIAgentClient(async_credential=credential),
```

```
            instructions="You are good at telling jokes."
        ) as agent,
    ):
        result = await agent.run("Tell me a joke about a pirate.")
        print(result.text)

if __name__ == "__main__":
    asyncio.run(main())
```

More Examples

For more detailed examples and advanced scenarios, see the Azure AI Agent Examples .

Next steps

Create and run agents

Running Agents

10/02/2025

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 `run` method.

Python

```
result = await agent.run("What is the weather like in Amsterdam?")
print(result.text)
```

For streaming, use the `run_stream` method.

Python

```
async for update in agent.run_stream("What is the weather like in
Amsterdam?"):
    if update.text:
        print(update.text, end="", flush=True)
```

Agent run options

Python agents support passing keyword arguments to customize each run. The specific options available depend on the agent type, but `ChatAgent` supports many chat client parameters that can be passed to both `run` and `run_stream` methods.

Common options for `ChatAgent` include:

- `max_tokens`: Maximum number of tokens to generate
- `temperature`: Controls randomness in response generation
- `model`: Override the model for this specific run
- `tools`: Add additional tools for this run only
- `response_format`: Specify the response format (e.g., structured output)

Python

```python
# Run with custom options
result = await agent.run(
    "What is the weather like in Amsterdam?",
    temperature=0.3,
    max_tokens=150,
    model="gpt-4o"
)

# Streaming with custom options
async for update in agent.run_stream(
    "Tell me a detailed weather forecast",
    temperature=0.7,
    tools=[additional_weather_tool]
):
    if update.text:
        print(update.text, end="", flush=True)
```

When both agent-level defaults and run-level options are provided, the run-level options take precedence.

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.

For the non-streaming case, everything is returned in one `AgentRunResponse` object. `AgentRunResponse` allows access to the produced messages via the `messages` property.

To extract the text result from a response, all `TextContent` items from all `ChatMessage` items need to be aggregated. To simplify this, we provide a `text` property on all response types that aggregates all `TextContent`.

Python

```python
response = await agent.run("What is the weather like in Amsterdam?")
print(response.text)
print(len(response.messages))

# Access individual messages
```

```
for message in response.messages:
    print(f"Role: {message.role}, Text: {message.text}")
```

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.

Python

```
async for update in agent.run_stream("What is the weather like in
Amsterdam?"):
    print(f"Update text: {update.text}")
    print(f"Content count: {len(update.contents)}")

    # Access individual content items
    for content in update.contents:
        if hasattr(content, 'text'):
            print(f"Content: {content.text}")
```

Message types

Input and output from agents are represented as messages. Messages are subdivided into content items.

The Python Agent Framework uses message and content types from the `agent_framework` package. Messages are represented by the `ChatMessage` class and all content classes inherit from the base `BaseContent` class.

Various `BaseContent` subclasses exist that are used to represent different types of content:

Expand table

Type	Description
`TextContent`	Textual content that can be both input and output from the agent. Typically contains the text result from an agent.
`DataContent`	Binary content represented as a data URI (e.g., base64-encoded images). Can be used to pass binary data to and from the agent.
`UriContent`	A URI that points to hosted content such as an image, audio file, or document.

Type	Description
`FunctionCallContent`	A request by an AI service to invoke a function tool.
`FunctionResultContent`	The result of a function tool invocation.
`ErrorContent`	Error information when processing fails.
`UsageContent`	Token usage and billing information from the AI service.

Here's how to work with different content types:

Python

```python
from agent_framework import ChatMessage, TextContent, DataContent,
UriContent

# Create a text message
text_message = ChatMessage(role="user", text="Hello!")

# Create a message with multiple content types
image_data = b"..."  # your image bytes
mixed_message = ChatMessage(
    role="user",
    contents=[
        TextContent("Analyze this image:"),
        DataContent(data=image_data, media_type="image/png"),
    ]
)

# Access content from responses
response = await agent.run("Describe the image")
for message in response.messages:
    for content in message.contents:
        if isinstance(content, TextContent):
            print(f"Text: {content.text}")
        elif isinstance(content, DataContent):
            print(f"Data URI: {content.uri}")
        elif isinstance(content, UriContent):
            print(f"External URI: {content.uri}")
```

Next steps

Agent Tools

Storing Chat History in 3rd Party Storage

10/02/2025

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

By default, when using `ChatAgent`, 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 or are not capable of the 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.

Creating a custom ChatMessage Store

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

Message storage and retrieval methods

The most important methods to implement are:

- `add_messages` - called to add new messages to the store.
- `list_messages` - called to retrieve the messages from the store.

`list_messages` should return the messages in ascending chronological order. All messages returned by it will be used by the `ChatAgent` when making calls to the underlying chat client. 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 `list_messages`.

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 `serialize_state` and `deserialize_state` methods of the `ChatMessageStore` protocol. These methods allow the store's state to be persisted and restored when resuming a thread.

Sample ChatMessageStore implementation

Let's look at a sample implementation that stores chat messages in Redis using the Redis Lists data structure.

In `add_messages` it stores messages in Redis using RPUSH to append them to the end of the list in chronological order.

`list_messages` retrieves the messages for the current thread from Redis using LRANGE, and returns them in ascending chronological 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 Redis for subsequent calls.

The unique key and other configuration are stored and can be serialized and deserialized using the `serialize_state` and `deserialize_state` methods. This state will therefore be persisted as part of the `AgentThread` state, allowing the thread to be resumed later and continue using the same chat history.

Python

```python
from collections.abc import Sequence
from typing import Any
from uuid import uuid4
from pydantic import BaseModel
import json
import redis.asyncio as redis
from agent_framework import ChatMessage


class RedisStoreState(BaseModel):
    """State model for serializing and deserializing Redis chat message
store data."""

    thread_id: str
```

```
    redis_url: str | None = None
    key_prefix: str = "chat_messages"
    max_messages: int | None = None


class RedisChatMessageStore:
    """Redis-backed implementation of ChatMessageStore using Redis
Lists."""

    def __init__(
        self,
        redis_url: str | None = None,
        thread_id: str | None = None,
        key_prefix: str = "chat_messages",
        max_messages: int | None = None,
    ) -> None:
        """Initialize the Redis chat message store.

        Args:
            redis_url: Redis connection URL (e.g.,
"redis://localhost:6379").
            thread_id: Unique identifier for this conversation thread.
                       If not provided, a UUID will be auto-generated.
            key_prefix: Prefix for Redis keys to namespace different ap-
plications.
            max_messages: Maximum number of messages to retain in Redis.
                          When exceeded, oldest messages are automatically
trimmed.
        """
        if redis_url is None:
            raise ValueError("redis_url is required for Redis connection")

        self.redis_url = redis_url
        self.thread_id = thread_id or f"thread_{uuid4()}"
        self.key_prefix = key_prefix
        self.max_messages = max_messages

        # Initialize Redis client
        self._redis_client = redis.from_url(redis_url,
decode_responses=True)

    @property
    def redis_key(self) -> str:
        """Get the Redis key for this thread's messages."""
        return f"{self.key_prefix}:{self.thread_id}"

    async def add_messages(self, messages: Sequence[ChatMessage]) -> None:
        """Add messages to the Redis store.

        Args:
            messages: Sequence of ChatMessage objects to add to the store.
        """
        if not messages:
            return
```

```
        # Serialize messages and add to Redis list
        serialized_messages = [self._serialize_message(msg) for msg in mes-
sages]
        await self._redis_client.rpush(self.redis_key, *serialized_mes-
sages)

        # Apply message limit if configured
        if self.max_messages is not None:
            current_count = await self._redis_client.llen(self.redis_key)
            if current_count > self.max_messages:
                # Keep only the most recent max_messages using LTRIM
                await self._redis_client.ltrim(self.redis_key, -self.-
max_messages, -1)

    async def list_messages(self) -> list[ChatMessage]:
        """Get all messages from the store in chronological order.

        Returns:
            List of ChatMessage objects in chronological order (oldest
first).
        """
        # Retrieve all messages from Redis list (oldest to newest)
        redis_messages = await self._redis_client.lrange(self.redis_key, 0,
-1)

        messages = []
        for serialized_message in redis_messages:
            message = self._deserialize_message(serialized_message)
            messages.append(message)

        return messages

    async def serialize_state(self, **kwargs: Any) -> Any:
        """Serialize the current store state for persistence.

        Returns:
            Dictionary containing serialized store configuration.
        """
        state = RedisStoreState(
            thread_id=self.thread_id,
            redis_url=self.redis_url,
            key_prefix=self.key_prefix,
            max_messages=self.max_messages,
        )
        return state.model_dump(**kwargs)

    async def deserialize_state(self, serialized_store_state: Any,
**kwargs: Any) -> None:
        """Deserialize state data into this store instance.

        Args:
            serialized_store_state: Previously serialized state data.
            **kwargs: Additional arguments for deserialization.
        """
        if serialized_store_state:
```

```
            state = RedisStoreState.model_validate(serialized_store_state,
**kwargs)
            self.thread_id = state.thread_id
            self.key_prefix = state.key_prefix
            self.max_messages = state.max_messages

            # Recreate Redis client if the URL changed
            if state.redis_url and state.redis_url != self.redis_url:
                self.redis_url = state.redis_url
                self._redis_client = redis.from_url(self.redis_url, de-
code_responses=True)

    def _serialize_message(self, message: ChatMessage) -> str:
        """Serialize a ChatMessage to JSON string."""
        message_dict = message.model_dump()
        return json.dumps(message_dict, separators=(",", ":"))

    def _deserialize_message(self, serialized_message: str) -> ChatMessage:
        """Deserialize a JSON string to ChatMessage."""
        message_dict = json.loads(serialized_message)
        return ChatMessage.model_validate(message_dict)

    async def clear(self) -> None:
        """Remove all messages from the store."""
        await self._redis_client.delete(self.redis_key)

    async def aclose(self) -> None:
        """Close the Redis connection."""
        await self._redis_client.aclose()
```

Using the custom ChatMessageStore with a ChatAgent

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

When creating a `ChatAgent`, you can provide the `chat_message_store_factory` parameter in addition to all other agent options.

Python

```
from azure.identity import AzureCliCredential
from agent_framework import ChatAgent
from agent_framework.openai import AzureOpenAIChatClient

# Create the chat agent with custom message store factory
agent = ChatAgent(
    chat_client=AzureOpenAIChatClient(
```

```
        endpoint="https://<myresource>.openai.azure.com",
        credential=AzureCliCredential(),
        ai_model_id="gpt-4o-mini"
    ),
    name="Joker",
    instructions="You are good at telling jokes.",
    chat_message_store_factory=lambda: RedisChatMessageStore(
        redis_url="redis://localhost:6379"
    )
)

# Use the agent with persistent chat history
thread = agent.get_new_thread()
response = await agent.run("Tell me a joke about pirates", thread=thread)
print(response.text)
```

Next steps

Adding Memory to an Agent

Using an agent as a function tool

10/02/2025

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 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 a `ChatAgent` as a function tool by calling `.as_tool()` 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 that will be used by our agent that is exposed as a function.

Python

```
from typing import Annotated
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
    """Get the weather for a given location."""
    return f"The weather in {location} is cloudy with a high of 15°C."
```

Create a `ChatAgent` that uses the function tool.

Python

```
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

weather_agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    name="WeatherAgent",
    description="An agent that answers questions about the weather.",
    instructions="You answer questions about the weather.",
```

```
    tools=get_weather
)
```

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

Python

```
main_agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    instructions="You are a helpful assistant who responds in French.",
    tools=weather_agent.as_tool()
)
```

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

Python

```
result = await main_agent.run("What is the weather like in Amsterdam?")
print(result.text)
```

You can also customize the tool name, description, and argument name when converting an agent to a tool:

Python

```
# Convert agent to tool with custom parameters
weather_tool = weather_agent.as_tool(
    name="WeatherLookup",
    description="Look up weather information for any location",
    arg_name="query",
    arg_description="The weather query or location"
)

main_agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    instructions="You are a helpful assistant who responds in French.",
    tools=weather_tool
)
```

Next steps

Exposing an agent as an MCP tool

Using function tools with an agent

10/02/2025

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 agents created via chat clients, which do support function tools.

Prerequisites

For prerequisites and installing Python 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 Python function into a function tool by passing it to the agent's `tools` parameter when creating the agent.

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 Python's type annotations with `Annotated` and Pydantic's `Field` to provide descriptions.

Here is an example of a simple function tool that fakes getting the weather for a given location. It uses type annotations to provide additional descriptions about the function and its location parameter to the agent.

Python

```
from typing import Annotated
from pydantic import Field

def get_weather(
    location: Annotated[str, Field(description="The location to get the
weather for.")],
) -> str:
```

```
    """Get the weather for a given location."""
    return f"The weather in {location} is cloudy with a high of 15°C."
```

You can also use the `ai_function` decorator to explicitly specify the function's name and description:

Python

```
from typing import Annotated
from pydantic import Field
from agent_framework import ai_function

@ai_function(name="weather_tool", description="Retrieves weather information for any location")
def get_weather(
    location: Annotated[str, Field(description="The location to get the weather for.")],
) -> str:
    return f"The weather in {location} is cloudy with a high of 15°C."
```

If you don't specify the `name` and `description` parameters in the `ai_function` decorator, the framework will automatically use the function's name and docstring as fallbacks.

When creating the agent, we can now provide the function tool to the agent, by passing it to the `tools` parameter.

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

agent = AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    instructions="You are a helpful assistant",
    tools=get_weather
)
```

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

Python

```
async def main():
    result = await agent.run("What is the weather like in Amsterdam?")
    print(result.text)
```

```
asyncio.run(main())
```

Creating a class with multiple function tools

You can also create a class that contains multiple function tools as methods. This can be useful for organizing related functions together or when you want to pass state between them.

Python

```
class WeatherTools:
    def __init__(self):
        self.last_location = None

    def get_weather(
        self,
        location: Annotated[str, Field(description="The location to get the weather for.")],
    ) -> str:
        """Get the weather for a given location."""
        return f"The weather in {location} is cloudy with a high of 15°C."

    def get_weather_details(self) -> int:
        """Get the detailed weather for the last requested location."""
        if self.last_location is None:
            return "No location specified yet."
        return f"The detailed weather in {self.last_location} is cloudy with a high of 15°C, low of 7°C, and 60% humidity."
```

When creating the agent, we can now provide all the methods of the class as functions:

Python

```
tools = WeatherTools()
agent = AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    instructions="You are a helpful assistant",
    tools=[tools.get_weather, tools.get_weather_details]
)
```

You can also decorate the functions with the same `ai_function` decorator as before.

Next steps

Using function tools with human in the loop approvals

Using function tools with human in the loop approvals

10/02/2025

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.Func-
tionCall.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.Cre-
ateResponse(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

Using images with an agent

10/02/2025

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 agent that is able to analyze images.

Python

```
import asyncio
from agent_framework.azure import AzureOpenAIChatClient
from azure.identity import AzureCliCredential

agent =
AzureOpenAIChatClient(credential=AzureCliCredential()).create_agent(
    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.

Python

```
from agent_framework import ChatMessage, TextContent, UriContent, Role

message = ChatMessage(
    role=Role.USER,
    contents=[
        TextContent(text="What do you see in this image?"),
        UriContent(

uri="https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wiscon-
sin-madison-the-nature-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-na-
```

```
ture-boardwalk.jpg",
            media_type="image/jpeg"
        )
    ]
)
```

You can also load an image from your local file system using `DataContent`:

Python

```python
from agent_framework import ChatMessage, TextContent, DataContent, Role

# Load image from local file
with open("path/to/your/image.jpg", "rb") as f:
    image_bytes = f.read()

message = ChatMessage(
    role=Role.USER,
    contents=[
        TextContent(text="What do you see in this image?"),
        DataContent(
            data=image_bytes,
            media_type="image/jpeg"
        )
    ]
)
```

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

Python

```python
async def main():
    result = await agent.run(message)
    print(result.text)

asyncio.run(main())
```

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

Next steps

Having a multi-turn conversation with an agent

Using MCP tools with Agents

10/02/2025

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 Python Agent Framework provides comprehensive support for integrating with Model Context Protocol (MCP) servers through multiple connection types. This allows your agents to access external tools and services seamlessly.

MCP Tool Types

The Agent Framework supports three types of MCP connections:

MCPStdioTool - Local MCP Servers

Use `MCPStdioTool` to connect to MCP servers that run as local processes using standard input/output:

Python

```python
import asyncio
from agent_framework import ChatAgent, MCPStdioTool
from agent_framework.openai import OpenAIChatClient

async def local_mcp_example():
    """Example using a local MCP server via stdio."""
    async with (
        MCPStdioTool(
            name="calculator",
            command="uvx",
            args=["mcp-server-calculator"]
        ) as mcp_server,
        ChatAgent(
            chat_client=OpenAIChatClient(),
            name="MathAgent",
            instructions="You are a helpful math assistant that can solve
calculations.",
        ) as agent,
    ):
        result = await agent.run(
            "What is 15 * 23 + 45?",
            tools=mcp_server
        )
        print(result)
```

```
if __name__ == "__main__":
    asyncio.run(local_mcp_example())
```

MCPStreamableHTTPTool - HTTP/SSE MCP Servers

Use `MCPStreamableHTTPTool` to connect to MCP servers over HTTP with Server-Sent Events:

Python

```
import asyncio
from agent_framework import ChatAgent, MCPStreamableHTTPTool
from agent_framework.azure import AzureAIAgentClient
from azure.identity.aio import AzureCliCredential

async def http_mcp_example():
    """Example using an HTTP-based MCP server."""
    async with (
        AzureCliCredential() as credential,
        MCPStreamableHTTPTool(
            name="Microsoft Learn MCP",
            url="https://learn.microsoft.com/api/mcp",
            headers={"Authorization": "Bearer your-token"},
        ) as mcp_server,
        ChatAgent(
            chat_client=AzureAIAgentClient(async_credential=credential),
            name="DocsAgent",
            instructions="You help with Microsoft documentation
questions.",
        ) as agent,
    ):
        result = await agent.run(
            "How to create an Azure storage account using az cli?",
            tools=mcp_server
        )
        print(result)

if __name__ == "__main__":
    asyncio.run(http_mcp_example())
```

MCPWebsocketTool - WebSocket MCP Servers

Use `MCPWebsocketTool` to connect to MCP servers over WebSocket connections:

Python

```
import asyncio
from agent_framework import ChatAgent, MCPWebsocketTool
from agent_framework.openai import OpenAIChatClient
```

```python
async def websocket_mcp_example():
    """Example using a WebSocket-based MCP server."""
    async with (
        MCPWebsocketTool(
            name="realtime-data",
            url="wss://api.example.com/mcp",
        ) as mcp_server,
        ChatAgent(
            chat_client=OpenAIChatClient(),
            name="DataAgent",
            instructions="You provide real-time data insights.",
        ) as agent,
    ):
        result = await agent.run(
            "What is the current market status?",
            tools=mcp_server
        )
        print(result)

if __name__ == "__main__":
    asyncio.run(websocket_mcp_example())
```

Popular MCP Servers

Common MCP servers you can use with Python Agent Framework:

- **Calculator**: `uvx mcp-server-calculator` - Mathematical computations
- **Filesystem**: `uvx mcp-server-filesystem` - File system operations
- **GitHub**: `npx @modelcontextprotocol/server-github` - GitHub repository access
- **SQLite**: `uvx mcp-server-sqlite` - Database operations

Each server provides different tools and capabilities that extend your agent's functionality while maintaining the security and standardization benefits of the Model Context Protocol.

Next steps

Using workflows as Agents

Visualizing Workflows

10/02/2025

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

```
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

```
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 .