Microsoft Agent Framework Workflows

10/02/2025

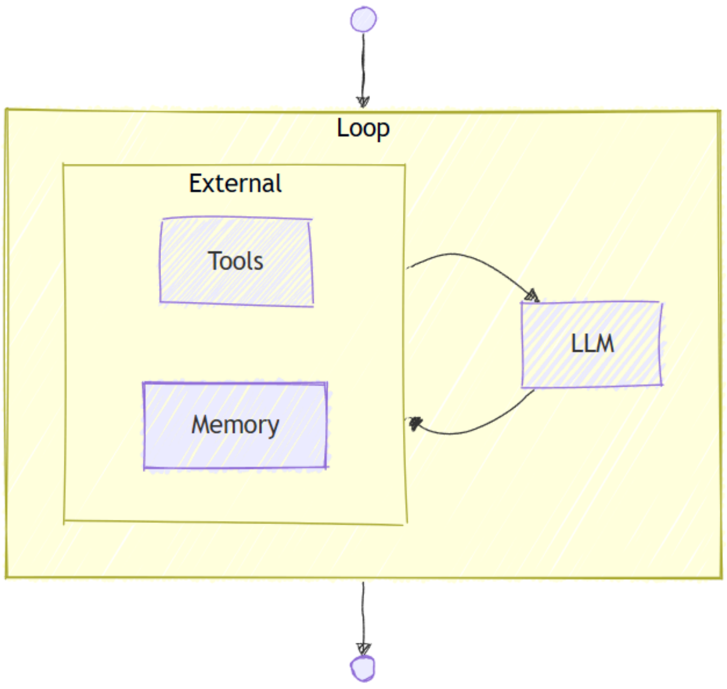
Overview

Microsoft Agent Framework Workflows empowers you to build intelligent automation systems that seamlessly blend AI agents with business processes. With its type-safe architecture and intuitive design, you can orchestrate complex workflows without getting bogged down in infrastructure complexity, allowing you to focus on your core business logic.

How is a Workflows different from an AI Agent?

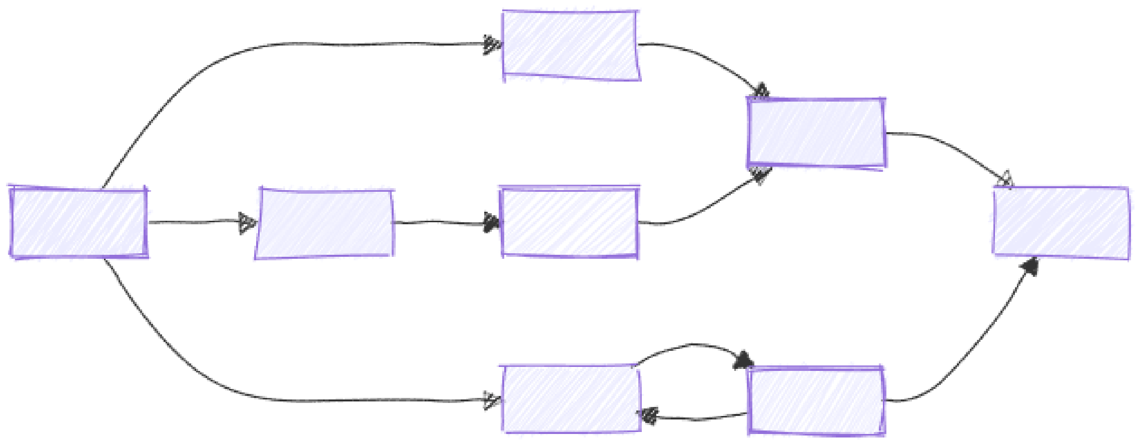
While an AI agent and a workflow can involve multiple steps to achieve a goal, they serve different purposes and operate at different levels of abstraction:

- **AI Agent**: An AI agent is typically driven by a large language model (LLM) and it has access to various tools to help it accomplish tasks. The steps an agent takes are dynamic and determined by the LLM based on the context of the conversation and the tools



available.
- **Workflow**: A workflow, on the other hand, is a predefined sequence of operations that can include AI agents as components. Workflows are designed to handle complex business processes that may involve multiple agents, human interactions, and integrations with external systems. The flow of a workflow is explicitly defined, allowing

for more control over the execution path.



Key Features

- **Type Safety**: Strong typing ensures messages flow correctly between components, with comprehensive validation that prevents runtime errors.
- **Flexible Control Flow**: Graph-based architecture allows for intuitive modeling of complex workflows with `executors` and `edges`. Conditional routing, parallel processing, and dynamic execution paths are all supported.
- **External Integration**: Built-in request/response patterns for seamless integration with external APIs, and human-in-the-loop scenarios.
- **Checkpointing**: Save workflow states via checkpoints, enabling recovery and resumption of long-running processes on server sides.
- **Multi-Agent Orchestration**: Built-in patterns for coordinating multiple AI agents, including sequential, concurrent, hand-off, and magentic.

Core Concepts

- **Executors**: represent individual processing units within a workflow. They can be AI agents or custom logic components. They receive input messages, perform specific tasks, and produce output messages.
- **Edges**: define the connections between executors, determining the flow of messages. They can include conditions to control routing based on message contents.
- **Workflows**: are directed graphs composed of executors and edges. They define the overall process, starting from an initial executor and proceeding through various paths based on conditions and logic defined in the edges.

Getting Started

Begin your journey with Microsoft Agent Framework Workflows by exploring our getting started samples:

- C# Getting Started Sample
- Python Getting Started Sample

Next Steps

Dive deeper into the concepts and capabilities of Microsoft Agent Framework Workflows by continuing to the Workflows Concepts page.

Microsoft Agent Framework Workflows - Checkpoints

10/02/2025

This page provides an overview of **Checkpoints** in the Microsoft Agent Framework Workflow system.

Overview

Checkpoints allow you to save the state of a workflow at specific points during its execution, and resume from those points later. This feature is particularly useful for the following scenarios:

- Long-running workflows where you want to avoid losing progress in case of failures.
- Long-running workflows where you want to pause and resume execution at a later time.
- Workflows that require periodic state saving for auditing or compliance purposes.
- Workflows that need to be migrated across different environments or instances.

When Are Checkpoints Created?

Remember that workflows are executed in **supersteps**, as documented in the core concepts. Checkpoints are created at the end of each superstep, after all executors in that superstep have completed their execution. A checkpoint captures the entire state of the workflow, including:

- The current state of all executors
- All pending messages in the workflow for the next superstep
- Pending requests and responses
- Shared states

Capturing Checkpoints

To enable check pointing, a `CheckpointStorage` needs to be provided when creating a workflow. A checkpoint then can be accessed via the storage.

Python

```python
from agent_framework import (
    InMemoryCheckpointStorage,
    WorkflowBuilder,
)
```

```python
# Create a checkpoint storage to manage checkpoints
# There are different implementations of CheckpointStorage, such as
InMemoryCheckpointStorage and FileCheckpointStorage.
checkpoint_storage = InMemoryCheckpointStorage()

# Build a workflow with checkpointing enabled
builder = WorkflowBuilder()
builder.set_start_executor(start_executor)
builder.add_edge(start_executor, executor_b)
builder.add_edge(executor_b, executor_c)
builder.add_edge(executor_b, end_executor)
workflow = builder.with_checkpointing(checkpoint_storage).build()

# Run the workflow
async for event in workflow.run_streaming(input):
    ...

# Access checkpoints from the storage
checkpoints = await checkpoint_storage.list_checkpoints()
```

Resuming from Checkpoints

You can resume a workflow from a specific checkpoint directly on the same workflow instance.

Python

```python
# Assume we want to resume from the 6th checkpoint
saved_checkpoint = checkpoints[5]
async for event in workflow.run_stream_from_checkpoint(saved_check-
point.checkpoint_id):
    ...
```

Rehydrating from Checkpoints

Or you can rehydrate a new workflow instance from a checkpoint.

Python

```python
from agent_framework import WorkflowBuilder

builder = WorkflowBuilder()
builder.set_start_executor(start_executor)
builder.add_edge(start_executor, executor_b)
builder.add_edge(executor_b, executor_c)
builder.add_edge(executor_b, end_executor)
# This workflow instance doesn't require checkpointing enabled.
workflow = builder.build()
```

```python
# Assume we want to resume from the 6th checkpoint
saved_checkpoint = checkpoints[5]
async for event in workflow.run_stream_from_checkpoint(
    saved_checkpoint.checkpoint_id,
    checkpoint_storage,
):
    ...
```

Save Executor States

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.

Microsoft Agent Framework Workflows - Observability

10/02/2025

Observability provides insights into the internal state and behavior of workflows during execution. This includes logging, metrics, and tracing capabilities that help monitor and debug workflows.

Aside from the standard GenAI telemetry, Agent Framework Workflows emits additional spans, logs, and metrics to provide deeper insights into workflow execution. These observability features help developers understand the flow of messages, the performance of executors, and any errors that may occur.

Enable Observability

Observability is enabled framework-wide by setting the `ENABLE_OTEL=true` environment variable or calling `setup_observability()` at the beginning of your application.

env

```env
# This is not required if you run `setup_observability()` in your code
ENABLE_OTEL=true
# Sensitive data (e.g., message content) will be included in logs and traces if this is set to true
ENABLE_SENSITIVE_DATA=true
```

Python

```python
from agent_framework.observability import setup_observability

setup_observability(enable_sensitive_data=True)
```

Workflow Spans

Expand table

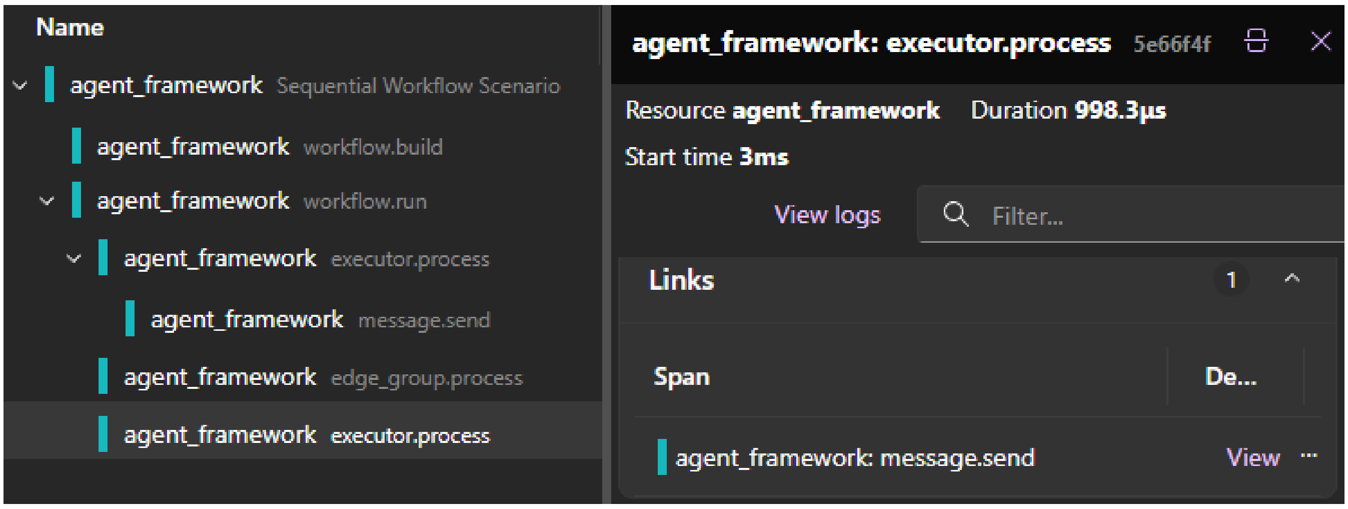
Span Name	Description
`workflow.build`	For each workflow build
`workflow.run`	For each workflow execution

Span Name	Description
`message.send`	For each message sent to an executor
`executor.process`	For each executor processing a message
`edge_group.process`	For each edge group processing a message

Links between Spans

When an executor sends a message to another executor, the `message.send` span is created as a child of the `executor.process` span. However, the `executor.process` span of the target executor will not be a child of the `message.send` span because the execution is not nested. Instead, the `executor.process` span of the target executor is linked to the `message.send` span of the source executor. This creates a traceable path through the workflow execution.

For example:



Name
agent_framework Sequential Workflow Scenario
agent_framework workflow.build
agent_framework workflow.run
agent_framework executor.process
agent_framework message.send
agent_framework edge_group.process
agent_framework executor.process

agent_framework: executor.process 5e66f4f
Resource **agent_framework** Duration **998.3µs**
Start time **3ms**

Span	De...
agent_framework: message.send	View

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Request and Response

10/02/2025

This page provides an overview of how **Request and Response** handling works in the Microsoft Agent Framework Workflow system.

Overview

Executors in a workflow can send requests to outside of the workflow and wait for responses. This is useful for scenarios where an executor needs to interact with external systems, such as human-in-the-loop interactions, or any other asynchronous operations.

Requests and responses are handled via a special built-in executor called `RequestInfoExecutor`.

Python

```python
from agent_framework import RequestInfoExecutor

# Create a RequestInfoExecutor with an ID
request_info_executor = RequestInfoExecutor(id="request-info-executor")
```

Add the `RequestInfoExecutor` to a workflow.

Python

```python
from agent_framework import WorkflowBuilder

executor_a = SomeExecutor()
workflow_builder = WorkflowBuilder()
workflow_builder.set_start_executor(request_info_executor)
workflow_builder.add_edge(request_info_executor, executor_a)
workflow = workflow_builder.build()
```

Now, because in the workflow we have `executor_a` connected to the `request_info_executor` in both directions, `executor_a` needs to be able to send requests and receive responses via the `request_info_executor`. Here is what we need to do in `SomeExecutor` to send a request and receive a response.

Python

```python
from agent_framework import (
    Executor,
    RequestResponse,
    WorkflowContext,
    handler,
)

class SomeExecutor(Executor):

    @handler
    async def handle(
        self,
        request: RequestResponse[CustomRequestType, CustomResponseType],
        context: WorkflowContext[CustomResponseType],
    ):
        # Process the response...
        ...
        # Send a request
        await context.send_message(CustomRequestType(...))
```

Alternatively, `SomeExecutor` can separate the request sending and response handling into two handlers.

Python

```python
class SomeExecutor(Executor):

    @handler
    async def handle_response(
        self,
        response: CustomResponseType[CustomRequestType,
CustomResponseType],
        context: WorkflowContext,
    ):
        # Process the response...
        ...

    @handler
    async def handle_other_data(
        self,
        data: OtherDataType,
        context: WorkflowContext[CustomRequestType],
    ):
        # Process the message...
        ...
        # Send a request
        await context.send_message(CustomRequestType(...))
```

Handling Requests and Responses

The `RequestInfoExecutor` emits a `RequestInfoEvent` when it receives a request. You can subscribe to these events to handle incoming requests from the workflow. When you receive a response from an external system, send it back to the workflow using the response mechanism. The framework automatically routes the response to the executor that sent the original request.

Python

```python
from agent_framework import RequestInfoEvent

while True:
    request_info_events : list[RequestInfoEvent] = []
    pending_responses : dict[str, CustomResponseType] = {}

    stream = workflow.run_stream(input) if not pending_responses else work-
flow.send_responses_streaming(pending_responses)

    async for event in stream:
        if isinstance(event, RequestInfoEvent):
            # Handle `RequestInfoEvent` from the workflow
            request_info_events.append(event)

    if not request_info_events:
        break

    for request_info_event in request_info_events:
        # Handle `RequestInfoEvent` from the workflow
        response = CustomResponseType(...)
        pending_responses[request_info_event.request_id] = response
```

Checkpoints and Requests

To learn more about checkpoints, please refer to this page.

When a checkpoint is created, pending requests are also saved as part of the checkpoint state. When you restore from a checkpoint, any pending requests will be re-emitted, allowing the workflow to continue processing from where it left off.

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Shared States

10/02/2025

This document provides an overview of **Shared States** in the Microsoft Agent Framework Workflow system.

Overview

Shared States allow multiple executors within a workflow to access and modify common data. This feature is essential for scenarios where different parts of the workflow need to share information where direct message passing is not feasible or efficient.

Writing to Shared States

Python

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

class FileReadExecutor(Executor):

    @handler
    async def handle(self, file_path: str, ctx: WorkflowContext[str]):
        # Read file content from embedded resource
        with open(file_path, 'r') as file:
            file_content = file.read()
        # Store file content in a shared state for access by other execu-
tors
        file_id = str(uuid.uuid4())
        await ctx.set_shared_state(file_id, file_content)

        await ctx.send_message(file_id)
```

Accessing Shared States

Python

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

class WordCountingExecutor(Executor):

    @handler
    async def handle(self, file_id: str, ctx: WorkflowContext[int]):
        # Retrieve the file content from the shared state
        file_content = await ctx.get_shared_state(file_id)
        if file_content is None:
            raise ValueError("File content state not found")

        await ctx.send_message(len(file_content.split()))
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Using workflows as Agents

10/02/2025

This document provides an overview of how to use **Workflows as Agents** in the Microsoft Agent Framework.

Overview

Developers can turn a workflow into an Agent Framework Agent and interact with the workflow as if it were an agent. This feature enables the following scenarios:

- Integrate workflows with APIs that already support the Agent interface.
- Use a workflow to drive single agent interactions, which can create more powerful agents.
- Close the loop between agents and workflows, creating opportunities for advanced compositions.

Creating a Workflow Agent

Create a workflow of any complexity and then wrap it as an agent.

Python

```python
workflow_agent = workflow.as_agent(name="Workflow Agent")
workflow_agent_thread = workflow_agent.get_new_thread()
```

Using a Workflow Agent

Then use the workflow agent like any other Agent Framework agent.

Python

```python
async for update in workflow_agent.run_streaming(input, workflow_agen-
t_thread):
    print(update)
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.

- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows - Visualization

10/02/2025

Sometimes a workflow that has multiple executors and complex interactions can be hard to understand from just reading the code. Visualization can help you see the structure of the workflow more clearly, so that you can verify that it has the intended design.

Workflow visualization is done via a `WorkflowViz` object that can be instantiated with a `Workflow` object. The `WorkflowViz` object can then generate visualizations in different formats, such as Graphviz DOT format or Mermaid diagram format.

Tip

To export visualization images you also need to install GraphViz .

Creating a `WorkflowViz` object is straightforward:

Python

```python
from agent_framework import WorkflowBuilder, WorkflowViz

# Create a workflow with a fan-out and fan-in pattern
workflow = (
    WorkflowBuilder()
    .set_start_executor(dispatcher)
    .add_fan_out_edges(dispatcher, [researcher, marketer, legal])
    .add_fan_in_edges([researcher, marketer, legal], aggregator)
    .build()
)

viz = WorkflowViz(workflow)
```

Then, you can create visualizations in different formats:

Python

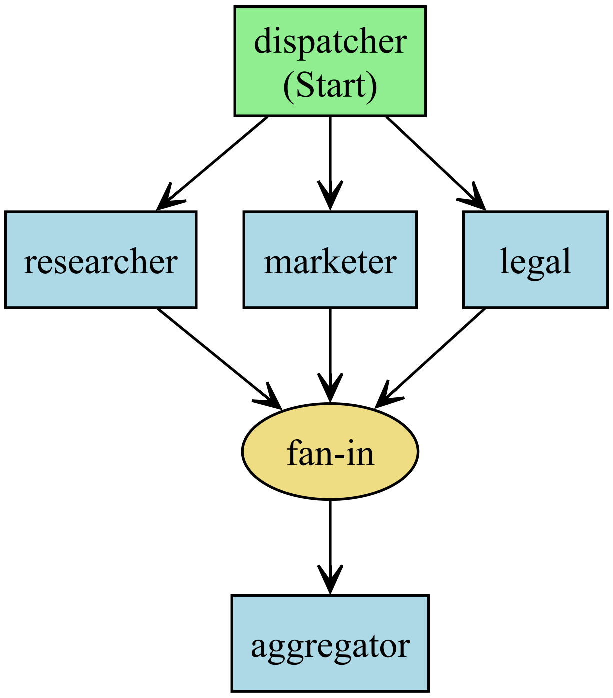
```python
# Mermaid diagram
print(viz.to_mermaid())
# DiGraph string
print(viz.to_digraph())
# Export to a file
print(viz.export(format="svg"))
```

The exported diagram will look similar to the following for the example workflow:

mermaid

```
flowchart TD
  dispatcher["dispatcher (Start)"];
  researcher["researcher"];
  marketer["marketer"];
  legal["legal"];
  aggregator["aggregator"];
  fan_in__aggregator__e3a4ff58((fan-in))
  legal --> fan_in__aggregator__e3a4ff58;
  marketer --> fan_in__aggregator__e3a4ff58;
  researcher --> fan_in__aggregator__e3a4ff58;
  fan_in__aggregator__e3a4ff58 --> aggregator;
  dispatcher --> researcher;
  dispatcher --> marketer;
  dispatcher --> legal;
```

or in Graphviz DOT format:



Microsoft Agent Framework Workflows - Working with Agents

10/02/2025

This page provides an overview of how to use **Agents** within the Microsoft Agent Framework Workflows.

Overview

To add intelligence to your workflows, you can leverage AI agents as part of your workflow execution. AI agents can be easily integrated into workflows, allowing you to create complex, intelligent solutions that were previously difficult to achieve.

Using the Built-in Agent Executor

You can add agents to your workflow via edges:

Python

```python
from agent_framework import WorkflowBuilder
from agent_framework.azure import AzureChatClient
from azure.identity import AzureCliCredential

# Create the agents first
chat_client = AzureChatClient(credential=AzureCliCredential())
writer_agent: ChatAgent = chat_client.create_agent(
    instructions=(
        "You are an excellent content writer. You create new content and
edit contents based on the feedback."
    ),
    name="writer_agent",
)
reviewer_agent = chat_client.create_agent(
    instructions=(
        "You are an excellent content reviewer."
        "Provide actionable feedback to the writer about the provided con-
tent."
        "Provide the feedback in the most concise manner possible."
    ),
    name="reviewer_agent",
)

# Build a workflow with the agents
builder = WorkflowBuilder()
builder.set_start_executor(writer_agent)
builder.add_edge(writer_agent, reviewer_agent)
workflow = builder.build()
```

Running the Workflow

Inside the workflow created above, the agents are actually wrapped inside an executor that handles the communication of the agent with other parts of the workflow. The executor can handle three message types:

- `str`: A single chat message in string format
- `ChatMessage`: A single chat message
- `List<ChatMessage>`: A list of chat messages

Whenever the executor receives a message of one of these types, it will trigger the agent to respond, and the response type will be an `AgentExecutorResponse` object. This class contains useful information about the agent's response, including:

- `executor_id`: The ID of the executor that produced this response
- `agent_run_response`: The full response from the agent
- `full_conversation`: The full conversation history up to this point

Two possible event type related to the agents' responses can be emitted when running the workflow:

- `AgentRunUpdateEvent` containing chunks of the agent's response as they are generated in streaming mode.
- `AgentRunEvent` containing the full response from the agent in non-streaming mode.

> By default, agents are wrapped in executors that run in streaming mode. You can customize this behavior by creating a custom executor. See the next section for more details.

Python

```python
last_executor_id = None
async for event in workflow.run_streaming("Write a short blog post about AI agents."):
    if isinstance(event, AgentRunUpdateEvent):
        if event.executor_id != last_executor_id:
            if last_executor_id is not None:
                print()
            print(f"{event.executor_id}:", end=" ", flush=True)
            last_executor_id = event.executor_id
        print(event.data, end="", flush=True)
```

Using a Custom Agent Executor

Sometimes you may want to customize how AI agents are integrated into a workflow. You can achieve this by creating a custom executor. This allows you to control:

- The invocation of the agent: streaming or non-streaming
- The message types the agent will handle, including custom message types
- The life cycle of the agent, including initialization and cleanup
- The usage of agent threads and other resources
- Additional events emitted during the agent's execution, including custom events
- Integration with other workflow features, such as shared states and requests/responses

Python

```python
from agent_framework import (
    ChatAgent,
    ChatMessage,
    Executor,
    WorkflowContext,
    handler
)

class Writer(Executor):

    agent: ChatAgent

    def __init__(self, chat_client: AzureChatClient, id: str = "writer"):
        # Create a domain specific agent using your configured
AzureChatClient.
        agent = chat_client.create_agent(
            instructions=(
                "You are an excellent content writer. You create new con-
tent and edit contents based on the feedback."
            ),
        )
        # Associate the agent with this executor node. The base Executor
stores it on self.agent.
        super().__init__(agent=agent, id=id)

    @handler
    async def handle(self, message: ChatMessage, ctx:
WorkflowContext[list[ChatMessage]]) -> None:
        """Handles a single chat message and forwards the accumulated mes-
sages to the next executor in the workflow."""
        # Invoke the agent with the incoming message and get the response
        messages: list[ChatMessage] = [message]
        response = await self.agent.run(messages)
        # Accumulate messages and send them to the next executor in the
workflow.
        messages.extend(response.messages)
        await ctx.send_message(messages)
```

Next Steps

- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows Core Concepts

10/02/2025

This page provides an overview of the core concepts and architecture of the Microsoft Agent Framework Workflow system. It covers the fundamental building blocks, execution model, and key features that enable developers to create robust, type-safe workflows.

Core Components

The workflow framework consists of four core layers that work together to create a flexible, type-safe execution environment:

- **Executors** and **Edges** form a directed graph representing the workflow structure
- **Workflows** orchestrate executor execution, message routing, and event streaming
- **Events** provide observability into the workflow execution

Next Steps

To dive deeper into each core component, explore the following sections:

- Executors
- Edges
- Workflows
- Events

Microsoft Agent Framework Workflows Core Concepts - Edges

10/02/2025

This document provides an in-depth look at the **Edges** component of the Microsoft Agent Framework Workflow system.

Overview

Edges define how messages flow between executors with optional conditions. They represent the connections in the workflow graph and determine the data flow paths.

Types of Edges

The framework supports several edge patterns:

1. **Direct Edges**: Simple one-to-one connections between executors
2. **Conditional Edges**: Edges with conditions that determine when messages should flow
3. **Fan-out Edges**: One executor sending messages to multiple targets
4. **Fan-in Edges**: Multiple executors sending messages to a single target

Direct Edges

The simplest form of connection between two executors:

Python

```python
from agent_framework import WorkflowBuilder

builder = WorkflowBuilder()
builder.add_edge(source_executor, target_executor)
builder.set_start_executor(source_executor)
workflow = builder.build()
```

Conditional Edges

Edges that only activate when certain conditions are met:

Python

```python
from agent_framework import WorkflowBuilder

builder = WorkflowBuilder()
builder.add_edge(spam_detector, email_processor, condition=lambda result:
isinstance(result, SpamResult) and not result.is_spam)
builder.add_edge(spam_detector, spam_handler, condition=lambda result:
isinstance(result, SpamResult) and result.is_spam)
builder.set_start_executor(spam_detector)
workflow = builder.build()
```

Switch-case Edges

Route messages to different executors based on conditions:

Python

```python
from agent_framework import (
    Case,
    Default,
    WorkflowBuilder,
)

builder = WorkflowBuilder()
builder.set_start_executor(router_executor)
builder.add_switch_case_edge_group(
    router_executor,
    [
        Case(
            condition=lambda message: message.priority < Priority.NORMAL,
            target=executor_a,
        ),
        Case(
            condition=lambda message: message.priority < Priority.HIGH,
            target=executor_b,
        ),
        Default(target=executor_c)
    ],
)
workflow = builder.build()
```

Fan-out Edges

Distribute messages from one executor to multiple targets:

Python

```python
from agent_framework import WorkflowBuilder
```

```
builder = WorkflowBuilder()
builder.set_start_executor(splitter_executor)
builder.add_fan_out_edges(splitter_executor, [worker1, worker2, worker3])
workflow = builder.build()

# Send to specific targets based on partitioner function
builder = WorkflowBuilder()
builder.set_start_executor(splitter_executor)
builder.add_fan_out_edges(
    splitter_executor,
    [worker1, worker2, worker3],
    selection_func=lambda message, target_ids: (
        [0] if message.priority == Priority.HIGH else
        [1, 2] if message.priority == Priority.NORMAL else
        list(range(target_count))
    )
)
workflow = builder.build()
```

Fan-in Edges

Collect messages from multiple sources into a single target:

Python

```
builder.add_fan_in_edge([worker1, worker2, worker3], aggregator_executor)
```

Next Step

- Learn about Workflows to understand how to build and execute workflows.

Microsoft Agent Framework Workflows Core Concepts - Events

10/02/2025

This document provides an in-depth look at the **Events** system of Workflows in the Microsoft Agent Framework.

Overview

There are built-in events that provide observability into the workflow execution.

Built-in Event Types

Python

```python
# Workflow lifecycle events
WorkflowStartedEvent    # Workflow execution begins
WorkflowOutputEvent     # Workflow produces an output
WorkflowErrorEvent      # Workflow encounters an error

# Executor events
ExecutorInvokeEvent     # Executor starts processing
ExecutorCompleteEvent   # Executor finishes processing

# Request events
RequestInfoEvent        # A request is issued
```

Consuming Events

Python

```python
from agent_framework import (
    ExecutorCompleteEvent,
    ExecutorInvokeEvent,
    WorkflowOutputEvent,
    WorkflowErrorEvent,
)

async for event in workflow.run_stream(input_message):
    match event:
        case ExecutorInvokeEvent() as invoke:
            print(f"Starting {invoke.executor_id}")
        case ExecutorCompleteEvent() as complete:
            print(f"Completed {complete.executor_id}: {complete.data}")
        case WorkflowOutputEvent() as output:
            print(f"Workflow produced output: {output.data}")
            return
        case WorkflowErrorEvent() as error:
            print(f"Workflow error: {error.exception}")
            return
```

Custom Events

Users can define and emit custom events during workflow execution for enhanced observability.

Python

```python
from agent_framework import (
    handler,
    Executor,
    WorkflowContext,
    WorkflowEvent,
)

class CustomEvent(WorkflowEvent):
    def __init__(self, message: str):
        super().__init__(message)

class CustomExecutor(Executor):

    @handler
    async def handle(self, message: str, ctx: WorkflowContext[str]) ->
None:
        await ctx.add_event(CustomEvent(f"Processing message: {message}"))
        # Executor logic...
```

Next Steps

- Learn how to use agents in workflows to build intelligent workflows.
- Learn how to use workflows as agents.
- Learn how to handle requests and responses in workflows.
- Learn how to manage state in workflows.
- Learn how to create checkpoints and resume from them.

Microsoft Agent Framework Workflows Core Concepts - Executors

10/02/2025

This document provides an in-depth look at the **Executors** component of the Microsoft Agent Framework Workflow system.

Overview

Executors are the fundamental building blocks that process messages in a workflow. They are autonomous processing units that receive typed messages, perform operations, and can produce output messages or events.

Executors inherit from the `Executor` base class. Each executor has a unique identifier and can handle specific message types using methods decorated with the `@handler` decorator. Handlers must have the proper annotation to specify the type of messages they can process.

Basic Executor Structure

Python

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

class UpperCase(Executor):

    @handler
    async def to_upper_case(self, text: str, ctx: WorkflowContext[str]) ->
None:
        """Convert the input to uppercase and forward it to the next node.

        Note: The WorkflowContext is parameterized with the type this han-
dler will
        emit. Here WorkflowContext[str] means downstream nodes should ex-
pect str.
        """
        await ctx.send_message(text.upper())
```

It is possible to create an executor from a function by using the `@executor` decorator:

Python

```
from agent_framework import (
    WorkflowContext,
    executor,
)

@executor(id="upper_case_executor")
async def upper_case(text: str, ctx: WorkflowContext[str]) -> None:
    """Convert the input to uppercase and forward it to the next node.

    Note: The WorkflowContext is parameterized with the type this handler 
will
    emit. Here WorkflowContext[str] means downstream nodes should expect 
str.
    """
    await ctx.send_message(text.upper())
```

It is also possible to handle multiple input types by defining multiple handlers:

Python

```
class SampleExecutor(Executor):

    @handler
    async def to_upper_case(self, text: str, ctx: WorkflowContext[str]) -> 
None:
        """Convert the input to uppercase and forward it to the next node.

        Note: The WorkflowContext is parameterized with the type this han-
dler will
        emit. Here WorkflowContext[str] means downstream nodes should ex-
pect str.
        """
        await ctx.send_message(text.upper())

    @handler
    async def double_integer(self, number: int, ctx: WorkflowContext[int]) 
-> None:
        """Double the input integer and forward it to the next node.

        Note: The WorkflowContext is parameterized with the type this han-
dler will
        emit. Here WorkflowContext[int] means downstream nodes should ex-
pect int.
        """
        await ctx.send_message(number * 2)
```

The `WorkflowContext` Object

The `WorkflowContext` object provides methods for the handler to interact with the workflow during execution. The `WorkflowContext` is parameterized with the type of messages the handler will emit and the type of outputs it can yield.

The most commonly used method is `send_message`, which allows the handler to send messages to connected executors.

Python

```python
from agent_framework import WorkflowContext

class SomeHandler(Executor):

    @handler
    async def some_handler(message: str, ctx: WorkflowContext[str]) ->
None:
        await ctx.send_message("Hello, World!")
```

A handler can use `yield_output` to produce outputs that will be considered as workflow outputs and be returned/streamed to the caller as an output event:

Python

```python
from agent_framework import WorkflowContext

class SomeHandler(Executor):

    @handler
    async def some_handler(message: str, ctx: WorkflowContext[Never, str])
-> None:
        await ctx.yield_output("Hello, World!")
```

If a handler neither sends messages nor yields outputs, no type parameter is needed for `WorkflowContext`:

Python

```python
from agent_framework import WorkflowContext

class SomeHandler(Executor):

    @handler
    async def some_handler(message: str, ctx: WorkflowContext) -> None:
        print("Doing some work...")
```

Next Step

- Learn about Edges to understand how executors are connected in a workflow.

Microsoft Agent Framework Workflows Core Concepts - Workflows

10/02/2025

This document provides an in-depth look at the **Workflows** component of the Microsoft Agent Framework Workflow system.

Overview

A Workflow ties everything together and manages execution. It's the orchestrator that coordinates executor execution, message routing, and event streaming.

Building Workflows

Workflows are constructed using the `WorkflowBuilder` class, which provides a fluent API for defining the workflow structure:

Python

```python
from agent_framework import WorkflowBuilder

processor = DataProcessor()
validator = Validator()
formatter = Formatter()

# Build workflow
builder = WorkflowBuilder()
builder.set_start_executor(processor)  # Set starting executor
builder.add_edge(processor, validator)
builder.add_edge(validator, formatter)
workflow = builder.build()
```

Workflow Execution

Workflows support both streaming and non-streaming execution modes:

Python

```python
from agent_framework import WorkflowCompletedEvent

# Streaming execution - get events as they happen
async for event in workflow.run_stream(input_message):
    if isinstance(event, WorkflowCompletedEvent):
        print(f"Workflow completed: {event.data}")

# Non-streaming execution - wait for completion
events = await workflow.run(input_message)
print(f"Final result: {events.get_completed_event()}")
```

Workflow Validation

The framework performs comprehensive validation when building workflows:

- **Type Compatibility**: Ensures message types are compatible between connected executors
- **Graph Connectivity**: Verifies all executors are reachable from the start executor
- **Executor Binding**: Confirms all executors are properly bound and instantiated
- **Edge Validation**: Checks for duplicate edges and invalid connections

Execution Model

The framework uses a modified Pregel execution model with clear data flow semantics and superstep-based processing.

Pregel-Style Supersteps

Workflow execution is organized into discrete supersteps, where each superstep processes all available messages in parallel:

text

```
Superstep N:
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│  Collect All    │───▶│ Route Messages  │───▶│  Execute All    │
│  Pending        │    │ Based on Type   │    │  Target         │
│  Messages       │    │ & Conditions    │    │  Executors      │
└─────────────────┘    └─────────────────┘    └─────────────────┘
                                                        │
                                                        ▼
┌─────────────────┐    ┌─────────────────┐             │
│  Start Next     │◀───│ Emit Events &   │◀────────────┘
│  Superstep      │    │ New Messages    │
└─────────────────┘    └─────────────────┘
```

Key Execution Characteristics

- **Superstep Isolation**: All executors in a superstep run concurrently without interfering with each other
- **Message Delivery**: Messages are delivered in parallel to all matching edges

- **Event Streaming**: Events are emitted in real-time as executors complete processing
- **Type Safety**: Runtime type validation ensures messages are routed to compatible handlers

Next Step

- Learn about events to understand how to monitor and observe workflow execution.

Microsoft Agent Framework Workflows Orchestrations

10/02/2025

Orchestrations are pre-built workflow patterns that allow developers to quickly create complex workflows by simply plugging in their own AI agents.

Why Multi-Agent?

Traditional single-agent systems are limited in their ability to handle complex, multi-faceted tasks. By orchestrating multiple agents, each with specialized skills or roles, we can create systems that are more robust, adaptive, and capable of solving real-world problems collaboratively.

Supported Orchestrations

Expand table

Pattern	Description	Typical Use Case
Concurrent	Broadcasts a task to all agents, collects results independently.	Parallel analysis, independent subtasks, ensemble decision making.
Sequential	Passes the result from one agent to the next in a defined order.	Step-by-step workflows, pipelines, multi-stage processing.
Handoff	Dynamically passes control between agents based on context or rules.	Dynamic workflows, escalation, fallback, or expert handoff scenarios.
Magentic	Inspired by MagenticOne .	Complex, generalist multi-agent collaboration.

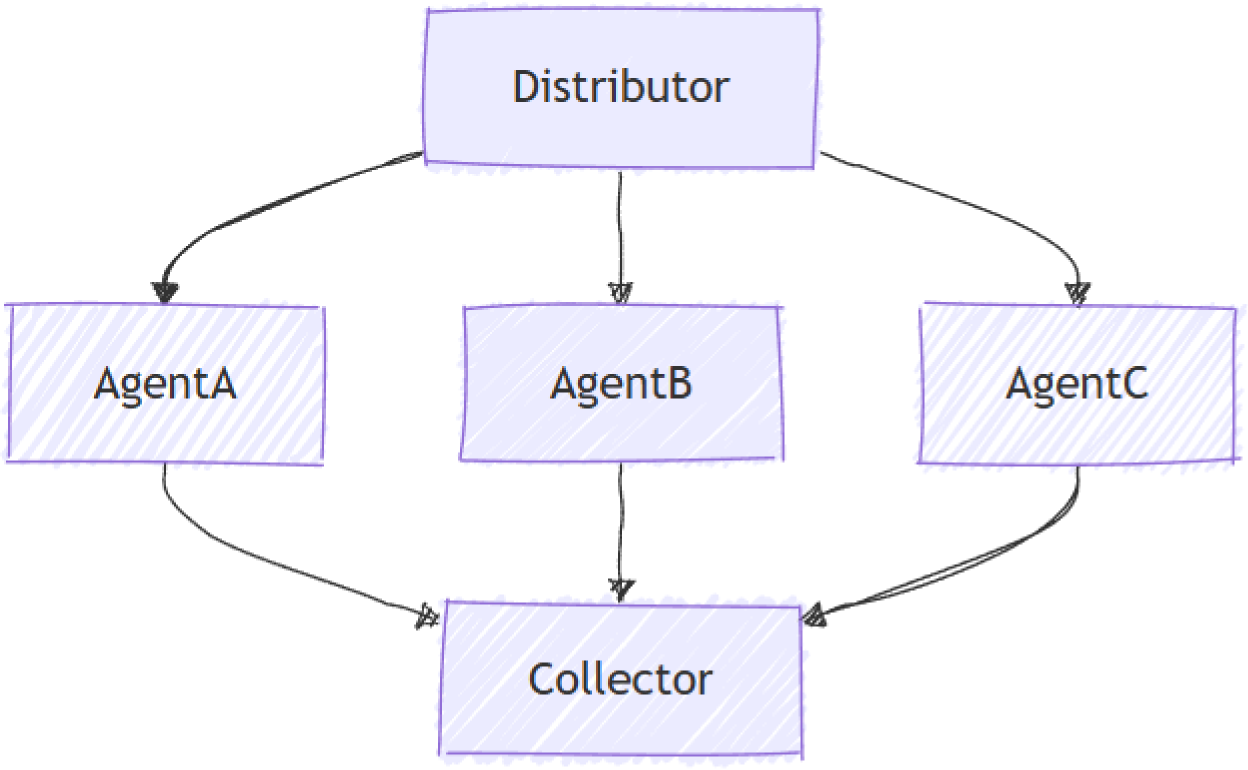
Next Steps

Explore the individual orchestration patterns to understand their unique features and how to use them effectively in your applications.

Microsoft Agent Framework Workflows Orchestrations - Concurrent

10/02/2025

Concurrent orchestration enables multiple agents to work on the same task in parallel. Each agent processes the input independently, and their results are collected and aggregated. This approach is well-suited for scenarios where diverse perspectives or solutions are valuable, such as brainstorming, ensemble reasoning, or voting systems.



What You'll Learn

- How to define multiple agents with different expertise
- How to orchestrate these agents to work concurrently on a single task
- How to collect and process the results

Agents are specialized entities that can process tasks. Here, we define three agents: a research expert, a marketing expert, and a legal expert.

Python

```python
from agent_framework.azure import AzureChatClient

# 1) Create three domain agents using AzureChatClient
chat_client = AzureChatClient(credential=AzureCliCredential())
```

```
researcher = chat_client.create_agent(
    instructions=(
        "You're an expert market and product researcher. Given a prompt,
provide concise, factual insights,"
        " opportunities, and risks."
    ),
    name="researcher",
)

marketer = chat_client.create_agent(
    instructions=(
        "You're a creative marketing strategist. Craft compelling value
propositions and target messaging"
        " aligned to the prompt."
    ),
    name="marketer",
)

legal = chat_client.create_agent(
    instructions=(
        "You're a cautious legal/compliance reviewer. Highlight con-
straints, disclaimers, and policy concerns"
        " based on the prompt."
    ),
    name="legal",
)
```

Set Up the Concurrent Orchestration

The `ConcurrentBuilder` class allows you to construct a workflow to run multiple agents in parallel. You pass the list of agents as participants.

Python

```python
from agent_framework import ConcurrentBuilder

# 2) Build a concurrent workflow
# Participants are either Agents (type of AgentProtocol) or Executors
workflow = ConcurrentBuilder().participants([researcher, marketer,
legal]).build()
```

Run the Concurrent Workflow and Collect the Results

Python

```
from agent_framework import ChatMessage, WorkflowCompletedEvent

# 3) Run with a single prompt, stream progress, and pretty-print the final
combined messages
completion: WorkflowCompletedEvent | None = None
async for event in workflow.run_stream("We are launching a new budget-
friendly electric bike for urban commuters."):
    if isinstance(event, WorkflowCompletedEvent):
        completion = event

if completion:
    print("===== Final Aggregated Conversation (messages) =====")
    messages: list[ChatMessage] | Any = completion.data
    for i, msg in enumerate(messages, start=1):
        name = msg.author_name if msg.author_name else "user"
        print(f"{'-' * 60}\n\n{i:02d} [{name}]:\n{msg.text}")
```

Sample Output

plaintext

```
Sample Output:

    ===== Final Aggregated Conversation (messages) =====
    ------------------------------------------------------------


    01 [user]:
    We are launching a new budget-friendly electric bike for urban com-
muters.
    ------------------------------------------------------------


    02 [researcher]:
    **Insights:**

    - **Target Demographic:** Urban commuters seeking affordable, eco-
friendly transport;
        likely to include students, young professionals, and price-sensi-
tive urban residents.
    - **Market Trends:** E-bike sales are growing globally, with increasing
urbanization,
        higher fuel costs, and sustainability concerns driving adoption.
    - **Competitive Landscape:** Key competitors include brands like Rad
Power Bikes, Aventon,
        Lectric, and domestic budget-focused manufacturers in North
America, Europe, and Asia.
    - **Feature Expectations:** Customers expect reliability, ease-of-use,
theft protection,
        lightweight design, sufficient battery range for daily city com-
mutes (typically 25-40 miles),
        and low-maintenance components.
    **Opportunities:**

    - **First-time Buyers:** Capture newcomers to e-biking by emphasizing
affordability, ease of
        operation, and cost savings vs. public transit/car ownership.
    ...
    ------------------------------------------------------------

    03 [marketer]:
    **Value Proposition:**
    "Empowering your city commute: Our new electric bike combines afford-
ability, reliability, and
        sustainable design—helping you conquer urban journeys without
breaking the bank."

    **Target Messaging:**

    *For Young Professionals:*
    ...
    ------------------------------------------------------------

    04 [legal]:
    **Constraints, Disclaimers, & Policy Concerns for Launching a Budget-
Friendly Electric Bike for Urban Commuters:**

    **1. Regulatory Compliance**
    - Verify that the electric bike meets all applicable federal, state,
and local regulations
        regarding e-bike classification, speed limits, power output, and
safety features.
    - Ensure necessary certifications (e.g., UL certification for batter-
ies, CE markings if sold internationally) are obtained.

    **2. Product Safety**
    - Include consumer safety warnings regarding use, battery handling,
charging protocols, and age restrictions.
```

Advanced: Custom Agent Executors

Concurrent orchestration supports custom executors that wrap agents with additional logic. This is useful when you need more control over how agents are initialized and how they process requests:

Define Custom Agent Executors

Python

```python
from agent_framework import (
    AgentExecutorRequest,
    AgentExecutorResponse,
    ChatAgent,
    Executor,
    WorkflowContext,
    handler,
)

class ResearcherExec(Executor):
    agent: ChatAgent

    def __init__(self, chat_client: AzureChatClient, id: str =
"researcher"):
        agent = chat_client.create_agent(
            instructions=(
                "You're an expert market and product researcher. Given a
prompt, provide concise, factual insights,"
                " opportunities, and risks."
            ),
            name=id,
        )
        super().__init__(agent=agent, id=id)

    @handler
    async def run(self, request: AgentExecutorRequest, ctx:
WorkflowContext[AgentExecutorResponse]) -> None:
        response = await self.agent.run(request.messages)
        full_conversation = list(request.messages) + list(response.mes-
sages)
        await ctx.send_message(AgentExecutorResponse(self.id, response,
full_conversation=full_conversation))

class MarketerExec(Executor):
    agent: ChatAgent

    def __init__(self, chat_client: AzureChatClient, id: str = "marketer"):
        agent = chat_client.create_agent(
            instructions=(
                "You're a creative marketing strategist. Craft compelling
value propositions and target messaging"
                " aligned to the prompt."
            ),
            name=id,
        )
        super().__init__(agent=agent, id=id)

    @handler
    async def run(self, request: AgentExecutorRequest, ctx:
WorkflowContext[AgentExecutorResponse]) -> None:
        response = await self.agent.run(request.messages)
        full_conversation = list(request.messages) + list(response.mes-
sages)
        await ctx.send_message(AgentExecutorResponse(self.id, response,
full_conversation=full_conversation))
```

Build a Workflow with Custom Executors

Python

```python
chat_client = AzureChatClient(credential=AzureCliCredential())

researcher = ResearcherExec(chat_client)
marketer = MarketerExec(chat_client)
legal = LegalExec(chat_client)

workflow = ConcurrentBuilder().participants([researcher, marketer,
legal]).build()
```

Advanced: Custom Aggregator

By default, concurrent orchestration aggregates all agent responses into a list of messages. You can override this behavior with a custom aggregator that processes the results in a specific way:

Define a Custom Aggregator

Python

```python
# Define a custom aggregator callback that uses the chat client to summa-
rize
async def summarize_results(results: list[Any]) -> str:
    # Extract one final assistant message per agent
    expert_sections: list[str] = []
    for r in results:
        try:
            messages = getattr(r.agent_run_response, "messages", [])
            final_text = messages[-1].text if messages and
hasattr(messages[-1], "text") else "(no content)"
            expert_sections.append(f"{getattr(r, 'executor_id',
'expert')}:\n{final_text}")
        except Exception as e:
            expert_sections.append(f"{getattr(r, 'executor_id', 'expert')}:
(error: {type(e).__name__}: {e})")

    # Ask the model to synthesize a concise summary of the experts' outputs
    system_msg = ChatMessage(
        Role.SYSTEM,
        text=(
            "You are a helpful assistant that consolidates multiple domain
expert outputs "
            "into one cohesive, concise summary with clear takeaways. Keep
it under 200 words."
        ),
    )
    user_msg = ChatMessage(Role.USER, text="\n\n".join(expert_sections))

    response = await chat_client.get_response([system_msg, user_msg])
    # Return the model's final assistant text as the completion result
    return response.messages[-1].text if response.messages else ""
```

Build a Workflow with Custom Aggregator

Python

```python
workflow = (
    ConcurrentBuilder()
    .participants([researcher, marketer, legal])
    .with_aggregator(summarize_results)
    .build()
)

completion: WorkflowCompletedEvent | None = None
async for event in workflow.run_stream("We are launching a new budget-
friendly electric bike for urban commuters."):
    if isinstance(event, WorkflowCompletedEvent):
        completion = event

if completion:
    print("===== Final Consolidated Output =====")
    print(completion.data)
```

Sample Output with Custom Aggregator

plaintext

```
===== Final Consolidated Output =====
Urban e-bike demand is rising rapidly due to eco-awareness, urban conges-
tion, and high fuel costs,
with market growth projected at a ~10% CAGR through 2030. Key customer con-
cerns are affordability,
easy maintenance, convenient charging, compact design, and theft protec-
tion. Differentiation opportunities
include integrating smart features (GPS, app connectivity), offering sub-
scription or leasing options, and
developing portable, space-saving designs. Partnering with local govern-
ments and bike shops can boost visibility.

Risks include price wars eroding margins, regulatory hurdles, battery qual-
ity concerns, and heightened expectations
for after-sales support. Accurate, substantiated product claims and trans-
parent marketing (with range disclaimers)
are essential. All e-bikes must comply with local and federal regulations
```

```
on speed, wattage, safety certification,
and labeling. Clear warranty, safety instructions (especially regarding
batteries), and inclusive, accessible
marketing are required. For connected features, data privacy policies and
user consents are mandatory.

Effective messaging should target young professionals, students, eco-con-
scious commuters, and first-time buyers,
emphasizing affordability, convenience, and sustainability. Slogan sugges-
tion: "Charge Ahead—City Commutes Made
Affordable." Legal review in each target market, compliance vetting, and
robust customer support policies are
critical before launch.
```

Key Concepts

- **Parallel Execution**: All agents work on the task simultaneously and independently
- **Result Aggregation**: Results are collected and can be processed by either the default or custom aggregator
- **Diverse Perspectives**: Each agent brings its unique expertise to the same problem
- **Flexible Participants**: You can use agents directly or wrap them in custom executors
- **Custom Processing**: Override the default aggregator to synthesize results in domain-specific ways

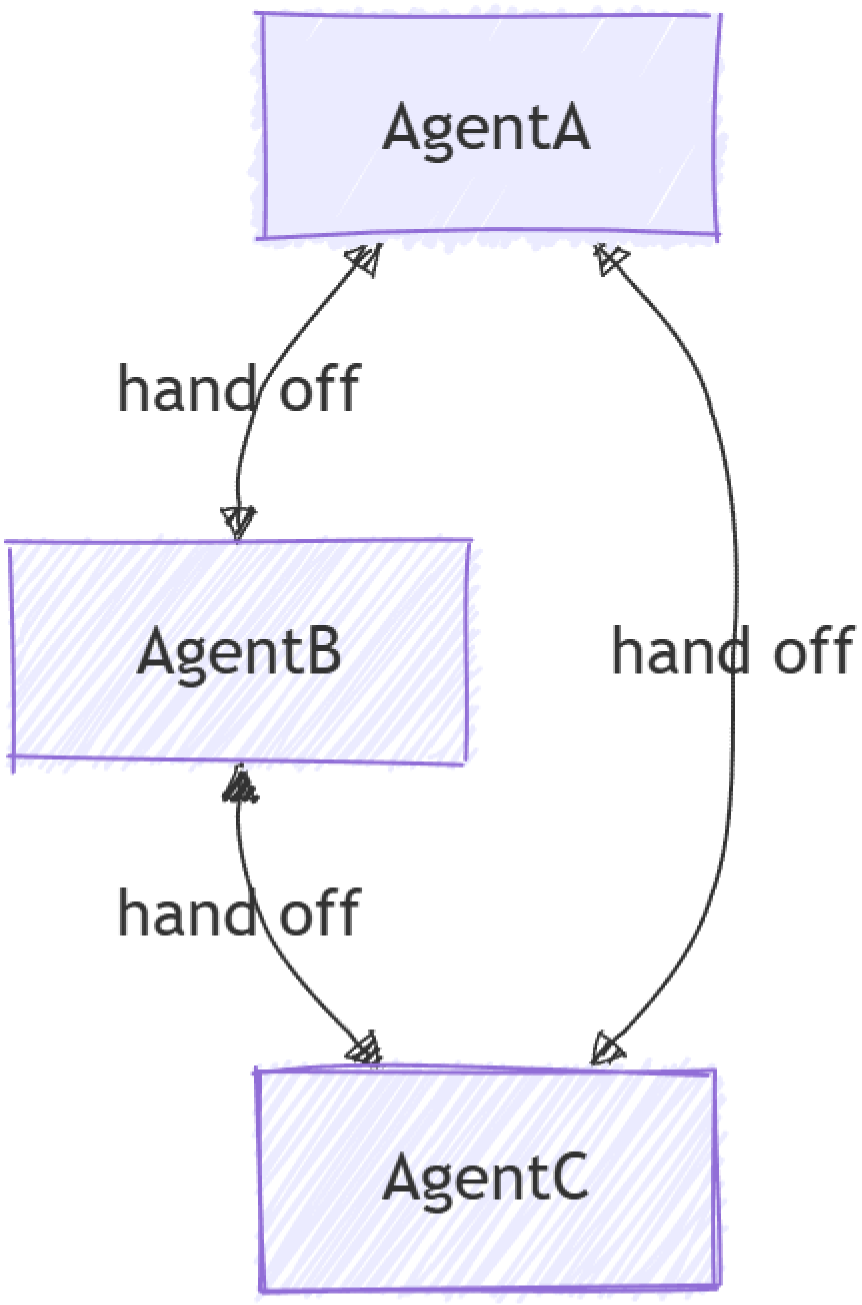
Next steps

Sequential Orchestration

Microsoft Agent Framework Workflows Orchestrations - Handoff

10/02/2025

Handoff orchestration allows agents to transfer control to one another based on the context or user request. Each agent can "handoff" the conversation to another agent with the appropriate expertise, ensuring that the right agent handles each part of the task. This is particularly useful in customer support, expert systems, or any scenario requiring dynamic delegation.



Differences Between Handoff and Agent-as-Tools

While agent-as-tools is commonly considered as a multi-agent pattern and it may look similar to handoff at first glance, there are fundamental differences between the two:

- **Control Flow**: In handoff orchestration, control is explicitly passed between agents based on defined rules. Each agent can decide to hand off the entire task to another agent. There is no central authority managing the workflow. In contrast, agent-as-tools involves a primary agent that delegates sub tasks to other agents and once the agent completes the sub task, control returns to the primary agent.
- **Task Ownership**: In handoff, the agent receiving the handoff takes full ownership of the task. In agent-as-tools, the primary agent retains overall responsibility for the task, while other agents are treated as tools to assist in specific subtasks.
- **Context Management**: In handoff orchestration, the conversation is handed off to another agent entirely. The receiving agent has full context of what has been done so far. In agent-as-tools, the primary agent manages the overall context and may provide only relevant information to the tool agents as needed.

What You'll Learn

- How to create specialized agents for different domains
- How to configure handoff rules between agents
- How to build interactive workflows with dynamic agent routing
- How to handle multi-turn conversations with agent switching

In handoff orchestration, agents can transfer control to one another based on context, allowing for dynamic routing and specialized expertise handling.

Set Up the Azure OpenAI Client

C#

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

// 1) Set up the Azure OpenAI client
var endpoint = Environment.GetEnvironmentVariable("AZURE_OPENAI_ENDPOINT")
??
    throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is not
set.");
var deploymentName = Environment.GetEnvironmentVariable("AZURE_OPENAI_DE-
PLOYMENT_NAME") ?? "gpt-4o-mini";
var client = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient(deploymentName)
    .AsIChatClient();
```

Define Your Specialized Agents

Create domain-specific agents and a triage agent for routing:

C#

```
// 2) Create specialized agents
ChatClientAgent historyTutor = new(client,
    "You provide assistance with historical queries. Explain important
events and context clearly. Only respond about history.",
    "history_tutor",
    "Specialist agent for historical questions");

ChatClientAgent mathTutor = new(client,
    "You provide help with math problems. Explain your reasoning at each
step and include examples. Only respond about math.",
    "math_tutor",
    "Specialist agent for math questions");

ChatClientAgent triageAgent = new(client,
    "You determine which agent to use based on the user's homework ques-
tion. ALWAYS handoff to another agent.",
    "triage_agent",
    "Routes messages to the appropriate specialist agent");
```

Configure Handoff Rules

Define which agents can hand off to which other agents:

C#

```
// 3) Build handoff workflow with routing rules
var workflow = AgentWorkflowBuilder.StartHandoffWith(triageAgent)
    .WithHandoff(triageAgent, [mathTutor, historyTutor])  // Triage can
route to either specialist
    .WithHandoff(mathTutor, triageAgent)                  // Math tutor can
return to triage
    .WithHandoff(historyTutor, triageAgent)               // History tutor
can return to triage
    .Build();
```

Run Interactive Handoff Workflow

Handle multi-turn conversations with dynamic agent switching:

C#

```
// 4) Process multi-turn conversations
List<ChatMessage> messages = new();

while (true)
{
    Console.Write("Q: ");
    string userInput = Console.ReadLine()!;
    messages.Add(new(ChatRole.User, userInput));

    // Execute workflow and process events
    StreamingRun run = await InProcessExecution.StreamAsync(workflow, mes-
sages);
    await run.TrySendMessageAsync(new TurnToken(emitEvents: true));

    List<ChatMessage> newMessages = new();
    await foreach (WorkflowEvent evt in run.WatchStreamAsync().Configure-
Await(false))
    {
        if (evt is AgentRunUpdateEvent e)
        {
            Console.WriteLine($"{e.ExecutorId}: {e.Data}");
        }
        else if (evt is WorkflowCompletedEvent completed)
        {
            newMessages = (List<ChatMessage>)completed.Data!;
            break;
        }
    }

    // Add new messages to conversation history
    messages.AddRange(newMessages.Skip(messages.Count));
}
```

Sample Interaction

plaintext

```
Q: What is the derivative of x^2?
triage_agent: This is a math question. I'll hand this off to the math tu-
tor.
math_tutor: The derivative of x^2 is 2x. Using the power rule, we bring
down the exponent (2) and multiply it by the coefficient (1), then reduce
the exponent by 1: d/dx(x^2) = 2x^(2-1) = 2x.
```

```
Q: Tell me about World War 2
triage_agent: This is a history question. I'll hand this off to the history
tutor.
history_tutor: World War 2 was a global conflict from 1939 to 1945. It be-
gan when Germany invaded Poland and involved most of the world's nations.
Key events included the Holocaust, Pearl Harbor attack, D-Day invasion, and
ended with atomic bombs on Japan.

Q: Can you help me with calculus integration?
triage_agent: This is another math question. I'll route this to the math
tutor.
math_tutor: I'd be happy to help with calculus integration! Integration is
the reverse of differentiation. The basic power rule for integration is:
∫x^n dx = x^(n+1)/(n+1) + C, where C is the constant of integration.
```

Key Concepts

- **Dynamic Routing**: Agents can decide which agent should handle the next interaction based on context
- **AgentWorkflowBuilder.StartHandoffWith()**: Defines the initial agent that starts the workflow
- **WithHandoff()**: Configures handoff rules between specific agents
- **Context Preservation**: Full conversation history is maintained across all handoffs
- **Multi-turn Support**: Supports ongoing conversations with seamless agent switching
- **Specialized Expertise**: Each agent focuses on their domain while collaborating through handoffs

Next steps

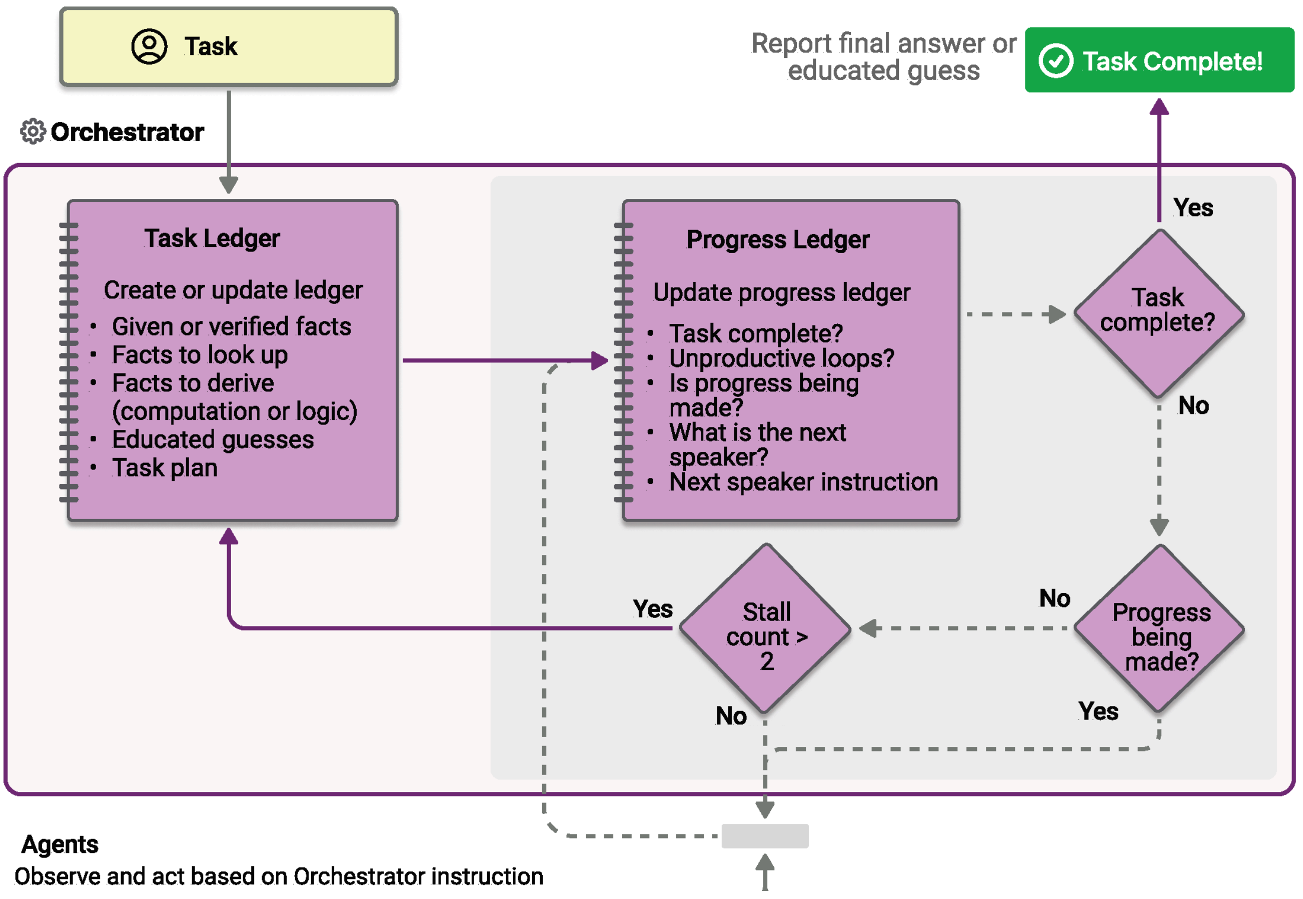
Magnetic Orchestration

Microsoft Agent Framework Workflows Orchestrations - Magentic

10/02/2025

Magentic orchestration is designed based on the Magentic-One system invented by AutoGen. It is a flexible, general-purpose multi-agent pattern designed for complex, open-ended tasks that require dynamic collaboration. In this pattern, a dedicated Magentic manager coordinates a team of specialized agents, selecting which agent should act next based on the evolving context, task progress, and agent capabilities.

The Magentic manager maintains a shared context, tracks progress, and adapts the workflow in real time. This enables the system to break down complex problems, delegate subtasks, and iteratively refine solutions through agent collaboration. The orchestration is especially well-suited for scenarios where the solution path is not known in advance and may require multiple rounds of reasoning, research, and computation.



What You'll Learn

- How to set up a Magentic manager to coordinate multiple specialized agents
- How to configure callbacks for streaming and event handling
- How to implement human-in-the-loop plan review

- How to track agent collaboration and progress through complex tasks

Define Your Specialized Agents

In Magentic orchestration, you define specialized agents that the manager can dynamically select based on task requirements:

Python

```python
from agent_framework import ChatAgent, HostedCodeInterpreterTool
from agent_framework.openai import OpenAIChatClient, OpenAIResponsesClient

researcher_agent = ChatAgent(
    name="ResearcherAgent",
    description="Specialist in research and information gathering",
    instructions=(
        "You are a Researcher. You find information without additional com-
putation or quantitative analysis."
    ),
    # This agent requires the gpt-4o-search-preview model to perform web
searches
    chat_client=OpenAIChatClient(ai_model_id="gpt-4o-search-preview"),
)

coder_agent = ChatAgent(
    name="CoderAgent",
    description="A helpful assistant that writes and executes code to
process and analyze data.",
    instructions="You solve questions using code. Please provide detailed
analysis and computation process.",
    chat_client=OpenAIResponsesClient(),
    tools=HostedCodeInterpreterTool(),
)
```

Set Up Event Callbacks

Magentic orchestration provides rich event callbacks to monitor the workflow progress in real-time:

Python

```python
from agent_framework import (
    MagenticAgentDeltaEvent,
    MagenticAgentMessageEvent,
    MagenticCallbackEvent,
    MagenticFinalResultEvent,
    MagenticOrchestratorMessageEvent,
)
```

```python
# Unified callback for all events
async def on_event(event: MagenticCallbackEvent) -> None:
    if isinstance(event, MagenticOrchestratorMessageEvent):
        # Manager's planning and coordination messages
        print(f"\n[ORCH:{event.kind}]\n\n{getattr(event.message, 'text',
'')}\n{'-' * 26}")

    elif isinstance(event, MagenticAgentDeltaEvent):
        # Streaming tokens from agents
        print(event.text, end="", flush=True)

    elif isinstance(event, MagenticAgentMessageEvent):
        # Complete agent responses
        msg = event.message
        if msg is not None:
            response_text = (msg.text or "").replace("\n", " ")
            print(f"\n[AGENT:{event.agent_id}] {msg.role.value}\n\n{respon-
se_text}\n{'-' * 26}")

    elif isinstance(event, MagenticFinalResultEvent):
        # Final synthesized result
        print("\n" + "=" * 50)
        print("FINAL RESULT:")
        print("=" * 50)
        if event.message is not None:
            print(event.message.text)
        print("=" * 50)
```

Build the Magentic Workflow

Use `MagenticBuilder` to configure the workflow with a standard manager:

Python

```python
from agent_framework import MagenticBuilder, MagenticCallbackMode

workflow = (
    MagenticBuilder()
    .participants(researcher=researcher_agent, coder=coder_agent)
    .on_event(on_event, mode=MagenticCallbackMode.STREAMING)
    .with_standard_manager(
        chat_client=OpenAIChatClient(),
        max_round_count=10,  # Maximum collaboration rounds
        max_stall_count=3,   # Maximum rounds without progress
        max_reset_count=2,   # Maximum plan resets allowed
    )
    .build()
)
```

Run the Workflow

Execute a complex task that requires multiple agents working together:

Python

```python
from agent_framework import WorkflowCompletedEvent

task = (
    "I am preparing a report on the energy efficiency of different machine 
learning model architectures. "
    "Compare the estimated training and inference energy consumption of 
ResNet-50, BERT-base, and GPT-2 "
    "on standard datasets (e.g., ImageNet for ResNet, GLUE for BERT, 
WebText for GPT-2). "
    "Then, estimate the CO2 emissions associated with each, assuming train-
ing on an Azure Standard_NC6s_v3 "
    "VM for 24 hours. Provide tables for clarity, and recommend the most 
energy-efficient model "
    "per task type (image classification, text classification, and text 
generation)."
)

completion_event = None
async for event in workflow.run_stream(task):
    if isinstance(event, WorkflowCompletedEvent):
        completion_event = event

if completion_event is not None:
    data = getattr(completion_event, "data", None)
    preview = getattr(data, "text", None) or (str(data) if data is not 
None else "")
    print(f"Workflow completed with result:\n\n{preview}")
```

Advanced: Human-in-the-Loop Plan Review

Enable human review and approval of the manager's plan before execution:

Configure Plan Review

Python

```python
from agent_framework import (
    MagenticPlanReviewDecision,
    MagenticPlanReviewReply,
    MagenticPlanReviewRequest,
    RequestInfoEvent,
)
```

```python
workflow = (
    MagenticBuilder()
    .participants(researcher=researcher_agent, coder=coder_agent)
    .on_event(on_event, mode=MagenticCallbackMode.STREAMING)
    .with_standard_manager(
        chat_client=OpenAIChatClient(),
        max_round_count=10,
        max_stall_count=3,
        max_reset_count=2,
    )
    .with_plan_review()  # Enable plan review
    .build()
)
```

Handle Plan Review Requests

Python

```python
completion_event: WorkflowCompletedEvent | None = None
pending_request: RequestInfoEvent | None = None

while True:
    # Run until completion or review request
    if pending_request is None:
        async for event in workflow.run_stream(task):
            if isinstance(event, WorkflowCompletedEvent):
                completion_event = event

            if isinstance(event, RequestInfoEvent) and event.request_type
is MagenticPlanReviewRequest:
                pending_request = event
                review_req = cast(MagenticPlanReviewRequest, event.data)
                if review_req.plan_text:
                    print(f"\n=== PLAN REVIEW REQUEST ===\n{review_req.-
plan_text}\n")

    # Check if completed
    if completion_event is not None:
        break

    # Respond to plan review
    if pending_request is not None:
        # Collect human decision (approve/reject/modify)
        # For demo, we auto-approve:
        reply =
MagenticPlanReviewReply(decision=MagenticPlanReviewDecision.APPROVE)

        # Or modify the plan:
        # reply = MagenticPlanReviewReply(
        #     decision=MagenticPlanReviewDecision.APPROVE,
        #     edited_plan="Modified plan text here..."
        # )
        async for event in workflow.send_responses_streaming({pending_re-
quest.request_id: reply}):
            if isinstance(event, WorkflowCompletedEvent):
                completion_event = event
            elif isinstance(event, RequestInfoEvent):
                # Another review cycle if needed
                pending_request = event
            else:
                pending_request = None
```

Key Concepts

- **Dynamic Coordination**: The Magentic manager dynamically selects which agent should act next based on the evolving context
- **Iterative Refinement**: The system can break down complex problems and iteratively refine solutions through multiple rounds
- **Progress Tracking**: Built-in mechanisms to detect stalls and reset the plan if needed
- **Flexible Collaboration**: Agents can be called multiple times in any order as determined by the manager
- **Human Oversight**: Optional human-in-the-loop review allows manual intervention and plan modification

Workflow Execution Flow

The Magentic orchestration follows this execution pattern:

1. **Planning Phase**: The manager analyzes the task and creates an initial plan
2. **Agent Selection**: The manager selects the most appropriate agent for each subtask
3. **Execution**: The selected agent executes their portion of the task
4. **Progress Assessment**: The manager evaluates progress and updates the plan
5. **Iteration**: Steps 2-4 repeat until the task is complete or limits are reached
6. **Final Synthesis**: The manager synthesizes all agent outputs into a final result

Error Handling

Add error handling to make your workflow robust:

Python

```python
def on_exception(exception: Exception) -> None:
    print(f"Exception occurred: {exception}")
    logger.exception("Workflow exception", exc_info=exception)
```

```python
workflow = (
    MagenticBuilder()
    .participants(researcher=researcher_agent, coder=coder_agent)
    .on_exception(on_exception)
    .on_event(on_event, mode=MagenticCallbackMode.STREAMING)
    .with_standard_manager(
        chat_client=OpenAIChatClient(),
        max_round_count=10,
        max_stall_count=3,
        max_reset_count=2,
    )
    .build()
)
```

Complete Example

Here's a full example that brings together all the concepts:

Python

```python
import asyncio
import logging
from typing import cast

from agent_framework import (
    ChatAgent,
    HostedCodeInterpreterTool,
    MagenticAgentDeltaEvent,
    MagenticAgentMessageEvent,
    MagenticBuilder,
    MagenticCallbackEvent,
    MagenticCallbackMode,
    MagenticFinalResultEvent,
    MagenticOrchestratorMessageEvent,
    WorkflowCompletedEvent,
)
from agent_framework.openai import OpenAIChatClient, OpenAIResponsesClient

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

async def main() -> None:
    # Define specialized agents
    researcher_agent = ChatAgent(
        name="ResearcherAgent",
        description="Specialist in research and information gathering",
        instructions=(
            "You are a Researcher. You find information without additional
"
            "computation or quantitative analysis."
        ),
        chat_client=OpenAIChatClient(ai_model_id="gpt-4o-search-preview"),
    )

    coder_agent = ChatAgent(
        name="CoderAgent",
        description="A helpful assistant that writes and executes code to
process and analyze data.",
        instructions="You solve questions using code. Please provide de-
tailed analysis and computation process.",
        chat_client=OpenAIResponsesClient(),
        tools=HostedCodeInterpreterTool(),
    )

    # State for streaming callback
    last_stream_agent_id: str | None = None
    stream_line_open: bool = False

    # Unified callback for all events
    async def on_event(event: MagenticCallbackEvent) -> None:
        nonlocal last_stream_agent_id, stream_line_open

        if isinstance(event, MagenticOrchestratorMessageEvent):
            print(f"\n[ORCH:{event.kind}]\n\n{getattr(event.message,
'text', '')}\n{'-' * 26}")

        elif isinstance(event, MagenticAgentDeltaEvent):
            if last_stream_agent_id != event.agent_id or not stream_-
line_open:
                if stream_line_open:
                    print()
                print(f"\n[STREAM:{event.agent_id}]: ", end="", flush=True)
                last_stream_agent_id = event.agent_id
                stream_line_open = True
            print(event.text, end="", flush=True)

        elif isinstance(event, MagenticAgentMessageEvent):
            if stream_line_open:
                print(" (final)")
                stream_line_open = False
                print()
            msg = event.message
            if msg is not None:
                response_text = (msg.text or "").replace("\n", " ")
                print(f"\n[AGENT:{event.agent_id}] {msg.role.value}\n\n{re-
sponse_text}\n{'-' * 26}")

        elif isinstance(event, MagenticFinalResultEvent):
            print("\n" + "=" * 50)
            print("FINAL RESULT:")
            print("=" * 50)
            if event.message is not None:
                print(event.message.text)
            print("=" * 50)

    # Build the workflow
```

```python
    print("\nBuilding Magentic Workflow...")

    workflow = (
        MagenticBuilder()
        .participants(researcher=researcher_agent, coder=coder_agent)
        .on_event(on_event, mode=MagenticCallbackMode.STREAMING)
        .with_standard_manager(
            chat_client=OpenAIChatClient(),
            max_round_count=10,
            max_stall_count=3,
            max_reset_count=2,
        )
        .build()
    )

    # Define the task
    task = (
        "I am preparing a report on the energy efficiency of different machine learning model architectures. "
        "Compare the estimated training and inference energy consumption of ResNet-50, BERT-base, and GPT-2 "
        "on standard datasets (e.g., ImageNet for ResNet, GLUE for BERT, WebText for GPT-2). "
        "Then, estimate the CO2 emissions associated with each, assuming training on an Azure Standard_NC6s_v3 "
        "VM for 24 hours. Provide tables for clarity, and recommend the most energy-efficient model "
        "per task type (image classification, text classification, and text generation)."
    )

    print(f"\nTask: {task}")
    print("\nStarting workflow execution...")

    # Run the workflow
    try:
        completion_event = None
        async for event in workflow.run_stream(task):
            print(f"Event: {event}")

            if isinstance(event, WorkflowCompletedEvent):
                completion_event = event

        if completion_event is not None:
            data = getattr(completion_event, "data", None)
            preview = getattr(data, "text", None) or (str(data) if data is not None else "")
            print(f"Workflow completed with result:\n\n{preview}")

    except Exception as e:
        print(f"Workflow execution failed: {e}")
        logger.exception("Workflow exception", exc_info=e)

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

Configuration Options

Manager Parameters

- `max_round_count`: Maximum number of collaboration rounds (default: 10)
- `max_stall_count`: Maximum rounds without progress before reset (default: 3)
- `max_reset_count`: Maximum number of plan resets allowed (default: 2)

Callback Modes

- `MagenticCallbackMode.STREAMING`: Receive incremental token updates
- `MagenticCallbackMode.COMPLETE`: Receive only complete messages

Plan Review Decisions

- `APPROVE`: Accept the plan as-is
- `REJECT`: Reject and request a new plan
- `APPROVE` with `edited_plan`: Accept with modifications

Sample Output

Coming soon...

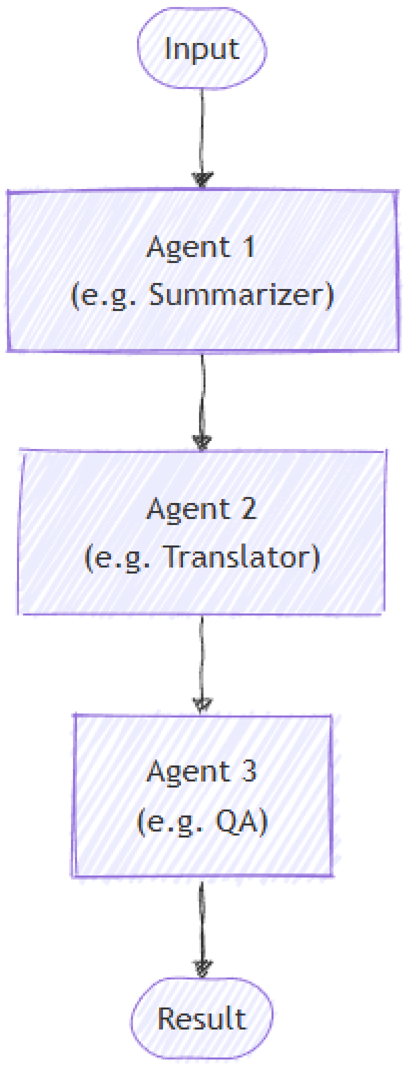
Next steps

Handoff Orchestration

Microsoft Agent Framework Workflows Orchestrations - Sequential

10/02/2025

In sequential orchestration, agents are organized in a pipeline. Each agent processes the task in turn, passing its output to the next agent in the sequence. This is ideal for workflows where each step builds upon the previous one, such as document review, data processing pipelines, or multi-stage reasoning.



What You'll Learn

- How to create a sequential pipeline of agents
- How to chain agents where each builds upon the previous output
- How to mix agents with custom executors for specialized tasks
- How to track the conversation flow through the pipeline

Define Your Agents

In sequential orchestration, each agent processes the task in turn, with output flowing from one to the next. Let's start by defining agents for a two-stage process:

Python

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

# 1) Create agents using AzureChatClient
chat_client = AzureChatClient(credential=AzureCliCredential())

writer = chat_client.create_agent(
    instructions=(
        "You are a concise copywriter. Provide a single, punchy marketing 
sentence based on the prompt."
    ),
    name="writer",
)

reviewer = chat_client.create_agent(
    instructions=(
        "You are a thoughtful reviewer. Give brief feedback on the previous 
assistant message."
    ),
    name="reviewer",
)
```

Set Up the Sequential Orchestration

The `SequentialBuilder` class creates a pipeline where agents process tasks in order. Each agent sees the full conversation history and adds their response:

Python

```python
from agent_framework import SequentialBuilder

# 2) Build sequential workflow: writer -> reviewer
workflow = SequentialBuilder().participants([writer, reviewer]).build()
```

Run the Sequential Workflow

Execute the workflow and collect the final conversation showing each agent's contribution:

Python

```
from agent_framework import ChatMessage, WorkflowCompletedEvent

# 3) Run and print final conversation
completion: WorkflowCompletedEvent | None = None
async for event in workflow.run_stream("Write a tagline for a budget-
friendly eBike."):
    if isinstance(event, WorkflowCompletedEvent):
        completion = event

if completion:
    print("===== Final Conversation =====")
    messages: list[ChatMessage] | Any = completion.data
    for i, msg in enumerate(messages, start=1):
        name = msg.author_name or ("assistant" if msg.role ==
Role.ASSISTANT else "user")
        print(f"{'-' * 60}\n{i:02d} [{name}]\n{msg.text}")
```

Sample Output

plaintext

```
===== Final Conversation =====
------------------------------------------------------------
01 [user]
Write a tagline for a budget-friendly eBike.
------------------------------------------------------------
02 [writer]
Ride farther, spend less—your affordable eBike adventure starts here.
------------------------------------------------------------
03 [reviewer]
This tagline clearly communicates affordability and the benefit of extended
travel, making it
appealing to budget-conscious consumers. It has a friendly and motivating
tone, though it could
be slightly shorter for more punch. Overall, a strong and effective
suggestion!
```

Advanced: Mixing Agents with Custom Executors

Sequential orchestration supports mixing agents with custom executors for specialized processing. This is useful when you need custom logic that doesn't require an LLM:

Define a Custom Executor

Python

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

class Summarizer(Executor):
    """Simple summarizer: consumes full conversation and appends an as-
sistant summary."""

    @handler
    async def summarize(
        self,
        conversation: list[ChatMessage],
        ctx: WorkflowContext[list[ChatMessage]]
    ) -> None:
        users = sum(1 for m in conversation if m.role == Role.USER)
        assistants = sum(1 for m in conversation if m.role ==
Role.ASSISTANT)
        summary = ChatMessage(
            role=Role.ASSISTANT,
            text=f"Summary -> users:{users} assistants:{assistants}"
        )
        await ctx.send_message(list(conversation) + [summary])
```

Build a Mixed Sequential Workflow

Python

```python
# Create a content agent
content = chat_client.create_agent(
    instructions="Produce a concise paragraph answering the user's
request.",
    name="content",
)

# Build sequential workflow: content -> summarizer
summarizer = Summarizer(id="summarizer")
workflow = SequentialBuilder().participants([content, summarizer]).build()
```

Sample Output with Custom Executor

plaintext

```
------------------------------------------------------------
01 [user]
Explain the benefits of budget eBikes for commuters.
------------------------------------------------------------
02 [content]
Budget eBikes offer commuters an affordable, eco-friendly alternative to
cars and public transport.
```

```
Their electric assistance reduces physical strain and allows riders to
cover longer distances quickly,
minimizing travel time and fatigue. Budget models are low-cost to maintain
and operate, making them accessible
for a wider range of people. Additionally, eBikes help reduce traffic con-
gestion and carbon emissions,
supporting greener urban environments. Overall, budget eBikes provide cost-
effective, efficient, and
sustainable transportation for daily commuting needs.
------------------------------------------------------------
03 [assistant]
Summary -> users:1 assistants:1
```

Key Concepts

- **Shared Context**: Each participant receives the full conversation history, including all previous messages
- **Order Matters**: Agents execute strictly in the order specified in the `participants()` list
- **Flexible Participants**: You can mix agents and custom executors in any order
- **Conversation Flow**: Each agent/executor appends to the conversation, building a complete dialogue

Next steps

Magentic Orchestration