ChatGPT

Microsoft Agent Framework Labs – Step-by-Step Guide

Setting Up the Project Solution

Before diving into the labs, ensure you have the necessary prerequisites and a proper project structure:

- **Install .NET SDK 8.0 or later** – The Agent Framework supports all supported .NET versions (we recommend .NET 8.0+) [1].
- **Azure OpenAI resource** – You'll need an Azure OpenAI resource with a deployed model (e.g. `gpt-4o-mini`) and appropriate access (Cognitive Services OpenAI User/Contributor role) [1].
- **Azure CLI** – Install Azure CLI and run `az login` to authenticate. The quickstarts below use `AzureCliCredential` for authentication, so being logged in is required [2]. *(If you prefer not to use Azure CLI, you can use an API key credential instead – this is discussed shortly.)*
- **Microsoft Agent Framework NuGet packages** – The labs will use the **Microsoft Agent Framework** libraries. The primary packages to install are: `Azure.AI.OpenAI`, `Azure.Identity`, and `Microsoft.Agents.AI.OpenAI` [3]. Some labs will require additional packages (noted in their steps).

Next, set up a single Visual Studio solution to contain all lab projects (so they build on each other but can run independently):

1. **Create a Solution:** In Visual Studio, create a new empty solution (e.g., **AgentFrameworkLabs**). Using one solution for all labs makes it easy to manage and avoids "dependency hell" by isolating each lab in its own project.
2. *CLI alternative:* Run `dotnet new sln -n AgentFrameworkLabs` to create a solution folder.
3. **Add a Console Project per Lab:** For each lab (1 through 11), add a new .NET console application project to the solution. For clarity, name them like **Lab01_SimpleAgent**, **Lab02_ImageAgent**, …, **Lab11_PersistConversation**. Each lab's code will reside in its own `Program.cs`.
4. *CLI alternative:* For example, `dotnet new console -n Lab01_SimpleAgent` then `dotnet sln add Lab01_SimpleAgent/Lab01_SimpleAgent.csproj`.
5. **Reference Agent Framework Packages:** For each project, add the NuGet packages:

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

These packages provide the Azure OpenAI client, Azure authentication, and the Agent Framework APIs [3]. (Use the `--prerelease` flag for Microsoft.Agents.AI.OpenAI if a stable version isn't available yet [3].)

6. **Configure Azure OpenAI Credentials:** Decide how your code will authenticate to Azure OpenAI:

7. **Using Azure CLI Credential (Recommended):** This uses your `az login` session. No keys are needed in code – the `AzureCliCredential` will utilize your logged-in identity [2]. Make sure you're logged in and have access to the OpenAI resource.

8. **Using an API Key:** If you prefer using an API key (or if CLI auth is problematic on your Mac), you can use an `AzureKeyCredential`. In this case, store your Azure OpenAI **endpoint URL** and **API key** in a config file or environment variables and load them in your code. For example, you might create an **appsettings.json** (and a template for it in your repo) with:

   ```
   {
     "AzureOpenAI": {
       "Endpoint": "https://<your-resource>.openai.azure.com/",
       "Deployment": "<your-model-deployment-name>",
       "ApiKey": "<your-api-key>"
     }
   }
   ```

   You can then read these in your program (using `ConfigurationBuilder` or `Environment.GetEnvironmentVariable`) and instantiate the client as:

   ```
   string endpoint = /* read from config or env */;
   string apiKey = /* read from config or env */;
   var client = new AzureOpenAIClient(new Uri(endpoint), new
   AzureKeyCredential(apiKey));
   ```

   This replaces the Azure CLI credential with a direct key. *(If using CLI credential, skip this – the code in labs will assume CLI auth by default.)* [4]

9. **Running the Labs:** You can run each lab project individually without conflict. In Visual Studio, right-click the project and choose **Set as Startup Project** (or launch via the Run dropdown) to run that lab. On the command line, you can `cd` into a project folder and run `dotnet run`. Each lab is self-contained, so you shouldn't hit dependency issues between them.

With the setup complete, let's walk through each lab in detail.

Lab 1: Create and Run a Simple Agent

Goal: Build a basic AI agent using Azure OpenAI as the large-language model backend. The agent will have simple instructions and will generate a text response to a user prompt.

Prerequisites: Ensure your environment is set up (see above). You need a valid Azure OpenAI endpoint and model deployment, and you should have installed the three packages (`Azure.AI.OpenAI`, `Azure.Identity`, `Microsoft.Agents.AI.OpenAI`) in this lab's project [3].

Steps:

1. **Create the Console Project:** If not already done, create a new console app for Lab 1 (e.g. **Lab01_SimpleAgent**) and add the required packages. In Program.cs, you'll be writing the code below.

2. **Import Namespaces:** At the top of *Program.cs*, import the necessary namespaces:

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

   These give you access to the OpenAI client, Azure credential, and Agent Framework types.

3. **Initialize the Azure OpenAI Client:** Inside `Main()`, create an Azure OpenAI chat client. You can use Azure CLI credentials as shown below, or an API key if configured:

```
var endpoint = new Uri("https://<your-resource>.openai.azure.com/");
var credential = new AzureCliCredential();
var chatClient = new AzureOpenAIClient(endpoint, credential)
                    .GetChatClient("<your-deployment-name>");
```

   Replace the placeholder URL with your Azure OpenAI endpoint, and `<your-deployment-name>` with the name of your deployed model (e.g., `gpt-4o-mini`) [5] [6]. This code creates a chat client that will communicate with the Azure OpenAI Chat Completion service.

4. **Create the AI Agent:** Use the chat client to create an AI agent with a given instruction (system prompt). For example:

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

   Here we instruct the agent to be "good at telling jokes." The agent is now ready to receive user messages. (You could also provide a `name` or omit it; naming can help identify the agent in multi-agent scenarios.) [7]

5. **Run the Agent with a Prompt:** To get a response from the agent, call its `RunAsync` method with a user prompt. For example:

```
string userPrompt = "Tell me a joke about a pirate.";
AgentRunResponse result = await agent.RunAsync(userPrompt);
Console.WriteLine(result.ToString());
```

This sends the prompt to the agent and prints out the response text [8]. The `AgentRunResponse.ToString()` (or `.Text`) will contain the model's answer [9] [10].

6. **Execute and Observe Output:** Run the Lab1 project. The agent should connect to Azure OpenAI and return a joke. For example, you might see output like:

```
Why did the pirate go to school?
Because he wanted to improve his "arrr-ticulation"!
```

(Your exact output may vary, but it should be a pirate joke given our instructions [11] [12].)

Tip: If you want to see the agent stream its answer token-by-token (instead of waiting for the full response), you can use `RunStreamingAsync`. For example:

```
await foreach (var update in agent.RunStreamingAsync(userPrompt))
{
    Console.Write(update.Text);
}
```

This will output the response as it's generated [13]. Streaming is optional but useful for real-time updates.

Lab 1 is now complete – you've built and run a simple AI agent! Next, we'll enhance the agent with additional capabilities.

Lab 2: Using Images with an Agent

Goal: Extend the agent to handle image input. This lab demonstrates how to send an image (by URL) to the agent along with a question, so the agent can analyze the image before responding.

Prerequisites: Continue using the Azure OpenAI setup from Lab 1. Ensure the `Microsoft.Agents.AI.OpenAI` package is installed (it provides classes like `TextContent` and `UriContent` for handling multi-modal input).

Steps:

1. **Create Project for Lab 2:** Add a new console project **Lab02_ImageAgent** (or reuse Lab1's code if you prefer continuity). Include the same `using` directives and Azure OpenAI client setup as in Lab 1 (client credentials, etc.).

2. **Create an Agent for Vision:** Instantiate an `AIAgent` similar to Lab 1, but give it instructions that acknowledge image analysis. For example:

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

   This agent is configured with a general ability to analyze images based on its instructions [14].

3. **Prepare an Image Query:** Create a `ChatMessage` that contains both text and an image. For example, to ask "What do you see in this image?" with an image URL:

```
var message = new ChatMessage(
    ChatRole.User,
    new ChatMessageContent[]
    {
        new TextContent("What do you see in this image?"),
        new UriContent("https://upload.wikimedia.org/..." /* image URL */,
"image/jpeg")
    });
```

   Here we use `TextContent` for the prompt text and `UriContent` for the image (JPEG in this case). Make sure the URL points to a publicly accessible image [15]. The `ChatMessage` combines both parts as a single user message.

4. **Send the Message to the Agent:** Use `await agent.RunAsync(message)` to send the multi-modal message and get a response. For example:

```
var response = await agent.RunAsync(message);
Console.WriteLine(response.Text);
```

   The agent will receive both the text and the image URL as context, allowing it to analyze the image before answering [16].

5. **Run Lab 2 and Observe:** When you run this project, the agent should output a description or analysis of the image. For instance, if the image was of a nature scene, the agent might describe the scenery or objects in it. *(The exact output depends on the model's vision capabilities and the image content.)* [17]

This lab shows how to pass images to an agent by including `UriContent` in a chat message. The agent's response will be based on both the visual content and the prompt text.

Lab 3: Multi-Turn Conversation with an Agent

Goal: Enable the agent to carry on a multi-turn conversation, remembering context from previous user queries. By default, each call to `RunAsync` is stateless, but here we'll maintain a conversation thread.

Prerequisites: Use the same basic agent setup as before. No extra packages needed beyond the core ones.

Steps:

1. **Create Project for Lab 3:** Add **Lab03_MultiTurn** console project, or continue from a previous lab. Set up the agent (with instructions of your choice) as in Lab 1. For example, an agent with instruction "You are a helpful assistant.".

2. **Obtain a Conversation Thread:** Agents do not automatically remember past messages, so you need an `AgentThread` to store the conversation state. Create a new thread from the agent:

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

   This thread will track the messages sent to and from the agent [18] [19].

3. **Have a Multi-Turn Dialog:** Call the agent multiple times with the same `thread`. For example:

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

   In the second query, because we pass the same thread, the agent remembers the joke it told and can modify it as requested instead of coming up with a new unrelated joke [19]. The conversation state (all prior messages) is stored in the `AgentThread` and sent to the model on each call, so the model receives the full context [20].

4. **Test Multi-Turn Memory:** Run the Lab 3 project. You should see that the second response builds on the first. For instance:

5. First response: *"Why did the pirate go to school? … (punchline)"*

6. Second response: *(Now the same joke with pirate slang and emojis, e.g., adding "☠ " and speaking as a parrot.)*
   This demonstrates that the agent retained context. The Agent Framework ensures the conversation history is included so long as you use the same `AgentThread` [21].

7. **Multiple Independent Conversations (Optional):** You can create multiple threads to handle separate conversations in parallel. For example:

```
AgentThread thread1 = agent.GetNewThread();
AgentThread thread2 = agent.GetNewThread();
// Start two separate conversations:
Console.WriteLine(await agent.RunAsync("Tell me a joke about a pirate.",
thread1));
Console.WriteLine(await agent.RunAsync("Tell me a joke about a robot.",
thread2));
// Continue each conversation:
Console.WriteLine(await agent.RunAsync("Add emojis and a parrot voice to
the pirate joke.", thread1));
Console.WriteLine(await
agent.RunAsync("Add emojis and a robot voice to the robot joke.",
thread2));
```

   Here, `thread1` and `thread2` maintain independent histories [22] [23]. Each conversation will only remember its own context.

Using threads for conversation state is crucial for multi-turn interactions [24] [19]. This lab showed how to implement that. Next, we'll explore giving the agent custom tools/functions to use.

Lab 4: Using Function Tools with an Agent

Goal: Extend the agent with a custom function (tool) that it can call to retrieve information. We'll define a simple function (e.g., a weather lookup) and register it as a tool for the agent. The agent can then decide to call this function when answering a relevant query.

Prerequisites: No new packages beyond the core ones. Ensure your project (e.g., **Lab04_FunctionTool**) references `System.ComponentModel` (for `[Description]` attributes) which is part of .NET base library.

Steps:

1. **Create a Custom Function:** Define a C# method that the agent can use. For example:

```
using System.ComponentModel;

[Description("Get the weather for a given location.")]
```

```
static string GetWeather([Description("The location to get the weather
for.")] string location)
{
    // In a real scenario, call a weather API. Here we fake a result:
    return $"The weather in {location} is cloudy with a high of 15°C.";
}
```

We use `[Description]` attributes to provide the agent with metadata about what the function does and its parameter [25] [26]. This helps the agent choose the right function when multiple are available.

2. **Create the Agent with the Function Tool:** When constructing the agent, pass the function as a tool. The Agent Framework provides `AIFunctionFactory.Create` to turn a C# method into an `AIFunction` object. For example:

```
AIAgent agent = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient("<deployment>")
    .CreateAIAgent(
        instructions: "You are a helpful assistant.",
        tools: new[] { AIFunctionFactory.Create(GetWeather) }
    );
```

Now the agent knows about the `GetWeather` tool and can call it if the user's query seems to require it [27] [28]. (We keep the instructions simple here, but you might remind the agent it can use tools for extra info.)

3. **Ask the Agent a Question Requiring the Tool:** For instance:

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

The agent will analyze the question, recognize that it can use the `GetWeather` function, and invoke it internally. The function returns the dummy weather string, which the agent then incorporates into its answer [29].

4. **Run Lab 4:** The output should be the weather info from our function. For example:

```
The weather in Amsterdam is cloudy with a high of 15°C.
```

The agent might wrap it in a sentence or simply output the function result, depending on its prompt and the model's behavior. The key is that it successfully called our C# function behind the scenes.

By using `AIFunctionFactory.Create` and providing the function in `tools`, we gave the AI agent a new capability [27]. Next, we'll explore how to handle cases where tool usage requires user approval (human in the loop).

Lab 5: Function Tools with Human-in-the-Loop Approvals

Goal: Some tool calls may require user approval (for safety or confirmation) before execution. In this lab, we modify our function tool so that the agent must ask for approval before using it. We'll then simulate the approval step.

Scenario: We'll reuse the `GetWeather` function from Lab 4, but now mark it as requiring approval. The agent will ask "May I use this function?" and wait for a user response.

Steps:

1. **Wrap the Function with Approval Required:** The Agent Framework provides `ApprovalRequiredAIFunction` to denote that a function needs user approval before execution. Wrap the `AIFunction` we created:

   ```
   AIFunction weatherFunc = AIFunctionFactory.Create(GetWeather);
   AIFunction approvalRequiredWeatherFunc = new
   ApprovalRequiredAIFunction(weatherFunc);
   ```

   Now `approvalRequiredWeatherFunc` is a tool that the agent will not execute immediately – it will request approval first [30] [31].

2. **Create the Agent with the Approval-Required Tool:**

   ```
   AIAgent agent = new AzureOpenAIClient(new Uri(endpoint), new
   AzureCliCredential())
       .GetChatClient("<deployment>")
       .CreateAIAgent(
           instructions: "You are a helpful assistant.",
           tools: new[] { approvalRequiredWeatherFunc }
       );
   AgentThread thread = agent.GetNewThread();
   ```

   We include the wrapped tool when creating the agent [32] [33]. We also start a new `AgentThread` since this interaction will span multiple turns (question -> approval -> answer).

3. **User Asks a Question:** For example:

```
AgentRunResponse response = await agent.RunAsync("What's the weather like
in Amsterdam?", thread);
```

Because the agent has a tool that needs approval, it will **not** directly call it. Instead, it should return a response that includes a **function approval request** [34] [35]. Essentially, the agent's answer might be along the lines of: *"I need to use a tool to get the weather. Do you approve?"* (The actual `AgentRunResponse` will contain a `FunctionApprovalRequestContent` object rather than a final answer.)

4. **Detect the Approval Request:** After the run, inspect the `AgentRunResponse` for any `FunctionApprovalRequestContent`. In code, you can filter the response messages, for example:

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

If `approvalRequests.Count > 0`, the agent is waiting for approval [34] [35]. (There may be multiple requests if several tools required approval, but in our case there's just one.)

5. **Provide Approval (Simulate User):** Normally, this is where a user would say "yes" or "no". We simulate approval by creating a special response message containing a **FunctionApprovalResponseContent**. The Agent Framework makes this easy: call `CreateResponse(true)` on the `FunctionApprovalRequestContent` to indicate approval (use `false` to deny) [36] [37]. For example:

```
var requestContent = approvalRequests.First();
Console.WriteLine($"We require approval to execute
'{requestContent.FunctionCall.Name}'.");
var approvalMessage = new ChatMessage(ChatRole.User, new[] {
requestContent.CreateResponse(true) });
var finalResponse = await agent.RunAsync(approvalMessage, thread);
Console.WriteLine(finalResponse.Text);
```

Here, we log that the agent requested to call a function (e.g., "GetWeather"), then create a user message granting approval [37] [38]. We call the agent again with this approval message (using the same thread to maintain context).

6. **Run Lab 5:** The first call will produce an approval request (which you might print or detect), and the second call (with approval) will produce the actual weather answer. For example, the sequence could be:

7. Agent (turn 1): *"I can get the weather for you, but I need your approval to use the weather tool. Do you approve?"*
8. User (approval): *"Yes"* (we simulated this)
9. Agent (turn 2): *"The weather in Amsterdam is cloudy with a high of 15°C."*
10. **Important:** Always check for `FunctionApprovalRequestContent` after each run when using approval-required functions, and keep calling the agent with approvals until no more requests remain [39].

Lab 5 demonstrated a human-in-the-loop pattern for tool usage. The agent pauses for confirmation, making it safer for potentially sensitive or privileged actions.

Lab 6: Producing Structured Output with Agents

Goal: Sometimes we want the agent's answer in a structured format (like JSON) instead of free-form text. In this lab, we configure the agent to return JSON output following a specific schema.

Steps:

1. **Define the Output Schema via a C# Class:** Decide what structured data you want. For example, define a class for personal info:

```
public class PersonInfo
{
    [JsonPropertyName("name")] public string? Name { get; set; }
    [JsonPropertyName("age")] public int? Age { get; set; }
    [JsonPropertyName("occupation")] public string? Occupation { get;
set; }
}
```

   This class represents the desired JSON structure (keys "name", "age", "occupation") [40] [41].

2. **Create a JSON Schema for the Class:** Use Agent Framework's utility to generate a schema that the LLM can follow. For example:

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

   This produces a JSON Schema (as a `JsonElement`) describing `PersonInfo` [42].

3. **Configure ChatOptions for Structured Output:** Prepare a `ChatOptions` object telling the chat client to format responses as JSON. For instance:

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

Here we set the `ResponseFormat` to a JSON format that conforms to our schema [43]. (We give the schema a name and description to help the model.)

4. **Create the Agent with Structured Output Enabled:** When building the agent, supply the `ChatOptions`. One way is using `ChatClientAgentOptions`. For example:

```
var chatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
                    .GetChatClient("<deployment>");
// Create agent with custom ChatOptions:
AIAgent agent = chatClient.CreateAIAgent(new ChatClientAgentOptions {
    Name = "HelpfulAssistant",
    Instructions = "You are a helpful assistant.",
    ChatOptions = chatOptions
});
```

This sets up an agent that by default will try to output JSON matching the `PersonInfo` schema [44] [45].

5. **Ask the Agent for Structured Data:** For example, if we want person information:

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

Because of the `ChatOptions`, the model knows it should output JSON. It might return something like:

```
{"name": "John Smith", "age": 35, "occupation": "Software Engineer"}
```

(with possible variations or additional text depending on the model).

6. **Deserialize the Agent's JSON Response:** Agent responses come as `AgentRunResponse`. You can directly deserialize to our C# class:

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

The `Deserialize<T>` helper will parse the JSON string into the `PersonInfo` object [46] [47]. Now you have structured data you can work with in code.

7. **(Optional) Streaming Consideration:** If using `RunStreamingAsync`, you'll receive multiple partial updates. You should accumulate all updates and only deserialize once you have the complete response. The framework provides an extension `ToAgentRunResponseAsync()` for this purpose [48].

Run Lab 6 with a suitable prompt. You should see that the output is parsed into the structured form. This lab is powerful for scenarios where you need machine-readable answers (JSON) rather than just plain text [49] [50].

Lab 7: Using an Agent as a Function Tool

Goal: Compose multiple agents by letting one agent use another as a tool. This enables delegation: a main agent can call a specialist agent to handle part of a task. In this lab, we'll have a primary agent that calls a secondary "weather agent" to get weather info.

Steps:

1. **Define or Reuse a Specialist Function:** We'll use the same `GetWeather` function from previous labs. Additionally, let's create a dedicated **Weather Agent** that could potentially do more complex weather-related reasoning (though in this simple case it might just use the same function).

2. **Create the Weather Specialist Agent:** Make an agent focused on weather queries. For example:

```
AIAgent weatherAgent = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient("<deployment>")
    .CreateAIAgent(
        instructions: "You answer questions about the weather.",
        name: "WeatherAgent",
        description: "An agent that provides weather information.",
        tools: new[] { AIFunctionFactory.Create(GetWeather) }
    );
```

This agent has one tool (the `GetWeather` function) and is instructed to answer weather questions [51] [52]. It could be more sophisticated (e.g., call a real API), but for our purpose it's a simple agent that knows how to use `GetWeather`.

3. **Create the Main Agent and Use the Weather Agent as a Tool:** Now, for the main agent, instead of giving it the raw `GetWeather` function, we give it the *weatherAgent* as a tool. We do this by calling `.AsAIFunction()` on the agent:

```
AIAgent mainAgent = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient("<deployment>")
    .CreateAIAgent(
        instructions: "You are a helpful assistant who responds in
French.",
        tools: new[] { weatherAgent.AsAIFunction() }
    );
```

We instruct the main agent to respond in French (to clearly see it processing the info from the weather agent in its own style) [53] . The `weatherAgent.AsAIFunction()` wraps the entire WeatherAgent as a callable tool for the main agent [54] [53] . This means when the main agent decides to use that tool, it will internally invoke the weather agent and get its result.

4. **Query the Main Agent:** Ask something that requires weather info, for example:

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

Here's what happens under the hood:

5. The main agent gets the question (in French, it might translate it internally or not, but instruction says respond in French).
6. It sees it has a tool (the weatherAgent) that can handle weather questions, so it calls that.
7. The weatherAgent (as a tool) executes (using `GetWeather`) and returns (in English, likely "The weather in Amsterdam is cloudy…").
8. The main agent takes that result and then produces a **French** answer to the user, e.g., *"Le temps à Amsterdam est nuageux avec une température maximale de 15°C."* (French translation of the weather info) [55] .
9. **Run Lab 7:** The output should be the weather information presented by the main agent. If you used an instruction to respond in French, expect a French sentence. The main agent successfully utilized another agent's capabilities as a sub-routine.

This lab highlights agent composition – you can build modular agents and have one agent call another for specialized tasks [56] [54] . It's a powerful pattern for multi-agent orchestration.

Lab 8: Exposing an Agent as an MCP Tool

Goal: Demonstrate how to expose an agent as a tool via the **Model Context Protocol (MCP)**. MCP is a protocol that allows tools (functions or even whole agents) to be called by external systems (including other agents) in a standardized way. Here we will host our agent as an MCP server so it can be invoked remotely.

Prerequisites: In addition to previous packages, install `Microsoft.Extensions.Hosting` and `ModelContextProtocol` NuGet packages (the Agent Framework provides integration with MCP) [57].

Steps:

1. **Add Required Packages:** In your **Lab08_McpAgent** project, install:

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

   These provide hosting utilities and MCP support [58].

2. **Create an Agent to Expose:** You can use any agent – let's use a simple joke-telling agent from Lab 1 (the "Joker" agent). For example:

   ```
   AIAgent agent = new AzureOpenAIClient(new Uri(endpoint), new
   AzureCliCredential())
       .GetChatClient("<deployment>")
       .CreateAIAgent(instructions: "You are good at telling jokes.", name:
   "Joker");
   ```

   This is the agent we will expose as a tool for MCP [59].

3. **Wrap Agent as an MCP Tool:** Convert the agent into a tool that an MCP server can host:

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

   We use `AsAIFunction()` again to turn the agent into a callable function, and then `McpServerTool.Create(...)` to get an MCP-compatible tool instance [60] [61]. The tool will carry the agent's name and description for identification.

4. **Configure and Start the MCP Server:** Use the Generic Host to run an MCP server that serves our tool. For example:

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

```
using ModelContextProtocol.Server;

var builder = Host.CreateDefaultBuilder();
builder.ConfigureServices(services =>
{
    services.AddMcpServer()
            .WithStdioServerTransport()   // use STDIN/STDOUT for tool
calls
            .WithTools(new[] { tool });   // register our tool
});
using IHost host = builder.Build();
await host.RunAsync();
```

This sets up an MCP server listening on standard input/output [62]. We add our `tool` to it and run the host to start serving. The program will now stay running, awaiting MCP requests.

5. **Run Lab 8 (MCP Server):** When you run this project, it won't print an immediate result like previous labs. Instead, it's hosting a service. You can test it by connecting an MCP-compatible client. For example, another agent or tool runner could call it. (In practice, to manually test, you might write a small MCP client or use the Agent Framework's client to list available tools on this MCP server.)

For now, verify that the server starts without errors – it means your agent "Joker" is exposed as a tool named "Joker" over MCP [63] [64]. Any MCP client connecting to this server (over STDIO or whichever transport you choose) can invoke the Joker agent as a function.

Notes: The above uses STDIO as the transport for simplicity (read/write console). In real scenarios, you might use other transports (e.g., named pipes, sockets) via MCP. Also, ensure only one instance runs on a given STDIO at a time to avoid conflicts.

Lab 9: Enabling Observability for Agents (OpenTelemetry)

Goal: Instrument the agent with OpenTelemetry to log tracing information for each operation. This allows you to monitor agent activity (like tool invocations, model calls) in a standardized way. We'll configure an OpenTelemetry tracer to write to console for demonstration.

Prerequisites: Install the `OpenTelemetry` and `OpenTelemetry.Exporter.Console` packages in your **Lab09_Observability** project [65].

Steps:

1. **Add OpenTelemetry Packages:**

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

This gives the OpenTelemetry APIs and a console exporter to output traces [65].

2. **Set Up a TracerProvider:** At the start of your program (before creating or running the agent), configure OpenTelemetry. For example:

```
using OpenTelemetry;
using OpenTelemetry.Trace;

using var tracerProvider = Sdk.CreateTracerProviderBuilder()
    .AddSource("agent-telemetry-source")
    .AddConsoleExporter()
    .Build();
```

This creates a tracer provider that listens to a source named `"agent-telemetry-source"` and writes traces to the console [66]. The provider remains active while in scope.

3. **Instrument the Agent with OpenTelemetry:** Create your agent as usual (e.g., the joke agent). Then, convert it to a builder and attach telemetry:

```
AIAgent baseAgent = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient("<deployment>")
    .CreateAIAgent(instructions: "You are good at telling jokes.", name:
"Joker");

AIAgent agentWithTelemetry = baseAgent.AsBuilder()
    .UseOpenTelemetry(sourceName: "agent-telemetry-source")
    .Build();
```

The `.UseOpenTelemetry(...)` call instruments the agent to emit tracing events under the source name we defined [67]. We then build a new agent instance that includes this telemetry middleware. (The source name string must match what the TracerProvider is listening to [68].)

4. **Run the Agent and View Traces:** Now use `agentWithTelemetry` as normal:

```
var reply = await agentWithTelemetry.RunAsync("Tell me a joke about a
pirate.");
Console.WriteLine(reply.Text);
```

When you run this lab, you will get the agent's answer (a pirate joke) as before, **and** you will see OpenTelemetry trace data in the console. The trace might look like:

```
Activity.TraceId: <...>
Activity.SpanId: <...>
Activity.DisplayName: invoke_agent Joker
...
gen_ai.agent.name: Joker
gen_ai.request.instructions: You are good at telling jokes.
gen_ai.usage.input_tokens: 26
gen_ai.usage.output_tokens: 29
...
```

followed by the joke text [69] [70]. These tags include operation name (`invoke_agent`), the agent's ID and name, token usage, etc., which are extremely useful for debugging and monitoring agent behavior.

5. **Interpretation:** Each call to the agent is traced as an Activity. The example above shows the agent "Joker" was invoked, and it took ~8.6 seconds (for example) and used certain tokens [71] [72]. You can extend this by adding other exporters (Jaeger, Zipkin, Azure Monitor, etc.) or integrating with a full telemetry pipeline. The console exporter is just for demo.

Lab 9 gives you visibility into the agent's internal operations, leveraging OpenTelemetry standards [73] [74]. This is valuable for performance analysis and debugging in complex applications.

Lab 10: Adding Middleware to Agents

Goal: Learn how to intercept and customize agent behavior by writing **middleware**. The Agent Framework allows injection of middleware at three levels:
- **Agent-run middleware:** Runs before/after each agent run (each prompt-response cycle).
- **Function-call middleware:** Runs around each function/tool invocation.
- **Chat client middleware:** Runs around the low-level LLM API calls (to intercept requests to the model).

We will create examples of each to see how they work.

Steps:

> **Note:** Middleware uses async delegates to wrap behavior. If the middleware should not short-circuit, it must call the inner function/agent to continue the operation (similar to ASP.NET middleware or decorator patterns) [75] [76].

Step 1: Create a Base Agent (with a simple tool to demonstrate middleware effects): For illustration, we use a trivial tool that gives current time. Define:

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

Create a base agent with this tool:

```
AIAgent baseAgent = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
    .GetChatClient("<deployment>")
    .CreateAIAgent(
        instructions: "You are an AI assistant that helps people find
information.",
        tools: new[] { AIFunctionFactory.Create(GetDateTime, name:
nameof(GetDateTime)) }
    );
```

Now `baseAgent` can respond to queries and can use the `GetDateTime` function if needed (not critical for middleware demo, but included) [77] [78].

Step 2: Create Agent-Run Middleware: This middleware will execute before and after each full agent run. For example, we can count messages in and out:

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

This function, when plugged in, will wrap the agent's `RunAsync`. It logs how many messages were in the request (usually 1 user message, but could be more in a batch) and how many messages the agent responded with (typically 1) [75] [79] [80]. It then returns the actual response unchanged.

Step 3: Attach Agent-Run Middleware: Use the agent's builder:

```
var middlewareAgent = baseAgent.AsBuilder()
    .Use(CustomAgentRunMiddleware)  // add our run middleware
    .Build();
```

This creates a new agent (`middlewareAgent`) that will execute `CustomAgentRunMiddleware` on each run [81] [82]. The original `baseAgent` remains unmodified (builders clone the agent).

Step 4: Test Agent-Run Middleware: Run a query with `middlewareAgent`, e.g.:

```
await middlewareAgent.RunAsync("What time is it?");
```

In the console, you should first see something like "Incoming message count: 1", and after the agent responds, "Outgoing message count: 1" [79] [83]. Between those logs, the agent likely called the `GetDateTime` tool to get the current time and answered with it. The logs are from our middleware.

Step 5: Create Function-Calling Middleware: Now, intercept function tool calls. For example:

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

This middleware logs the name of any function being invoked and its result [84] [85]. It calls `next(context)` to actually execute the function, then logs the result. (If you wanted, you could modify the context or result here.)

Step 6: Attach Function-Call Middleware: Similar to before:

```
middlewareAgent = baseAgent.AsBuilder()
    .Use(CustomFunctionCallingMiddleware)  // add function middleware
    .Build();
```

Now `middlewareAgent` will apply our function middleware on every tool invocation [86].

Step 7: Test Function-Call Middleware: Query the agent in a way that triggers a tool. For example, ask the time again:

```
await middlewareAgent.RunAsync("What time is it?");
```

Now the console output should include our logs: e.g., "Function Name: GetDateTime" followed by "Function Call Result: <the current timestamp>" [84] [85]. This confirms our middleware intercepted the tool call.

> **Note:** Function middleware only works for agent types that allow custom functions (e.g., `ChatClientAgent` as used here) [87]. Some agent types with fixed toolsets might not support it.

Step 8: Create Chat Client Middleware: This intercepts calls between the agent and the underlying LLM service (the `IChatClient`). For instance:

```
async Task<ChatResponse> CustomChatClientMiddleware(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    IChatClient innerClient,
    CancellationToken cancellationToken)
{
    Console.WriteLine($"LLM Request: {messages.Count()} message(s)");
    var response = await innerClient.GetResponseAsync(messages, options,
cancellationToken);
    Console.WriteLine($"LLM Response: {response.Messages.Count} message(s)");
    return response;
}
```

This logs how many messages were sent to the model and how many came back [88]. It then returns the response unmodified.

Step 9: Attach Chat Client Middleware: To use this, you need to get the `IChatClient` your agent is using and wrap it. For example:

```
var rawChatClient = new AzureOpenAIClient(new Uri(endpoint), new
AzureCliCredential())
                        .GetChatClient("<deployment>")
                        .AsIChatClient();
var instrumentedChatClient = rawChatClient.AsBuilder()
    .Use(getResponseFunc: CustomChatClientMiddleware, getStreamingResponseFunc:
null)
    .Build();
AIAgent agent = new ChatClientAgent(instrumentedChatClient, instructions: "You
are a helpful assistant.");
```

Here we took the Azure OpenAI chat client, got it as an `IChatClient`, then applied our middleware using `.Use()` and built it [89]. Finally, we create a `ChatClientAgent` with this wrapped client. Now any calls the agent makes to the OpenAI service will go through our `CustomChatClientMiddleware`. (Alternatively, the Agent Framework allows passing a `clientFactory` lambda to `CreateAIAgent` to inject middleware – either approach is fine [90] [91].)

Step 10: Test Chat Client Middleware: Run a query on this new agent. You should see logs like "LLM Request: 1 message(s)" just before the model is called, and "LLM Response: 1 message(s)" after a response is received, in addition to the agent's actual answer [88] [92].

Through these middleware examples, you've seen how to intercept different parts of the agent's operation: - **Agent-run middleware** can wrap around the entire prompt/response cycle (good for logging, input/output transformations, or short-circuiting responses) [75] [93]. - **Function-call middleware** can monitor or alter tool usage (e.g., logging as we did, or blocking certain tools dynamically) [84] [85]. - **Chat client middleware** can log or modify raw requests to the LLM (useful for e.g. censoring prompts or responses, injecting headers, etc.) [88] [94].

These can be powerful for debugging and extending agent functionality in a clean, modular way.

Lab 11: Persisting and Resuming Agent Conversations

Goal: Show how to **persist an AgentThread** (conversation state) to external storage and later **resume** the conversation. This is useful for long-running conversations or scenarios where you need to save state (e.g., to a database between user sessions).

Steps:

1. **Set Up an Agent and Thread:** Use a similar setup to Lab 3. For example, in **Lab11_PersistConversation**, create an agent and start a new conversation thread:

   ```
   AIAgent agent = new AzureOpenAIClient(new Uri(endpoint), new
   AzureCliCredential())
       .GetChatClient("<deployment>")
       .CreateAIAgent(instructions: "You are a helpful assistant.", name:
   "Assistant");
   AgentThread thread = agent.GetNewThread();
   ```

   We now have an agent and an empty conversation thread.

2. **Have a Conversation Turn:** Send a message and get a response, so the thread has some content:

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

   Suppose the agent replies with a pirate joke. The `thread` now contains that Q&A exchange [95] [96].

3. **Serialize the Conversation State:** To persist, convert the `AgentThread` to a JSON representation. The Agent Framework supports this via `thread.Serialize()` which returns a `JsonElement`. You can then serialize that to a string for storage. For example:

```
JsonElement serializedThread = thread.Serialize();
string threadJson = JsonSerializer.Serialize(serializedThread,
JsonSerializerOptions.Web);
// Save threadJson to file or database
string filePath = Path.Combine(Path.GetTempPath(), "agent_thread.json");
await File.WriteAllTextAsync(filePath, threadJson);
```

Here we serialize the thread, then write it to a file "agent_thread.json" for simplicity [97] [98]. In a real app, you might store it in a database or blob storage.

4. **Simulate Ending the Session:** (For demo, you can just end the program here and start a new one, or continue in code as if it's a new run.)

5. **Reload the Conversation State:** Later, to resume, read the saved JSON and deserialize it back into an `AgentThread`:

```
string loadedJson = await File.ReadAllTextAsync(filePath);
JsonElement reloaded = JsonSerializer.Deserialize<JsonElement>(loadedJson);
AgentThread resumedThread = agent.DeserializeThread(reloaded);
```

We read the JSON string, parse it to a `JsonElement`, then call the agent's `DeserializeThread` to get a live `AgentThread` object tied to the same agent [99] [100]. It's important to use the *same agent or an agent of the same type with identical configuration* for deserialization [101].

6. **Continue the Conversation:** Now use `resumedThread` to carry on the dialogue:

```
Console.WriteLine(await agent.RunAsync("Now tell that joke in the voice of
a parrot.", resumedThread));
```

Because `resumedThread` contains the earlier pirate joke context, the agent's new response will build on it (making the joke parrot-themed, for example) [102].

7. **Run Lab 11:** Initially, you get a joke. After serialization and resumption, you get a follow-up response that clearly uses the initial context. This proves the conversation state was successfully persisted and restored.

A few notes on persistence: - The `AgentThread.Serialize()` produces a JSON snapshot of the conversation (for Azure OpenAI chat agents, this includes all messages in the thread) [103] [97]. - The `DeserializeThread(JsonElement)` reconstructs the thread. Under the hood, it ensures the new thread is compatible with the agent's model (you cannot load a thread from a different type of agent or different instructions and expect coherence) [101]. - Always store the serialized thread securely if it contains sensitive conversation data (it might include user queries and model responses).

With Lab 11, we conclude our step-by-step walkthrough. You have built a series of .NET projects demonstrating the Microsoft Agent Framework's capabilities – from simple prompt-response agents to complex multi-agent workflows with tools, approvals, observability, and state management. Each lab builds on the concepts of the previous ones, and by structuring them in one solution with separate projects, you can easily run and experiment with each feature in isolation. Happy coding!

Sources: The above labs and code samples were developed with guidance from the *Microsoft Agent Framework* documentation and tutorials [8] [19] [33], ensuring best practices and proper usage of the Agent Framework preview (as of Oct 2025). For more details, refer to the official Microsoft Learn guide and Agent Framework repository [1] [87].

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30]
[31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60]
[61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90]
[91] [92] [93] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] agent-framework-pdfguide.pdf
file://file-Cg5Nt8gMzcHpyewAHqTdiG