Daniel Costea

Microsoft Agent Framework in .NET

Build production-ready AI agents and multi-agent systems in C#

Microsoft Agent Framework in .NET

Build production-ready AI agents

and multi-agent systems in C#

Early Access Edition

Release 2026.08.10

Free Sample Edition

This free sample includes Chapters 2 and 9

Daniel Costea

Copyright and Legal Notice

Early Access edition
Release 2026.08.10

Published independently by Daniel Costea.

Complete book and early access updates:
https://leanpub.com/agentframework

The complete source code for this book is available at:
https://github.com/dcostea/AgentFrameworkBook

Book Status and Release Notes

This is an Early Access edition of Microsoft Agent Framework in .NET.

This release includes Chapters 1–11. Chapters 12–16 are in development and will be published in future releases.

Purchasers will receive access to updated editions as new content, examples, corrections, and framework updates are published.

Current release

Release 2026.08.10

- Initial Early Access release

- Includes Chapters 1–11

- Covers Agent Framework fundamentals, Microsoft.Extensions.AI, agents, model providers, context, tools, observability, and enterprise-ready patterns

Planned content

The following chapters are planned for future releases:

- Chapter 12 — Orchestrating agents using sequential and concurrent patterns

- Chapter 13 — Iterative reasoning using the group chat pattern

- Chapter 14 — Conditional routing using the handoff pattern

- Chapter 15 — Orchestrating agents dynamically using AI skills

- Chapter 16 — Building custom orchestration using workflows

Planned chapter titles, ordering, and scope may change as Microsoft Agent Framework evolves and the book is refined.

Welcome

Thank you for purchasing Microsoft Agent Framework in .NET: Build production-ready AI agents and multi-agent systems in C#.

This book helps .NET developers bridge the gap between conventional application development and agentic AI. You'll learn how to use Microsoft Agent Framework—the unified SDK that brings together strengths from Semantic Kernel and AutoGen—to build practical, reliable AI agents and multi-agent systems in C#.

I wrote this book for intermediate and advanced C# developers who want to apply generative AI without first becoming AI specialists. Through clear explanations and hands-on examples, you'll learn to design agents that integrate models, tools, context, workflows, and enterprise concerns such as observability, reliability, and governance.

This book goes beyond simple LLM integration. It equips you with practical patterns for building reliable, scalable agentic applications that address real-world needs—whether you are developing enterprise systems, experimenting with personal projects, or creating new products.

Happy coding and AI exploration!

—Daniel Costea

.NET software engineer, AI specialist, technical author, and Microsoft MVP for AI and .NET

Table of Contents

2

Building your first agent step by step

This chapter covers

- Building a simple AI-powered console application with Agent Framework
- Obtaining and securing API keys for OpenAI and Azure OpenAI
- Creating a basic console application for prompting a GPT model
- Creating your first AI agent

Let's dive into practical applications with Agent Framework and see how to build AI-powered apps that use large language models. We'll start with a simple console app that interacts with OpenAI's GPT models, then move on to an agent that can control a robot car. You'll also learn how to obtain and secure API keys, set up your development environment, connect Agent Framework to GPT models, and expose native (C#) functions as tools.

2.1 A Robot Car Story

Imagine a robot car that can sense fire danger and escape independently. While this concept may sound like something out of science fiction, it represents a real pet project that sparked the journey behind this book. The idea of creating an autonomous vehicle capable of detecting and responding to potential fire hazards highlights the fascinating intersection of robotics, sensor technology, and machine learning. This project was built on a very low budget (less than 100 USD), with a focus on exploring how these technologies could run in the .NET ecosystem on an IoT (Internet of Things) device.

2.1.1 A Fire-Detecting Robot Car

The story began several years ago when I decided to build a robotic car as part of my academic exploration of machine learning. This project was meant to power demonstrations for upcoming speeches, not to be a commercially viable product. The primary function of the robot car is to read its surroundings and react to potential fire hazards.

I based the project on a low-cost model, using a Raspberry Pi (a low-power mini-PC with an ARM architecture) and a motor controller add-on board. The motor controller drives the wheels through a software API (Application Programming Interface). While it may not be sophisticated, this setup lets the car move forward and backward, turn left and right, and stop. To improve its awareness of its surroundings, I added a camera and sensors for distance, light, infrared, and temperature.

Next, I developed software to control the robot car and trained a machine learning model to classify inputs into two states: *Fire* and *Safe*. The Fire state indicates danger, while the Safe state signifies the absence of such a threat.

IMPORTANT You don't need a physical robot car to build an agent. The robot car is a visual aid to help illustrate concepts. Your agent will operate entirely in the digital realm.

With this machine learning model in place, the car could make predictions from sensor readings and react accordingly, showing the potential of low-cost robotics to address real-world challenges.

2.1.2 Challenges in Programming Complex Movements

Initially, I programmed basic steps supported by the robot car API into a step-by-step sequence that simulates an escape maneuver. As I assembled these steps, I immediately realized a limitation: this approach doesn't scale well to new movement patterns. Each new movement would require programming a new sequence of steps. An escape maneuver can be coded manually, but more complex patterns—following a circular path, zigzagging, imitating a dance, or avoiding obstacles—quickly lead to more hard-coded, user-defined step sequences. Figure 2.1 suggests how easy it is for a robot car to do simple moves, and how difficult it can become to do complex ones.

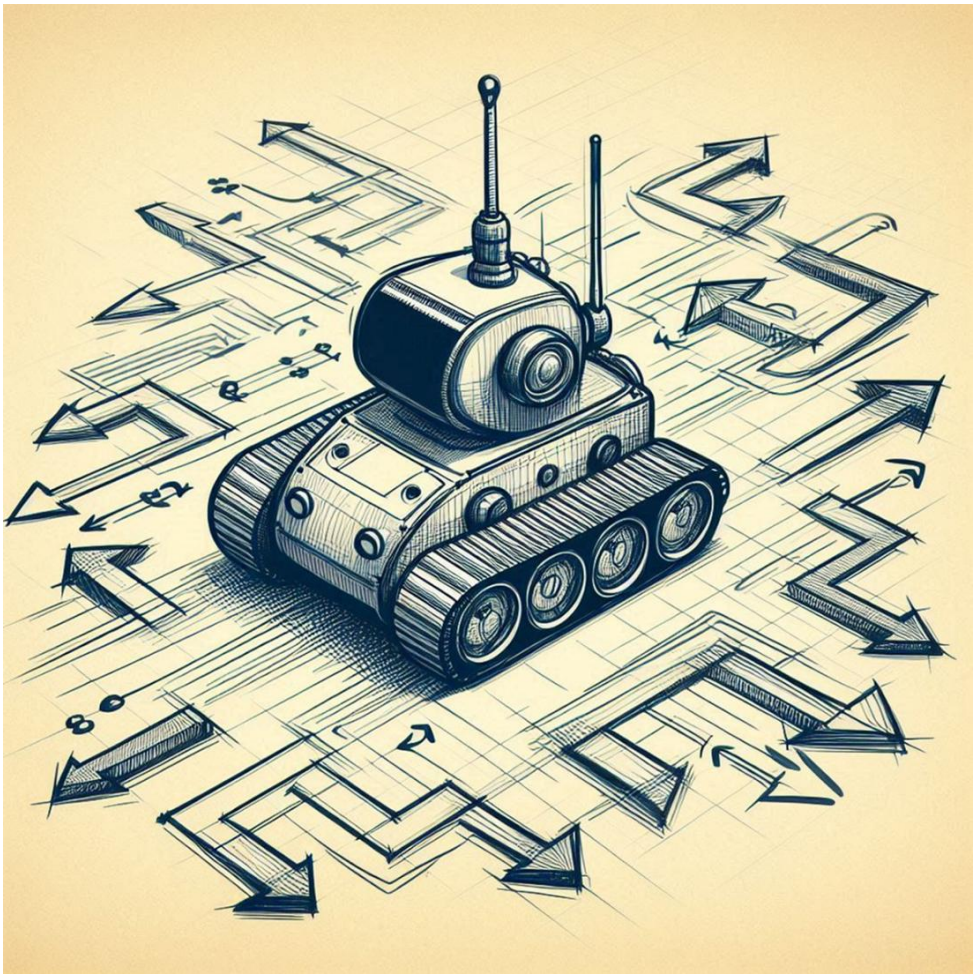

Figure 2.1 The robot car understands five basic commands: forward, backward, left turn, right turn, and stop. Any complex instruction must be broken down into a sequence of these fundamental movements, allowing the vehicle

to execute more sophisticated maneuvers by combining its core functions.

Even though I recognized a key element of this challenge, the semantic link between simple and complex commands, I lacked an elegant solution at the time and had to abandon my efforts to automate motion patterns until a better opportunity arose.

2.1.3 Revisiting the Problem with Generative AI

A few years later, the emergence of generative artificial intelligence reawakened my interest in solving the complex movement decomposition problem. Through experimentation with ChatGPT, I discovered LLMs' remarkable ability to translate complex, and thus more abstract, commands into executable sequences, a capability rooted in their "functional nature." This inspired me to use generative AI as an intermediary layer between the user and the robot's API to create motion models on the fly.

EXPERIMENTATION WITH CHATGPT

Eager to explore this possibility, I experimented with ChatGPT to break down complex movements into basic commands that the robot car API could understand. Here are the findings, which you can also verify using ChatGPT or Copilot.

User prompt:

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn left, turn right, and stop. You have to break down the complex command "There is a tree directly in front of the car. Avoid it and then return to the original path." into basic moves you know.

The prompt is fluent and concise, written in the style we're used to with ChatGPT, so the answer should be coherent and to the point.

ChatGPT response:

Let's break down the complex command into a series of basic moves:

1. **Stop**: If the car is moving, first make it stop.
2. **Turn Left**: Turn the car left to start moving around the tree.
3. **Move Forward**: Move forward a bit to clear the tree.
4. **Turn Right**: Turn the car right to straighten it out.
5. **Move Forward**: Continue moving forward to clear the tree completely.
6. **Turn Right**: Turn the car right again to start moving back towards the original path.
7. **Move Forward**: Move forward a bit to align the car with the original path.
8. **Turn Left**: Turn the car left to return to the original path.
9. **Move Forward**: Continue moving forward along the original path.

This way, the car can avoid the tree and get back on its original path!

If you have any specific parameters like distances or turning angles, I can refine the steps further.

The result was good enough. Much of the conditional logic that had instructed the robot car to move in complex patterns was now delegated to the AI.

Let's run it again!

ChatGPT response:

Absolutely. Let's break it down:

1. **Turn right** to start avoiding the tree.
2. **Move forward** for a short distance to bypass the tree.
3. **Turn left** to align parallel to the original path.
4. **Move forward** to pass the tree.
5. **Turn left** again to face the original path.
6. **Move forward** to return to the original path.
7. **Turn right** to get back on the original path.
8. **Move forward** to continue forward.
9. **Stop** to stop the car.

Feel free to ask if you need further assistance or have other tasks for the robot car!

Whether the model decides to start by turning right or left, the maneuver is pretty much the same. This shows how an LLM can introduce random variation in its responses.

Sometimes, it's much easier to grasp these steps when you can visualize them, so I created a simple diagram. See figure 2.2 for a clearer picture of the robot car's journey around the tree.

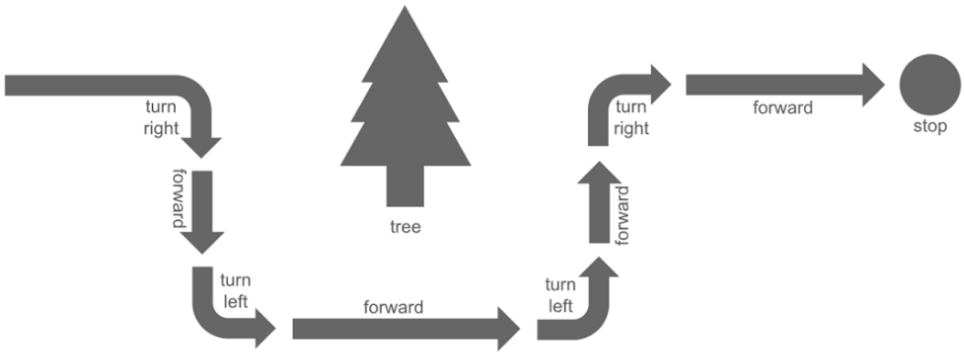

Figure 2.2 The LLM generates a textual response that details the car robot's movements around the tree, but visualizing this sequence as a diagram makes the detour easier to understand.

These steps clearly describe a path around the tree to avoid an imminent collision.

THE NON-DETERMINISTIC NATURE OF LLMS

The result from ChatGPT was promising, delegating much of the conditional logic for complex movement patterns to AI. Resending the same prompt reveals a well-known limitation: LLMs are non-deterministic and can produce varied outputs for identical inputs. While this flexibility is valuable for creative applications, machine-to-machine communication needs more predictable responses.

This functional decomposition stems from key architectural features of LLMs:

- *Task decomposition through pattern recognition*: LLMs can semantically interpret instructions such as "avoid the tree" and map them to sequences of actions. The model captures the intent behind the command and produces step-by-step outputs grounded in learned semantic patterns. This creates a key link between complex commands and basic moves.
- *Contextual Understanding Through Attention*: Although LLMs are inherently stateless, transformer-based attention mechanisms let them approximate conversational memory. By attending to earlier tokens within the context window, they can adapt later reasoning or actions based on prior information.
- *Structured Output with Prompt Engineering*: Through carefully designed prompts, LLMs can be guided to produce structured, machine-readable outputs such as JSON or XML. Methods like the *PACT framework* help enforce schema alignment and output consistency.

This example shows functional translation from natural language into structured data, helping integration with robotic control systems. With this paradigm shift, large language models are no longer simple text generators; they become functional components that link human instructions to system execution.

Despite discovering these LLM capabilities, I still wasn't convinced to return to my old problem of decomposing complex motions into basic moves. There were no .NET libraries available yet, and working directly with the OpenAI API wasn't appealing.

2.1.4 The Spark that Ignites My Robot Car

A few weeks later, during a discussion with a colleague who shared my passion for .NET, I was asked whether I had started using Microsoft Semantic Kernel for C#. I had seen a few posts about this new tool, but that conversation sparked a deeper exploration. What I discovered transformed my approach to the robot machine project: Semantic Kernel provided the missing link between the semantic understanding of generative AI and the deterministic execution of the system, which is what I needed to solve the motion decomposition problem.

Since Semantic Kernel joined forces with AutoGen to create the newer, more powerful Agent Framework, we'll use Agent Framework for all subsequent examples.

2.2 Acquiring the API Keys

Let's look at how to transform the original ChatGPT prompt, through a series of incremental steps, into an agent capable of communicating semantically with the robot-car API commands. It's worth noting that working with advanced GPT models is not as straightforward as you might assume. These powerful language models aren't freely accessible; instead, they're hosted as services provided by OpenAI or Azure OpenAI. Access typically comes with usage-based and operational costs, which can vary by usage and by the service plan you choose.

2.2.1 How to Get an OpenAI API Key

OpenAI is a pioneer in generative AI, known for its GPT family of language models. Many of the innovations we've seen came first from OpenAI. To use OpenAI's capabilities, you need an API key, which lets you integrate AI features into your applications. Next, we'll walk through how to get an OpenAI API key.

2.2.2 How to Get an Azure OpenAI API Key

Azure OpenAI is Microsoft's implementation of OpenAI's powerful language models and AI technologies, integrated into the Azure cloud platform. Obtaining an Azure OpenAI API key may take longer than obtaining the OpenAI API key described in the previous step because it involves two steps: creating an Azure account and requesting access to the Azure OpenAI service. For details on creating an Azure account and requesting Azure OpenAI service access, see appendix A, section *A.2 Creating an Azure OpenAI account and obtaining an API key.*

2.2.3 Securing API Keys

In .NET, you can store API keys securely using several best practices and options:

ENVIRONMENT VARIABLES

1. Store API keys as environment variables on the server or development machine.

   For Windows:

```
setx OPENAI_API_KEY "YOUR_OPENAI_API_KEY"
```

   For Linux:

```
export OPENAI_API_KEY="YOUR_OPENAI_API_KEY"
```

2. Access environment variables in code by reading the corresponding OpenAI key.

```
var apiKey = Environment.GetEnvironmentVariable("OPENAI_API_KEY");
```

The API key is retrieved from environment variables.

SECRET MANAGER

1. Use the Secret Manager tool provided by .NET for development only. This tool helps store secrets outside the application's folders, preventing accidental commits to Git or other repositories. Secret Manager stores secrets unencrypted in a JSON file within the user's profile folder, making it suitable for local development but not for production, so never use it in production.
2. Store secrets locally using the `dotnet` tool.

```
dotnet user-secrets init
dotnet user-secrets set "OpenAI:ApiKey" "api-key"
```

   `init` creates the secret file, `set` sets the values.

3. Build and access the secrets through the `IConfiguration` interface.

```
var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>()
  .Build();
var apiKey = configuration["OpenAI:ApiKey"];
```

The API key is programmatically retrieved from the local secrets store.

More techniques to deal with API keys

We've covered several methods for handling API keys, but production environments often call for more advanced techniques, including:

Azure Key Vault: A cloud service for securely storing and accessing secrets, including API keys.

AWS Secrets Manager: A service that helps protect access to your applications, services, and IT resources without needing to hardcode sensitive information.

Database Storage: Store encrypted API keys in a secure database and access them only when needed.

Containerized secrets management: Tools such as Docker Secrets or Kubernetes Secrets for managing sensitive data in containerized environments.

Advanced API key management techniques are critical in production environments because they enhance security through encryption and access control, improve scalability through centralized management, ensure compliance with audit trails, and integrate seamlessly with DevOps practices. These methods significantly reduce security risks in large-scale, complex applications.

While these solutions offer robust security for API key management in production settings, a detailed discussion of how to implement them is beyond the scope of this book. For most applications, the methods discussed earlier should provide sufficient security when implemented properly.

2.3 Your First AI Agent

Ready to return to the problem of decomposing complex movements into basic movements that the robot car API can understand? Agent Framework makes it easy to integrate into a .NET console application. We'll start with a simple example and gradually enhance it to create a dynamic, interactive AI assistant for a robot car. You'll have an agent that can break down complex movements into basic commands, and you'll be eager to expand its capabilities even further.

Open Visual Studio Code to create a new console application and add the necessary NuGet packages. If you prefer a more full-featured IDE, you can use Visual Studio instead. Open a terminal and run the commands shown in listing 2.1. Note that the package versions aren't specified because the package names are expected to remain consistent throughout future releases, ensuring compatibility with newer versions.

Listing 2.1 Creating a new console application – Terminal

```
dotnet new console -n RobotCarAssistant  ❶
cd RobotCarAssistant  ❷
dotnet add package Microsoft.Agents.AI  ❸
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

❶ Creates a new console application
❷ Navigates to the newly created folder and get ready to add some packages
❸ Adds Agent Framework library dependency

This approach ensures your project is set up correctly, with the required dependencies for working with Agent Framework and securely managing secrets during development.

2.3.1 Crafting a Good Prompt

I know you're eager to send your first prompt to an LLM from your C# application, but I encourage you to have a bit of patience. When working with generative AI, nothing is more important than crafting a well-designed prompt. Before jumping into implementation, it's beneficial to experiment with tools like ChatGPT, Copilot, or any GPT-powered sandbox to refine your prompts. As shown in section 2.1.3 with the robot car AI assistant prompt, an unrefined prompt can lead to inconsistent or suboptimal results. Structuring and improving prompts are essential to guide the model toward accurate, reliable responses.

Structuring a prompt into distinct parts enhances a model's ability to understand and execute tasks by breaking the prompt into specific components, making it easier to interpret and generate effective responses. It also helps separate the original input into key elements such as *user messages*, *system messages*, and *chat message history* for maintaining context. Message types and chat history are covered in more detail in later chapters.

PACT (Persona-Action-Context-Template) Prompt Engineering Framework

To create effective prompts, I use the PACT framework, my adaptation of the Role-Task-Format (RTF) framework. PACT is for Agent Framework because it aligns with its modular, structured approach. The framework consists of four components:

Persona (Who): Defines the role or identity the model should assume when generating a response. For example, you might ask the model to act as a teacher, a journalist, or an AI assistant.

Action (What): Specify the task or action you want the model to perform. Clear instructions guide its behavior (for example, "write an article" or "break down a command").

Context (With What): Provides relevant background information or details that help the model generate accurate, contextually appropriate responses.

Template (How): Defines the desired structure or format of the response, such as JSON arrays, tables, or plain text.

This structured approach makes PACT easy to use and highly effective for crafting precise, reliable prompts. For more complex scenarios, PACT can be enriched with few-shot learning and zero-shot chain-of-thought techniques.

Let's revisit the first unstructured prompt and its generated response, then analyze its structure. By focusing on the text's semantics, we can identify three main components:

1. *The Persona* (or role or actor): "You are an AI assistant controlling a robot car...", highlighted in light gray.
2. *The Action* (or task): "You have to break down complex commands..."—that is, the unhighlighted text.
3. *Context*: "There is a tree directly in front of the car...", highlighted in dark gray.

Let's analyze the user prompt:

User prompt:

You are an AI assistant controlling a robot car that can perform basic moves: forward, backward, turn left, turn right, and stop. Break down the complex command "There is a tree directly in front of the car. Avoid it and then return to the original path." into the basic moves you know.

While this prompt provides all the necessary information in a single block of text, it lacks a clear structure. By applying *PACT* principles (Persona, Action, Context, and Template) and separating key elements into distinct prompt parts, we can improve clarity and guide the model more effectively.

Our analysis of the original prompt identified the Persona, Action, and Context components of PACT, but it was missing a crucial element: the Template. The Template part of PACT specifies the desired output format, which helps guide the model to produce a structured, consistent response.

We can improve that by adding something like "Use a JSON array for the response" to the prompt to force structured output. This provides an immediate benefit: asking for a JSON response promotes more deterministic output. JSON's strict structure guides the model to follow specific syntax and key-value arrangements, reducing variability and leading to a more deterministic response.

The user prompt looks like this:

User prompt:

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn left, turn right, and stop.

You have to break down the provided complex commands into the basic moves you know.

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path.

Use a JSON array for the response.

Requesting the response in JSON format makes it easier for Agent Framework to parse the output into a native type. This is key to seamlessly integrating AI-generated content into your application's workflow.

For learning purposes, we can keep the labels for the prompt parts (Persona, Action, Context, Template) in the prompt. Note, though, that explicitly labeling these components in the prompt isn't necessary.

Let's look at the refined user prompt in the PACT framework, with the prompt-part labels:
User prompt:

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path."

Template

Use a JSON array for the response.

Let's look at the generated response:

ChatGPT response:

Here's a breakdown of the complex command into basic moves for the robot car:

```json
```

```
[
 {"move": "turn left"},
 {"move": "forward"},
 {"move": "turn right"},
 {"move": "forward"},
 {"move": "turn right"},
 {"move": "forward"},
 {"move": "turn left"}
]
```

These steps should help the robot car avoid the tree and return to its original path.

If we run the prompt again, we'll get similar responses. Try running it a few times to see how similar they are.

ChatGPT response:

We can break the complex command down into basic moves:

```json
[
 {"action": "stop"},
 {"action": "turn right"},
 {"action": "forward"},
 {"action": "turn left"},
 {"action": "forward"},
 {"action": "turn left"},
 {"action": "forward"},
 {"action": "turn right"}
]
```

I've broken down the complex command into basic moves for the robot car. The car will stop, turn right to avoid the tree, move forward to pass it, then turn left and continue forward in the original direction.

We're making substantial progress. The JSON part of the response is what we want, but most of the time it's wrapped in explanatory text. Let's clean that up by adding a new rule: "Respond only with

the moves, without any additional explanations.” This will instruct the model to return only the JSON output.

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree in front of the car. Avoid it, then return to the original path."

Template

Use a JSON array for the response.

Respond only with the moves, without any additional explanations.

Response:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

It looks much better: the JSON content is now isolated. But let’s not celebrate yet! Run it a few more times (as I advised) to see how stable the response is:

ChatGPT response:

```
[
 "turn right",
 "forward",
 "turn left",
 "forward",
 "turn left",
 "forward",
 "turn right"
]
```

Almost there! The result is now cleanly formatted as a JSON array, without unnecessary explanatory text. During testing across multiple runs, some variation in output formatting may still occur. Sometimes, items in the JSON array are key-value pairs, and even the key names can vary. Other times, the JSON response is formatted on a single line or split across multiple lines.

TIP Include explicit instructions in the prompt, such as "`Respond only with JSON with properties: name, age, profession`", or "`Respond only with JSON arrays like [item1, item2, item3]`", or like `[key1:value1, key2:value2, key3:value3]`.

Let's fix this by providing a formal schema: "`Use a JSON array like [move1, move2, move3] for the response.`"

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path."

Template

Use a JSON array like [move1, move2, move3] for the response.

Respond only with the moves, without any additional explanations.

Let's see what the response is now:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

I ran it a few times, and it's obvious that the response becomes more stable and adheres nicely to the desired JSON schema.

While I recommend the PACT framework for its clarity, modularity, and Agent Framework readiness, other prompt engineering frameworks, such as *RTF* (Role, Task, Format), can also be effective, depending on the task and context.

2.3.2 Structuring Prompts for Reusability and Scalability

The core of Agent Framework is the agent. The agent's `Instructions` property defines the agent's persona, a foundational element that remains constant and helps characterize the agent. This property

acts as the system message. Breaking down the prompt into semantic parts (a method introduced earlier with the PACT prompting framework) is especially beneficial here.

Let's decompose the prompt into smaller semantic parts, as shown in figure 2.3.

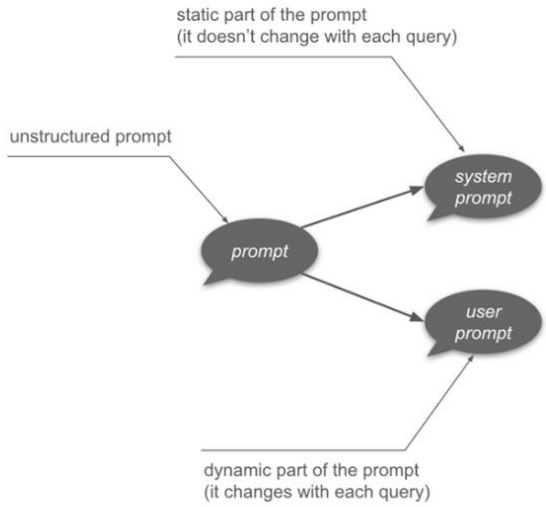

Figure 2.3 A prompt split into user prompt (the dynamic part that changes with each new interaction with an AI model) and system prompt (the static part that doesn't change during interaction with an AI model)

On one hand, the static parts—such as Persona, Actions, and Output Template—aren't expected to change when you send multiple queries to the model. On the other hand, Context is the dynamic part of the prompt and is expected to change with each prompt. The static (or general, if you prefer) parts go into the system prompt (a.k.a. instructions in the agents' world), and the dynamic part goes into the user prompt.

NOTE To help distinguish the components of a prompt, such as the user prompt, system prompt, and instructions, I use the term *prompt* for the dynamic portion (the user prompt). The static portion (the system prompt) corresponds to *instructions*, and I'll keep this convention throughout the book.

Now that we've split the well-structured prompt into a prompt and instructions, we're ready to use them in our first Agent Framework application.

2.3.3 Building a Stateless Agent with Agent Framework

When creating a new application with Agent Framework, the first step is to choose an AI provider and create a chat client. The provider might be OpenAI, Azure OpenAI, Ollama, or others. Most providers require a private API key for authentication. It's crucial to understand that these keys are often tied

to your account's billing system. If someone intercepts your key, they could misuse it and run up charges against your account. You don't want that.

There are several ways to securely manage sensitive information like API keys. For development purposes, I recommend using the .NET Secret Manager tool because it's straightforward to implement and maintain. We'll use it in examples in this chapter and the next. Although Secret Manager is convenient during development, it isn't suitable for production environments because of its limited security features.

In production, use more secure options such as environment variables, Azure Key Vault, or AWS Secrets Manager. These tools manage sensitive information in live environments and help protect it from unauthorized access. If you want to use these methods during development, adapt the examples as needed. The right choice depends on your project requirements and deployment environment.

IMPORTANT Never underestimate the importance of protecting API keys. Strong security measures help prevent accidental exposure. Common risks include editing solution files, committing keys to Git, and showing them during screen sharing or code demos. Exposed API keys can lead to unauthorized access, financial losses, and compromised data. Use secure methods such as environment variables or secret-management tools to handle API keys.

Now that we have the necessary setup, agent instructions, and prompt, let's start coding our console application to demonstrate how to use the OpenAI API to break complex movements into basic commands the robot car API can understand.

Step-by-step explanation:

1. Namespaces: Import the `Microsoft.Extensions.Configuration` namespace to manage user secrets, `Microsoft.Agents.AI` to work with agents, and `OpenAI` to initialize the OpenAI services provider.
2. Configuration setup: Uses `ConfigurationBuilder` to load user secrets, a secure way to store sensitive information such as API keys.
3. OpenAI Client Initialization: An `OpenAIClient` instance is created using the model ID and API key stored in user secrets. This chat client will handle the agent's internal communication with OpenAI services.
4. Agent initialization: Creates an AI agent using the previously created chat client with the provided instructions.
5. Prompt (User Prompt) Definition: The prompt is declared using a raw string literal. This feature is particularly useful because it allows more readable and maintainable multiline strings without escape characters, especially double quotes. The prompt string defines the task, or the goal we expect the agent to complete; more specifically, it instructs the AI model to control a robot car using basic moves.
6. Prompting the API. The `RunAsync` method is called with the prompt argument, sending a request to OpenAI for a chat completion.
7. Output: The `AgentResponse` text response from the AI model is printed to the console.

Open the `Program.cs` file and replace its contents with the following code (listing 2.2):

Listing 2.2 Agent sends a prompt to an AI model

```
using Microsoft.Extensions.Configuration;  //❶
```

```
using Microsoft.Agents.AI;  //❷
using OpenAI;  //❷

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>()
  .Build();  //❸

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❹

AIAgent agent = chatClient.AsAIAgent("""
  ## Persona
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.

  ## Action
  You have to break down the provided complex commands into the basic
  moves you know.

  ## Template
  Use a JSON array like [move1, move2, move3] for the response.
  Respond only with the moves, without any additional explanations.
  """;  //❺

var prompt = """
  ## Context
  Complex command:
  "There is a tree directly in front of the car. Avoid it and then
  return to the original path."
  """;  //❻

AgentResponse response = await agent.RunAsync(prompt);
Console.WriteLine(response.Text);  //❼
```

❶ Imports namespace for reading the user secrets
❷ Imports namespaces for working with OpenAI provider
❸ Reads the secrets to configuration object
❹ Creates the chat client
❺ Creates the AI agent using the chat client and the instructions
❻ Declares the prompt
❼ Prints the text response to the console

Let's run the code. Open the Terminal and run the following command.

```
dotnet run
```

Are you getting a response like this?

Assistant:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

If your response is similar, everything is fine because you just replicated what ChatGPT does when a user sends a prompt to the model, except this time you did it through the OpenAI API with an AI agent running in a console application.

I encourage you to try a few prompt variations to see how large language models "understand" the intent behind a user prompt.

Try these queries (one at a time!):

- a straightforward command for basic movements:

```
"Go like: turning left, forward, turning right, backward, stop"
```

- This relies on semantic understanding of geometric paths:

```
"Go on a semi-circle"
```

- This relies, again, on semantic understanding of geometric paths:

```
"Go on a square path"
```

- The word "randomly" introduces variability in the output:

```
"Go 10 steps where each step is a randomly selected step"
```

- a longer command combining multiple steps into a sequence:

```
"Do a full circle by turning left, followed by a full circle by turning
right"
```

- This challenges the LLM to understand the initial position and navigate back to it:

```
"Move forward, turn left, forward, and return to the same place where it
started"
```

- Let's get creative:

```
"Do the moonwalk dancing"
```

- Let's look at some stops and moves that mimic jellyfish movement:

```
"Move like a jellyfish"
```

2.3.4 Troubleshooting the application

There are relatively few things that can go wrong with the previous example. If problems arise, they're most likely related to API key management or endpoint configuration. Double-check these elements before running your application to minimize problems. Mishandling API keys or endpoints can result in errors such as `ClientResultException`. Error descriptions are usually self-explanatory and help you solve the problem (table 2.1):

Table 2.1 Troubleshooting the Application

ClientResultException	Description
`HTTP 401 (invalid_request_error: invalid_api_key)`	Access denied due to invalid subscription key or wrong API endpoint. Make sure to provide a valid key for an active subscription and use a correct regional API endpoint for your resource.
`HTTP 404 (invalid_request_error: model_not_found)`	The model `…` does not exist or you do not have access to it
`HTTP 429 (insufficient_quota: insufficient_quota)`	You exceeded your current quota, please check your plan and billing details. For more information on this error, read the docs: https://platform.openai.com/docs/guides/

Hopefully, everything worked as expected! If so, congratulations on completing this example of a generative AI application using Agent Framework. Whether this is your first experience with Agent

Framework or you're already familiar with it, this marks an important first step toward building more advanced AI-driven applications.

2.3.5 Building a stateful agent with memory

The code sends a prompt to the model and then prints the result. This behavior is stateless and follows a request-response pattern, which works well when we need the model to respond to a single request. Simply put, it doesn't remember anything from previous messages. What if we need to simulate an agent that remembers?

Let's turn our sample into an interactive agent that recalls conversation history (see listing 2.3) by adding the `AgentSession` object, which stores the message history.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Listing 2.3 Agent sends a prompt to an AI model using AgentSession

```
using Microsoft.Extensions.Configuration;  //❶
using Microsoft.Agents.AI;  //❷
using OpenAI;  //❷

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❸

AIAgent agent = chatClient.AsAIAgent("""
  ## Persona
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.

  ## Action
  You have to break down the provided complex commands into the basic
  moves you know.

  ## Template
  Use a JSON array like [move1, move2, move3] for the response.
  Respond only with the moves, without any additional explanations.
  """);  //❹

AgentSession session = await agent.CreateSessionAsync();  //❺

while (true)
{
  Console.Write("User: ");
  var input = Console.ReadLine();  //❻
  if (string.IsNullOrEmpty(input)) break;  //❼

  var prompt = $"""
    ## Context
    Complex command:
    "{input}"
    """;  //❽

  var response = await agent.RunAsync(prompt, session);  //❾
  Console.WriteLine($"Assistant: {response.Text}");  //❿
}
```

```
Console.WriteLine("\nHistory:");

if (session is ChatClientAgentSession
  { MessageStore: not null } chatSession)  //⓫
{
  IEnumerable<ChatMessage> history = await chatSession
    .MessageStore.GetMessagesAsync();  //⓬

  foreach (ChatMessage message in history)  //⓭
  {
    Console.WriteLine($"{message.Role}: {message.Text}");
  }
}
```

❶ Imports namespace for reading the user secrets
❷ Imports namespaces for working with Agent Framework agents OpenAI provider
❸ Reads the secrets to configuration object
❹ Creates the chat client
❺ Creates the AI agent using the chat client and the instructions
❻ Reads the input from user
❼ If the user inputs empty string, the loop stops
❽ Wraps the input into a prompt
❾ Agent queries the AI service with the session for a response
❿ Prints the text response to the console
⓫ Checks if session is chat client agent session with message store
⓬ Reads the messages from the chat session message store
⓭ Iterates each message for printing to console

This enhanced version creates an interactive experience where the agent remembers previous interactions and maintains context throughout the conversation. Notice how the session object is built and passed back to the agent.

When you run the code from listing 2.3, your agent will remember every previous interaction, as shown in listing 2.4. For example:

Listing 2.4 Console output from code in Listing 2.3

```
User: There is a tree directly in front of the car.
Avoid it and return to the original path.
Assistant: ["turn right", "forward", "turn left", "forward", "turn left",
"forward", "turn right"]
User: Now move forward two times.
Assistant: ["forward", "forward"]
User:

History:
user: ## Context
Complex command:
"There is a tree directly in front of the car.
Avoid it and return to the original path."
assistant: ["turn right", "forward", "turn left", "forward", "turn left",
"forward", "turn right"]
user: ## Context
Complex command:
"Now move forward two times."
assistant: ["forward", "forward"]
```

Congratulations! You've just created an agent that can intelligently break down complex movements for a robot car and that remembers previous interactions!

2.3.6 Streaming Agent Responses

To capture the essence of an interactive agent, we need to implement response streaming. This feature enhances the user experience and improves perceived response speed. With streaming, users

can start reading the AI's response as it's generated, instead of waiting for the entire message to complete.

Let's upgrade our agent from listing 2.3 with this streaming capability. Locate this line:

```
AgentResponse response = await agent.RunAsync(prompt, session);
Console.WriteLine($"Assistant: {response.Text}");
```

And replace it with:

```
Console.Write($"Assistant: ");
await foreach (AgentResponseUpdate update in agent
  .RunStreamingAsync(prompt, session))
{
  Console.Write(update.Text);
}
Console.WriteLine();
```

Run the application and enter the following user prompt at the console:

```
There is a tree directly in front of the car.
Avoid it and then come back to the original path.
```

The response should be like what we saw earlier, but streaming responses improve user experience by reducing perceived latency, since users see the AI's reply almost immediately. This creates a more natural, interactive flow that mimics human conversation.

IMPORTANT When responding with structured information, streaming is less effective because you need the full response to ensure clarity and coherence. In those cases, a non-streaming response is more suitable.

While our agent can now hold a conversation and maintain context, it's still a passive system. It responds to user input, but it doesn't take any independent actions. Next, we'll enhance the agent's capabilities by enabling it to directly control the robot car based on its understanding of the conversation.

2.3.7 Building a Tool Agent with Actions

While having a responsive agent is impressive, the real power lies in taking action based on AI's responses. This lets the vehicle make decisions more independently, rather than relying on preprogrammed handling of the model's output. We can use Agent Framework's ability to automate actions as the model generates responses, enabling the agent to execute those actions based on its language understanding.

DEFINING TOOLS WITH NATIVE FUNCTIONS

Native functions let you define custom tools that agents can use to interact with your application's logic or data. A tool is typically a native function (conventional code) that's wrapped and registered so an agent can invoke it as needed.

Listing 2.5 shows the `MotorTools` class, which shows how to create tools from native C# methods:

Listing 2.5 Motor AI tools class

```
using Microsoft.Extensions.AI;  //❶
using System.ComponentModel;

public static class MotorTools  //❷
{
  private const int Delay = 1000;  //❸
```

```
  static public IEnumerable<AITool> AsAITools()  //❹
  {
    yield return AIFunctionFactory.Create(TurnRight);
    yield return AIFunctionFactory.Create(TurnLeft);
    yield return AIFunctionFactory.Create(Stop);
    yield return AIFunctionFactory.Create(Forward);
    yield return AIFunctionFactory.Create(Backward);
  }

  [Description("Basic command: Moves the robot car backward.")]
  public static async Task<string> Backward(
    [Description("The distance (in meters) backward.")] int distance)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Backward:
{distance}m");
    await Task.Delay(Delay);  //❺
    return $"moved backward for {distance} meters.";
  }

  [Description("Basic command: Moves the robot car forward.")]
  public static async Task<string> Forward(
    [Description("The distance (in meters) to move the robot car forward.")]
    int distance)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Forward:
{distance}m");
    await Task.Delay(Delay);
    return $"moved forward for {distance} meters.";
  }

  [Description("Basic command: Stops the robot car.")]
  public static async Task<string> Stop()
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Stop");
    await Task.Delay(Delay);
    return "stopped.";
  }

  [Description("Basic command: Turns the robot car anticlockwise.")]
  public static async Task<string> TurnLeft(
    [Description("The angle (in degrees) to turn the robot car anticlockwise.")]
    int angle)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: TurnLeft:
{angle}°");
    await Task.Delay(Delay);
    return $"turned anticlockwise {angle}°.";
  }

  [Description("Basic command: Turns the robot car clockwise.")]
  public static async Task<string> TurnRight(
    [Description("The angle (in degrees) to turn the robot car clockwise.")]
    int angle)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: TurnRight:
{angle}°");
    await Task.Delay(Delay);
    return $"turned clockwise {angle}°.";
  }
}
```

❶ Imports the namespace for AITool class used in user-defined AsAITools method
❷ Defines the MotorTools class that keep the domain-specific functions together
❸ Declares a property that simulates 1000 milliseconds delay
❹ Defines a method that returns the native functions as tools
❺ Introduces a delay to simulate an action that takes time to execute

Native functions in Agent Framework use method annotations such as `DescriptionAttribute`. Classes and parameters also have their own descriptions. In generative AI, these semantic descriptions are as crucial as syntax is in traditional programming.

IMPORTANT If no description is provided for function names or arguments, Agent Framework uses the names themselves by default. This is a compact way to inform the AI model about their purpose. Clear, meaningful naming conventions are still essential, as the model relies heavily on these names when selecting and invoking functions.

Advanced large language models use these descriptions (or, if descriptions aren't provided, their names) to:

- understand a function's purpose
- use functions correctly
- identify appropriate functions to call.

Comprehensive descriptions of native functions in Agent Framework are essential for effective AI interactions because large language models (LLMs) choose whether to call functions based on context.

When LLMs prepare a response, they may decide to call none, one, or multiple functions associated with the prompt to provide a more accurate, informed response, depending on the agent setting. For example, if we ask the model about the current weather in a specific location, it likely doesn't have real-time weather data. Without access to relevant functions, it might hallucinate a response or state that it can't provide accurate information. But when the prompt includes well-described functions, the LLM can fill information gaps by calling the right ones.

I propose the term *semantic signature* to describe this set of contextual conventions, analogous to a traditional method's syntax signature, which defines its name, parameters, and data types. A semantic signature extends this notion by capturing a function's meaning, intent, and capability, so the AI can use it more intelligently in suitable contexts.

IMPORTANT Semantic signatures capture both the structural and descriptive details of a function, encompassing its name, arguments, and descriptions. This concept is crucial because it gives an AI model a deeper understanding of each function's intent and capabilities. As a result, AI models can select functions and Agent Framework can invoke them more intelligently and contextually, improving reliability and enabling more advanced orchestration in AI-driven solutions.

While LLMs can decide whether to call functions based on context, the underlying decision process varies across implementations and model architectures.

IMPORTING AI TOOLS FROM A CLASS

To build an agent with AI tools, use `Tools` property:

Listing 2.6 Create a tool agent

```
var agent = chatClient.AsAIAgent("""
  instructions: """
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.
```

```
  You have to break down the provided complex commands into the basic
  moves you know.
  Respond only with the moves, without any additional explanations.
  """,
  tools: [.. MotorTools.AsAITools()]
);
```

With these enhancements (history, streaming, and tools), we've transformed our agent into a stateful, tool-using agent. It can now:

- Engage in natural language conversations
- Stream responses for more fluid interaction
- Access external data and services
- Execute real-world actions through native functions

This foundation opens exciting possibilities. Imagine expanding this system to control various aspects of a smart home, assist in complex data analysis, or even guide autonomous systems through challenging scenarios.

BRINGING ALL CHANGES TOGETHER

Listing 2.7 shows the full code for creating an agent with tools. It uses the class defined already in listing 2.5.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Listing 2.7 Agent sends a prompt to an AI model using tools

```
using Microsoft.Agents.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;
using AITools;

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❶

AIAgent agent = chatClient.AsAIAgent("""
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.
  You have to break down the provided complex commands into the basic
  moves you know.
  Respond only with the moves, without any additional explanations.
  """,  //❷
  tools: [..MotorTools.AsAITools()]  //❸
);

AgentSession session = await agent.CreateSessionAsync();

while (true)
{
  Console.Write("User: ");
  var input = Console.ReadLine();  //❹
  if (string.IsNullOrEmpty(input)) break;  //❹

  var prompt = $"""
    Complex command:
```

```
    "{input}"
    """;  //❺

  AgentResponse response = await agent.RunAsync(prompt, session);
  Console.WriteLine($"Assistant: {response.Text}");  //❻
}
```

❶ **Creates the chat client**
❷ **Assigns the tools from MotorTools**
❸ **Creates the AI agent using the chat client, instructions and tools**
❹ **Reads the input from user until the input is empty**
❺ **Wraps the input into a prompt**
❻ **Prints the text response to the console**

Run the code. The output should look like this:

Listing 2.8 Console output for listing 2.7

```
User: There is a tree directly in front of the car.
Avoid it and then return to the original path.
[06:01:43:339] MOTORS: TurnRight: 90°
[06:01:44:361] MOTORS: Forward: 5m
[06:01:45:366] MOTORS: TurnLeft: 90°
[06:01:46:369] MOTORS: Forward: 10m
[06:01:47:372] MOTORS: TurnLeft: 90°
[06:01:48:378] MOTORS: Forward: 5m
[06:01:49:384] MOTORS: TurnRight: 90°
Assistant: The robot car successfully avoided the tree and returned to the
original path.
```

Notice the tool responses (for example, `MOTORS: TurnRight: 90°`) and the assistant response produced by the AI model. Each step in the sequence triggered the corresponding tool from `MotorTools`, as expected.

IMPORTANT Unlike listing 2.4 that shows the output when there are no tools involved, this response was produced after the agent invoked one or more provided tools to complete the task.

2.4 Conclusion

As we've seen so far, we've only scratched the surface of Agent Framework's capabilities. Our simple console application is the first step in an exciting journey into generative AI development. Agent Framework streamlines interactions with large language models, abstracting away complex API calls and prompt engineering.

Summary

- Motivation for agents and LLMs: a robot car story that links complex commands to low-level device APIs.
- Limitations of hardcoded movement sequences and the need for a more scalable approach to complex robot-car behaviors.
- Use assistants such as ChatGPT or Copilot to translate natural language movement descriptions into executable step sequences for the robot car API.
- An understanding of LLM nondeterminism and its effects on machine-to-machine scenarios that require predictable, structured outputs.
- The PACT prompt engineering framework is a practical method for designing clear, reusable

prompts for control scenarios.

- Techniques for shaping free-form prompts into JSON responses that are easy to parse and integrate into .NET applications.
- Steps for obtaining OpenAI and Azure OpenAI API keys, plus core options for securing them across environments.
- Build your first stateless Agent Framework console agent that sends queries to an LLM and prints structured results.
- Evolving to a stateful agent with `AgentSession` so conversation history and prior context are preserved across interactions.
- Separating static instructions from dynamic queries, and mapping them to Agent Framework concepts for modular, maintainable prompts.

9

Building enterprise-ready agents with ChatClient middleware

This chapter covers

- Why middleware is essential for production AI agents
- The two-layer middleware architecture: ChatClient vs Agent
- Three middleware types: SharedFunction, Response, and FunctionCalling
- Composing a complete middleware pipeline

Robby worked well in development because nothing bad was at stake. In production, the same agent became dangerous: no token limits, no input/output data redaction, and no guard around tool calls. The root problem was architectural; the agent talked straight to the LLM. Low-level *ChatClient* middleware is the caller-agnostic layer between any chat client and the model where you attach shared, organization-wide infrastructure guardrails for all agents without cluttering their logic with cross-cutting concerns. It turns a "works in dev" agent into an enterprise-ready one by enforcing global guardrails for cost, safety, and basic sanitization without rewriting the agent.

9.1 Applying ChatClient Middleware to Agents

Robby's AI agent worked flawlessly in development. Then production happened, and that's when we learned that *middleware*, the layer that sits between our chat calls and the LLM to enforce guardrails, is not optional. It is essential.

If we come from ASP.NET, this should feel familiar. HTTP middleware wraps requests and responses in a pipeline. Here we apply the same idea to chat calls and LLM responses: middleware surrounds every call and handles cross-cutting concerns such as validation, logging, budgets, and safety.

9.1.1 Explaining the Middleware Pipeline Pattern

Middleware looks abstract until you see how a request actually flows through the pipeline. The next two diagrams show the two essential cases you will use throughout this chapter.

HAPPY-FLOW MIDDLEWARE

In the normal case as shown in figure 9.1, every piece of middleware lets the call pass through to the LLM and then sees the response on the way back.

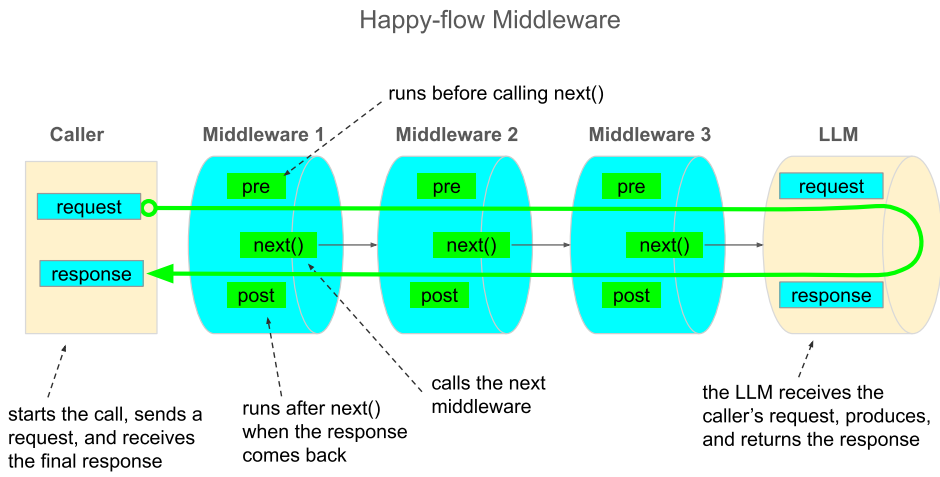

Figure 9.1 The middleware pipeline wraps the LLM call: each middleware runs pre logic, calls next(), then runs post logic as the response returns, so every component executes on both the way in and the way out.

In the happy-flow diagram, the caller starts the operation by sending a request into the pipeline. *Middleware 1* receives that request and runs its pre logic first (e.g., logging, input shaping, or attaching metadata). It then calls `next()`, which passes control to the next middleware in the chain rather than to the LLM directly.

Middleware 2 and *Middleware 3* repeat the same pattern. Each one gets the request, runs its own pre logic, and calls `next()` to move the request further down the pipeline. Eventually the request reaches the LLM, which receives the caller's request, produces an answer, and returns a response.

On the way back, the response travels through the pipeline in reverse order. *Middleware 3* runs its post logic, then returns control to *Middleware 2*, which runs its own post logic, and finally *Middleware 1* does the same before the response reaches the caller. In this happy flow, every middleware in the chain runs on both sides of the call: once before `next()` and once after `next()` completes.

SHORT-CIRCUIT MIDDLEWARE

Sometimes middleware needs to stop the operation early (e.g., when a token budget is exhausted or a user is not authorized) as seen in figure 9.2.

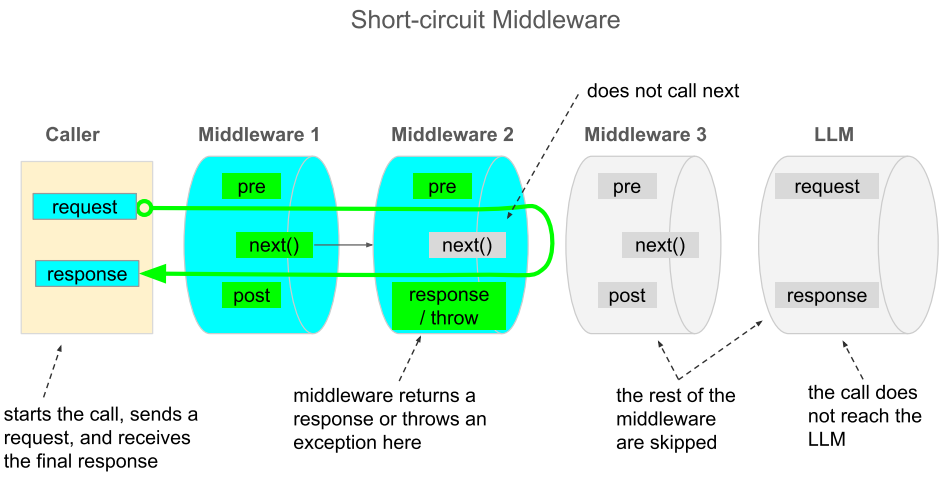

Figure 9.2 Any middleware can short-circuit the pipeline by not calling next() and instead returning a response or throwing, so later middleware and the LLM are skipped and only earlier middleware run their post logic.

The short-circuit diagram shows the same pipeline, but this time *Middleware 2* decides not to call `next()`. The caller still sends a request and Middleware 1 still runs its pre logic and calls `next()`, so the request reaches *Middleware 2* in the usual way. *Middleware 2* runs its own pre logic, but instead of forwarding the request, it either returns a response directly or throws an exception.

At that moment, the pipeline ended. *Middleware 3* and the LLM are skipped entirely because `next()` was never invoked in *Middleware 2*. The response (or error) flows back from *Middleware 2* to *Middleware 1*, which can still run its post logic, and then back to the caller. This short-circuiting behavior is what lets you implement guardrails such as token budgets, authorization checks, or validation rules: a middleware can stop the operation early and return a controlled outcome without changing any of the downstream middleware or the agent itself.

9.1.2 Defining ChatClient Middleware

ChatClient middleware is the production layer that sits between a chat client call and the AI model. It does not define the agent's reasoning or behavior, but it enforces how that behavior is executed in a real system.

A useful way to think about middleware is as the safety systems in a car. The engine (the agent) provides power, but safe driving depends on the surrounding control systems:

- Seatbelts (validation) prevent bad or malformed inputs from causing damage.
- Airbags (error handling) absorb failures, so a single fault doesn't bring everything down.
- Speed limiters (rate limiting) keep calls, costs, and side effects within safe bounds.
- Dashcam (audit logging) records what happened, when, and why for debugging, compliance, and incident response.
- Collision warning (human approval) pauses before high-risk actions and asks a human to confirm.

In production systems, these safeguards are not optional. Middleware provides the guardrails that let us operate powerful agents safely, keeping control over risk, cost, and compliance.

9.1.3 Visualizing the Two-Layer Middleware Architecture

In Agent Framework, middleware operates at two distinct layers: the *ChatClient* layer (`IChatClient` from Microsoft.Extensions.AI) and the *Agent* layer (Microsoft.Agents.AI).

The diagram in figure 9.3 shows a two-layer middleware architecture, where both layers operate independently but can be combined.

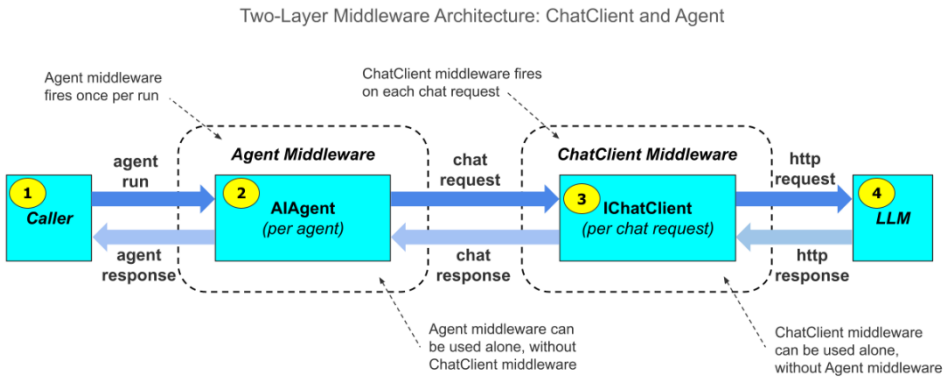

Figure 9.3 Diagrams show the two-layer middleware model. Agent middleware is at higher level, and it runs once per RunAsync() call (agent run), and ChatClient middleware is a lower level, and it runs on each call to LLM (chat request).

These are the two-layer diagram steps:

- Step 1: Caller Invokes Agent

  The caller initiates the flow on the AIAgent (component 2). This triggers the execution pipeline, starting with any registered agent middleware.

- Step 2: *Agent* Middleware Executes

  Agent middleware wraps the entire agent running lifecycle. It fires once per agent invocation, allowing inspection and modification of both the incoming chat request and outgoing agent response. Importantly, agent middleware can run standalone without requiring *ChatClient* middleware.

- Step 3: *ChatClient* Middleware Executes

  After the agent processes the request, it sends a chat request to the `IChatClient` (component 3). At this point, *ChatClient* middleware intercepts the HTTP request to LLM. This middleware fires on each chat request, meaning it can execute multiple times within a single agent run if the agent makes multiple model calls. *ChatClient* middleware can also operate independently, without *Agent* middleware.

- Step 4: LLM Interaction

  The `IChatClient` sends the HTTP request to the LLM (component 4) and receives the HTTP response. The response then flows back through the middleware chain in reverse order: first

through any *ChatClient* middleware for post-processing, then through agent middleware, and finally back to the caller.

This two-layer design provides fine-grained control over where we intercept execution, allowing different concerns to be handled at the appropriate level.

NOTE ChatClient layer is your universal safety net, it sees every chat request to the LLM and applies shared behavior across all agents. The Agent layer is your fine-grained control, it knows which agent, which session, and enforces per-agent instructions.

In this chapter, we focus on the lower layer, *ChatClient* middleware.

SOLVING PRODUCTION PROBLEMS WITH MIDDLEWARE

Middleware handles everything that isn't core agent logic but is absolutely essential for production (table 9.1):

Table 9.1 Production scenarios handled using middleware

Production Concern	Without Middleware	With Middleware
Security	PII (Personally Identifiable Information) sent to LLM providers	Automatically redacted
Cost Control	Unlimited API calls or token usage	Enforced quotas and budgets
Compliance	No audit trail	Complete logging with timestamps
Safety	All operations auto-approved	Human-in-the-loop for dangerous ops
Performance	No visibility into bottlenecks	Metrics and monitoring
Reliability	Errors cascade unpredictably	Graceful handling and recovery

Middleware does not add new capabilities to the agent. It centralizes guardrails such as cost limits, data protection, tool supervision, and auditing, so production risks are handled outside the agent definition.

RUNNING AGENTS WITHOUT MIDDLEWARE

Before adding any middleware, let's see what happens when agents run unprotected. The baseline project demonstrates all three stories in action: email leaking to the LLM provider, no cost control, and dangerous operations executing without validation.

In listing 9.1 we have a simple agent encountering a few challenges mentioned before.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Code path: `/MiddlewareBaseline/Program.cs`

Listing 9.1 Agent without Middleware

```
using AITools;
using Helpers;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;
```

```
var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient();  //❶
// NO .AsBuilder() - NO middleware pipeline!

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are the MotorsAgent controlling a robot car.
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
      """,
    Tools = [.. MotorTools.AsAITools()],
  }
});

AgentSession session = await motorsAgent.CreateSessionAsync();

Console.WriteLine("""
  TEST 1: Email Leak — Step 1 gap (no RemoveEmail)
  ChatClientSharedFunctionMiddleware would redact the address.
  """);
var prompt1 = """
  Navigate to original position.
  Contact me at john.doe@example.com for updates.
  """;
Console.WriteLine($"PROMPT: {prompt1}");
AgentResponse result1 = await motorsAgent
  .RunAsync(prompt1, session);  //❷
Console.WriteLine($"\nRESULT: {result1}\n");
Console.WriteLine("""
  >>> The email was NOT redacted —
  sent directly to the LLM provider (GDPR violation).
  """);

Console.WriteLine("""
  TEST 2: Unlimited Spend —
  Step 1 + Step 2 gap (no LimitRequests, no EnforceTokenBudget)
  ChatClientSharedFunctionMiddleware would limit requests.
  ChatClientResponseMiddleware would enforce a token budget.
  """);

var prompt2 = "Move forward 3 meters, turn right 90 degrees, move forward 3
meters";
Console.WriteLine($"PROMPT: {prompt2}");
AgentResponse result2 = await motorsAgent
  .RunAsync(prompt2, session);  //❸
Console.WriteLine($"\nRESULT: {result2}\n");
Console.WriteLine("""
  >>> No request limit hit, no token budget enforced —
  costs accumulate silently.
  """);
```

```
Console.WriteLine("""
  TEST 3: Unconstrained Backward —
  Step 3 gap (no ConstrainDistance, no AuditFunctionCalling)
  ChatClientFunctionCallingMiddleware would cap backward to 5 m
  and audit every tool call.
  """);

var prompt3 = "Move forward 10 meters then go backward 10 meters";
Console.WriteLine($"PROMPT: {prompt3}");
AgentResponse result3 = await motorsAgent
  .RunAsync(prompt3, session);  //❹
Console.WriteLine($"\nRESULT: {result3}\n");
Console.WriteLine("""
  >>> Backward ran the full 10 m — no constraint, no audit.
  The robot could hit a wall.
  """);
```

❶ ChatClient that has NO middleware protection
❷ Queries the agent with a sensitive input data (email)
❸ Queries the agent with no requests or budget limit
❹ Queries the agent with no audit or action supervision

Code output:

```
TEST 1: Email Leak — Step 1 gap (no RemoveEmail)
ChatClientSharedFunctionMiddleware would redact the address.
PROMPT: Navigate to original position.
Contact me at john.doe@example.com for updates.

RESULT: stop

>>> The email was NOT redacted —
sent directly to the LLM provider (GDPR violation).
TEST 2: Unlimited Spend —
Step 1 + Step 2 gap (no LimitRequests, no EnforceTokenBudget)
ChatClientSharedFunctionMiddleware would limit requests.
ChatClientResponseMiddleware would enforce a token budget.
PROMPT: Move forward 3 meters, turn right 90 degrees, move forward 3 meters
[01:37:34:935] MOTORS: Forward: 3m
[01:37:37:099] MOTORS: TurnRight: 90°
[01:37:39:217] MOTORS: Forward: 3m
[01:37:41:095] MOTORS: Stop

RESULT: forward 3
turn right 90
forward 3
stop

>>> No request limit hit, no token budget enforced — costs accumulate silently.
TEST 3: Unconstrained Backward —
Step 3 gap (no ConstrainDistance, no AuditFunctionCalling)
ChatClientFunctionCallingMiddleware would cap backward to 5 m
and audit every tool call.
PROMPT: Move forward 10 meters then go backward 10 meters
[01:37:43:921] MOTORS: Forward: 10m
[01:37:48:027] MOTORS: Backward: 10m
[01:37:49:975] MOTORS: Stop

RESULT: forward 10
backward 10
stop

>>> Backward ran the full 10 m — no constraint, no audit.
The robot could hit a wall.
```

All three tests expose the same root problem: the agent is an unguarded conduit between the caller and the LLM. Sensitive data travels to the LLM provider unchanged, every command executes

immediately regardless of risk, and nothing stands between a runaway request loop and a five-figure cloud bill.

The critical insight is that these concerns are cross-cutting. They apply to every agent, every session, and every LLM call, yet none of them belong in business logic. This is exactly the problem middleware is designed to solve, following the principle of *Separation of Concerns*: keeping different responsibilities in various parts of the system so each part has a clear, focused job. Middleware intercepts the pipeline at the right layer, enforces rules once, and protects every agent transparently, without touching a single line of agent.

9.2 Walking Through ChatClient Middleware Layer

ChatClient middleware fires on each chat request between `IChatClient` and the LLM, operating without agent session context or identity. This lack of context is deliberate: it treats every invocation as an independent HTTP call, applying shared behavior uniformly across all agents. Without *ChatClient* middleware layer, agents talk directly to LLM with no place to protect input or output, supervise tool calls, or audit what happened.

Diagram in figure 9.4 shows the *ChatClient* middleware.

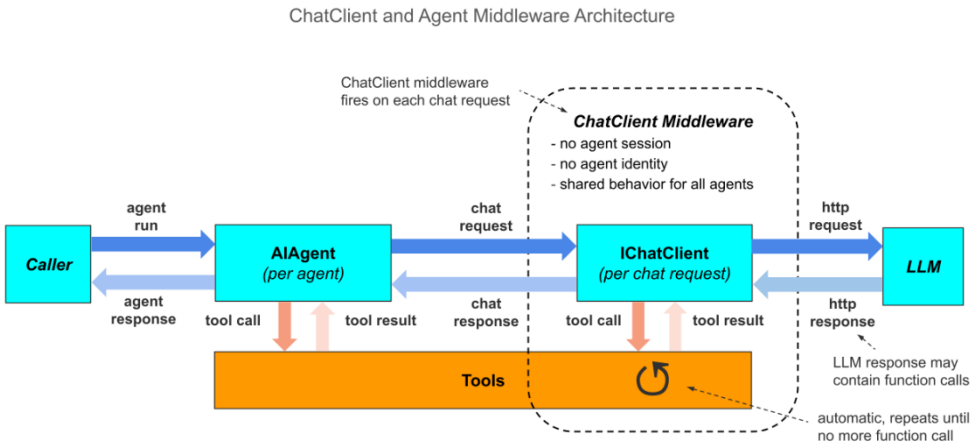

Figure 9.4 ChatClient middleware in tool-calling flow: operates without agent context, intercepts every chat request to the LLM, and fires repeatedly during iterative tool execution loops.

During tool calling, the flow becomes iterative. When the LLM returns function calls in the chat request area, the agent executes them via the *Tools* area and sends results back through `IChatClient`. This loop repeats automatically until no more function calls are requested. *ChatClient* middleware intercepts every round trip, potentially firing multiple times per agent run. The chat request area handles both responses and shared functions, while the *Tools* area manages function calling execution. This interception prevents issues like unredacted sensitive data reaching the LLM, unconstrained tool calls causing damage, and uncontrolled token consumption accumulating costs.

To address these concerns effectively, *ChatClient* middleware is implemented using three distinct types, each targeting a specific stage in the chat request-response lifecycle.

9.2.1 Choosing the Right ChatClient Middleware Type

Robby's production issues fall into three buckets: what we send to the LLM, what we get back, and how tools are invoked. *ChatClient* middleware mirrors this with three types, so we must pick the right one based on whether we need to shape the request, inspect the response, or supervise a tool call. *SharedFunction* prepares outgoing requests (sanitizing or redacting data before it reaches the LLM). Response handles incoming responses (tracking tokens, enforcing budgets, or inspecting content). *FunctionCalling* supervises tool invocations (constraining arguments, auditing calls, or blocking unsafe executions). Together, these types cover the prepare-handle-invoke pattern across the chat request area and tool execution flow.

Figure 9.5 breaks down the three *ChatClient* middleware types side by side, showing their inputs, outputs, and where each sit relative to the LLM response.

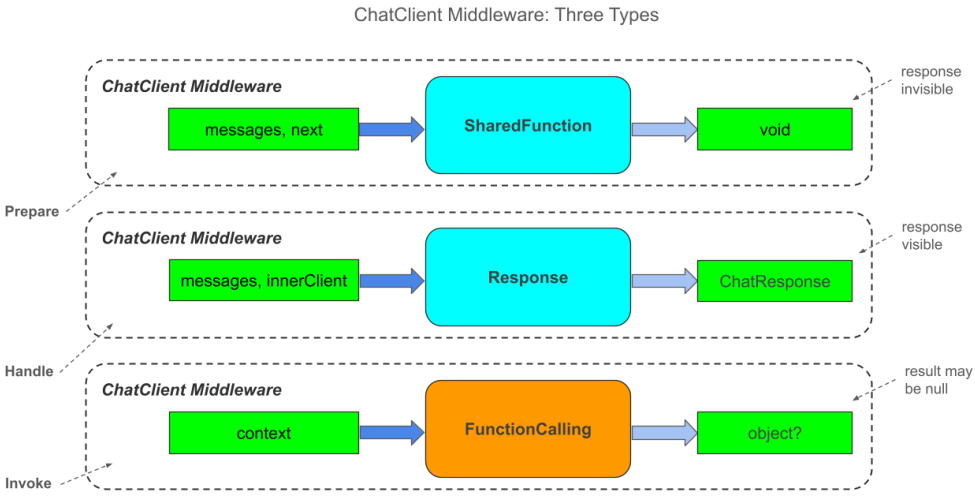

Figure 9.5 The diagram shows the three types of ChatClient middleware: SharedFunction, Response, and FunctionCalling.

The three types form a layered progression. *SharedFunction* is the lightest: it can inspect and modify input messages before they reach the LLM, but it never sees the response. *Response* wraps the entire LLM call, giving you access to both the request and the full `ChatResponse`, including token usage and latency. *FunctionCalling* is specialized: it does not participate in the LLM round-trip at all. Instead, it intercepts each tool invocation, letting you validate, modify, or reject individual function calls before they execute.

- *SharedFunction*: **prepares** the input (blind to `ChatResponse`)
- *Response*: **handles** the input and output (sees `ChatResponse`)
- *FunctionCalling*: **invokes** each tool (innermost, fires per tool call)

We'll build up the middleware pipeline gradually. *SharedFunction* is the simplest type: it controls what goes into the LLM but is structurally blind to what comes out (`ChatResponse`). *Response* unlocks the response; you get both input and output control. *FunctionCalling* is a different category entirely: it doesn't intercept LLM calls at all, it intercepts individual tool invocations (conventional code).

9.2.2 Preparing Requests with SharedFunction Middleware

SharedFunction uses the classic `next`-delegate middleware pattern. Each *SharedFunction* component receives messages, options, a next delegate, and a cancellation token. It can transform the messages, throw an exception to block the request, or call `await next(…)` to pass control to the `next` middleware and later resume execution after that call. The signature encodes this pattern. The `next` delegate returns `Task` (void). There is no `ChatResponse` value to return or modify here, which is why `ChatResponse` never appears in this middleware type.

Figure 9.6 shows the *SharedFunction* middleware type.

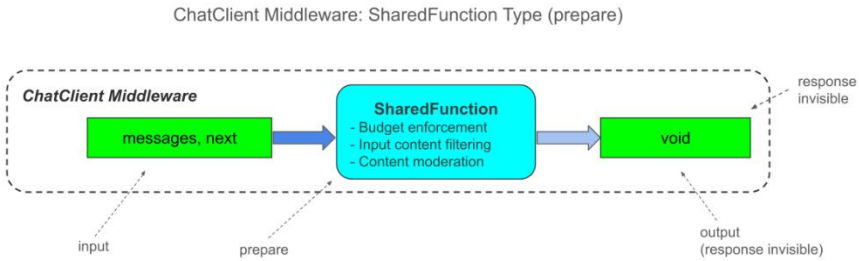

Figure 9.6 The SharedFunction middleware receives messages and a next delegate. The SharedFunction returns no value, and there is no ChatResponse to read or alter. Typical use cases include request limiting, input content filtering, prompt injection detection, and content moderation. All of these run before any token is consumed.

Registration order defines the pipeline order: each middleware wraps the next one, and if it does not call next, the pipeline short-circuits. Typical use cases include request limiting, input content filtering, and prompt injection detection. All of these run before any token is consumed.

This makes *SharedFunction* the natural first layer: validate and adjust the input at this level. If something is wrong with the request, we fail here, before tokens are consumed and before the response path runs at all.

PRACTICAL EXAMPLE: EMAIL REMOVAL

Let me tell you a production story: The GDPR Nightmare Story.

For a European client, we deploy Robby as an intelligent delivery robot car in their office building. Employees could summon Robby, providing their name, office number, and phone number for delivery confirmation. Six months later, a security audit revealed that every employee's phone number, email, and office location had been sent to their LLM provider and logged in plain text.

Robby worked correctly. It just never sanitized the data before processing. This is what we want:

```
Input:  "Navigate to GPS. Contact me at john@example.com for updates."
Output: "Navigate to GPS. Contact me at [REDACTED-EMAIL] for updates."
```

Let's see in listing 9.2 how we can prevent sensitive information leakage.

Code path: `/ChatClientSharedFunctionMiddleware/ChatClientSharedFunctions.cs`

Listing 9.2 RemoveEmail

```
public static class ChatClientSharedFunctions
{
  private const string EmailPattern =
    @"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b";  //❶
  private const string EmailMask = "[REDACTED-EMAIL]";  //❷

  public static async Task RemoveEmail(
```

```
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    Func<IEnumerable<ChatMessage>, ChatOptions?,
      CancellationToken, Task> next,
    CancellationToken cancellationToken)  //❸
  {
    Console.WriteLine($"[ChatClient] [SharedFunction] [Email] "
      + "PRE: Scanning {messages.Count()} messages...");

    bool emailFound = false;  //❹
    List<ChatMessage> sanitizedMessages = [];

    foreach (var message in messages)
    {
      if (message.Role == ChatRole.User && message.Text is not null)  //❺
      {
        var sanitized = Regex
          .Replace(message.Text, EmailPattern, EmailMask);
        if (sanitized != message.Text) emailFound = true;
        sanitizedMessages.Add(new ChatMessage(message.Role, sanitized));
      }
      else
      {
        sanitizedMessages.Add(message);
      }
    }

    Console.WriteLine(emailFound
      ? "[ChatClient] [SharedFunction] [Email] Email detected and removed!"
      : "[ChatClient] [SharedFunction] [Email] No email found");

    await next(sanitizedMessages, options, cancellationToken);  //❻

    Console.WriteLine($"[ChatClient] [SharedFunction] [Email] POST: "
      + "Completed");
  }
}
```

❶ Defines Regex pattern standard email address format
❷ Defines replacement mask for any detected email
❸ Defines the SharedFunction
❹ Tracks whether any email was found
❺ Only user messages are sanitized, tool calls and tool results are not
❻ Passes sanitized messages downstream

Let's see in listing 9.3 how to wire it into the *ChatClient* pipeline.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Code path: `/ChatClientSharedFunctionMiddleware/Program.cs`

Listing 9.3 SharedFunction Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❷
  .Build();  //❸

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
```

```
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are an AI assistant controlling a robot car capable of performing
      basic moves: forward, backward, turn left, turn right, and stop.
      You have to break down the provided complex commands into the basic
      moves you know.
      Use a JSON array like [move1, move2, move3] for the response.
      Respond only with the moves and their parameters (angle or distance),
      without any additional explanations.
      """
  }
});  //❹

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt = """
  There is a tree directly in front of the car.
  Avoid it and then return to the original path.
  """;
Console.WriteLine($"PROMPT: {prompt}");
var result = await motorsAgent.RunAsync(prompt, session);  //❺
Console.WriteLine($"\nRESULT: {result}\n");
```

❶ Converts IChatClient into a builder to enable middleware registration
❷ Registers RemoveEmail
❸ Builds the IChatClient with the middleware pipeline applied
❹ Creates the agent
❺ Runs the agent

Email sanitization is now in place, so Robby will never leak an email address to the LLM provider. But Robby can still make unlimited LLM calls. The $10,847 bill is still entirely possible. The next middleware adds the request limit guard, and once both are defined, we will compose them into a single pipeline and see how they interact.

NOTE Transient vs. persistent sanitization. At the *ChatClient* layer, sanitization is *transient*: the LLM sees clean data, but the agent's session chat history retains the original text (email stays intact). This is safe because the middleware re-applies transparently on every LLM round-trip. Contrast with the *Agent* layer (as we will find out in the next chapter where we discuss *Agent* middleware layer), where such `RemoveEmail` function is *persistent*, sanitized messages are stored in the session history, so all future round-trips already see clean data without re-sanitization. At the *ChatClient* layer, the session is not accessible, so transient re-application is the only option, but it is the developer's responsibility to choose which middleware layer fits best the desired use-case.

PRACTICAL EXAMPLE: REQUEST LIMITING

Let me tell you a production story: The $10,000 Weekend.

Late on a Friday afternoon, Robby navigated obstacles, followed voice commands, and captivated visitors. On Monday morning, the CTO opened an email from OpenAI. The weekend bill was $10,847. The cause was simple but costly. A bug in Robby's retry logic had triggered 47,000 API calls as it continuously re-evaluated its environment. Without middleware to enforce rate or request limits, nothing stopped it.

Listing 9.4 shows how such a costly incident can be prevented.

Code path: `/ChatClientSharedFunctionMiddleware/ChatClientSharedFunctions.cs`

Listing 9.4 LimitRequests

```
public static class ChatClientSharedFunctions
{
  private static int _requestCount = 0;  //❶
  private const int MaxRequests = 2;  //❷

  public static async Task LimitRequests(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    Func<IEnumerable<ChatMessage>, ChatOptions?, CancellationToken, Task> next,
    CancellationToken cancellationToken)  //❸
  {
    var currentCount = Interlocked.Increment(ref _requestCount);  //❹

    Console.WriteLine($"[ChatClient] [SharedFunction] [Limit] PRE: "
      + "Request {currentCount} of {MaxRequests} max");

    if (currentCount > MaxRequests)  //❺
    {
      throw new LimitExceededException(currentCount, MaxRequests);  //❻
    }

    await next(messages, options, cancellationToken);  //❼

    Console.WriteLine($"[ChatClient] [SharedFunction] [Limit] POST: "
      + "Request {currentCount} of {MaxRequests} max completed");
  }
}
```

❶ **Declares shared counter across all LLM round-trips**
❷ **Declares maximum number of LLM round-trips allowed**
❸ **Declares SharedFunction**
❹ **Thread-safe increment**
❺ **Checks the request budget**
❻ **Throws a custom exception carrying current count and limit**
❼ **Limit not exceeded, continue the pipeline toward the LLM**

Key teaching points (for listing 9.4):

- `Interlocked.Increment` for thread-safe counting across concurrent agents

  PRE phase: check requests limit, throw custom `Exception` if exceeded — the request never reaches the LLM

- `await next(messages, options, cancellationToken)` continues the pipeline

  POST phase: log completion after the entire inner pipeline has returned

---

WARNING With tool-calling agents, a single user prompt triggers *multiple* LLM round-trips and the limit depletes faster than the number of user queries suggests

---

Listing 9.5 shows the custom exception we build specially for capturing the limit exceeding requests.

Listing 9.5 LimitExceededException.cs

```
public class LimitExceededException(int currentCount, int maxRequests)
  : Exception($"Request limit exceeded: request {currentCount} "
    + "of {maxRequests} blocked!")
{ }
```

Fail fast saves tokens. Because *SharedFunction* is outermost, `LimitRequests` throws before *Response* middleware even starts. No LLM call is made, no tokens are consumed, no money is spent.

With both functions defined, we now wire them into a single middleware pipeline. Registration order determines execution order: `LimitRequests` registers first, so it sits outermost. If the budget is exceeded, `RemoveEmail` is never entered and no sanitization work is wasted.

Let's see in listing 9.6 how the previously defined *ChatClient* middleware functions (`RemoveEmail` and `LimitRequests`) kick in.

Code path: `/ChatClientSharedFunctionMiddleware/Program.cs`

Listing 9.6 SharedFunction Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientSharedFunctions.LimitRequests)  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❷
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are an AI assistant controlling a robot car capable of performing
      basic moves: forward, backward, turn left, turn right, and stop.
      You have to break down the provided complex commands into the basic
      moves you know.
      Use a JSON array like [move1, move2, move3] for the response.
      Respond only with the moves and their parameters (angle or distance),
      without any additional explanations.
      """
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt1 = """
  There is a tree directly in front of the car.
  Avoid it and then return to the original path.
  """;
Console.WriteLine($"PROMPT: {prompt1}");
var result1 = await motorsAgent.RunAsync(prompt1, session);  //❹
Console.WriteLine($"RESULT: {result1}\n");

var prompt2 = """
  Navigate to original position.
  Contact me at john.doe@example.com for updates.
  """;
Console.WriteLine($"PROMPT: {prompt2}");
try
{
  var result2 = await motorsAgent.RunAsync(prompt2, session);  //❺
  Console.WriteLine($"RESULT: {result2}\n");
}
catch (LimitExceededException ex)  //❻
{
  Console.WriteLine($"EXCEPTION: {ex.Message}\n");
}
```

```
var prompt3 = "Stop and email me at sara.doe@example.com for "
  + "further instructions.";
Console.WriteLine($"PROMPT: {prompt3}");
try
{
  var result3 = await motorsAgent.RunAsync(prompt3, session);  //❼
  Console.WriteLine($"RESULT: {result3}\n");
}
catch (LimitExceededException ex)  //❽
{
  Console.WriteLine($"EXCEPTION: {ex.Message}\n");
}
```

❶ LimitRequests is outermost
❷ RemoveEmail is inner
❸ Defines the agent
❹ First prompt — no email present, limit not exceeded
❺ Second prompt — email detected and removed before reaching the LLM
❻ Catches LimitExceededException from the second prompt if the limit was hit
❼ Third prompt — limit already exhausted from previous round-trips
❽ Catches LimitExceededException

Every prompt is wrapped in try-catch. The source treats all three queries uniformly, because in a real system you cannot assume which round-trip will exhaust the budget.

Code output:

```
PROMPT: There is a tree directly in front of the car. Avoid it and then return to
the original path.
[ChatClient] [SharedFunction] [Limit] PRE: Request 1 of 2 max
[ChatClient] [SharedFunction] [Email] PRE: Scanning 1 messages...
[ChatClient] [SharedFunction] [Email] No email found
[ChatClient] [SharedFunction] [Email] POST: Completed
[ChatClient] [SharedFunction] [Limit] POST: Request 1 of 2 max completed
RESULT: [
  {"move":"turn right","angle":45},
  {"move":"forward","distance":3},
  {"move":"turn left","angle":45},
  {"move":"forward","distance":5},
  {"move":"turn left","angle":45},
  {"move":"forward","distance":3},
  {"move":"turn right","angle":45}
]

PROMPT: Navigate to original position. Contact me at john.doe@example.com for
updates.
[ChatClient] [SharedFunction] [Limit] PRE: Request 2 of 2 max
[ChatClient] [SharedFunction] [Email] PRE: Scanning 3 messages...
[ChatClient] [SharedFunction] [Email] Email detected and removed!
[ChatClient] [SharedFunction] [Email] POST: Completed
[ChatClient] [SharedFunction] [Limit] POST: Request 2 of 2 max completed
RESULT: [
  {"move":"backward","distance":3},
  {"move":"turn right","angle":45},
  {"move":"backward","distance":5},
  {"move":"turn right","angle":45},
  {"move":"backward","distance":3},
  {"move":"turn left","angle":45}
]

PROMPT: Stop and email me at sara.doe@example.com for further instructions.
[ChatClient] [SharedFunction] [Limit] PRE: Request 3 of 2 max
EXCEPTION: Request limit exceeded: request 3 of 2 blocked!
```

Every successful round-trip shows the same sequence:

```
Limit PRE → Email PRE → Email POST → Limit POST.
```

Fail fast confirmed. Prompt 3 shows `[Limit] PRE` immediately followed by the exception, so `[Email] PRE` never appears. As expected, no sanitization work is done on a rejected request.

EXERCISE

Clone `RemoveEmail` function and name it `HideIp` to also detect and redact IP addresses (e.g., 192.168.1.1 → [REDACTED-IP]). Hint: use `Regex.Replace` call for the pattern

```
private const string IpPattern = @"\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b";
private const string IpMask = "[REDACTED-IP]";
```

then call the function using:

```
.Use(ChatClientSharedFunctions.HideIp)
```

The middleware function should mask the IPs as well, and you may want to rename the method accordingly, but nothing stops you to follow the best practices and having two middleware functions instead of one with two responsibilities.

9.2.3 Handling Requests and Responses with Response Middleware

Response middleware type is where the chat response is revealed. Everything *SharedFunction* can do, like transform input messages, block requests, log, *Response* can do too.

There is no *next* delegate here, but the pipeline still exists. The framework builds it by wrapping: it starts from the real chat client, then passes it as `innerClient` to the first middleware, which becomes the new client, then passes that to the next middleware, and so on. Each middleware receives the previously wrapped client as its `innerClient`, calls `GetResponseAsync` to get the response, and returns it with or without modifications. This recursive wrapping is what simulates the *next*-like pipeline without an explicit next parameter.

Figure 9.7 shows the *Response* middleware type.

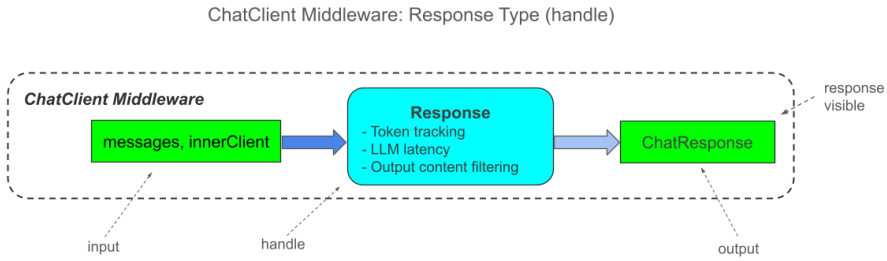

Figure 9.7 The Response middleware receives messages and an IChatClient innerClient, and it returns a ChatResponse. It starts from the real chat client, then repeatedly wraps it, passing the previously built client as innerClient. Typical use cases include token tracking, LLM latency, response caching, and output content filtering.

Response has full control over the `innerClient` to call `GetResponseAsync()` and can read and modify the `ChatResponse`0. With *Response* we can track tokens and latency, attach metadata, enforce budgets and timeouts, or even fabricate a response to short-circuit the pipeline. For example, `ChatResponse.Usage` exposes tokens counts that are only available at this *ChatClient* layer, not in the `AgentResponse` type we see in the *Agent* middleware layer.

This makes it the natural home for token tracking, content filtering, and budget enforcement.

PRACTICAL EXAMPLE: ENFORCE TOKEN BUDGET

Do you recall the $10,000 Weekend story from section 9.2.2, where on Monday morning we find in the inbox a $10,847 bill, because a bug in retry logic? Beyond request limiting, we also need token budget enforcement.

Every LLM call has a cost in tokens. Running out of token budget silently degrades user experience; uncontrolled consumption racks up costs. At the *ChatClient* layer we can measure cumulative token usage — the total tokens consumed across all round-trips — and enforce a hard ceiling before the next call is dispatched.

In listing 9.7 we see the *Response* function.

Code path: `/ChatClientResponseMiddleware/ChatClientResponses.cs`

Listing 9.7 Enforce Token Budget

```
public static class ChatClientResponses
{
  private static long _tokensCount = 0;
  private const long MaxTokens = 2000;  //❶

  public static async Task<ChatResponse> EnforceTokenBudget(
  IEnumerable<ChatMessage> messages,
  ChatOptions? options,
  IChatClient innerClient,
  CancellationToken cancellationToken)  //❷
  {
    var currentTokens = Interlocked.Read(ref _tokensCount);  //❸
    if (currentTokens > MaxTokens)  //❹
    {
      Console.WriteLine($"[ChatClient] [Response] [Tokens] "
        + "Budget exhausted ({currentTokens} / {MaxTokens}) "
        + "— LLM call skipped");
      return new ChatResponse([new ChatMessage(ChatRole.Assistant,
        "Token budget exhausted.")]);  //❺
    }

    var response = await innerClient
      .GetResponseAsync(messages, options, cancellationToken);  //❻

    if (response.Usage is not null)
    {
      var totalTokens = response.Usage.TotalTokenCount
        ?? (response.Usage.InputTokenCount ?? 0) +
           (response.Usage.OutputTokenCount ?? 0);  //❼
      Interlocked.Add(ref _tokensCount, totalTokens);  //❽
    }

    return response;  //❾
  }
}
```

❶ **Defines token limit (Real apps would have much higher limits)**
❷ **Declares Response function (gives access to call and chat response)**
❸ **Reads the current token count**
❹ **Checks if max tokens are exceeded**
❺ **Returns a synthetic assistant message (innerClient is never called)**
❻ **Explicitly calls innerClient for a chat response**
❼ **Computes the response token count**
❽ **Adds the current tokens count**
❾ **Returns the chat response**

When the budget is exhausted, the middleware short-circuits: it returns a synthetic `ChatResponse` with a human-readable message and never calls the LLM.

Practical Example: Timestamp Adding

Production systems often require audit trails that capture when each response was generated. Timestamping chat responses enables compliance tracking, debugging temporal issues, and correlating agent behavior with external events.

Response middleware type has access to `ChatResponse`.

In listing 9.8 we see the `AddTimestamp` *ChatClient Response* middleware function.

Code path: `/ChatClientResponseMiddleware/ChatClientResponses.cs`

Listing 9.8 AddTimestamp

```
public static class ChatClientResponses
{
  public static async Task<ChatResponse> AddTimestamp(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    IChatClient innerClient,
    CancellationToken cancellationToken)  //❶
  {
    ChatResponse response = await innerClient
      .GetResponseAsync(messages, options, cancellationToken);  //❷

    foreach (ChatMessage message in response.Messages)  //❸
    {
      if (!string.IsNullOrEmpty(message.Text))  //❹
      {
        string timestamp = DateTimeOffset.UtcNow.ToString("o");  //❺
        Console.WriteLine($"[ChatClient] [Response] [Timestamp] "
          + "Stamping response with [{timestamp}]");
        message.Contents = [new TextContent(
          $"[{timestamp}] {message.Text}")];  //❻
      }
    }

    return response;
  }
}
```

❶ **Declares Response middleware function**
❷ **Calls innerClient.GetResponseAsync**
❸ **Iterates through all messages**
❹ **Identifies the valid text contents**
❺ **Builds the timestamp string**
❻ **Prefixes the message contents with the timestamp**

Let's see in listing 9.9 how to wire both *Response* functions into the middleware pipeline.

Code path: `/ChatClientResponseMiddleware/Program.cs`

Listing 9.9 Response Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientResponses.EnforceTokenBudget, null)  //❶
  .Use(ChatClientResponses.AddTimestamp, null)  //❷
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are the MotorsAgent controlling a robot car.
```

```
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
    """,
    Tools = [.. MotorTools.AsAITools()],
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt1 = "Move forward 3 meters";
Console.WriteLine($"PROMPT: {prompt1}");
var result1 = await motorsAgent.RunAsync(prompt1, session);  //❹
Console.WriteLine($"\nRESULT: {result1}\n");

var prompt2 = "Turn right 45 degrees";
Console.WriteLine($"PROMPT: {prompt2}");
var result2 = await motorsAgent.RunAsync(prompt2, session);  //❺
Console.WriteLine($"\nRESULT: {result2}\n");
```

❶ **EnforceTokenBudget registered as outer**
❷ **Registered as inner**
❸ **Declares the agent**
❹ **First prompt that triggers timestamp adding**
❺ **Second prompt that triggers limit exceeding response**

WARNING This agent has tools — `MotorTools.AsAITools()`. As side effect, each user prompt triggers multiple LLM round-trips (one to decide which tool to call, one after the tool result is returned). Token consumption accumulates across all round-trips, which is why the budget can be exhausted mid-prompt.

Code output:

```
PROMPT: Move forward 3 meters
[08:26:06:940] MOTORS: Forward: 3m
[08:26:09:244] MOTORS: Stop
[ChatClient] [Response] [Timestamp] Stamping response with [2026-04-
08T10:24:35.1407499+00:00]

RESULT: [2026-04-06 18:26:11 UTC] - forward 3 meters
- stop

PROMPT: Turn right 45 degrees
[08:26:12:384] MOTORS: TurnRight: 45°
[08:26:14:816] MOTORS: Stop
[ChatClient] [Response] [Tokens] Budget exhausted (2068 / 2000) — LLM call skipped

RESULT: Token budget exhausted.
```

Each prompt produces two metric blocks — not one. The first is the initial LLM call (assistant message) where the model decides which tool to invoke; the second is the follow-up call (tool message) after the tool result is returned. This is the same multi-round-trip behavior described in section 9.1.1: *ChatClient* middleware fires on every LLM round-trip, not once per user prompt.

NOTE Token tracking is *ChatClient* middleware layer exclusive. `response.Usage` is present on `ChatResponse` (returned by the LLM provider) but is NOT available on `AgentResponse` (returned by the *Agent* middleware layer). Token budget enforcement, cost estimation, and per-

call token logging are impossible to implement at the *Agent* layer. This is the primary reason `EnforceTokenBudget` must live at the *ChatClient* layer.

EXERCISE

Create a `CacheResponse` *Response* middleware function that caches LLM responses by the content of the outgoing messages. On a cache hit, return the cached `ChatResponse` directly without calling `innerClient`. On a cache miss, call `innerClient`, store the result, and return it. You may use a dictionary to preserve the queries:

```
private static readonly Dictionary<string, ChatResponse> _cache = new();
```

Then check if we have a cache hit or a cache miss like this:

```
var key = string.Concat(messages.Select(m => m.Text));
if (_cache.TryGetValue(key, out ChatResponse? cached))
{
  return cached;  //❶
}
ChatResponse response = await innerClient
  .GetResponseAsync(messages, options, cancellationToken);  //❷
_cache[key] = response;  //❸
return response;  //❹
```

❶ **Returns cached response if we have cache hit**
❷ **Gets a new response**
❸ **Caches the current response**
❹ **Return new response if we have a cache miss**

9.2.4 Supervising Tool Calls with FunctionCalling Middleware

FunctionCalling middleware does not wrap the chat request itself. It runs when the LLM response contains a tool call and the framework is about to execute that tool, where the model has already decided which function to call and with which arguments.

Unlike *SharedFunction* and *Response*, *FunctionCalling* has no next delegate and no `innerClient`. The only way to execute the tool is through `FunctionInvocationContext`:

```
context.Function.InvokeAsync(context.Arguments, cancellationToken);  //❶
```

❶ **Invokes the function with arguments and cancelation token**

`FunctionInvocationContext` provides access to the function name, arguments, chat messages, and result, allowing middleware to validate, modify, or reject tool calls before they execute.

Figure 9.8 shows the *FunctionCalling* middleware type.

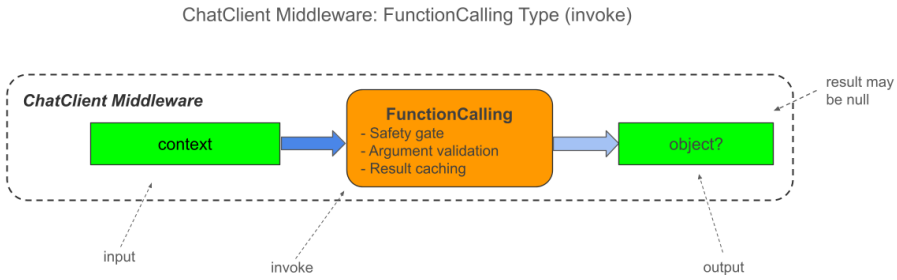

Figure 9.8 The FunctionCalling middleware receives a FunctionInvocationContext and returns ValueTask<object?>. There is no next delegate and no innerClient; the tool is invoked directly through context.Function.InvokeAsync(). Typical use cases include safety checks, audit logging, argument validation, and result caching.

Because there is no *next* delegate, composition relies on a null/non-null result convention. A middleware that returns `null` acts as an *interceptor*: it can inspect and modify arguments but does not call `InvokeAsync`, signaling that the pipeline should continue. A middleware that returns a non-null value acts as the *executor*: it calls `InvokeAsync`, owns the result, and terminates the chain. We can have multiple interceptors but only one executor, and the executor is always last.

Figure 9.9 illustrates the interceptor-executor pattern.

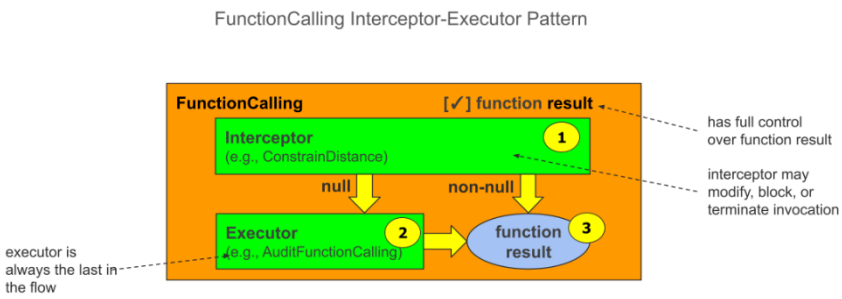

Figure 9.9 FunctionCalling interceptor-executor pattern. The interceptor (step 2) runs first, inspecting and potentially modifying the function invocation. If it returns null, control flows to the executor (step 3), which calls InvokeAsync and produces the final function result (step 4). If the interceptor returns non-null, the flow short-circuits immediately.

In interceptor-executor pattern. The interceptor (step 1) runs first, with full control over the function result — it may modify, block, or terminate the invocation. If it returns null, control flows to the executor (step 2), which calls `InvokeAsync` and produces the final function result (step 3). If the interceptor returns non-null, the flow short-circuits immediately returning the function result (step 3), bypassing the executor entirely. The executor is always last in the flow.

WARNING Agent Framework does not enforce interceptor-executor pattern described here. We are fully responsible for following the null/non-null convention. A mistake can result in double tool execution or skipped safety checks.

FunctionCalling at the *ChatClient* layer is also agent-agnostic. `FunctionInvocationContext` does not carry agent identity, so the same logic applies uniformly across all agents sharing that chat client.

WARNING *FunctionCalling* must be innermost. `UseFunctionInvocation` must be the last middleware registered before `.Build()`. If placed outermost, its internal tool loop drives repeatedly passes back through *SharedFunction* and *Response*, re-running budgets, sanitization, and timing once per tool rather than once per chat request.

PRACTICAL EXAMPLE: DISTANCE CONSTRAINING

Let me tell you a production story: The Runaway Robot

Robby, the autonomous robot car intelligent companion, received the command: `"Danger ahead! Move backward immediately!"` Robby dutifully called, for example, `backward(100)` and drove straight into a wall behind it.

No one had implemented a safety check. The agent just executed. We should have a distance constraint for safety.

In listing 9.10 we see how a *FunctionCalling* middleware function for preventing such incidents is implemented.

Code path:
`/ChatClientFunctionCallingMiddleware/ChatClientFunctionCallings.cs`

Listing 9.10 ConstrainDistance

```
public static class ChatClientFunctionCallings
{
  public static async Task<object?> ConstrainDistance(
    FunctionInvocationContext context,
    CancellationToken cancellationToken)  //❶
  {
    const int MaxBackwardDistance = 5;  //❷
    bool isBackward = context.Function.Name.Contains("backward",
      StringComparison.OrdinalIgnoreCase);  //❸
    bool hasDistance = context.Arguments
      .TryGetValue("distance", out object? value);  //❹
    if (isBackward && hasDistance)  //❺
    {
      int distance = value is JsonElement jsonElement
        ? jsonElement.GetInt32()
        : Convert.ToInt32(value);  //❻
      if (distance > MaxBackwardDistance)  //❼
      {
        context.Arguments["distance"] = JsonSerializer
          .SerializeToElement(MaxBackwardDistance);  //❽
        Console.WriteLine($"[ChatClient] [FunctionCall] [Constrain] "
          + "Backward distance constrained from {distance}m "
          + "to {MaxBackwardDistance}m");
      }
    }

    return null;  //❾
  }
}
```

❶ **Declares middleware function**
❷ **Defines the max allowed backward distance**
❸ **Identifies if the current function is "backward"**
❹ **Identifies if it has "distance" argument**
❺ **Checks if it's "backward" and has "distance"**
❻ **Reads the "distance" value**
❼ **Checks if the distance exceeds the max allowed backward distance**
❽ **Persists as JsonElement for downstream consistency**
❾ **Returns null to signal "proceed to the terminal invoker"**

The `null` return value is what makes the `??` operator in the `FunctionInvoker` fall through to `AuditFunctionCalling`.

Practical Example: Function Calling Auditing

When a tool call causes damage (like Robby hitting a wall), we need a complete record of which function was invoked, with which arguments, and what result it returned. *FunctionCalling* middleware provides this audit trail for debugging, compliance, and security reviews.

In listing 9.11 we see how a *FunctionCalling* middleware function is implemented.

Code path:
`/ChatClientFunctionCallingMiddleware/ChatClientFunctionCallings.cs`

Listing 9.11 AuditFunctionCalling

```
public static class ChatClientFunctionCallings
{
  public static async Task<object?> AuditFunctionCalling(
    FunctionInvocationContext context,
    CancellationToken cancellationToken)  //❶
  {
    var functionName = context.Function.Name;
    var timestamp = DateTime.UtcNow;
    var stopwatch = Stopwatch.StartNew();

    var result = await context.Function
      .InvokeAsync(context.Arguments, cancellationToken);  //❷
    stopwatch.Stop();
    Console.WriteLine($"[ChatClient] [FunctionCall] [Audit] "
      + "Function call '{functionName}' [{timestamp:HH:mm:ss}] "
      + "COMPLETED in {stopwatch.ElapsedMilliseconds}ms | "
      + "Result: {result}");  //❸
    return result;  //❹
  }
}
```

❶ **Defines the middleware function**
❷ **Invokes the function call**
❸ **Logs the successful call**
❹ **Returns the function result**

`AuditFunctionCalling` is the terminal invoker: it is the only method that calls `context.Function.InvokeAsync()`. Because `ConstrainDistance` always returns `null`, the `??` operator guarantees `AuditFunctionCalling` always runs exactly once per tool call.

Let's see in listing 9.12 how we can call multiple *FunctionCalling* middleware functions.

Code path:
`/ChatClientFunctionCallingMiddleware/Program.cs`

Listing 9.12 FunctionCalling Middleware – Program.cs

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .UseFunctionInvocation(loggerFactory: null, configure: options =>
  {
    options.FunctionInvoker = async (context, ct) =>
      await Middleware.ChatClientFunctionCallings
        .ConstrainDistance(context, ct)
          ?? await Middleware.ChatClientFunctionCallings
        .AuditFunctionCalling(context, ct);
  })  //❶
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
```

```
    Instructions = """
      You are the MotorsAgent controlling a robot car.
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
      """,
    Tools = [.. MotorTools.AsAITools()],  //❷
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt = " Move forward 10 meters then go backward 8 meters";
Console.WriteLine($"PROMPT: {prompt}");
var result = await motorsAgent.RunAsync(prompt, session);  //❹
Console.WriteLine($"\nRESULT: {result}\n");
```

❶ **Defines the FunctionCalling layer**
❷ **Assigns the AI tools**
❸ **Declares the agent**
❹ **Runs the agent with a single prompt that triggers "backward"**

Code output:

```
PROMPT: Move forward 10 meters then go backward 8 meters
[08:52:40:833] MOTORS: Forward: 10m
[ChatClient] [FunctionCall] [Audit] Function call 'Forward' [18:52:40] COMPLETED
in 1019ms | Result: moved forward for 10 meters.
[ChatClient] [FunctionCall] [Constrain] Backward distance constrained from 8m to
5m
[08:52:42:830] MOTORS: Backward: 5m
[ChatClient] [FunctionCall] [Audit] Function call 'Backward' [18:52:42] COMPLETED
in 1008ms | Result: moved backward for 5 meters.
[08:52:44:687] MOTORS: Backward: 3m
[ChatClient] [FunctionCall] [Audit] Function call 'Backward' [18:52:44] COMPLETED
in 1007ms | Result: moved backward for 3 meters.
[08:52:46:838] MOTORS: Stop
[ChatClient] [FunctionCall] [Audit] Function call 'Stop' [18:52:46] COMPLETED in
1006ms | Result: stopped.

RESULT: - Forward 10 m
- Backward 8 m
- Stop
```

Forward 10 meters pass through unconstrained. Backward 8 meters, on the other hand, is silently capped to 5 meters by `ConstrainDistance` — the LLM asked for 8, but only 5 were executed. The LLM then call `Backward(3)` again to cover the remaining distance, based on the LLM's plan. The result summary reports the LLM's intended plan (8 meters), not the actual constrained execution — the agent is unaware its argument was modified.

NOTE *Agent* vs. *ChatClient* middleware. At the *Agent* layer, `AuditFunctionCalls` is chainable: it delegates to next function in the pipeline, and it has access to the agent identity via `agent.Name`. At the *ChatClient* layer, it is terminal and anonymous: it calls `InvokeAsync` directly and has no agent context.

The registration order matters. `ConstrainDistance` (argument adjustment) runs first and returns `null` to signal that the pipeline should continue. Only when that check passes does `AuditFunctionCalls` run as the terminal invoker, calling `InvokeAsync`. It is the responsibility of the last middleware function in the chain to call `InvokeAsync`, not every middleware registered functions in the pipeline.

EXERCISE

Modify `ConstrainDistance` function to limit the `forward` distance to `20 meters`.

You should notice an extra adjustment when calling the `forward` tool when the distance is larger than 20 meters.

9.3 Composing the Full ChatClient Middleware Pipeline

Now that we've seen each *ChatClient* middleware type in isolation, here is how they compose into a complete pipeline.

The pipeline calling order works like a stack. As with any pipeline, we can return early and short-circuit execution, or we can keep adding middleware until we reach the top of the stack: the chat client. Registration order equals execution order: whatever we add first runs first.

9.3.1 Stacking SharedFunction, Response, and FunctionCalling

The pipeline calling order works like a stack. As with any pipeline, we can return early and short-circuit execution, or we can keep adding middleware until we reach the top of the stack: the chat client. Registration order equals execution order: whatever we add first runs first.

Figure 9.10 shows how we mix and order the three *ChatClient* middleware types.

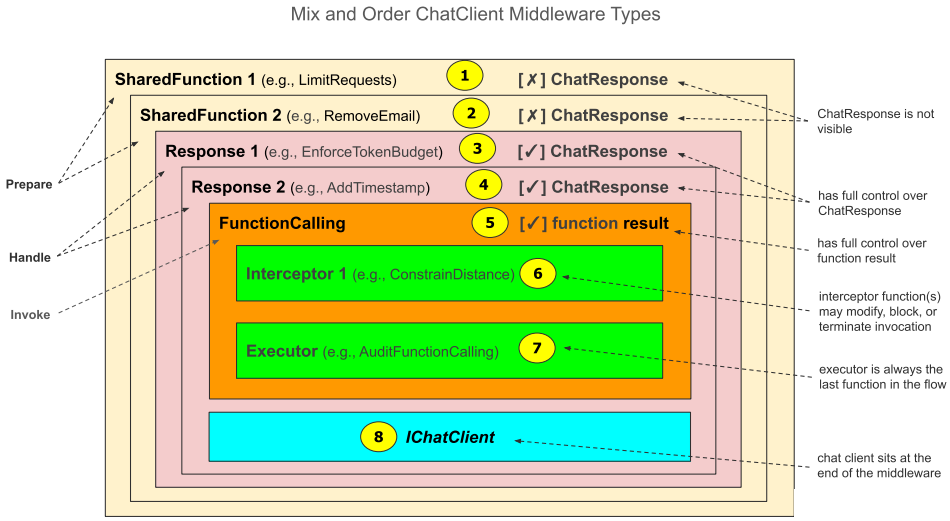

Figure 9.10 All three ChatClient middleware types composing together. SharedFunction functions (outermost, steps 1-2) prepares input, ChatResponse invisible. Response functions (middle, steps 3-4) wrap the LLM exchange, ChatResponse visible and controllable. FunctionCalling (innermost, step 5) intercepts tool calls via interceptor/executor pattern implemented with FunctionCalling functions (steps 6-7). IChatClient (step 8) sits at the end of the middleware pipeline. Registration order equals execution order: prepare → handle → invoke.

ChatClient middleware steps:

1. `SharedFunction 1`: `LimitRequests` starts the pipeline in the *Prepare* stage and can count or limit requests before they reach the chat client; `ChatResponse` is not visible here.
2. `SharedFunction 2`: `RemoveEmail` adds another *Prepare*-stage rule that sanitizes input, such as removing email addresses before the request is sent to the model; `ChatResponse` is still not visible.

3. `Response 1`: `EnforceTokenBudget` enters the *Handle* stage, where `ChatResponse` is visible and token usage or cost limits can be inspected and enforced.
4. `Response 2`: `AddTimestamp` adds another *Handle*-stage middleware that can decorate, inspect, or modify the completed `ChatResponse`.
5. *FunctionCalling* enters the Invoke stage, where middleware works with individual function results instead of the full `ChatResponse`.
6. *Interceptor*: `ConstrainDistance` runs before final function execution and can modify, block, terminate, or short-circuit the function invocation.
7. *Executor*: `AuditFunctionCalling` runs if the interceptor returns null, audits the invocation, and produces the final function result.
8. `IChatClient` sits at the end of the middleware stack, sends the request to the model, and produces the `ChatResponse` that flows back outward through the pipeline.

The recommended order for *ChatClient* middleware types is:

- *SharedFunction* Layer — *Prepare*

  The *SharedFunction* layer is the outermost part of the *ChatClient* middleware pipeline. Its job is to prepare the request before the model sees it. At this point, there is no `ChatResponse` yet, so these middleware functions cannot inspect or modify the final response.

  This makes *SharedFunction* middleware a good fit for input-oriented concerns: request limits, validation, privacy, and prompt/message shaping. In this sample, `LimitRequests` protects the pipeline from too many calls, while `RemoveEmail` prevents sensitive data from reaching the LLM provider.

- *Response* Layer — *Handle*

  The *Response* layer surrounds the actual model call. This is the first layer where `ChatResponse` is visible. Because of that, *Response* middleware is the right place for output-oriented concerns.

  This layer can inspect, decorate, replace, or reject the response. In this sample, `EnforceTokenBudget` uses response information to control cost, while `AddTimestamp` decorates the returned content.

- *FunctionCalling* Layer — *Invoke*

  The *FunctionCalling* layer is the innermost *ChatClient* middleware area. It does not work with the full `ChatResponse`; it works with individual function invocation results.

  This is the correct place for tool-level rules. In this sample, `ConstrainDistance` acts as an *interceptor* that can constrain unsafe tool arguments, while `AuditFunctionCalling` acts as the final *executor* for the function call.

- `IChatClient` — End of the Pipeline

  At the end of the pipeline sits the `IChatClient`. It sends the request to the model and produces the `ChatResponse`, which then flows back outward through the middleware layers.

IMPORTANT The important mental model is: 1) *SharedFunction* middleware prepares the request and then calls the next layer; 2) *Response* middleware handles the completed `ChatResponse`; 3) *FunctionCalling* middleware controls tool invocation, not the whole chat response. 4) `IChatClient` ends the middleware.

Reading the stack from bottom to top gives us the registration order, which becomes execution order in the pipeline.

Applying the Prepare–Handle–Invoke Middleware Pattern

This is the canonical composition for the *ChatClient* middleware types. Each concern is placed where it has the right visibility and the right timing: validation and sanitization before tokens are spent, response control around the model call, and tool supervision only when a function is about to be executed.

Listing 9.13 shows the full pipeline registration following the *Prepare → Handle → Invoke* mental model.

Listing 9.13 Full Pipeline Registration

```
IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientSharedFunctions.LimitRequests)  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❶
  .Use(ChatClientResponses.EnforceTokenBudget, null)  //❷
  .Use(ChatClientResponses.AddTimestamp, null)  //❷
  .UseFunctionInvocation(loggerFactory: null, configure: options =>
  {
    options.FunctionInvoker = async (context, ct) =>
    {
      var response = await Middleware.ChatClientFunctionCallings
        .ConstrainDistance(context, ct)  //❸
          ?? await Middleware.ChatClientFunctionCallings
        .AuditFunctionCalling(context, ct);  //❹
      return response;
    };
  })  //❺
  .Build();  //❻
```

❶ **Prepare layer: fail fast, sanitize input**
❷ **Handle layer: track tokens, enforce budget**
❸ **Interceptor: modify or block unsafe arguments**
❹ **Executor: invoke function and audit**
❺ **Invoke layer: MUST be last before .Build()**
❻ **Builds the ChatClient middleware for the chat client**

When all three middleware types compose correctly, the pipeline reads like a checklist. *SharedFunction* validates and sanitizes before any token is spent. *Response* wraps the LLM exchange and enforces the token budget. *FunctionCalling* controls each tool invocation individually, with the gate pattern ensuring the function executes exactly once. Remove a layer and lose a capability; misorder them and a concern may fire too late to matter.

IMPORTANT In practice, this is what we want from a *ChatClient* pipeline: outer layers that fail fast and sanitize, a middle layer that owns cost and observability, and an inner layer that supervises tools without leaking any of those responsibilities into agent code.

Common Middleware Anti-Patterns

Some common pitfalls in production:

Wrong registration order. Registration order equals execution order. Logging before sanitization means the log captures sensitive data before removal. Always sanitize first:

```
.Use(SharedFunctionMiddleware.RemoveEmail)  //❶
```

```
.Use(ResponseMiddleware.LogMessages, null)  //❷
```

❶ Correct: sanitizes first (e.g., "[REDACTED-EMAIL]")
❷ Then logs sanitized version (e.g., "[REDACTED-EMAIL]")

FunctionCalling is not innermost. `UseFunctionInvocation` must be the last middleware registered before `.Build()`. If placed outermost, its internal tool loop drives repeated calls back through the entire pipeline: budget checks, sanitization, and timing all fire once per tool call instead of once per request.

Firing frequency misunderstanding. SharedFunction and *Response* fire once per chat request, not once per agent run. A single agent run with tool calls sends multiple chat requests (one initial call and one per tool round-trip), so `LimitRequests` can increment multiple times within a single user prompt. Budget middleware depletes faster than the number of user queries suggests.

No ChatClient-layer protection. Relying only on agent middleware means rogue or misconfigured agents can bypass protection. The *ChatClient* layer acts as a universal failsafe because every call, from any agent or plain code, passes through it.

ChatClient middleware operates at the HTTP boundary with no agent context. This makes it ideal for universal concerns like cost control and privacy.

In the next chapter, we explore *Agent* middleware, where we have access to agent identity, session state, and the familiar next delegate pattern for all middleware types, including *FunctionCalling*.

Summary

- Middleware is essential for production agents. Without it, sensitive data leaks, costs spiral, and dangerous operations execute unchecked.
- Two middleware layers serve different purposes. *ChatClient* is the universal safety net for all agents. *Agent* provides per-agent control with session context and identity.
- Three middleware types map to prepare → handle → invoke. *SharedFunction* prepares input (ChatResponse invisible). *Response* handles LLM exchange (`ChatResponse` visible). *FunctionCalling* invokes tools (sees function result).
- Registration order determines execution order. *SharedFunction* outermost (fail fast). *Response* middle (track tokens). *FunctionCalling* innermost (must be last before `.Build()`).
- *FunctionCalling* uses interceptor/executor pattern. Interceptors return null to modify or block without invoking. Executor calls `InvokeAsync()` and returns non-null. One executor, multiple interceptors allowed.
- *FunctionCalling* must be innermost. If not last, its tool loop drives repeated pipeline passes. *SharedFunction* and *Response* fire per tool call instead of per chat request.
- Firing frequency differs by layer. *ChatClient* fires per chat request (multiple times per agent run with tools). Agent fires once per `RunAsync()` call.
- Sanitization differs by layer. *ChatClient* is transient (LLM sees clean data, session retains original). *Agent* is persistent (cleaned messages stored in history).
- Defense in depth requires both layers. Implement critical controls like email removal at both *ChatClient* (universal) and *Agent* (per-agent) layers.

2

Building your first agent step by step

This chapter covers

- Building a simple AI-powered console application with Agent Framework
- Obtaining and securing API keys for OpenAI and Azure OpenAI
- Creating a basic console application for prompting a GPT model
- Creating your first AI agent

Let's dive into practical applications with Agent Framework and see how to build AI-powered apps that use large language models. We'll start with a simple console app that interacts with OpenAI's GPT models, then move on to an agent that can control a robot car. You'll also learn how to obtain and secure API keys, set up your development environment, connect Agent Framework to GPT models, and expose native (C#) functions as tools.

2.1 A Robot Car Story

Imagine a robot car that can sense fire danger and escape independently. While this concept may sound like something out of science fiction, it represents a real pet project that sparked the journey behind this book. The idea of creating an autonomous vehicle capable of detecting and responding to potential fire hazards highlights the fascinating intersection of robotics, sensor technology, and machine learning. This project was built on a very low budget (less than 100 USD), with a focus on exploring how these technologies could run in the .NET ecosystem on an IoT (Internet of Things) device.

2.1.1 A Fire-Detecting Robot Car

The story began several years ago when I decided to build a robotic car as part of my academic exploration of machine learning. This project was meant to power demonstrations for upcoming speeches, not to be a commercially viable product. The primary function of the robot car is to read its surroundings and react to potential fire hazards.

I based the project on a low-cost model, using a Raspberry Pi (a low-power mini-PC with an ARM architecture) and a motor controller add-on board. The motor controller drives the wheels through a software API (Application Programming Interface). While it may not be sophisticated, this setup lets the car move forward and backward, turn left and right, and stop. To improve its awareness of its surroundings, I added a camera and sensors for distance, light, infrared, and temperature.

Next, I developed software to control the robot car and trained a machine learning model to classify inputs into two states: *Fire* and *Safe*. The Fire state indicates danger, while the Safe state signifies the absence of such a threat.

IMPORTANT You don't need a physical robot car to build an agent. The robot car is a visual aid to help illustrate concepts. Your agent will operate entirely in the digital realm.

With this machine learning model in place, the car could make predictions from sensor readings and react accordingly, showing the potential of low-cost robotics to address real-world challenges.

2.1.2 Challenges in Programming Complex Movements

Initially, I programmed basic steps supported by the robot car API into a step-by-step sequence that simulates an escape maneuver. As I assembled these steps, I immediately realized a limitation: this approach doesn't scale well to new movement patterns. Each new movement would require programming a new sequence of steps. An escape maneuver can be coded manually, but more complex patterns—following a circular path, zigzagging, imitating a dance, or avoiding obstacles—quickly lead to more hard-coded, user-defined step sequences. Figure 2.1 suggests how easy it is for a robot car to do simple moves, and how difficult it can become to do complex ones.

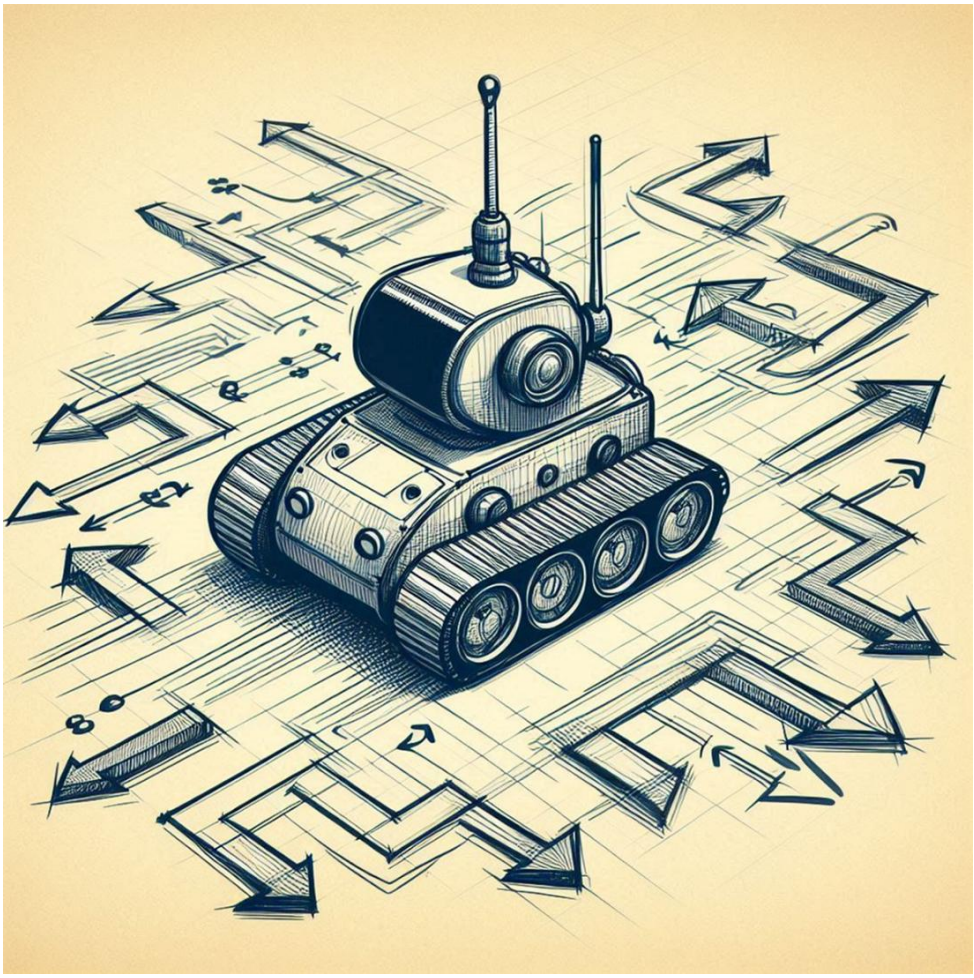

Figure 2.1 The robot car understands five basic commands: forward, backward, left turn, right turn, and stop. Any complex instruction must be broken down into a sequence of these fundamental movements, allowing the vehicle

to execute more sophisticated maneuvers by combining its core functions.

Even though I recognized a key element of this challenge, the semantic link between simple and complex commands, I lacked an elegant solution at the time and had to abandon my efforts to automate motion patterns until a better opportunity arose.

2.1.3 Revisiting the Problem with Generative AI

A few years later, the emergence of generative artificial intelligence reawakened my interest in solving the complex movement decomposition problem. Through experimentation with ChatGPT, I discovered LLMs' remarkable ability to translate complex, and thus more abstract, commands into executable sequences, a capability rooted in their "functional nature." This inspired me to use generative AI as an intermediary layer between the user and the robot's API to create motion models on the fly.

EXPERIMENTATION WITH CHATGPT

Eager to explore this possibility, I experimented with ChatGPT to break down complex movements into basic commands that the robot car API could understand. Here are the findings, which you can also verify using ChatGPT or Copilot.

User prompt:

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn left, turn right, and stop. You have to break down the complex command "There is a tree directly in front of the car. Avoid it and then return to the original path." into basic moves you know.

The prompt is fluent and concise, written in the style we're used to with ChatGPT, so the answer should be coherent and to the point.

ChatGPT response:

Let's break down the complex command into a series of basic moves:

1. **Stop**: If the car is moving, first make it stop.
2. **Turn Left**: Turn the car left to start moving around the tree.
3. **Move Forward**: Move forward a bit to clear the tree.
4. **Turn Right**: Turn the car right to straighten it out.
5. **Move Forward**: Continue moving forward to clear the tree completely.
6. **Turn Right**: Turn the car right again to start moving back towards the original path.
7. **Move Forward**: Move forward a bit to align the car with the original path.
8. **Turn Left**: Turn the car left to return to the original path.
9. **Move Forward**: Continue moving forward along the original path.

This way, the car can avoid the tree and get back on its original path!

If you have any specific parameters like distances or turning angles, I can refine the steps further.

The result was good enough. Much of the conditional logic that had instructed the robot car to move in complex patterns was now delegated to the AI.

Let's run it again!

ChatGPT response:

Absolutely. Let's break it down:

1. **Turn right** to start avoiding the tree.
2. **Move forward** for a short distance to bypass the tree.
3. **Turn left** to align parallel to the original path.
4. **Move forward** to pass the tree.
5. **Turn left** again to face the original path.
6. **Move forward** to return to the original path.
7. **Turn right** to get back on the original path.
8. **Move forward** to continue forward.
9. **Stop** to stop the car.

Feel free to ask if you need further assistance or have other tasks for the robot car!

Whether the model decides to start by turning right or left, the maneuver is pretty much the same. This shows how an LLM can introduce random variation in its responses.

Sometimes, it's much easier to grasp these steps when you can visualize them, so I created a simple diagram. See figure 2.2 for a clearer picture of the robot car's journey around the tree.

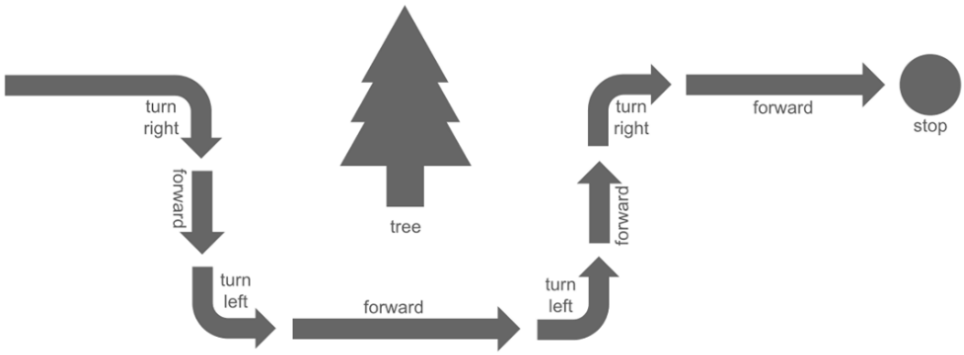

Figure 2.2 The LLM generates a textual response that details the car robot's movements around the tree, but visualizing this sequence as a diagram makes the detour easier to understand.

These steps clearly describe a path around the tree to avoid an imminent collision.

THE NON-DETERMINISTIC NATURE OF LLMS

The result from ChatGPT was promising, delegating much of the conditional logic for complex movement patterns to AI. Resending the same prompt reveals a well-known limitation: LLMs are non-deterministic and can produce varied outputs for identical inputs. While this flexibility is valuable for creative applications, machine-to-machine communication needs more predictable responses.

This functional decomposition stems from key architectural features of LLMs:

- *Task decomposition through pattern recognition*: LLMs can semantically interpret instructions such as "avoid the tree" and map them to sequences of actions. The model captures the intent behind the command and produces step-by-step outputs grounded in learned semantic patterns. This creates a key link between complex commands and basic moves.
- *Contextual Understanding Through Attention*: Although LLMs are inherently stateless, transformer-based attention mechanisms let them approximate conversational memory. By attending to earlier tokens within the context window, they can adapt later reasoning or actions based on prior information.
- *Structured Output with Prompt Engineering*: Through carefully designed prompts, LLMs can be guided to produce structured, machine-readable outputs such as JSON or XML. Methods like the *PACT framework* help enforce schema alignment and output consistency.

This example shows functional translation from natural language into structured data, helping integration with robotic control systems. With this paradigm shift, large language models are no longer simple text generators; they become functional components that link human instructions to system execution.

Despite discovering these LLM capabilities, I still wasn't convinced to return to my old problem of decomposing complex motions into basic moves. There were no .NET libraries available yet, and working directly with the OpenAI API wasn't appealing.

2.1.4 The Spark that Ignites My Robot Car

A few weeks later, during a discussion with a colleague who shared my passion for .NET, I was asked whether I had started using Microsoft Semantic Kernel for C#. I had seen a few posts about this new tool, but that conversation sparked a deeper exploration. What I discovered transformed my approach to the robot machine project: Semantic Kernel provided the missing link between the semantic understanding of generative AI and the deterministic execution of the system, which is what I needed to solve the motion decomposition problem.

Since Semantic Kernel joined forces with AutoGen to create the newer, more powerful Agent Framework, we'll use Agent Framework for all subsequent examples.

2.2 Acquiring the API Keys

Let's look at how to transform the original ChatGPT prompt, through a series of incremental steps, into an agent capable of communicating semantically with the robot-car API commands. It's worth noting that working with advanced GPT models is not as straightforward as you might assume. These powerful language models aren't freely accessible; instead, they're hosted as services provided by OpenAI or Azure OpenAI. Access typically comes with usage-based and operational costs, which can vary by usage and by the service plan you choose.

2.2.1 How to Get an OpenAI API Key

OpenAI is a pioneer in generative AI, known for its GPT family of language models. Many of the innovations we've seen came first from OpenAI. To use OpenAI's capabilities, you need an API key, which lets you integrate AI features into your applications. Next, we'll walk through how to get an OpenAI API key.

2.2.2 How to Get an Azure OpenAI API Key

Azure OpenAI is Microsoft's implementation of OpenAI's powerful language models and AI technologies, integrated into the Azure cloud platform. Obtaining an Azure OpenAI API key may take longer than obtaining the OpenAI API key described in the previous step because it involves two steps: creating an Azure account and requesting access to the Azure OpenAI service. For details on creating an Azure account and requesting Azure OpenAI service access, see appendix A, section *A.2 Creating an Azure OpenAI account and obtaining an API key.*

2.2.3 Securing API Keys

In .NET, you can store API keys securely using several best practices and options:

ENVIRONMENT VARIABLES

1. Store API keys as environment variables on the server or development machine.

   For Windows:

```
setx OPENAI_API_KEY "YOUR_OPENAI_API_KEY"
```

   For Linux:

```
export OPENAI_API_KEY="YOUR_OPENAI_API_KEY"
```

2. Access environment variables in code by reading the corresponding OpenAI key.

```
var apiKey = Environment.GetEnvironmentVariable("OPENAI_API_KEY");
```

The API key is retrieved from environment variables.

SECRET MANAGER

1. Use the Secret Manager tool provided by .NET for development only. This tool helps store secrets outside the application's folders, preventing accidental commits to Git or other repositories. Secret Manager stores secrets unencrypted in a JSON file within the user's profile folder, making it suitable for local development but not for production, so never use it in production.
2. Store secrets locally using the `dotnet` tool.

```
dotnet user-secrets init
dotnet user-secrets set "OpenAI:ApiKey" "api-key"
```

   `init` creates the secret file, `set` sets the values.

3. Build and access the secrets through the `IConfiguration` interface.

```
var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>()
  .Build();
var apiKey = configuration["OpenAI:ApiKey"];
```

The API key is programmatically retrieved from the local secrets store.

More techniques to deal with API keys

We've covered several methods for handling API keys, but production environments often call for more advanced techniques, including:

Azure Key Vault: A cloud service for securely storing and accessing secrets, including API keys.

AWS Secrets Manager: A service that helps protect access to your applications, services, and IT resources without needing to hardcode sensitive information.

Database Storage: Store encrypted API keys in a secure database and access them only when needed.

Containerized secrets management: Tools such as Docker Secrets or Kubernetes Secrets for managing sensitive data in containerized environments.

Advanced API key management techniques are critical in production environments because they enhance security through encryption and access control, improve scalability through centralized management, ensure compliance with audit trails, and integrate seamlessly with DevOps practices. These methods significantly reduce security risks in large-scale, complex applications.

While these solutions offer robust security for API key management in production settings, a detailed discussion of how to implement them is beyond the scope of this book. For most applications, the methods discussed earlier should provide sufficient security when implemented properly.

2.3 Your First AI Agent

Ready to return to the problem of decomposing complex movements into basic movements that the robot car API can understand? Agent Framework makes it easy to integrate into a .NET console application. We'll start with a simple example and gradually enhance it to create a dynamic, interactive AI assistant for a robot car. You'll have an agent that can break down complex movements into basic commands, and you'll be eager to expand its capabilities even further.

Open Visual Studio Code to create a new console application and add the necessary NuGet packages. If you prefer a more full-featured IDE, you can use Visual Studio instead. Open a terminal and run the commands shown in listing 2.1. Note that the package versions aren't specified because the package names are expected to remain consistent throughout future releases, ensuring compatibility with newer versions.

Listing 2.1 Creating a new console application – Terminal

```
dotnet new console -n RobotCarAssistant  ❶
cd RobotCarAssistant  ❷
dotnet add package Microsoft.Agents.AI  ❸
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

❶ Creates a new console application
❷ Navigates to the newly created folder and get ready to add some packages
❸ Adds Agent Framework library dependency

This approach ensures your project is set up correctly, with the required dependencies for working with Agent Framework and securely managing secrets during development.

2.3.1 Crafting a Good Prompt

I know you're eager to send your first prompt to an LLM from your C# application, but I encourage you to have a bit of patience. When working with generative AI, nothing is more important than crafting a well-designed prompt. Before jumping into implementation, it's beneficial to experiment with tools like ChatGPT, Copilot, or any GPT-powered sandbox to refine your prompts. As shown in section 2.1.3 with the robot car AI assistant prompt, an unrefined prompt can lead to inconsistent or suboptimal results. Structuring and improving prompts are essential to guide the model toward accurate, reliable responses.

Structuring a prompt into distinct parts enhances a model's ability to understand and execute tasks by breaking the prompt into specific components, making it easier to interpret and generate effective responses. It also helps separate the original input into key elements such as *user messages*, *system messages*, and *chat message history* for maintaining context. Message types and chat history are covered in more detail in later chapters.

PACT (Persona-Action-Context-Template) Prompt Engineering Framework

To create effective prompts, I use the PACT framework, my adaptation of the Role-Task-Format (RTF) framework. PACT is for Agent Framework because it aligns with its modular, structured approach. The framework consists of four components:

Persona (Who): Defines the role or identity the model should assume when generating a response. For example, you might ask the model to act as a teacher, a journalist, or an AI assistant.

Action (What): Specify the task or action you want the model to perform. Clear instructions guide its behavior (for example, "write an article" or "break down a command").

Context (With What): Provides relevant background information or details that help the model generate accurate, contextually appropriate responses.

Template (How): Defines the desired structure or format of the response, such as JSON arrays, tables, or plain text.

This structured approach makes PACT easy to use and highly effective for crafting precise, reliable prompts. For more complex scenarios, PACT can be enriched with few-shot learning and zero-shot chain-of-thought techniques.

Let's revisit the first unstructured prompt and its generated response, then analyze its structure. By focusing on the text's semantics, we can identify three main components:

1. *The Persona* (or role or actor): "You are an AI assistant controlling a robot car...", highlighted in light gray.
2. *The Action* (or task): "You have to break down complex commands..."—that is, the unhighlighted text.
3. *Context*: "There is a tree directly in front of the car...", highlighted in dark gray.

Let's analyze the user prompt:

User prompt:

You are an AI assistant controlling a robot car that can perform basic moves: forward, backward, turn left, turn right, and stop. Break down the complex command "There is a tree directly in front of the car. Avoid it and then return to the original path." into the basic moves you know.

While this prompt provides all the necessary information in a single block of text, it lacks a clear structure. By applying *PACT* principles (Persona, Action, Context, and Template) and separating key elements into distinct prompt parts, we can improve clarity and guide the model more effectively.

Our analysis of the original prompt identified the Persona, Action, and Context components of PACT, but it was missing a crucial element: the Template. The Template part of PACT specifies the desired output format, which helps guide the model to produce a structured, consistent response.

We can improve that by adding something like "Use a JSON array for the response" to the prompt to force structured output. This provides an immediate benefit: asking for a JSON response promotes more deterministic output. JSON's strict structure guides the model to follow specific syntax and key-value arrangements, reducing variability and leading to a more deterministic response.

The user prompt looks like this:

User prompt:

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn left, turn right, and stop.

You have to break down the provided complex commands into the basic moves you know.

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path.

Use a JSON array for the response.

Requesting the response in JSON format makes it easier for Agent Framework to parse the output into a native type. This is key to seamlessly integrating AI-generated content into your application's workflow.

For learning purposes, we can keep the labels for the prompt parts (Persona, Action, Context, Template) in the prompt. Note, though, that explicitly labeling these components in the prompt isn't necessary.

Let's look at the refined user prompt in the PACT framework, with the prompt-part labels:
User prompt:

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path."

Template

Use a JSON array for the response.

Let's look at the generated response:

ChatGPT response:

Here's a breakdown of the complex command into basic moves for the robot car:

```json
```

```
[
 {"move": "turn left"},
 {"move": "forward"},
 {"move": "turn right"},
 {"move": "forward"},
 {"move": "turn right"},
 {"move": "forward"},
 {"move": "turn left"}
]
```

These steps should help the robot car avoid the tree and return to its original path.

If we run the prompt again, we'll get similar responses. Try running it a few times to see how similar they are.

ChatGPT response:

We can break the complex command down into basic moves:

```json
[
 {"action": "stop"},
 {"action": "turn right"},
 {"action": "forward"},
 {"action": "turn left"},
 {"action": "forward"},
 {"action": "turn left"},
 {"action": "forward"},
 {"action": "turn right"}
]
```

I've broken down the complex command into basic moves for the robot car. The car will stop, turn right to avoid the tree, move forward to pass it, then turn left and continue forward in the original direction.

We're making substantial progress. The JSON part of the response is what we want, but most of the time it's wrapped in explanatory text. Let's clean that up by adding a new rule: "Respond only with

the moves, without any additional explanations.” This will instruct the model to return only the JSON output.

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree in front of the car. Avoid it, then return to the original path."

Template

Use a JSON array for the response.

Respond only with the moves, without any additional explanations.

Response:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

It looks much better: the JSON content is now isolated. But let’s not celebrate yet! Run it a few more times (as I advised) to see how stable the response is:

ChatGPT response:

```
[
 "turn right",
 "forward",
 "turn left",
 "forward",
 "turn left",
 "forward",
 "turn right"
]
```

Almost there! The result is now cleanly formatted as a JSON array, without unnecessary explanatory text. During testing across multiple runs, some variation in output formatting may still occur. Sometimes, items in the JSON array are key-value pairs, and even the key names can vary. Other times, the JSON response is formatted on a single line or split across multiple lines.

TIP Include explicit instructions in the prompt, such as "`Respond only with JSON with properties: name, age, profession`", or "`Respond only with JSON arrays like [item1, item2, item3]`", or like `[key1:value1, key2:value2, key3:value3]`.

Let's fix this by providing a formal schema: "`Use a JSON array like [move1, move2, move3] for the response.`"

User prompt:

Persona

You are an AI assistant controlling a robot car capable of performing basic moves: forward, backward, turn left, turn right, and stop.

Action

You have to break down the provided complex commands into the basic moves you know.

Context

Complex command: "There is a tree directly in front of the car. Avoid it and then return to the original path."

Template

Use a JSON array like [move1, move2, move3] for the response.

Respond only with the moves, without any additional explanations.

Let's see what the response is now:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

I ran it a few times, and it's obvious that the response becomes more stable and adheres nicely to the desired JSON schema.

While I recommend the PACT framework for its clarity, modularity, and Agent Framework readiness, other prompt engineering frameworks, such as *RTF* (Role, Task, Format), can also be effective, depending on the task and context.

2.3.2 Structuring Prompts for Reusability and Scalability

The core of Agent Framework is the agent. The agent's `Instructions` property defines the agent's persona, a foundational element that remains constant and helps characterize the agent. This property

acts as the system message. Breaking down the prompt into semantic parts (a method introduced earlier with the PACT prompting framework) is especially beneficial here.

Let's decompose the prompt into smaller semantic parts, as shown in figure 2.3.

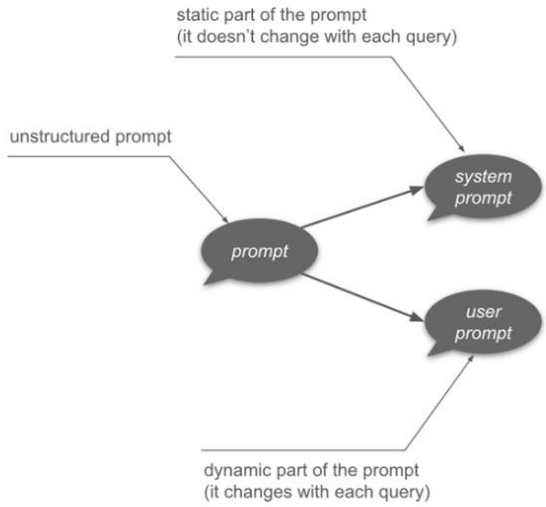

Figure 2.3 A prompt split into user prompt (the dynamic part that changes with each new interaction with an AI model) and system prompt (the static part that doesn't change during interaction with an AI model)

On one hand, the static parts—such as Persona, Actions, and Output Template—aren't expected to change when you send multiple queries to the model. On the other hand, Context is the dynamic part of the prompt and is expected to change with each prompt. The static (or general, if you prefer) parts go into the system prompt (a.k.a. instructions in the agents' world), and the dynamic part goes into the user prompt.

NOTE To help distinguish the components of a prompt, such as the user prompt, system prompt, and instructions, I use the term *prompt* for the dynamic portion (the user prompt). The static portion (the system prompt) corresponds to *instructions*, and I'll keep this convention throughout the book.

Now that we've split the well-structured prompt into a prompt and instructions, we're ready to use them in our first Agent Framework application.

2.3.3 Building a Stateless Agent with Agent Framework

When creating a new application with Agent Framework, the first step is to choose an AI provider and create a chat client. The provider might be OpenAI, Azure OpenAI, Ollama, or others. Most providers require a private API key for authentication. It's crucial to understand that these keys are often tied

to your account's billing system. If someone intercepts your key, they could misuse it and run up charges against your account. You don't want that.

There are several ways to securely manage sensitive information like API keys. For development purposes, I recommend using the .NET Secret Manager tool because it's straightforward to implement and maintain. We'll use it in examples in this chapter and the next. Although Secret Manager is convenient during development, it isn't suitable for production environments because of its limited security features.

In production, use more secure options such as environment variables, Azure Key Vault, or AWS Secrets Manager. These tools manage sensitive information in live environments and help protect it from unauthorized access. If you want to use these methods during development, adapt the examples as needed. The right choice depends on your project requirements and deployment environment.

IMPORTANT Never underestimate the importance of protecting API keys. Strong security measures help prevent accidental exposure. Common risks include editing solution files, committing keys to Git, and showing them during screen sharing or code demos. Exposed API keys can lead to unauthorized access, financial losses, and compromised data. Use secure methods such as environment variables or secret-management tools to handle API keys.

Now that we have the necessary setup, agent instructions, and prompt, let's start coding our console application to demonstrate how to use the OpenAI API to break complex movements into basic commands the robot car API can understand.

Step-by-step explanation:

1. Namespaces: Import the `Microsoft.Extensions.Configuration` namespace to manage user secrets, `Microsoft.Agents.AI` to work with agents, and `OpenAI` to initialize the OpenAI services provider.
2. Configuration setup: Uses `ConfigurationBuilder` to load user secrets, a secure way to store sensitive information such as API keys.
3. OpenAI Client Initialization: An `OpenAIClient` instance is created using the model ID and API key stored in user secrets. This chat client will handle the agent's internal communication with OpenAI services.
4. Agent initialization: Creates an AI agent using the previously created chat client with the provided instructions.
5. Prompt (User Prompt) Definition: The prompt is declared using a raw string literal. This feature is particularly useful because it allows more readable and maintainable multiline strings without escape characters, especially double quotes. The prompt string defines the task, or the goal we expect the agent to complete; more specifically, it instructs the AI model to control a robot car using basic moves.
6. Prompting the API. The `RunAsync` method is called with the prompt argument, sending a request to OpenAI for a chat completion.
7. Output: The `AgentResponse` text response from the AI model is printed to the console.

Open the `Program.cs` file and replace its contents with the following code (listing 2.2):

Listing 2.2 Agent sends a prompt to an AI model

```
using Microsoft.Extensions.Configuration;  //❶
```

```
using Microsoft.Agents.AI;  //❷
using OpenAI;  //❷

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>()
  .Build();  //❸

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❹

AIAgent agent = chatClient.AsAIAgent("""
  ## Persona
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.

  ## Action
  You have to break down the provided complex commands into the basic
  moves you know.

  ## Template
  Use a JSON array like [move1, move2, move3] for the response.
  Respond only with the moves, without any additional explanations.
  """;  //❺

var prompt = """
  ## Context
  Complex command:
  "There is a tree directly in front of the car. Avoid it and then
  return to the original path."
  """;  //❻

AgentResponse response = await agent.RunAsync(prompt);
Console.WriteLine(response.Text);  //❼
```

❶ Imports namespace for reading the user secrets
❷ Imports namespaces for working with OpenAI provider
❸ Reads the secrets to configuration object
❹ Creates the chat client
❺ Creates the AI agent using the chat client and the instructions
❻ Declares the prompt
❼ Prints the text response to the console

Let's run the code. Open the Terminal and run the following command.

```
dotnet run
```

Are you getting a response like this?

Assistant:

ChatGPT response:

```json
 ["turn right", "forward", "turn left", "forward", "turn left", "forward", "turn right"]
```

If your response is similar, everything is fine because you just replicated what ChatGPT does when a user sends a prompt to the model, except this time you did it through the OpenAI API with an AI agent running in a console application.

I encourage you to try a few prompt variations to see how large language models "understand" the intent behind a user prompt.

Try these queries (one at a time!):

- a straightforward command for basic movements:

```
"Go like: turning left, forward, turning right, backward, stop"
```

- This relies on semantic understanding of geometric paths:

```
"Go on a semi-circle"
```

- This relies, again, on semantic understanding of geometric paths:

```
"Go on a square path"
```

- The word "randomly" introduces variability in the output:

```
"Go 10 steps where each step is a randomly selected step"
```

- a longer command combining multiple steps into a sequence:

```
"Do a full circle by turning left, followed by a full circle by turning
right"
```

- This challenges the LLM to understand the initial position and navigate back to it:

```
"Move forward, turn left, forward, and return to the same place where it
started"
```

- Let's get creative:

```
"Do the moonwalk dancing"
```

- Let's look at some stops and moves that mimic jellyfish movement:

```
"Move like a jellyfish"
```

2.3.4 Troubleshooting the application

There are relatively few things that can go wrong with the previous example. If problems arise, they're most likely related to API key management or endpoint configuration. Double-check these elements before running your application to minimize problems. Mishandling API keys or endpoints can result in errors such as `ClientResultException`. Error descriptions are usually self-explanatory and help you solve the problem (table 2.1):

Table 2.1 Troubleshooting the Application

ClientResultException	Description
`HTTP 401 (invalid_request_error: invalid_api_key)`	Access denied due to invalid subscription key or wrong API endpoint. Make sure to provide a valid key for an active subscription and use a correct regional API endpoint for your resource.
`HTTP 404 (invalid_request_error: model_not_found)`	The model `…` does not exist or you do not have access to it
`HTTP 429 (insufficient_quota: insufficient_quota)`	You exceeded your current quota, please check your plan and billing details. For more information on this error, read the docs: https://platform.openai.com/docs/guides/

Hopefully, everything worked as expected! If so, congratulations on completing this example of a generative AI application using Agent Framework. Whether this is your first experience with Agent

Framework or you're already familiar with it, this marks an important first step toward building more advanced AI-driven applications.

2.3.5 Building a stateful agent with memory

The code sends a prompt to the model and then prints the result. This behavior is stateless and follows a request-response pattern, which works well when we need the model to respond to a single request. Simply put, it doesn't remember anything from previous messages. What if we need to simulate an agent that remembers?

Let's turn our sample into an interactive agent that recalls conversation history (see listing 2.3) by adding the `AgentSession` object, which stores the message history.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Listing 2.3 Agent sends a prompt to an AI model using AgentSession

```
using Microsoft.Extensions.Configuration;  //❶
using Microsoft.Agents.AI;  //❷
using OpenAI;  //❷

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❸

AIAgent agent = chatClient.AsAIAgent("""
  ## Persona
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.

  ## Action
  You have to break down the provided complex commands into the basic
  moves you know.

  ## Template
  Use a JSON array like [move1, move2, move3] for the response.
  Respond only with the moves, without any additional explanations.
  """);  //❹

AgentSession session = await agent.CreateSessionAsync();  //❺

while (true)
{
  Console.Write("User: ");
  var input = Console.ReadLine();  //❻
  if (string.IsNullOrEmpty(input)) break;  //❼

  var prompt = $"""
    ## Context
    Complex command:
    "{input}"
    """;  //❽

  var response = await agent.RunAsync(prompt, session);  //❾
  Console.WriteLine($"Assistant: {response.Text}");  //❿
}
```

```
Console.WriteLine("\nHistory:");

if (session is ChatClientAgentSession
  { MessageStore: not null } chatSession)  //⓫
{
  IEnumerable<ChatMessage> history = await chatSession
    .MessageStore.GetMessagesAsync();  //⓬

  foreach (ChatMessage message in history)  //⓭
  {
    Console.WriteLine($"{message.Role}: {message.Text}");
  }
}
```

❶ Imports namespace for reading the user secrets
❷ Imports namespaces for working with Agent Framework agents OpenAI provider
❸ Reads the secrets to configuration object
❹ Creates the chat client
❺ Creates the AI agent using the chat client and the instructions
❻ Reads the input from user
❼ If the user inputs empty string, the loop stops
❽ Wraps the input into a prompt
❾ Agent queries the AI service with the session for a response
❿ Prints the text response to the console
⓫ Checks if session is chat client agent session with message store
⓬ Reads the messages from the chat session message store
⓭ Iterates each message for printing to console

This enhanced version creates an interactive experience where the agent remembers previous interactions and maintains context throughout the conversation. Notice how the session object is built and passed back to the agent.

When you run the code from listing 2.3, your agent will remember every previous interaction, as shown in listing 2.4. For example:

Listing 2.4 Console output from code in Listing 2.3

```
User: There is a tree directly in front of the car.
Avoid it and return to the original path.
Assistant: ["turn right", "forward", "turn left", "forward", "turn left",
"forward", "turn right"]
User: Now move forward two times.
Assistant: ["forward", "forward"]
User:

History:
user: ## Context
Complex command:
"There is a tree directly in front of the car.
Avoid it and return to the original path."
assistant: ["turn right", "forward", "turn left", "forward", "turn left",
"forward", "turn right"]
user: ## Context
Complex command:
"Now move forward two times."
assistant: ["forward", "forward"]
```

Congratulations! You've just created an agent that can intelligently break down complex movements for a robot car and that remembers previous interactions!

2.3.6 Streaming Agent Responses

To capture the essence of an interactive agent, we need to implement response streaming. This feature enhances the user experience and improves perceived response speed. With streaming, users

can start reading the AI's response as it's generated, instead of waiting for the entire message to complete.

Let's upgrade our agent from listing 2.3 with this streaming capability. Locate this line:

```
AgentResponse response = await agent.RunAsync(prompt, session);
Console.WriteLine($"Assistant: {response.Text}");
```

And replace it with:

```
Console.Write($"Assistant: ");
await foreach (AgentResponseUpdate update in agent
  .RunStreamingAsync(prompt, session))
{
  Console.Write(update.Text);
}
Console.WriteLine();
```

Run the application and enter the following user prompt at the console:

```
There is a tree directly in front of the car.
Avoid it and then come back to the original path.
```

The response should be like what we saw earlier, but streaming responses improve user experience by reducing perceived latency, since users see the AI's reply almost immediately. This creates a more natural, interactive flow that mimics human conversation.

IMPORTANT When responding with structured information, streaming is less effective because you need the full response to ensure clarity and coherence. In those cases, a non-streaming response is more suitable.

While our agent can now hold a conversation and maintain context, it's still a passive system. It responds to user input, but it doesn't take any independent actions. Next, we'll enhance the agent's capabilities by enabling it to directly control the robot car based on its understanding of the conversation.

2.3.7 Building a Tool Agent with Actions

While having a responsive agent is impressive, the real power lies in taking action based on AI's responses. This lets the vehicle make decisions more independently, rather than relying on preprogrammed handling of the model's output. We can use Agent Framework's ability to automate actions as the model generates responses, enabling the agent to execute those actions based on its language understanding.

DEFINING TOOLS WITH NATIVE FUNCTIONS

Native functions let you define custom tools that agents can use to interact with your application's logic or data. A tool is typically a native function (conventional code) that's wrapped and registered so an agent can invoke it as needed.

Listing 2.5 shows the `MotorTools` class, which shows how to create tools from native C# methods:

Listing 2.5 Motor AI tools class

```
using Microsoft.Extensions.AI;  //❶
using System.ComponentModel;

public static class MotorTools  //❷
{
  private const int Delay = 1000;  //❸
```

```
  static public IEnumerable<AITool> AsAITools()  //❹
  {
    yield return AIFunctionFactory.Create(TurnRight);
    yield return AIFunctionFactory.Create(TurnLeft);
    yield return AIFunctionFactory.Create(Stop);
    yield return AIFunctionFactory.Create(Forward);
    yield return AIFunctionFactory.Create(Backward);
  }

  [Description("Basic command: Moves the robot car backward.")]
  public static async Task<string> Backward(
    [Description("The distance (in meters) backward.")] int distance)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Backward:
{distance}m");
    await Task.Delay(Delay);  //❺
    return $"moved backward for {distance} meters.";
  }

  [Description("Basic command: Moves the robot car forward.")]
  public static async Task<string> Forward(
    [Description("The distance (in meters) to move the robot car forward.")]
    int distance)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Forward:
{distance}m");
    await Task.Delay(Delay);
    return $"moved forward for {distance} meters.";
  }

  [Description("Basic command: Stops the robot car.")]
  public static async Task<string> Stop()
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: Stop");
    await Task.Delay(Delay);
    return "stopped.";
  }

  [Description("Basic command: Turns the robot car anticlockwise.")]
  public static async Task<string> TurnLeft(
    [Description("The angle (in degrees) to turn the robot car anticlockwise.")]
    int angle)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: TurnLeft:
{angle}°");
    await Task.Delay(Delay);
    return $"turned anticlockwise {angle}°.";
  }

  [Description("Basic command: Turns the robot car clockwise.")]
  public static async Task<string> TurnRight(
    [Description("The angle (in degrees) to turn the robot car clockwise.")]
    int angle)
  {
    Console.WriteLine($"[{DateTime.Now:hh:mm:ss:fff}] MOTORS: TurnRight:
{angle}°");
    await Task.Delay(Delay);
    return $"turned clockwise {angle}°.";
  }
}
```

❶ Imports the namespace for AITool class used in user-defined AsAITools method
❷ Defines the MotorTools class that keep the domain-specific functions together
❸ Declares a property that simulates 1000 milliseconds delay
❹ Defines a method that returns the native functions as tools
❺ Introduces a delay to simulate an action that takes time to execute

Native functions in Agent Framework use method annotations such as `DescriptionAttribute`. Classes and parameters also have their own descriptions. In generative AI, these semantic descriptions are as crucial as syntax is in traditional programming.

IMPORTANT If no description is provided for function names or arguments, Agent Framework uses the names themselves by default. This is a compact way to inform the AI model about their purpose. Clear, meaningful naming conventions are still essential, as the model relies heavily on these names when selecting and invoking functions.

Advanced large language models use these descriptions (or, if descriptions aren't provided, their names) to:

- understand a function's purpose
- use functions correctly
- identify appropriate functions to call.

Comprehensive descriptions of native functions in Agent Framework are essential for effective AI interactions because large language models (LLMs) choose whether to call functions based on context.

When LLMs prepare a response, they may decide to call none, one, or multiple functions associated with the prompt to provide a more accurate, informed response, depending on the agent setting. For example, if we ask the model about the current weather in a specific location, it likely doesn't have real-time weather data. Without access to relevant functions, it might hallucinate a response or state that it can't provide accurate information. But when the prompt includes well-described functions, the LLM can fill information gaps by calling the right ones.

I propose the term *semantic signature* to describe this set of contextual conventions, analogous to a traditional method's syntax signature, which defines its name, parameters, and data types. A semantic signature extends this notion by capturing a function's meaning, intent, and capability, so the AI can use it more intelligently in suitable contexts.

IMPORTANT Semantic signatures capture both the structural and descriptive details of a function, encompassing its name, arguments, and descriptions. This concept is crucial because it gives an AI model a deeper understanding of each function's intent and capabilities. As a result, AI models can select functions and Agent Framework can invoke them more intelligently and contextually, improving reliability and enabling more advanced orchestration in AI-driven solutions.

While LLMs can decide whether to call functions based on context, the underlying decision process varies across implementations and model architectures.

IMPORTING AI TOOLS FROM A CLASS

To build an agent with AI tools, use `Tools` property:

Listing 2.6 Create a tool agent

```
var agent = chatClient.AsAIAgent("""
  instructions: """
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.
```

```
  You have to break down the provided complex commands into the basic
  moves you know.
  Respond only with the moves, without any additional explanations.
  """,
  tools: [.. MotorTools.AsAITools()]
);
```

With these enhancements (history, streaming, and tools), we've transformed our agent into a stateful, tool-using agent. It can now:

- Engage in natural language conversations
- Stream responses for more fluid interaction
- Access external data and services
- Execute real-world actions through native functions

This foundation opens exciting possibilities. Imagine expanding this system to control various aspects of a smart home, assist in complex data analysis, or even guide autonomous systems through challenging scenarios.

BRINGING ALL CHANGES TOGETHER

Listing 2.7 shows the full code for creating an agent with tools. It uses the class defined already in listing 2.5.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Listing 2.7 Agent sends a prompt to an AI model using tools

```
using Microsoft.Agents.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;
using AITools;

var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];
var chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model);  //❶

AIAgent agent = chatClient.AsAIAgent("""
  You are an AI assistant controlling a robot car capable of performing
  basic moves: forward, backward, turn left, turn right, and stop.
  You have to break down the provided complex commands into the basic
  moves you know.
  Respond only with the moves, without any additional explanations.
  """,  //❷
  tools: [..MotorTools.AsAITools()]  //❸
);

AgentSession session = await agent.CreateSessionAsync();

while (true)
{
  Console.Write("User: ");
  var input = Console.ReadLine();  //❹
  if (string.IsNullOrEmpty(input)) break;  //❹

  var prompt = $"""
    Complex command:
```

```
    "{input}"
    """;  //❺

  AgentResponse response = await agent.RunAsync(prompt, session);
  Console.WriteLine($"Assistant: {response.Text}");  //❻
}
```

❶ **Creates the chat client**
❷ **Assigns the tools from MotorTools**
❸ **Creates the AI agent using the chat client, instructions and tools**
❹ **Reads the input from user until the input is empty**
❺ **Wraps the input into a prompt**
❻ **Prints the text response to the console**

Run the code. The output should look like this:

Listing 2.8 Console output for listing 2.7

```
User: There is a tree directly in front of the car.
Avoid it and then return to the original path.
[06:01:43:339] MOTORS: TurnRight: 90°
[06:01:44:361] MOTORS: Forward: 5m
[06:01:45:366] MOTORS: TurnLeft: 90°
[06:01:46:369] MOTORS: Forward: 10m
[06:01:47:372] MOTORS: TurnLeft: 90°
[06:01:48:378] MOTORS: Forward: 5m
[06:01:49:384] MOTORS: TurnRight: 90°
Assistant: The robot car successfully avoided the tree and returned to the
original path.
```

Notice the tool responses (for example, `MOTORS: TurnRight: 90°`) and the assistant response produced by the AI model. Each step in the sequence triggered the corresponding tool from `MotorTools`, as expected.

IMPORTANT Unlike listing 2.4 that shows the output when there are no tools involved, this response was produced after the agent invoked one or more provided tools to complete the task.

2.4 Conclusion

As we've seen so far, we've only scratched the surface of Agent Framework's capabilities. Our simple console application is the first step in an exciting journey into generative AI development. Agent Framework streamlines interactions with large language models, abstracting away complex API calls and prompt engineering.

Summary

- Motivation for agents and LLMs: a robot car story that links complex commands to low-level device APIs.
- Limitations of hardcoded movement sequences and the need for a more scalable approach to complex robot-car behaviors.
- Use assistants such as ChatGPT or Copilot to translate natural language movement descriptions into executable step sequences for the robot car API.
- An understanding of LLM nondeterminism and its effects on machine-to-machine scenarios that require predictable, structured outputs.
- The PACT prompt engineering framework is a practical method for designing clear, reusable

prompts for control scenarios.

- Techniques for shaping free-form prompts into JSON responses that are easy to parse and integrate into .NET applications.
- Steps for obtaining OpenAI and Azure OpenAI API keys, plus core options for securing them across environments.
- Build your first stateless Agent Framework console agent that sends queries to an LLM and prints structured results.
- Evolving to a stateful agent with `AgentSession` so conversation history and prior context are preserved across interactions.
- Separating static instructions from dynamic queries, and mapping them to Agent Framework concepts for modular, maintainable prompts.

9

Building enterprise-ready agents with ChatClient middleware

This chapter covers

- Why middleware is essential for production AI agents
- The two-layer middleware architecture: ChatClient vs Agent
- Three middleware types: SharedFunction, Response, and FunctionCalling
- Composing a complete middleware pipeline

Robby worked well in development because nothing bad was at stake. In production, the same agent became dangerous: no token limits, no input/output data redaction, and no guard around tool calls. The root problem was architectural; the agent talked straight to the LLM. Low-level *ChatClient* middleware is the caller-agnostic layer between any chat client and the model where you attach shared, organization-wide infrastructure guardrails for all agents without cluttering their logic with cross-cutting concerns. It turns a "works in dev" agent into an enterprise-ready one by enforcing global guardrails for cost, safety, and basic sanitization without rewriting the agent.

9.1 Applying ChatClient Middleware to Agents

Robby's AI agent worked flawlessly in development. Then production happened, and that's when we learned that *middleware*, the layer that sits between our chat calls and the LLM to enforce guardrails, is not optional. It is essential.

If we come from ASP.NET, this should feel familiar. HTTP middleware wraps requests and responses in a pipeline. Here we apply the same idea to chat calls and LLM responses: middleware surrounds every call and handles cross-cutting concerns such as validation, logging, budgets, and safety.

9.1.1 Explaining the Middleware Pipeline Pattern

Middleware looks abstract until you see how a request actually flows through the pipeline. The next two diagrams show the two essential cases you will use throughout this chapter.

HAPPY-FLOW MIDDLEWARE

In the normal case as shown in figure 9.1, every piece of middleware lets the call pass through to the LLM and then sees the response on the way back.

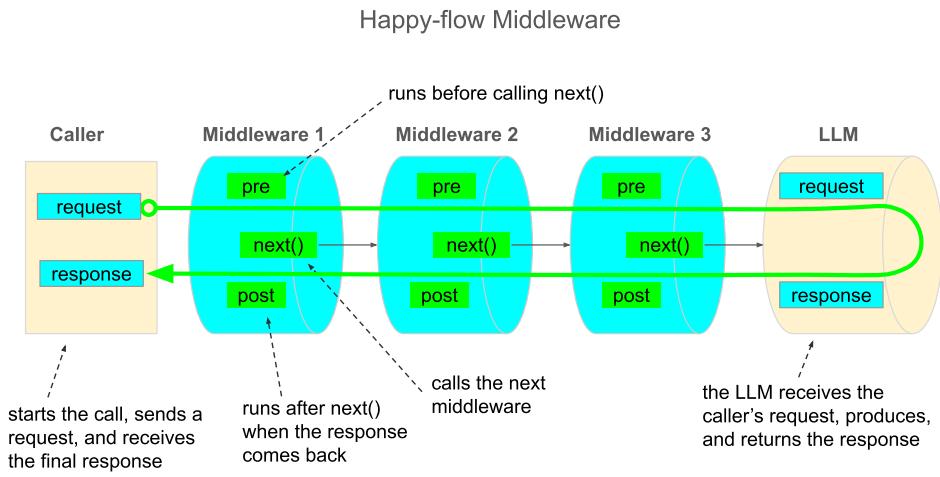

Figure 9.1 The middleware pipeline wraps the LLM call: each middleware runs pre logic, calls next(), then runs post logic as the response returns, so every component executes on both the way in and the way out.

In the happy-flow diagram, the caller starts the operation by sending a request into the pipeline. *Middleware 1* receives that request and runs its pre logic first (e.g., logging, input shaping, or attaching metadata). It then calls `next()`, which passes control to the next middleware in the chain rather than to the LLM directly.

Middleware 2 and *Middleware 3* repeat the same pattern. Each one gets the request, runs its own pre logic, and calls `next()` to move the request further down the pipeline. Eventually the request reaches the LLM, which receives the caller's request, produces an answer, and returns a response.

On the way back, the response travels through the pipeline in reverse order. *Middleware 3* runs its post logic, then returns control to *Middleware 2*, which runs its own post logic, and finally *Middleware 1* does the same before the response reaches the caller. In this happy flow, every middleware in the chain runs on both sides of the call: once before `next()` and once after `next()` completes.

SHORT-CIRCUIT MIDDLEWARE

Sometimes middleware needs to stop the operation early (e.g., when a token budget is exhausted or a user is not authorized) as seen in figure 9.2.

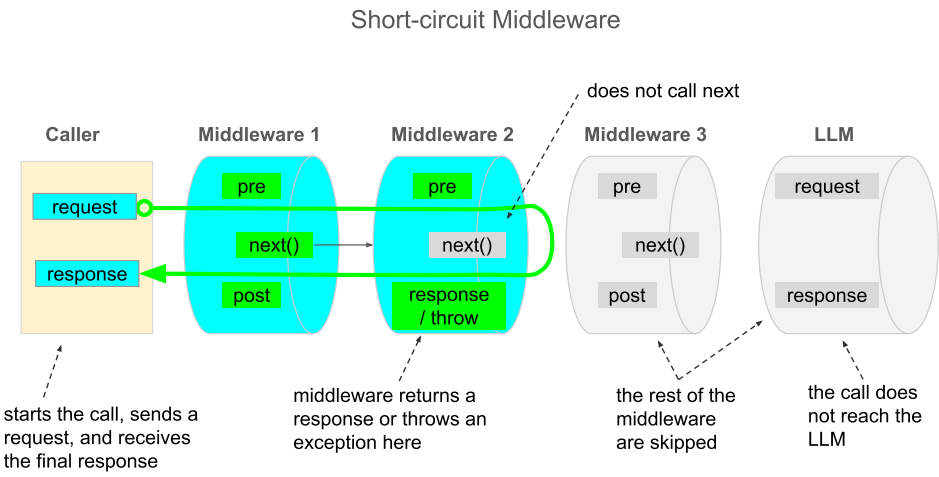

Figure 9.2 Any middleware can short-circuit the pipeline by not calling next() and instead returning a response or throwing, so later middleware and the LLM are skipped and only earlier middleware run their post logic.

The short-circuit diagram shows the same pipeline, but this time *Middleware 2* decides not to call `next()`. The caller still sends a request and Middleware 1 still runs its pre logic and calls `next()`, so the request reaches *Middleware 2* in the usual way. *Middleware 2* runs its own pre logic, but instead of forwarding the request, it either returns a response directly or throws an exception.

At that moment, the pipeline ended. *Middleware 3* and the LLM are skipped entirely because `next()` was never invoked in *Middleware 2*. The response (or error) flows back from *Middleware 2* to *Middleware 1*, which can still run its post logic, and then back to the caller. This short-circuiting behavior is what lets you implement guardrails such as token budgets, authorization checks, or validation rules: a middleware can stop the operation early and return a controlled outcome without changing any of the downstream middleware or the agent itself.

9.1.2 Defining ChatClient Middleware

ChatClient middleware is the production layer that sits between a chat client call and the AI model. It does not define the agent's reasoning or behavior, but it enforces how that behavior is executed in a real system.

A useful way to think about middleware is as the safety systems in a car. The engine (the agent) provides power, but safe driving depends on the surrounding control systems:

- Seatbelts (validation) prevent bad or malformed inputs from causing damage.
- Airbags (error handling) absorb failures, so a single fault doesn't bring everything down.
- Speed limiters (rate limiting) keep calls, costs, and side effects within safe bounds.
- Dashcam (audit logging) records what happened, when, and why for debugging, compliance, and incident response.
- Collision warning (human approval) pauses before high-risk actions and asks a human to confirm.

In production systems, these safeguards are not optional. Middleware provides the guardrails that let us operate powerful agents safely, keeping control over risk, cost, and compliance.

9.1.3 Visualizing the Two-Layer Middleware Architecture

In Agent Framework, middleware operates at two distinct layers: the *ChatClient* layer (`IChatClient` from Microsoft.Extensions.AI) and the *Agent* layer (Microsoft.Agents.AI).

The diagram in figure 9.3 shows a two-layer middleware architecture, where both layers operate independently but can be combined.

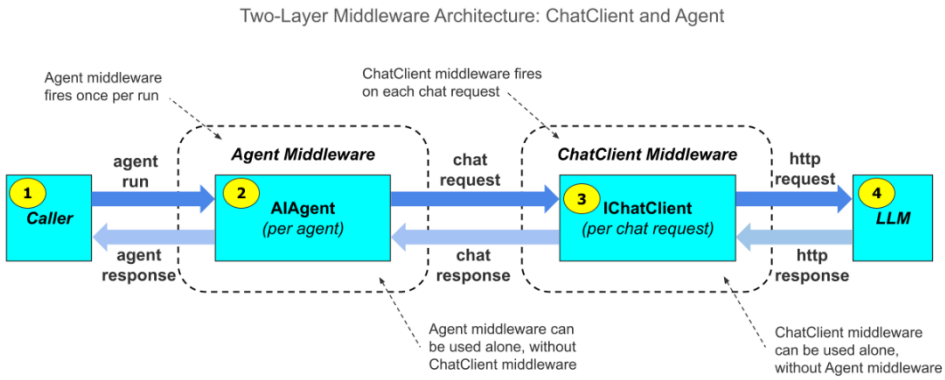

Figure 9.3 Diagrams show the two-layer middleware model. Agent middleware is at higher level, and it runs once per RunAsync() call (agent run), and ChatClient middleware is a lower level, and it runs on each call to LLM (chat request).

These are the two-layer diagram steps:

- Step 1: Caller Invokes Agent

  The caller initiates the flow on the AIAgent (component 2). This triggers the execution pipeline, starting with any registered agent middleware.

- Step 2: *Agent* Middleware Executes

  Agent middleware wraps the entire agent running lifecycle. It fires once per agent invocation, allowing inspection and modification of both the incoming chat request and outgoing agent response. Importantly, agent middleware can run standalone without requiring *ChatClient* middleware.

- Step 3: *ChatClient* Middleware Executes

  After the agent processes the request, it sends a chat request to the `IChatClient` (component 3). At this point, *ChatClient* middleware intercepts the HTTP request to LLM. This middleware fires on each chat request, meaning it can execute multiple times within a single agent run if the agent makes multiple model calls. *ChatClient* middleware can also operate independently, without *Agent* middleware.

- Step 4: LLM Interaction

  The `IChatClient` sends the HTTP request to the LLM (component 4) and receives the HTTP response. The response then flows back through the middleware chain in reverse order: first

through any *ChatClient* middleware for post-processing, then through agent middleware, and finally back to the caller.

This two-layer design provides fine-grained control over where we intercept execution, allowing different concerns to be handled at the appropriate level.

NOTE ChatClient layer is your universal safety net, it sees every chat request to the LLM and applies shared behavior across all agents. The Agent layer is your fine-grained control, it knows which agent, which session, and enforces per-agent instructions.

In this chapter, we focus on the lower layer, *ChatClient* middleware.

SOLVING PRODUCTION PROBLEMS WITH MIDDLEWARE

Middleware handles everything that isn't core agent logic but is absolutely essential for production (table 9.1):

Table 9.1 Production scenarios handled using middleware

Production Concern	Without Middleware	With Middleware
Security	PII (Personally Identifiable Information) sent to LLM providers	Automatically redacted
Cost Control	Unlimited API calls or token usage	Enforced quotas and budgets
Compliance	No audit trail	Complete logging with timestamps
Safety	All operations auto-approved	Human-in-the-loop for dangerous ops
Performance	No visibility into bottlenecks	Metrics and monitoring
Reliability	Errors cascade unpredictably	Graceful handling and recovery

Middleware does not add new capabilities to the agent. It centralizes guardrails such as cost limits, data protection, tool supervision, and auditing, so production risks are handled outside the agent definition.

RUNNING AGENTS WITHOUT MIDDLEWARE

Before adding any middleware, let's see what happens when agents run unprotected. The baseline project demonstrates all three stories in action: email leaking to the LLM provider, no cost control, and dangerous operations executing without validation.

In listing 9.1 we have a simple agent encountering a few challenges mentioned before.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Code path: `/MiddlewareBaseline/Program.cs`

Listing 9.1 Agent without Middleware

```
using AITools;
using Helpers;
using Microsoft.Agents.AI;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;
```

```
var configuration = new ConfigurationBuilder()
  .AddUserSecrets<Program>().Build();

var model = configuration["OpenAI:ModelId"];
var apiKey = configuration["OpenAI:ApiKey"];

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient();  //❶
// NO .AsBuilder() - NO middleware pipeline!

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are the MotorsAgent controlling a robot car.
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
      """,
    Tools = [.. MotorTools.AsAITools()],
  }
});

AgentSession session = await motorsAgent.CreateSessionAsync();

Console.WriteLine("""
  TEST 1: Email Leak — Step 1 gap (no RemoveEmail)
  ChatClientSharedFunctionMiddleware would redact the address.
  """);
var prompt1 = """
  Navigate to original position.
  Contact me at john.doe@example.com for updates.
  """;
Console.WriteLine($"PROMPT: {prompt1}");
AgentResponse result1 = await motorsAgent
  .RunAsync(prompt1, session);  //❷
Console.WriteLine($"\nRESULT: {result1}\n");
Console.WriteLine("""
  >>> The email was NOT redacted —
  sent directly to the LLM provider (GDPR violation).
  """);

Console.WriteLine("""
  TEST 2: Unlimited Spend —
  Step 1 + Step 2 gap (no LimitRequests, no EnforceTokenBudget)
  ChatClientSharedFunctionMiddleware would limit requests.
  ChatClientResponseMiddleware would enforce a token budget.
  """);

var prompt2 = "Move forward 3 meters, turn right 90 degrees, move forward 3
meters";
Console.WriteLine($"PROMPT: {prompt2}");
AgentResponse result2 = await motorsAgent
  .RunAsync(prompt2, session);  //❸
Console.WriteLine($"\nRESULT: {result2}\n");
Console.WriteLine("""
  >>> No request limit hit, no token budget enforced —
  costs accumulate silently.
  """);
```

```
Console.WriteLine("""
  TEST 3: Unconstrained Backward —
  Step 3 gap (no ConstrainDistance, no AuditFunctionCalling)
  ChatClientFunctionCallingMiddleware would cap backward to 5 m
  and audit every tool call.
  """);

var prompt3 = "Move forward 10 meters then go backward 10 meters";
Console.WriteLine($"PROMPT: {prompt3}");
AgentResponse result3 = await motorsAgent
  .RunAsync(prompt3, session);  //❹
Console.WriteLine($"\nRESULT: {result3}\n");
Console.WriteLine("""
  >>> Backward ran the full 10 m — no constraint, no audit.
  The robot could hit a wall.
  """);
```

❶ ChatClient that has NO middleware protection
❷ Queries the agent with a sensitive input data (email)
❸ Queries the agent with no requests or budget limit
❹ Queries the agent with no audit or action supervision

Code output:

```
TEST 1: Email Leak — Step 1 gap (no RemoveEmail)
ChatClientSharedFunctionMiddleware would redact the address.
PROMPT: Navigate to original position.
Contact me at john.doe@example.com for updates.

RESULT: stop

>>> The email was NOT redacted —
sent directly to the LLM provider (GDPR violation).
TEST 2: Unlimited Spend —
Step 1 + Step 2 gap (no LimitRequests, no EnforceTokenBudget)
ChatClientSharedFunctionMiddleware would limit requests.
ChatClientResponseMiddleware would enforce a token budget.
PROMPT: Move forward 3 meters, turn right 90 degrees, move forward 3 meters
[01:37:34:935] MOTORS: Forward: 3m
[01:37:37:099] MOTORS: TurnRight: 90°
[01:37:39:217] MOTORS: Forward: 3m
[01:37:41:095] MOTORS: Stop

RESULT: forward 3
turn right 90
forward 3
stop

>>> No request limit hit, no token budget enforced — costs accumulate silently.
TEST 3: Unconstrained Backward —
Step 3 gap (no ConstrainDistance, no AuditFunctionCalling)
ChatClientFunctionCallingMiddleware would cap backward to 5 m
and audit every tool call.
PROMPT: Move forward 10 meters then go backward 10 meters
[01:37:43:921] MOTORS: Forward: 10m
[01:37:48:027] MOTORS: Backward: 10m
[01:37:49:975] MOTORS: Stop

RESULT: forward 10
backward 10
stop

>>> Backward ran the full 10 m — no constraint, no audit.
The robot could hit a wall.
```

All three tests expose the same root problem: the agent is an unguarded conduit between the caller and the LLM. Sensitive data travels to the LLM provider unchanged, every command executes

immediately regardless of risk, and nothing stands between a runaway request loop and a five-figure cloud bill.

The critical insight is that these concerns are cross-cutting. They apply to every agent, every session, and every LLM call, yet none of them belong in business logic. This is exactly the problem middleware is designed to solve, following the principle of *Separation of Concerns*: keeping different responsibilities in various parts of the system so each part has a clear, focused job. Middleware intercepts the pipeline at the right layer, enforces rules once, and protects every agent transparently, without touching a single line of agent.

9.2 Walking Through ChatClient Middleware Layer

ChatClient middleware fires on each chat request between `IChatClient` and the LLM, operating without agent session context or identity. This lack of context is deliberate: it treats every invocation as an independent HTTP call, applying shared behavior uniformly across all agents. Without *ChatClient* middleware layer, agents talk directly to LLM with no place to protect input or output, supervise tool calls, or audit what happened.

Diagram in figure 9.4 shows the *ChatClient* middleware.

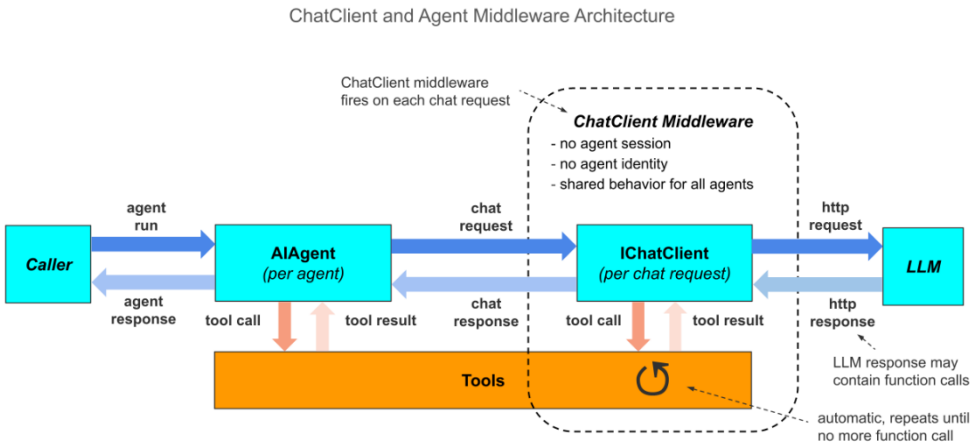

Figure 9.4 ChatClient middleware in tool-calling flow: operates without agent context, intercepts every chat request to the LLM, and fires repeatedly during iterative tool execution loops.

During tool calling, the flow becomes iterative. When the LLM returns function calls in the chat request area, the agent executes them via the *Tools* area and sends results back through `IChatClient`. This loop repeats automatically until no more function calls are requested. *ChatClient* middleware intercepts every round trip, potentially firing multiple times per agent run. The chat request area handles both responses and shared functions, while the *Tools* area manages function calling execution. This interception prevents issues like unredacted sensitive data reaching the LLM, unconstrained tool calls causing damage, and uncontrolled token consumption accumulating costs.

To address these concerns effectively, *ChatClient* middleware is implemented using three distinct types, each targeting a specific stage in the chat request-response lifecycle.

9.2.1 Choosing the Right ChatClient Middleware Type

Robby's production issues fall into three buckets: what we send to the LLM, what we get back, and how tools are invoked. *ChatClient* middleware mirrors this with three types, so we must pick the right one based on whether we need to shape the request, inspect the response, or supervise a tool call. *SharedFunction* prepares outgoing requests (sanitizing or redacting data before it reaches the LLM). Response handles incoming responses (tracking tokens, enforcing budgets, or inspecting content). *FunctionCalling* supervises tool invocations (constraining arguments, auditing calls, or blocking unsafe executions). Together, these types cover the prepare-handle-invoke pattern across the chat request area and tool execution flow.

Figure 9.5 breaks down the three *ChatClient* middleware types side by side, showing their inputs, outputs, and where each sit relative to the LLM response.

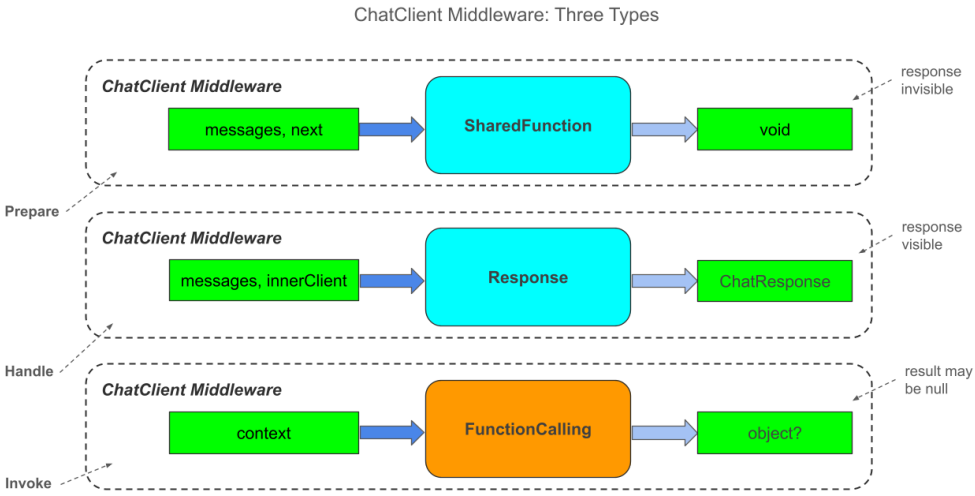

Figure 9.5 The diagram shows the three types of ChatClient middleware: SharedFunction, Response, and FunctionCalling.

The three types form a layered progression. *SharedFunction* is the lightest: it can inspect and modify input messages before they reach the LLM, but it never sees the response. *Response* wraps the entire LLM call, giving you access to both the request and the full `ChatResponse`, including token usage and latency. *FunctionCalling* is specialized: it does not participate in the LLM round-trip at all. Instead, it intercepts each tool invocation, letting you validate, modify, or reject individual function calls before they execute.

- *SharedFunction*: **prepares** the input (blind to `ChatResponse`)
- *Response*: **handles** the input and output (sees `ChatResponse`)
- *FunctionCalling*: **invokes** each tool (innermost, fires per tool call)

We'll build up the middleware pipeline gradually. *SharedFunction* is the simplest type: it controls what goes into the LLM but is structurally blind to what comes out (`ChatResponse`). *Response* unlocks the response; you get both input and output control. *FunctionCalling* is a different category entirely: it doesn't intercept LLM calls at all, it intercepts individual tool invocations (conventional code).

9.2.2 Preparing Requests with SharedFunction Middleware

SharedFunction uses the classic `next`-delegate middleware pattern. Each *SharedFunction* component receives messages, options, a next delegate, and a cancellation token. It can transform the messages, throw an exception to block the request, or call `await next(…)` to pass control to the `next` middleware and later resume execution after that call. The signature encodes this pattern. The `next` delegate returns `Task` (void). There is no `ChatResponse` value to return or modify here, which is why `ChatResponse` never appears in this middleware type.

Figure 9.6 shows the *SharedFunction* middleware type.

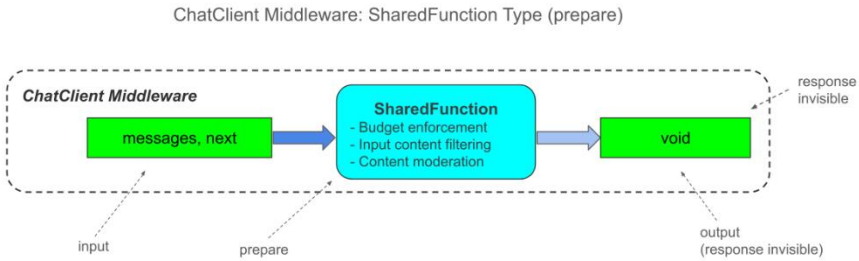

Figure 9.6 The SharedFunction middleware receives messages and a next delegate. The SharedFunction returns no value, and there is no ChatResponse to read or alter. Typical use cases include request limiting, input content filtering, prompt injection detection, and content moderation. All of these run before any token is consumed.

Registration order defines the pipeline order: each middleware wraps the next one, and if it does not call next, the pipeline short-circuits. Typical use cases include request limiting, input content filtering, and prompt injection detection. All of these run before any token is consumed.

This makes *SharedFunction* the natural first layer: validate and adjust the input at this level. If something is wrong with the request, we fail here, before tokens are consumed and before the response path runs at all.

PRACTICAL EXAMPLE: EMAIL REMOVAL

Let me tell you a production story: The GDPR Nightmare Story.

For a European client, we deploy Robby as an intelligent delivery robot car in their office building. Employees could summon Robby, providing their name, office number, and phone number for delivery confirmation. Six months later, a security audit revealed that every employee's phone number, email, and office location had been sent to their LLM provider and logged in plain text.

Robby worked correctly. It just never sanitized the data before processing. This is what we want:

```
Input:  "Navigate to GPS. Contact me at john@example.com for updates."
Output: "Navigate to GPS. Contact me at [REDACTED-EMAIL] for updates."
```

Let's see in listing 9.2 how we can prevent sensitive information leakage.

Code path: `/ChatClientSharedFunctionMiddleware/ChatClientSharedFunctions.cs`

Listing 9.2 RemoveEmail

```
public static class ChatClientSharedFunctions
{
  private const string EmailPattern =
    @"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b";  //❶
  private const string EmailMask = "[REDACTED-EMAIL]";  //❷

  public static async Task RemoveEmail(
```

```
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    Func<IEnumerable<ChatMessage>, ChatOptions?,
      CancellationToken, Task> next,
    CancellationToken cancellationToken)  //❸
  {
    Console.WriteLine($"[ChatClient] [SharedFunction] [Email] "
      + "PRE: Scanning {messages.Count()} messages...");

    bool emailFound = false;  //❹
    List<ChatMessage> sanitizedMessages = [];

    foreach (var message in messages)
    {
      if (message.Role == ChatRole.User && message.Text is not null)  //❺
      {
        var sanitized = Regex
          .Replace(message.Text, EmailPattern, EmailMask);
        if (sanitized != message.Text) emailFound = true;
        sanitizedMessages.Add(new ChatMessage(message.Role, sanitized));
      }
      else
      {
        sanitizedMessages.Add(message);
      }
    }

    Console.WriteLine(emailFound
      ? "[ChatClient] [SharedFunction] [Email] Email detected and removed!"
      : "[ChatClient] [SharedFunction] [Email] No email found");

    await next(sanitizedMessages, options, cancellationToken);  //❻

    Console.WriteLine($"[ChatClient] [SharedFunction] [Email] POST: "
      + "Completed");
  }
}
```

❶ Defines Regex pattern standard email address format
❷ Defines replacement mask for any detected email
❸ Defines the SharedFunction
❹ Tracks whether any email was found
❺ Only user messages are sanitized, tool calls and tool results are not
❻ Passes sanitized messages downstream

Let's see in listing 9.3 how to wire it into the *ChatClient* pipeline.

Requirements (NuGet packages):

```
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
dotnet add package Microsoft.Extensions.AI.OpenAI
```

Code path: `/ChatClientSharedFunctionMiddleware/Program.cs`

Listing 9.3 SharedFunction Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❷
  .Build();  //❸

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
```

```
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are an AI assistant controlling a robot car capable of performing
      basic moves: forward, backward, turn left, turn right, and stop.
      You have to break down the provided complex commands into the basic
      moves you know.
      Use a JSON array like [move1, move2, move3] for the response.
      Respond only with the moves and their parameters (angle or distance),
      without any additional explanations.
      """
  }
});  //❹

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt = """
  There is a tree directly in front of the car.
  Avoid it and then return to the original path.
  """;
Console.WriteLine($"PROMPT: {prompt}");
var result = await motorsAgent.RunAsync(prompt, session);  //❺
Console.WriteLine($"\nRESULT: {result}\n");
```

❶ Converts IChatClient into a builder to enable middleware registration
❷ Registers RemoveEmail
❸ Builds the IChatClient with the middleware pipeline applied
❹ Creates the agent
❺ Runs the agent

Email sanitization is now in place, so Robby will never leak an email address to the LLM provider. But Robby can still make unlimited LLM calls. The $10,847 bill is still entirely possible. The next middleware adds the request limit guard, and once both are defined, we will compose them into a single pipeline and see how they interact.

NOTE Transient vs. persistent sanitization. At the *ChatClient* layer, sanitization is *transient*: the LLM sees clean data, but the agent's session chat history retains the original text (email stays intact). This is safe because the middleware re-applies transparently on every LLM round-trip. Contrast with the *Agent* layer (as we will find out in the next chapter where we discuss *Agent* middleware layer), where such `RemoveEmail` function is *persistent*, sanitized messages are stored in the session history, so all future round-trips already see clean data without re-sanitization. At the *ChatClient* layer, the session is not accessible, so transient re-application is the only option, but it is the developer's responsibility to choose which middleware layer fits best the desired use-case.

PRACTICAL EXAMPLE: REQUEST LIMITING

Let me tell you a production story: The $10,000 Weekend.

Late on a Friday afternoon, Robby navigated obstacles, followed voice commands, and captivated visitors. On Monday morning, the CTO opened an email from OpenAI. The weekend bill was $10,847. The cause was simple but costly. A bug in Robby's retry logic had triggered 47,000 API calls as it continuously re-evaluated its environment. Without middleware to enforce rate or request limits, nothing stopped it.

Listing 9.4 shows how such a costly incident can be prevented.

Code path: `/ChatClientSharedFunctionMiddleware/ChatClientSharedFunctions.cs`

Listing 9.4 LimitRequests

```
public static class ChatClientSharedFunctions
{
  private static int _requestCount = 0;  //❶
  private const int MaxRequests = 2;  //❷

  public static async Task LimitRequests(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    Func<IEnumerable<ChatMessage>, ChatOptions?, CancellationToken, Task> next,
    CancellationToken cancellationToken)  //❸
  {
    var currentCount = Interlocked.Increment(ref _requestCount);  //❹

    Console.WriteLine($"[ChatClient] [SharedFunction] [Limit] PRE: "
      + "Request {currentCount} of {MaxRequests} max");

    if (currentCount > MaxRequests)  //❺
    {
      throw new LimitExceededException(currentCount, MaxRequests);  //❻
    }

    await next(messages, options, cancellationToken);  //❼

    Console.WriteLine($"[ChatClient] [SharedFunction] [Limit] POST: "
      + "Request {currentCount} of {MaxRequests} max completed");
  }
}
```

❶ **Declares shared counter across all LLM round-trips**
❷ **Declares maximum number of LLM round-trips allowed**
❸ **Declares SharedFunction**
❹ **Thread-safe increment**
❺ **Checks the request budget**
❻ **Throws a custom exception carrying current count and limit**
❼ **Limit not exceeded, continue the pipeline toward the LLM**

Key teaching points (for listing 9.4):

- `Interlocked.Increment` for thread-safe counting across concurrent agents

  PRE phase: check requests limit, throw custom `Exception` if exceeded — the request never reaches the LLM

- `await next(messages, options, cancellationToken)` continues the pipeline

  POST phase: log completion after the entire inner pipeline has returned

---

WARNING With tool-calling agents, a single user prompt triggers *multiple* LLM round-trips and the limit depletes faster than the number of user queries suggests

---

Listing 9.5 shows the custom exception we build specially for capturing the limit exceeding requests.

Listing 9.5 LimitExceededException.cs

```
public class LimitExceededException(int currentCount, int maxRequests)
  : Exception($"Request limit exceeded: request {currentCount} "
    + "of {maxRequests} blocked!")
{ }
```

Fail fast saves tokens. Because *SharedFunction* is outermost, `LimitRequests` throws before *Response* middleware even starts. No LLM call is made, no tokens are consumed, no money is spent.

With both functions defined, we now wire them into a single middleware pipeline. Registration order determines execution order: `LimitRequests` registers first, so it sits outermost. If the budget is exceeded, `RemoveEmail` is never entered and no sanitization work is wasted.

Let's see in listing 9.6 how the previously defined *ChatClient* middleware functions (`RemoveEmail` and `LimitRequests`) kick in.

Code path: `/ChatClientSharedFunctionMiddleware/Program.cs`

Listing 9.6 SharedFunction Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientSharedFunctions.LimitRequests)  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❷
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are an AI assistant controlling a robot car capable of performing
      basic moves: forward, backward, turn left, turn right, and stop.
      You have to break down the provided complex commands into the basic
      moves you know.
      Use a JSON array like [move1, move2, move3] for the response.
      Respond only with the moves and their parameters (angle or distance),
      without any additional explanations.
      """
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt1 = """
  There is a tree directly in front of the car.
  Avoid it and then return to the original path.
  """;
Console.WriteLine($"PROMPT: {prompt1}");
var result1 = await motorsAgent.RunAsync(prompt1, session);  //❹
Console.WriteLine($"RESULT: {result1}\n");

var prompt2 = """
  Navigate to original position.
  Contact me at john.doe@example.com for updates.
  """;
Console.WriteLine($"PROMPT: {prompt2}");
try
{
  var result2 = await motorsAgent.RunAsync(prompt2, session);  //❺
  Console.WriteLine($"RESULT: {result2}\n");
}
catch (LimitExceededException ex)  //❻
{
  Console.WriteLine($"EXCEPTION: {ex.Message}\n");
}
```

```
var prompt3 = "Stop and email me at sara.doe@example.com for "
  + "further instructions.";
Console.WriteLine($"PROMPT: {prompt3}");
try
{
  var result3 = await motorsAgent.RunAsync(prompt3, session);  //❼
  Console.WriteLine($"RESULT: {result3}\n");
}
catch (LimitExceededException ex)  //❽
{
  Console.WriteLine($"EXCEPTION: {ex.Message}\n");
}
```

❶ LimitRequests is outermost
❷ RemoveEmail is inner
❸ Defines the agent
❹ First prompt — no email present, limit not exceeded
❺ Second prompt — email detected and removed before reaching the LLM
❻ Catches LimitExceededException from the second prompt if the limit was hit
❼ Third prompt — limit already exhausted from previous round-trips
❽ Catches LimitExceededException

Every prompt is wrapped in try-catch. The source treats all three queries uniformly, because in a real system you cannot assume which round-trip will exhaust the budget.

Code output:

```
PROMPT: There is a tree directly in front of the car. Avoid it and then return to
the original path.
[ChatClient] [SharedFunction] [Limit] PRE: Request 1 of 2 max
[ChatClient] [SharedFunction] [Email] PRE: Scanning 1 messages...
[ChatClient] [SharedFunction] [Email] No email found
[ChatClient] [SharedFunction] [Email] POST: Completed
[ChatClient] [SharedFunction] [Limit] POST: Request 1 of 2 max completed
RESULT: [
  {"move":"turn right","angle":45},
  {"move":"forward","distance":3},
  {"move":"turn left","angle":45},
  {"move":"forward","distance":5},
  {"move":"turn left","angle":45},
  {"move":"forward","distance":3},
  {"move":"turn right","angle":45}
]

PROMPT: Navigate to original position. Contact me at john.doe@example.com for
updates.
[ChatClient] [SharedFunction] [Limit] PRE: Request 2 of 2 max
[ChatClient] [SharedFunction] [Email] PRE: Scanning 3 messages...
[ChatClient] [SharedFunction] [Email] Email detected and removed!
[ChatClient] [SharedFunction] [Email] POST: Completed
[ChatClient] [SharedFunction] [Limit] POST: Request 2 of 2 max completed
RESULT: [
  {"move":"backward","distance":3},
  {"move":"turn right","angle":45},
  {"move":"backward","distance":5},
  {"move":"turn right","angle":45},
  {"move":"backward","distance":3},
  {"move":"turn left","angle":45}
]

PROMPT: Stop and email me at sara.doe@example.com for further instructions.
[ChatClient] [SharedFunction] [Limit] PRE: Request 3 of 2 max
EXCEPTION: Request limit exceeded: request 3 of 2 blocked!
```

Every successful round-trip shows the same sequence:

```
Limit PRE → Email PRE → Email POST → Limit POST.
```

Fail fast confirmed. Prompt 3 shows `[Limit] PRE` immediately followed by the exception, so `[Email] PRE` never appears. As expected, no sanitization work is done on a rejected request.

EXERCISE

Clone `RemoveEmail` function and name it `HideIp` to also detect and redact IP addresses (e.g., 192.168.1.1 → [REDACTED-IP]). Hint: use `Regex.Replace` call for the pattern

```
private const string IpPattern = @"\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b";
private const string IpMask = "[REDACTED-IP]";
```

then call the function using:

```
.Use(ChatClientSharedFunctions.HideIp)
```

The middleware function should mask the IPs as well, and you may want to rename the method accordingly, but nothing stops you to follow the best practices and having two middleware functions instead of one with two responsibilities.

9.2.3 Handling Requests and Responses with Response Middleware

Response middleware type is where the chat response is revealed. Everything *SharedFunction* can do, like transform input messages, block requests, log, *Response* can do too.

There is no *next* delegate here, but the pipeline still exists. The framework builds it by wrapping: it starts from the real chat client, then passes it as `innerClient` to the first middleware, which becomes the new client, then passes that to the next middleware, and so on. Each middleware receives the previously wrapped client as its `innerClient`, calls `GetResponseAsync` to get the response, and returns it with or without modifications. This recursive wrapping is what simulates the *next*-like pipeline without an explicit next parameter.

Figure 9.7 shows the *Response* middleware type.

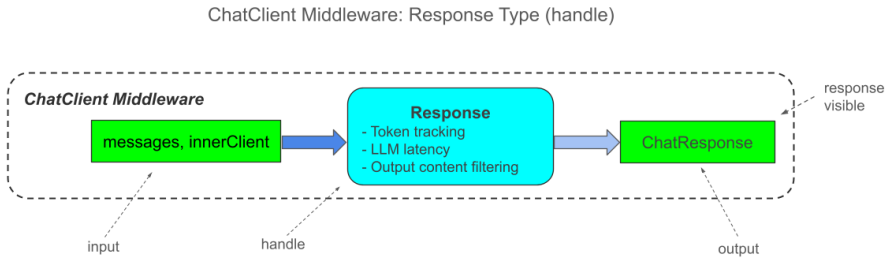

Figure 9.7 The Response middleware receives messages and an IChatClient innerClient, and it returns a ChatResponse. It starts from the real chat client, then repeatedly wraps it, passing the previously built client as innerClient. Typical use cases include token tracking, LLM latency, response caching, and output content filtering.

Response has full control over the `innerClient` to call `GetResponseAsync()` and can read and modify the `ChatResponse`0. With *Response* we can track tokens and latency, attach metadata, enforce budgets and timeouts, or even fabricate a response to short-circuit the pipeline. For example, `ChatResponse.Usage` exposes tokens counts that are only available at this *ChatClient* layer, not in the `AgentResponse` type we see in the *Agent* middleware layer.

This makes it the natural home for token tracking, content filtering, and budget enforcement.

PRACTICAL EXAMPLE: ENFORCE TOKEN BUDGET

Do you recall the $10,000 Weekend story from section 9.2.2, where on Monday morning we find in the inbox a $10,847 bill, because a bug in retry logic? Beyond request limiting, we also need token budget enforcement.

Every LLM call has a cost in tokens. Running out of token budget silently degrades user experience; uncontrolled consumption racks up costs. At the *ChatClient* layer we can measure cumulative token usage — the total tokens consumed across all round-trips — and enforce a hard ceiling before the next call is dispatched.

In listing 9.7 we see the *Response* function.

Code path: `/ChatClientResponseMiddleware/ChatClientResponses.cs`

Listing 9.7 Enforce Token Budget

```
public static class ChatClientResponses
{
  private static long _tokensCount = 0;
  private const long MaxTokens = 2000;  //❶

  public static async Task<ChatResponse> EnforceTokenBudget(
  IEnumerable<ChatMessage> messages,
  ChatOptions? options,
  IChatClient innerClient,
  CancellationToken cancellationToken)  //❷
  {
    var currentTokens = Interlocked.Read(ref _tokensCount);  //❸
    if (currentTokens > MaxTokens)  //❹
    {
      Console.WriteLine($"[ChatClient] [Response] [Tokens] "
        + "Budget exhausted ({currentTokens} / {MaxTokens}) "
        + "— LLM call skipped");
      return new ChatResponse([new ChatMessage(ChatRole.Assistant,
        "Token budget exhausted.")]);  //❺
    }

    var response = await innerClient
      .GetResponseAsync(messages, options, cancellationToken);  //❻

    if (response.Usage is not null)
    {
      var totalTokens = response.Usage.TotalTokenCount
        ?? (response.Usage.InputTokenCount ?? 0) +
           (response.Usage.OutputTokenCount ?? 0);  //❼
      Interlocked.Add(ref _tokensCount, totalTokens);  //❽
    }

    return response;  //❾
  }
}
```

❶ **Defines token limit (Real apps would have much higher limits)**
❷ **Declares Response function (gives access to call and chat response)**
❸ **Reads the current token count**
❹ **Checks if max tokens are exceeded**
❺ **Returns a synthetic assistant message (innerClient is never called)**
❻ **Explicitly calls innerClient for a chat response**
❼ **Computes the response token count**
❽ **Adds the current tokens count**
❾ **Returns the chat response**

When the budget is exhausted, the middleware short-circuits: it returns a synthetic `ChatResponse` with a human-readable message and never calls the LLM.

Practical Example: Timestamp Adding

Production systems often require audit trails that capture when each response was generated. Timestamping chat responses enables compliance tracking, debugging temporal issues, and correlating agent behavior with external events.

Response middleware type has access to `ChatResponse`.

In listing 9.8 we see the `AddTimestamp` *ChatClient Response* middleware function.

Code path: `/ChatClientResponseMiddleware/ChatClientResponses.cs`

Listing 9.8 AddTimestamp

```
public static class ChatClientResponses
{
  public static async Task<ChatResponse> AddTimestamp(
    IEnumerable<ChatMessage> messages,
    ChatOptions? options,
    IChatClient innerClient,
    CancellationToken cancellationToken)  //❶
  {
    ChatResponse response = await innerClient
      .GetResponseAsync(messages, options, cancellationToken);  //❷

    foreach (ChatMessage message in response.Messages)  //❸
    {
      if (!string.IsNullOrEmpty(message.Text))  //❹
      {
        string timestamp = DateTimeOffset.UtcNow.ToString("o");  //❺
        Console.WriteLine($"[ChatClient] [Response] [Timestamp] "
          + "Stamping response with [{timestamp}]");
        message.Contents = [new TextContent(
          $"[{timestamp}] {message.Text}")];  //❻
      }
    }

    return response;
  }
}
```

❶ **Declares Response middleware function**
❷ **Calls innerClient.GetResponseAsync**
❸ **Iterates through all messages**
❹ **Identifies the valid text contents**
❺ **Builds the timestamp string**
❻ **Prefixes the message contents with the timestamp**

Let's see in listing 9.9 how to wire both *Response* functions into the middleware pipeline.

Code path: `/ChatClientResponseMiddleware/Program.cs`

Listing 9.9 Response Middleware

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientResponses.EnforceTokenBudget, null)  //❶
  .Use(ChatClientResponses.AddTimestamp, null)  //❷
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
    Instructions = """
      You are the MotorsAgent controlling a robot car.
```

```
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
    """,
    Tools = [.. MotorTools.AsAITools()],
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt1 = "Move forward 3 meters";
Console.WriteLine($"PROMPT: {prompt1}");
var result1 = await motorsAgent.RunAsync(prompt1, session);  //❹
Console.WriteLine($"\nRESULT: {result1}\n");

var prompt2 = "Turn right 45 degrees";
Console.WriteLine($"PROMPT: {prompt2}");
var result2 = await motorsAgent.RunAsync(prompt2, session);  //❺
Console.WriteLine($"\nRESULT: {result2}\n");
```

❶ **EnforceTokenBudget registered as outer**
❷ **Registered as inner**
❸ **Declares the agent**
❹ **First prompt that triggers timestamp adding**
❺ **Second prompt that triggers limit exceeding response**

WARNING This agent has tools — `MotorTools.AsAITools()`. As side effect, each user prompt triggers multiple LLM round-trips (one to decide which tool to call, one after the tool result is returned). Token consumption accumulates across all round-trips, which is why the budget can be exhausted mid-prompt.

Code output:

```
PROMPT: Move forward 3 meters
[08:26:06:940] MOTORS: Forward: 3m
[08:26:09:244] MOTORS: Stop
[ChatClient] [Response] [Timestamp] Stamping response with [2026-04-
08T10:24:35.1407499+00:00]

RESULT: [2026-04-06 18:26:11 UTC] - forward 3 meters
- stop

PROMPT: Turn right 45 degrees
[08:26:12:384] MOTORS: TurnRight: 45°
[08:26:14:816] MOTORS: Stop
[ChatClient] [Response] [Tokens] Budget exhausted (2068 / 2000) — LLM call skipped

RESULT: Token budget exhausted.
```

Each prompt produces two metric blocks — not one. The first is the initial LLM call (assistant message) where the model decides which tool to invoke; the second is the follow-up call (tool message) after the tool result is returned. This is the same multi-round-trip behavior described in section 9.1.1: *ChatClient* middleware fires on every LLM round-trip, not once per user prompt.

NOTE Token tracking is *ChatClient* middleware layer exclusive. `response.Usage` is present on `ChatResponse` (returned by the LLM provider) but is NOT available on `AgentResponse` (returned by the *Agent* middleware layer). Token budget enforcement, cost estimation, and per-

call token logging are impossible to implement at the *Agent* layer. This is the primary reason `EnforceTokenBudget` must live at the *ChatClient* layer.

EXERCISE

Create a `CacheResponse` *Response* middleware function that caches LLM responses by the content of the outgoing messages. On a cache hit, return the cached `ChatResponse` directly without calling `innerClient`. On a cache miss, call `innerClient`, store the result, and return it. You may use a dictionary to preserve the queries:

```
private static readonly Dictionary<string, ChatResponse> _cache = new();
```

Then check if we have a cache hit or a cache miss like this:

```
var key = string.Concat(messages.Select(m => m.Text));
if (_cache.TryGetValue(key, out ChatResponse? cached))
{
  return cached;  //❶
}
ChatResponse response = await innerClient
  .GetResponseAsync(messages, options, cancellationToken);  //❷
_cache[key] = response;  //❸
return response;  //❹
```

❶ **Returns cached response if we have cache hit**
❷ **Gets a new response**
❸ **Caches the current response**
❹ **Return new response if we have a cache miss**

9.2.4 Supervising Tool Calls with FunctionCalling Middleware

FunctionCalling middleware does not wrap the chat request itself. It runs when the LLM response contains a tool call and the framework is about to execute that tool, where the model has already decided which function to call and with which arguments.

Unlike *SharedFunction* and *Response*, *FunctionCalling* has no next delegate and no `innerClient`. The only way to execute the tool is through `FunctionInvocationContext`:

```
context.Function.InvokeAsync(context.Arguments, cancellationToken);  //❶
```

❶ **Invokes the function with arguments and cancelation token**

`FunctionInvocationContext` provides access to the function name, arguments, chat messages, and result, allowing middleware to validate, modify, or reject tool calls before they execute.

Figure 9.8 shows the *FunctionCalling* middleware type.

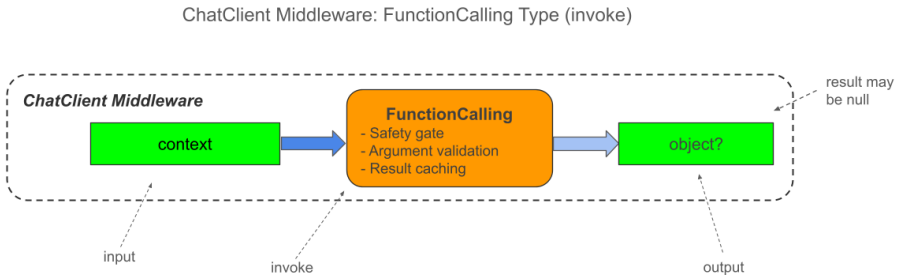

Figure 9.8 The FunctionCalling middleware receives a FunctionInvocationContext and returns ValueTask<object?>. There is no next delegate and no innerClient; the tool is invoked directly through context.Function.InvokeAsync(). Typical use cases include safety checks, audit logging, argument validation, and result caching.

Because there is no *next* delegate, composition relies on a null/non-null result convention. A middleware that returns `null` acts as an *interceptor*: it can inspect and modify arguments but does not call `InvokeAsync`, signaling that the pipeline should continue. A middleware that returns a non-null value acts as the *executor*: it calls `InvokeAsync`, owns the result, and terminates the chain. We can have multiple interceptors but only one executor, and the executor is always last.

Figure 9.9 illustrates the interceptor-executor pattern.

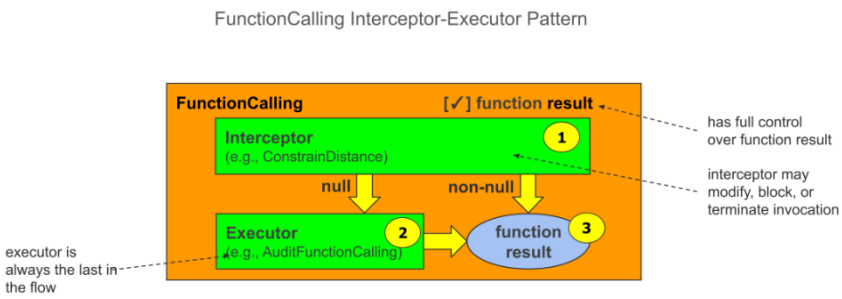

Figure 9.9 FunctionCalling interceptor-executor pattern. The interceptor (step 2) runs first, inspecting and potentially modifying the function invocation. If it returns null, control flows to the executor (step 3), which calls InvokeAsync and produces the final function result (step 4). If the interceptor returns non-null, the flow short-circuits immediately.

In interceptor-executor pattern. The interceptor (step 1) runs first, with full control over the function result — it may modify, block, or terminate the invocation. If it returns null, control flows to the executor (step 2), which calls `InvokeAsync` and produces the final function result (step 3). If the interceptor returns non-null, the flow short-circuits immediately returning the function result (step 3), bypassing the executor entirely. The executor is always last in the flow.

WARNING Agent Framework does not enforce interceptor-executor pattern described here. We are fully responsible for following the null/non-null convention. A mistake can result in double tool execution or skipped safety checks.

FunctionCalling at the *ChatClient* layer is also agent-agnostic. `FunctionInvocationContext` does not carry agent identity, so the same logic applies uniformly across all agents sharing that chat client.

WARNING *FunctionCalling* must be innermost. `UseFunctionInvocation` must be the last middleware registered before `.Build()`. If placed outermost, its internal tool loop drives repeatedly passes back through *SharedFunction* and *Response*, re-running budgets, sanitization, and timing once per tool rather than once per chat request.

PRACTICAL EXAMPLE: DISTANCE CONSTRAINING

Let me tell you a production story: The Runaway Robot

Robby, the autonomous robot car intelligent companion, received the command: `"Danger ahead! Move backward immediately!"` Robby dutifully called, for example, `backward(100)` and drove straight into a wall behind it.

No one had implemented a safety check. The agent just executed. We should have a distance constraint for safety.

In listing 9.10 we see how a *FunctionCalling* middleware function for preventing such incidents is implemented.

Code path:
`/ChatClientFunctionCallingMiddleware/ChatClientFunctionCallings.cs`

Listing 9.10 ConstrainDistance

```
public static class ChatClientFunctionCallings
{
  public static async Task<object?> ConstrainDistance(
    FunctionInvocationContext context,
    CancellationToken cancellationToken)  //❶
  {
    const int MaxBackwardDistance = 5;  //❷
    bool isBackward = context.Function.Name.Contains("backward",
      StringComparison.OrdinalIgnoreCase);  //❸
    bool hasDistance = context.Arguments
      .TryGetValue("distance", out object? value);  //❹
    if (isBackward && hasDistance)  //❺
    {
      int distance = value is JsonElement jsonElement
        ? jsonElement.GetInt32()
        : Convert.ToInt32(value);  //❻
      if (distance > MaxBackwardDistance)  //❼
      {
        context.Arguments["distance"] = JsonSerializer
          .SerializeToElement(MaxBackwardDistance);  //❽
        Console.WriteLine($"[ChatClient] [FunctionCall] [Constrain] "
          + "Backward distance constrained from {distance}m "
          + "to {MaxBackwardDistance}m");
      }
    }

    return null;  //❾
  }
}
```

❶ **Declares middleware function**
❷ **Defines the max allowed backward distance**
❸ **Identifies if the current function is "backward"**
❹ **Identifies if it has "distance" argument**
❺ **Checks if it's "backward" and has "distance"**
❻ **Reads the "distance" value**
❼ **Checks if the distance exceeds the max allowed backward distance**
❽ **Persists as JsonElement for downstream consistency**
❾ **Returns null to signal "proceed to the terminal invoker"**

The `null` return value is what makes the `??` operator in the `FunctionInvoker` fall through to `AuditFunctionCalling`.

Practical Example: Function Calling Auditing

When a tool call causes damage (like Robby hitting a wall), we need a complete record of which function was invoked, with which arguments, and what result it returned. *FunctionCalling* middleware provides this audit trail for debugging, compliance, and security reviews.

In listing 9.11 we see how a *FunctionCalling* middleware function is implemented.

Code path:
`/ChatClientFunctionCallingMiddleware/ChatClientFunctionCallings.cs`

Listing 9.11 AuditFunctionCalling

```
public static class ChatClientFunctionCallings
{
  public static async Task<object?> AuditFunctionCalling(
    FunctionInvocationContext context,
    CancellationToken cancellationToken)  //❶
  {
    var functionName = context.Function.Name;
    var timestamp = DateTime.UtcNow;
    var stopwatch = Stopwatch.StartNew();

    var result = await context.Function
      .InvokeAsync(context.Arguments, cancellationToken);  //❷
    stopwatch.Stop();
    Console.WriteLine($"[ChatClient] [FunctionCall] [Audit] "
      + "Function call '{functionName}' [{timestamp:HH:mm:ss}] "
      + "COMPLETED in {stopwatch.ElapsedMilliseconds}ms | "
      + "Result: {result}");  //❸
    return result;  //❹
  }
}
```

❶ **Defines the middleware function**
❷ **Invokes the function call**
❸ **Logs the successful call**
❹ **Returns the function result**

`AuditFunctionCalling` is the terminal invoker: it is the only method that calls `context.Function.InvokeAsync()`. Because `ConstrainDistance` always returns `null`, the `??` operator guarantees `AuditFunctionCalling` always runs exactly once per tool call.

Let's see in listing 9.12 how we can call multiple *FunctionCalling* middleware functions.

Code path:
`/ChatClientFunctionCallingMiddleware/Program.cs`

Listing 9.12 FunctionCalling Middleware – Program.cs

```
// 'usings' and API key fetching omitted for brevity

IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .UseFunctionInvocation(loggerFactory: null, configure: options =>
  {
    options.FunctionInvoker = async (context, ct) =>
      await Middleware.ChatClientFunctionCallings
        .ConstrainDistance(context, ct)
          ?? await Middleware.ChatClientFunctionCallings
        .AuditFunctionCalling(context, ct);
  })  //❶
  .Build();

ChatClientAgent motorsAgent = chatClient
  .AsAIAgent(new ChatClientAgentOptions
{
  Name = "MotorsAgent",
  Description = "Controls robot car movements.",
  ChatOptions = new ChatOptions
  {
```

```
    Instructions = """
      You are the MotorsAgent controlling a robot car.
      Break down complex movement commands into basic moves:
      forward, backward, turn left, turn right, stop.
      Respond with the sequence of moves needed to accomplish the task.
      """,
    Tools = [.. MotorTools.AsAITools()],  //❷
  }
});  //❸

AgentSession session = await motorsAgent.CreateSessionAsync();

var prompt = " Move forward 10 meters then go backward 8 meters";
Console.WriteLine($"PROMPT: {prompt}");
var result = await motorsAgent.RunAsync(prompt, session);  //❹
Console.WriteLine($"\nRESULT: {result}\n");
```

❶ **Defines the FunctionCalling layer**
❷ **Assigns the AI tools**
❸ **Declares the agent**
❹ **Runs the agent with a single prompt that triggers "backward"**

Code output:

```
PROMPT: Move forward 10 meters then go backward 8 meters
[08:52:40:833] MOTORS: Forward: 10m
[ChatClient] [FunctionCall] [Audit] Function call 'Forward' [18:52:40] COMPLETED
in 1019ms | Result: moved forward for 10 meters.
[ChatClient] [FunctionCall] [Constrain] Backward distance constrained from 8m to
5m
[08:52:42:830] MOTORS: Backward: 5m
[ChatClient] [FunctionCall] [Audit] Function call 'Backward' [18:52:42] COMPLETED
in 1008ms | Result: moved backward for 5 meters.
[08:52:44:687] MOTORS: Backward: 3m
[ChatClient] [FunctionCall] [Audit] Function call 'Backward' [18:52:44] COMPLETED
in 1007ms | Result: moved backward for 3 meters.
[08:52:46:838] MOTORS: Stop
[ChatClient] [FunctionCall] [Audit] Function call 'Stop' [18:52:46] COMPLETED in
1006ms | Result: stopped.

RESULT: - Forward 10 m
- Backward 8 m
- Stop
```

Forward 10 meters pass through unconstrained. Backward 8 meters, on the other hand, is silently capped to 5 meters by `ConstrainDistance` — the LLM asked for 8, but only 5 were executed. The LLM then call `Backward(3)` again to cover the remaining distance, based on the LLM's plan. The result summary reports the LLM's intended plan (8 meters), not the actual constrained execution — the agent is unaware its argument was modified.

NOTE *Agent* vs. *ChatClient* middleware. At the *Agent* layer, `AuditFunctionCalls` is chainable: it delegates to next function in the pipeline, and it has access to the agent identity via `agent.Name`. At the *ChatClient* layer, it is terminal and anonymous: it calls `InvokeAsync` directly and has no agent context.

The registration order matters. `ConstrainDistance` (argument adjustment) runs first and returns `null` to signal that the pipeline should continue. Only when that check passes does `AuditFunctionCalls` run as the terminal invoker, calling `InvokeAsync`. It is the responsibility of the last middleware function in the chain to call `InvokeAsync`, not every middleware registered functions in the pipeline.

EXERCISE

Modify `ConstrainDistance` function to limit the `forward` distance to `20 meters`.

You should notice an extra adjustment when calling the `forward` tool when the distance is larger than 20 meters.

9.3 Composing the Full ChatClient Middleware Pipeline

Now that we've seen each *ChatClient* middleware type in isolation, here is how they compose into a complete pipeline.

The pipeline calling order works like a stack. As with any pipeline, we can return early and short-circuit execution, or we can keep adding middleware until we reach the top of the stack: the chat client. Registration order equals execution order: whatever we add first runs first.

9.3.1 Stacking SharedFunction, Response, and FunctionCalling

The pipeline calling order works like a stack. As with any pipeline, we can return early and short-circuit execution, or we can keep adding middleware until we reach the top of the stack: the chat client. Registration order equals execution order: whatever we add first runs first.

Figure 9.10 shows how we mix and order the three *ChatClient* middleware types.

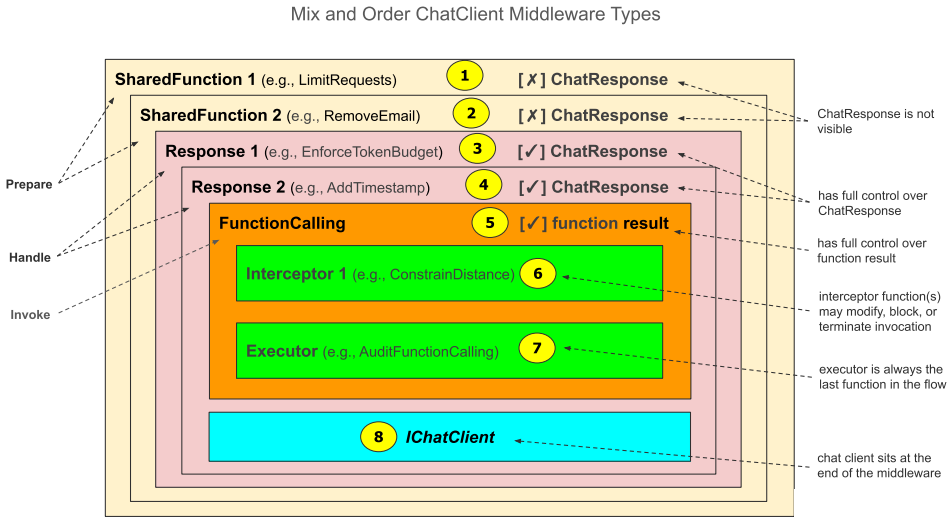

Figure 9.10 All three ChatClient middleware types composing together. SharedFunction functions (outermost, steps 1-2) prepares input, ChatResponse invisible. Response functions (middle, steps 3-4) wrap the LLM exchange, ChatResponse visible and controllable. FunctionCalling (innermost, step 5) intercepts tool calls via interceptor/executor pattern implemented with FunctionCalling functions (steps 6-7). IChatClient (step 8) sits at the end of the middleware pipeline. Registration order equals execution order: prepare → handle → invoke.

ChatClient middleware steps:

1. `SharedFunction 1`: `LimitRequests` starts the pipeline in the *Prepare* stage and can count or limit requests before they reach the chat client; `ChatResponse` is not visible here.
2. `SharedFunction 2`: `RemoveEmail` adds another *Prepare*-stage rule that sanitizes input, such as removing email addresses before the request is sent to the model; `ChatResponse` is still not visible.

3. `Response 1`: `EnforceTokenBudget` enters the *Handle* stage, where `ChatResponse` is visible and token usage or cost limits can be inspected and enforced.
4. `Response 2`: `AddTimestamp` adds another *Handle*-stage middleware that can decorate, inspect, or modify the completed `ChatResponse`.
5. *FunctionCalling* enters the Invoke stage, where middleware works with individual function results instead of the full `ChatResponse`.
6. *Interceptor*: `ConstrainDistance` runs before final function execution and can modify, block, terminate, or short-circuit the function invocation.
7. *Executor*: `AuditFunctionCalling` runs if the interceptor returns null, audits the invocation, and produces the final function result.
8. `IChatClient` sits at the end of the middleware stack, sends the request to the model, and produces the `ChatResponse` that flows back outward through the pipeline.

The recommended order for *ChatClient* middleware types is:

- *SharedFunction* Layer — *Prepare*

  The *SharedFunction* layer is the outermost part of the *ChatClient* middleware pipeline. Its job is to prepare the request before the model sees it. At this point, there is no `ChatResponse` yet, so these middleware functions cannot inspect or modify the final response.

  This makes *SharedFunction* middleware a good fit for input-oriented concerns: request limits, validation, privacy, and prompt/message shaping. In this sample, `LimitRequests` protects the pipeline from too many calls, while `RemoveEmail` prevents sensitive data from reaching the LLM provider.

- *Response* Layer — *Handle*

  The *Response* layer surrounds the actual model call. This is the first layer where `ChatResponse` is visible. Because of that, *Response* middleware is the right place for output-oriented concerns.

  This layer can inspect, decorate, replace, or reject the response. In this sample, `EnforceTokenBudget` uses response information to control cost, while `AddTimestamp` decorates the returned content.

- *FunctionCalling* Layer — *Invoke*

  The *FunctionCalling* layer is the innermost *ChatClient* middleware area. It does not work with the full `ChatResponse`; it works with individual function invocation results.

  This is the correct place for tool-level rules. In this sample, `ConstrainDistance` acts as an *interceptor* that can constrain unsafe tool arguments, while `AuditFunctionCalling` acts as the final *executor* for the function call.

- `IChatClient` — End of the Pipeline

  At the end of the pipeline sits the `IChatClient`. It sends the request to the model and produces the `ChatResponse`, which then flows back outward through the middleware layers.

IMPORTANT The important mental model is: 1) *SharedFunction* middleware prepares the request and then calls the next layer; 2) *Response* middleware handles the completed `ChatResponse`; 3) *FunctionCalling* middleware controls tool invocation, not the whole chat response. 4) `IChatClient` ends the middleware.

Reading the stack from bottom to top gives us the registration order, which becomes execution order in the pipeline.

Applying the Prepare–Handle–Invoke Middleware Pattern

This is the canonical composition for the *ChatClient* middleware types. Each concern is placed where it has the right visibility and the right timing: validation and sanitization before tokens are spent, response control around the model call, and tool supervision only when a function is about to be executed.

Listing 9.13 shows the full pipeline registration following the *Prepare → Handle → Invoke* mental model.

Listing 9.13 Full Pipeline Registration

```
IChatClient chatClient = new OpenAIClient(apiKey)
  .GetChatClient(model)
  .AsIChatClient()
  .AsBuilder()
  .Use(ChatClientSharedFunctions.LimitRequests)  //❶
  .Use(ChatClientSharedFunctions.RemoveEmail)  //❶
  .Use(ChatClientResponses.EnforceTokenBudget, null)  //❷
  .Use(ChatClientResponses.AddTimestamp, null)  //❷
  .UseFunctionInvocation(loggerFactory: null, configure: options =>
  {
    options.FunctionInvoker = async (context, ct) =>
    {
      var response = await Middleware.ChatClientFunctionCallings
        .ConstrainDistance(context, ct)  //❸
          ?? await Middleware.ChatClientFunctionCallings
        .AuditFunctionCalling(context, ct);  //❹
      return response;
    };
  })  //❺
  .Build();  //❻
```

❶ **Prepare layer: fail fast, sanitize input**
❷ **Handle layer: track tokens, enforce budget**
❸ **Interceptor: modify or block unsafe arguments**
❹ **Executor: invoke function and audit**
❺ **Invoke layer: MUST be last before .Build()**
❻ **Builds the ChatClient middleware for the chat client**

When all three middleware types compose correctly, the pipeline reads like a checklist. *SharedFunction* validates and sanitizes before any token is spent. *Response* wraps the LLM exchange and enforces the token budget. *FunctionCalling* controls each tool invocation individually, with the gate pattern ensuring the function executes exactly once. Remove a layer and lose a capability; misorder them and a concern may fire too late to matter.

IMPORTANT In practice, this is what we want from a *ChatClient* pipeline: outer layers that fail fast and sanitize, a middle layer that owns cost and observability, and an inner layer that supervises tools without leaking any of those responsibilities into agent code.

Common Middleware Anti-Patterns

Some common pitfalls in production:

Wrong registration order. Registration order equals execution order. Logging before sanitization means the log captures sensitive data before removal. Always sanitize first:

```
.Use(SharedFunctionMiddleware.RemoveEmail)  //❶
```

```
.Use(ResponseMiddleware.LogMessages, null)  //❷
```

❶ Correct: sanitizes first (e.g., "[REDACTED-EMAIL]")
❷ Then logs sanitized version (e.g., "[REDACTED-EMAIL]")

FunctionCalling is not innermost. `UseFunctionInvocation` must be the last middleware registered before `.Build()`. If placed outermost, its internal tool loop drives repeated calls back through the entire pipeline: budget checks, sanitization, and timing all fire once per tool call instead of once per request.

Firing frequency misunderstanding. SharedFunction and *Response* fire once per chat request, not once per agent run. A single agent run with tool calls sends multiple chat requests (one initial call and one per tool round-trip), so `LimitRequests` can increment multiple times within a single user prompt. Budget middleware depletes faster than the number of user queries suggests.

No ChatClient-layer protection. Relying only on agent middleware means rogue or misconfigured agents can bypass protection. The *ChatClient* layer acts as a universal failsafe because every call, from any agent or plain code, passes through it.

ChatClient middleware operates at the HTTP boundary with no agent context. This makes it ideal for universal concerns like cost control and privacy.

In the next chapter, we explore *Agent* middleware, where we have access to agent identity, session state, and the familiar next delegate pattern for all middleware types, including *FunctionCalling*.

Summary

- Middleware is essential for production agents. Without it, sensitive data leaks, costs spiral, and dangerous operations execute unchecked.
- Two middleware layers serve different purposes. *ChatClient* is the universal safety net for all agents. *Agent* provides per-agent control with session context and identity.
- Three middleware types map to prepare → handle → invoke. *SharedFunction* prepares input (ChatResponse invisible). *Response* handles LLM exchange (`ChatResponse` visible). *FunctionCalling* invokes tools (sees function result).
- Registration order determines execution order. *SharedFunction* outermost (fail fast). *Response* middle (track tokens). *FunctionCalling* innermost (must be last before `.Build()`).
- *FunctionCalling* uses interceptor/executor pattern. Interceptors return null to modify or block without invoking. Executor calls `InvokeAsync()` and returns non-null. One executor, multiple interceptors allowed.
- *FunctionCalling* must be innermost. If not last, its tool loop drives repeated pipeline passes. *SharedFunction* and *Response* fire per tool call instead of per chat request.
- Firing frequency differs by layer. *ChatClient* fires per chat request (multiple times per agent run with tools). Agent fires once per `RunAsync()` call.
- Sanitization differs by layer. *ChatClient* is transient (LLM sees clean data, session retains original). *Agent* is persistent (cleaned messages stored in history).
- Defense in depth requires both layers. Implement critical controls like email removal at both *ChatClient* (universal) and *Agent* (per-agent) layers.