9

Safeguarding the Kernel Function with Filters

This chapter covers

- What are a semi-autonomous agent and a copilot? Comparison.
- Introduction to filters
- Filter types: function invocation filters, prompt render filters, and auto function invocation filters
- OpenTelemetry with Semantic Kernel using Console Exporter

Imagine Robby the robot car driving through a challenging environment. To keep Robby safe and on the right track, we sometimes need to tweak his actions or even stop him for a quick human check. Semantic Kernel *filters* work the same way for your AI models: they step in when taking actions, such as when a function is called, or when a request is about to be sent to the AI. This is especially useful for scenarios like human-in-the-loop validation, where a person reviews the AI's actions before they proceed. Filters also help you monitor function calls, collect telemetry, and add custom logic for logging, error handling, or blocking unsafe content, just like Robby's sensors warn him about obstacles ahead.

In this chapter, we will learn how to build semi-autonomous agents and copilots.

9.1 What is a Semi-Autonomous Agent

A semi-autonomous agent in Semantic Kernel works like an autonomous agent but operates with constraints or checkpoints. Filters act as guardrails, intercepting plugin calls, prompt generation, or function execution to enforce compliance, safety, or human oversight. This allows for human-in-the-loop scenarios, compliance checks, or safety validation, where actions may need approval before proceeding. By using filters, developers can balance automation with control, ensuring the agent remains responsive while still governed by necessary rules or approvals.

9.2 What is a Copilot

A Semantic Kernel copilot acts as an intelligent assistant that executes tasks via function calls or planning. Unlike basic chatbots, it analyzes user intent, selects relevant functions from plugins, and runs them to achieve complex goals across services. The kernel manages step sequencing and dependencies, allowing copilots to chain actions and return results in user-friendly formats. This enables AI-driven productivity tools, workflow automation, or domain-specific assistants where the system not only understands requests but performs actual work.

We defined autonomous agents, chatbots, in the last chapters, and semi-autonomous agents, and copilots in the current one, so now is a good time to compare them side by side.

9.3 Comparing Chatbots, Copilots, and Agents

For a better understanding of the subtle differences between the chatbots, copilots, and agents, we will use a table.

Chatbot, Copilot, Semi-autonomous Agent, and Autonomous Agent side-by-side

Role	Autonomy Level	User Involvement	Typical Orchestration	Description / Typical Use Case
Chatbot	Low	Very High (conversation)	None / Static (query-response)	Conversational Q&A, follows user prompts, no actions or tool calls
Copilot	Semi-autonomous	High (interactive)	Static / Dynamic (user-guided, can use planners)	Assists user, suggests actions, expects user input/approval
Semi-autonomous Agent	Semi-autonomous	Moderate to High (user confirmation)	Static / Dynamic (user-supervised, can use planners, but expects user input at key points)	Executing with confirmation (human-in-the-loop)
Autonomous Agent	Autonomous	Low (goal-oriented)	Dynamic (planner-driven, goal-oriented)	Goals-oriented with minimal oversight

Table notes:

- Chatbot: Primarily for conversation and basic information retrieval; does not execute actions or chain tools.
- Copilot: Designed to augment user productivity, often by suggesting actions or automating steps, but always keeps the user "in the loop".
- Semi-autonomous Agent: Blends copilot and agent traits, can execute plans or workflows, but typically pauses for user confirmation on more complex or critical decisions.

- Autonomous Agent: Operates with the most independence, using planners and dynamic orchestration to achieve goals, often without user intervention.

Following, we will discuss the architectural diagrams of copilots and agents.

- AI Applications Layer
    - Copilot: Semi-autonomous assistant that suggests actions but requires human validation. For example, a coding assistant that proposes function calls but waits for approval.
    - Agent: Autonomous system that executes plans independently. For example, Robby the robot car avoiding obstacles without human intervention.
- AI Orchestration Layer:
    - Manual Planning: Direct execution of predefined plugin workflows. For example, explicitly calling `MotorsPlugin.forward 5` in code.
    - Automatic Planning: Uses Planners to dynamically generate sequences of plugin functions. For example, LLM breaking "avoid tree" into `turn_left → forward → turn_right` based on semantic similarity.
- Core Components:
    - Plugins: Native Functions and Semantic Functions.
    - Planners: Function calling
    - Filters: Audit and approvals
    - Connectors: Services and APIs
    - History: Chat conversation context
    - Memory: Long-term knowledge storage

This architecture enables flexible AI applications combining LLM reasoning (semantic functions) with deterministic execution (native functions), while maintaining safety through filters.

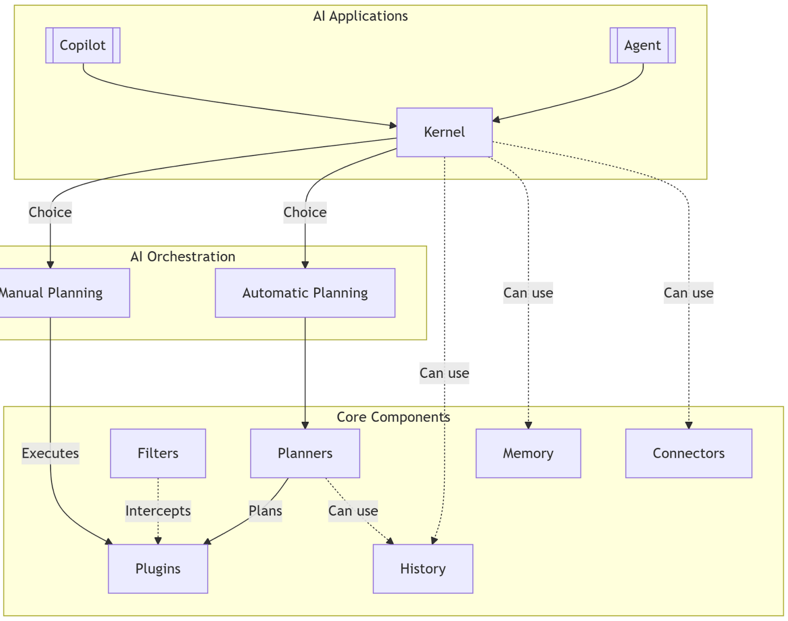

Figure 9.1 The diagram illustrates the architecture and dependencies in an AI orchestration system, highlighting the roles of Copilot and Agent, and how they interact with the core orchestration components.

Let’s explore next what the filters are and the different types of filters available in Semantic Kernel and how to implement them in your workflows.

9.4 Introducing Filtering

Filters in Semantic Kernel are interceptors that provide control and visibility over function execution and prompt handling.

Imagine Robby is about to cross a new, unpredictable road. His filters constantly read data from sensors to assess conditions ahead. If the filters detect deep mud, slippery spots, or hidden obstacles from these sensors, they stop Robby from driving forward, preventing him from getting stuck or damaged. By interpreting sensor data in real time, filters help Robby avoid trouble before it happens.

Or, let’s suppose Robby gets into trouble getting damaged due to unexpected road or weather conditions, his *black box* logs every sensor reading and decision. A black box is a device that records telemetry, sensor data, control commands, and other critical information for both unmanned aerial and terrestrial vehicles. This recorded data helps engineers investigate what happened and improve Robby’s safety for future trips.

As shown in the next Semantic Kernel high-level components diagram (figure 9.2), filters are core components responsible for intercepting and optionally modifying both input and output data.

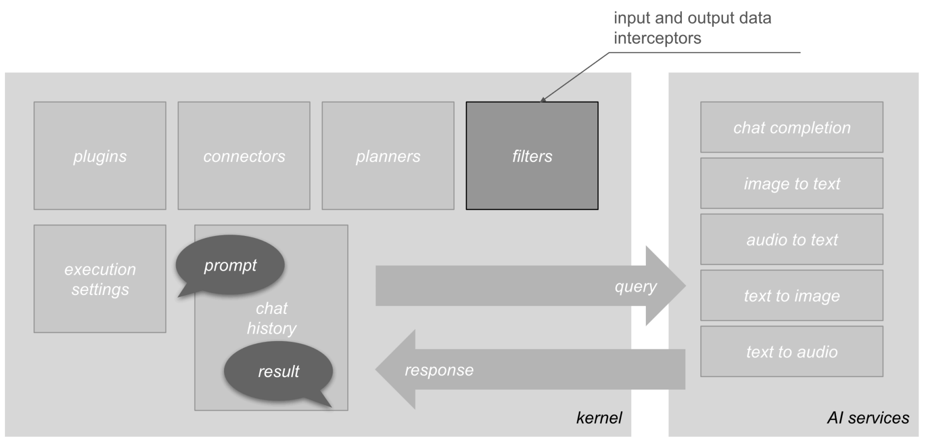

Figure 9.2 In Semantic Kernel high-level components diagram, filters are core components responsible for intercepting both input and output data.

Filters in Semantic Kernel are like security checkpoints: they inspect, validate, and can modify what goes into and comes out of your AI system. This ensures that both inputs and outputs are clean, safe, and appropriate. Filters are a critical layer for implementing security, observability, and control in your applications.

9.4.1 Filter Pipeline Architecture

Semantic Kernel's filter architecture is similar to ASP.NET *middleware*. In ASP.NET, middleware refers to software components arranged in a pipeline, each handling a specific concern such as authentication, logging, or error handling. Key aspects of this architecture include:

- Sequential Processing: Filters are arranged in a specific order. Each filter can inspect or modify data before and after the core function or prompt execution.
- Control Flow using next Delegate: Each filter receives a next delegate. Calling `await next(context)` passes control to the next filter in the sequence or, if it's the last filter, to the target function/LLM. Code executed after `await next(context)` runs in the reverse order of filter registration.
- Modularity and Separation of Concerns: This design allows encapsulating specific tasks like authentication, logging, validation, or content modification into distinct, reusable filter components, leading to cleaner and more maintainable code.
- Short-Circuiting: A filter can decide not to call `await next(context)`, effectively stopping the pipeline's execution early. This is useful for scenarios like input validation failures, authorization checks, or returning cached responses, which improves efficiency and security by preventing unnecessary downstream processing.
- Support for Responsible AI: The pipeline facilitates implementing responsible AI

practices by providing hooks to consistently enforce security, validation, and observability policies. (Responsible AI practices are principles and processes that ensure AI systems are developed and used ethically, transparently, safely, and fairly, prioritizing accountability, privacy, and minimizing bias or harm to individuals and society)

Middleware Architecture Diagram

Figure 9.3 illustrates the middleware architecture, showing the flow from kernel to LLM. Each filter processes requests and responses sequentially, enabling modular pre- and post-processing between the kernel and the LLM.

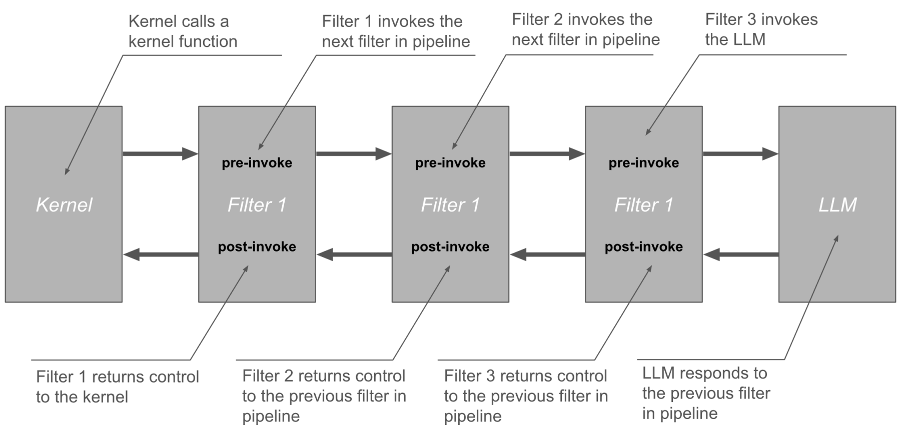
Kernel calls a kernel function
Filter 1 invokes the next filter in pipeline
Filter 2 invokes the next filter in pipeline
Filter 3 invokes the LLM
Kernel
pre-invoke
Filter 1
post-invoke
pre-invoke
Filter 1
post-invoke
pre-invoke
Filter 1
post-invoke
LLM
Filter 1 returns control to the kernel
Filter 2 returns control to the previous filter in pipeline
Filter 3 returns control to the previous filter in pipeline
LLM responds to the previous filter in pipeline

Figure 9.3 The image shows the middleware architecture with the flow from kernel to LLM, where requests pass sequentially through Filter 1, Filter 2, and Filter 3 before reaching the LLM, and responses travel back through the filters in reverse order to the kernel.

Figure 9.4 shows how the flow can stop at Filter 2, returning the response directly to the kernel and bypassing subsequent filters and the LLM:

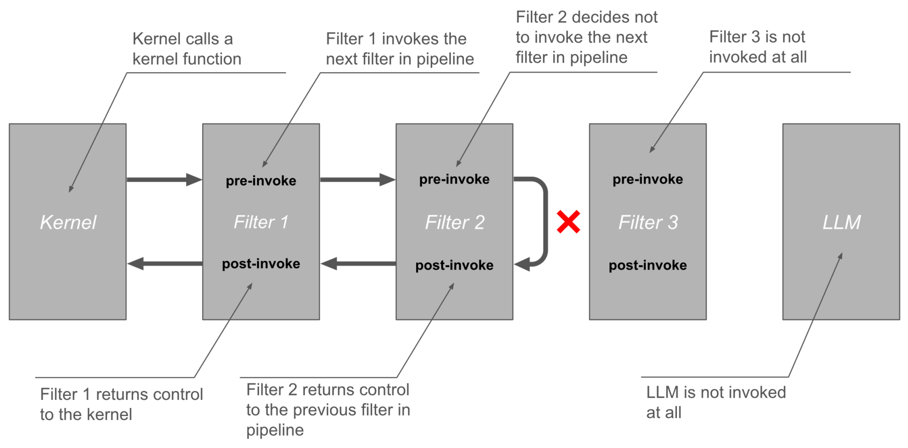
Kernel calls a kernel function
Filter 1 invokes the next filter in pipeline
Filter 2 decides not to invoke the next filter in pipeline
Filter 3 is not invoked at all
Kernel
pre-invoke
Filter 1
post-invoke
pre-invoke
Filter 2
post-invoke
pre-invoke
Filter 3
post-invoke
LLM
Filter 1 returns control to the kernel
Filter 2 returns control to the previous filter in pipeline
LLM is not invoked at all

Figure 9.4 The diagram shows middleware short-circuiting: processing stops at Filter 2, which returns a response directly to the Kernel, skipping later steps like Filter 3 and the LLM. This enables early termination for efficiency or validation.

Filters can be registered with the kernel in two ways:

1. Using dependency injection, before building the kernel:

```
var builder = Kernel.CreateBuilder();
builder.Services.AddSingleton<IFunctionInvocationFilter,
FirstFunctionFilter>();
builder.Services.AddSingleton<IFunctionInvocationFilter,
SecondFunctionFilter>();
builder.Services.AddSingleton<IFunctionInvocationFilter,
ThirdFunctionFilter>();
var kernel = builder.Build();
```

2. Directly on the kernel instance, using kernel properties:

```
kernel.FunctionInvocationFilters.Add(new FirstFunctionFilter());
kernel.FunctionInvocationFilters.Add(new ThirdFunctionFilter());
kernel.FunctionInvocationFilters.Insert(1, new SecondFunctionFilter());
```

Remember, registering filters directly on the kernel instance allows explicit control over their order.

The order of filters is important. As shown in figures 9.2 and 9.3, filters are invoked in the order they are registered. After the context is sent to the LLM, the response is processed by the filters in reverse order. In the example below, outbound processing occurs before calling `await next(context)`, and inbound processing happens after. This means outbound filters cascade in registration order, then, after the LLM responds, inbound filters cascade in reverse order.

Practical Example

First, let's create three simple and similar filters (listing 9.1). The purpose is to exemplify how the middleware works.

Listing 9.1 Function Invocation Filters for Middleware Exemplifying

```
using Microsoft.SemanticKernel;

namespace Filters;

public sealed class FirstFunctionFilter : IFunctionInvocationFilter    #A
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #B
  {
    Console.WriteLine($"  {nameof(FirstFunctionFilter)}.invoking
{context.Function.Name}");    #C
    await next(context);    #D
    Console.WriteLine($"  {nameof(FirstFunctionFilter)} invoked
{context.Function.Name}");    #E
  }
}

public sealed class SecondFunctionFilter : IFunctionInvocationFilter
#F
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)
  {
    Console.WriteLine($"  {nameof(SecondFunctionFilter)} invoking
{context.Function.Name}");
    await next(context);
    Console.WriteLine($"  {nameof(SecondFunctionFilter)} invoked
{context.Function.Name}");
  }
}

public sealed class ThirdFunctionFilter : IFunctionInvocationFilter    #G
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)
  {
    Console.WriteLine($"  {nameof(ThirdFunctionFilter)} invoking
{context.Function.Name}");
    await next(context);
    Console.WriteLine($"  {nameof(ThirdFunctionFilter)} invoked
{context.Function.Name}");
  }
}
```

#A Function invocation filter class
#B Async function that gets triggered when a function is invoked
#C Code that executes just before the function is called
#D Execute the function with the current context
#E Code that executes after the function is called
#F Another function invocation filter class
#G Another function invocation filter class

Each class (FirstFunctionFilter, SecondFunctionFilter, ThirdFunctionFilter) represents a distinct piece of middleware. The key method is `OnFunctionInvocationAsync`. This method wraps the actual function call.

Code before `await next(context)`: This is the "request" side of the middleware. It executes before the next filter in the chain, or the actual kernel function is called. In our example, it prints the "invoking" message.

Code `await next(context)`: This crucial line passes control to the next component in the pipeline. If there's another filter registered, next calls that filter's `OnFunctionInvocationAsync`. If it's the last filter, the next triggers the execution of the target kernel function.

Code after `await next(context)`: This is the "response" side of the middleware. It executes after the next call returns, meaning all subsequent filters and the kernel function itself have completed. In our example, it prints the "invoked" message.

Let’s observe in listing 9.2 the full code that uses our three filters.

Listing 9.2 Working with Function Invocation Filters Middleware

```
using Microsoft.SemanticKernel;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel.ChatCompletion;
using System.Diagnostics;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Plugins.Native;
using Filters;

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

var builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion(
  modelId: configuration["OpenAI:ModelId"]!,
  apiKey: configuration["OpenAI:ApiKey"]!);
builder.Services.AddSingleton<IFunctionInvocationFilter,
FirstFunctionFilter>();    #A
builder.Services.AddSingleton<IFunctionInvocationFilter,
SecondFunctionFilter>();    #B
builder.Services.AddSingleton<IFunctionInvocationFilter,
ThirdFunctionFilter>();    #C
var kernel = builder.Build();    #D

kernel.ImportPluginFromType<MotorsPlugin>();    #E

var executionSettings = new OpenAIPromptExecutionSettings
{
  FunctionChoiceBehavior = FunctionChoiceBehavior.Required()
};    #F

var history = new ChatHistory();    #G
history.AddSystemMessage("""
  You are an AI assistant controlling a robot car.
  """);    #H
history.AddUserMessage("""
  Perform these steps:
    {{forward 100}}

  Respond only with the moves, without any additional explanations.
  """);    #I

var chat = kernel.GetRequiredService<IChatCompletionService>();    #J
var response = await chat.GetChatMessageContentAsync(history,
executionSettings, kernel);    #K
Console.WriteLine($"RESPONSE: {response}");    #L
```
#A Register FirstFunctionFilter
#B Register SecondFunctionFilter
#C Register ThirdFunctionFilter
#D Build the kernel
#E Import a plugin with function kernels to be invoked
#F Set function choice behavior to Required (but can be Auto as well)
#G Initialize chat history
#H Add system message to chat history
#I Add user message to chat history
#J Get chat instance using dependency injection
#K Invoke response with chat history
#L Print response to console

Output:
```
  FirstFunctionFilter.invoking forward
  SecondFunctionFilter invoking forward
  ThirdFunctionFilter invoking forward
[09:43:48:321] ACTION: Forward: 100m
  ThirdFunctionFilter invoked forward
  SecondFunctionFilter invoked forward
  FirstFunctionFilter invoked forward
RESPONSE: moved forward for 100 meters.
```
We can notice that middleware with Filter1 added first, Filter2 added second, and Filter3 added last, creates a nested execution flow as follows:

Request In □
 `FirstFunctionFilter`: "invoking forward" (Before await next)
  `SecondFunctionFilter`: "invoking forward" (Before await next)
   `ThirdFunctionFilter`: "invoking forward" (Before await next)
    □ Kernel executes the 'forward' function (ACTION: Forward: 100m) □
   `ThirdFunctionFilter`: "invoked forward" (After await next)
  `SecondFunctionFilter`: "invoked forward" (After await next)
 `FirstFunctionFilter`: "invoked forward" (After await next)
□ Response Out

The filters are registered using dependency injection, and they are triggered when the kernel processes prompt messages that include functions (tools). The filters are not triggered by default because the function calls are not triggered by default. We need the function choice behavior to be set. Let's say it in other words, if the function choice behavior is null or set to `None()`, the filters will not get triggered, because the function calling is not active. The `Auto()` or `Required()` function choice behavior is responsible for determining when function invocation occurs.

There are more filter types in Semantic Kernel, covering a large variety of scenarios. We will discuss the filter types in the next section.

9.5 Types of filters

Semantic Kernel supports several filter types, each designed to intercept and process different stages of kernel execution:

- Function Invocation Filters: Triggered every time a `KernelFunction` is invoked. These filters provide access to function metadata and arguments, support exception handling, result overriding, and retries.
- Prompt Render Filters: Activated before prompt rendering, allowing inspection and modification of prompts sent to AI services. Useful for enforcing compliance, redacting sensitive information, or applying semantic caching.
- Auto Function Invocation Filters: Manage function calls automatically initiated by the AI model, providing context such as chat history and execution state.

9.5.1 Function Invocation Filters

Function invocation filters enable inspection, validation, modification, and control over function execution. They are commonly used for input validation, access control, rate limiting, caching, and fallback handling.

Basic Example

An implementation example for a filter dealing with authentication:

```
public class AuthFilter : IFunctionInvocationFilter    #A
{
    public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #B
    {
        var userRole = context.Arguments["userRole"]?.ToString();    #C
        if (userRole != "Admin")
            throw new UnauthorizedAccessException("Insufficient
permissions");    #D
        await next(context);    #E
    }
}
```

#A Function invocation filter class
#B Async function that gets triggered when a function is invoked
#C Argument userRole is read from context
#D Throw exception if the role doesn't match
#E Proceed to the next filter if authorized

In the previous code sample, let's observe that the `next(context)` may or may not be called, depending on the provided argument "`userRole`".

If the user role doesn't match, then the filter will throw an exception, short-circuiting the middleware, therefore request doesn't reach the LLM.

Some other common use cases:

- Input Validation: Reject malformed JSON in API parameters.
- Rate Limiting: Track function calls per user to prevent abuse.
- Caching: Return cached results for expensive functions when possible.
- Fallback Handling: Retry with a different AI model if the initial call fails.

Practical Example

Next, let's look at some filters (in listing 9.3) that will be used in the complete code example below. We want to learn how various function invocation filters work.

Filter examples:

- `BackwardConfirmationFilter` (listing 9.3) – a filter that checks if the called function name matches the value `backward` and requests approval.
- `FunctionVerboseFilter` (listing 9.4) – a filter that audits the function calling.

- `HumanInTheLoopFilter` (listing 9.5) – a filter that requests approval for any function to run.
- `MissingArgumentFilter` (listing 9.6) – a filter that provides the argument default values if they are empty.

Listing 9.3 Function Invocation Filters - BackwardConfirmationFilter

```
using Microsoft.SemanticKernel;
namespace Filters;

public sealed class BackwardConfirmationFilter :
IFunctionInvocationFilter    #A
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #B
  {
    if (context.Function.Name == "backward")    #C
    {
      string message;
      Console.WriteLine($"  Moving backward is cowardly, are you sure
([y]/n)?");    #D
      var yesNoResponse = Console.ReadKey(true);    #E
      if (yesNoResponse.Key == ConsoleKey.Y || yesNoResponse.Key ==
ConsoleKey.Enter)    #F
      {
        await next(context);    #G
      }
      else
      {
        message = "  Moving backward cancelled! Continue.";
        Console.WriteLine(message);    #H
      }
    }
    else
    {
      await next(context);    #I
    }
  }
}
```

#A Filter class for intercepting 'backward' movement
#B Async method triggered when a function is invoked
#C Check for movement type if it's 'backward'
#D Ask user if they agrees or not
#E Read the typed key without echo (typed key is not shown in console)
#F Check if key is 'y' or ‘Enter’
#G Calls the next middleware filter
#H Print a feedback to console if key is not 'y' or 'Enter'
#I Call the next middleware filter if the movement is not 'backward'

Listing 9.4 Function Invocation Filters - FunctionVerboseFilter

```
using Microsoft.SemanticKernel;
namespace Filters;

public sealed class FunctionVerboseFilter : IFunctionInvocationFilter
#A
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #B
```

```
  {
    Console.WriteLine($"  {nameof(FunctionVerboseFilter)} invoking
{context.Function.Name}");    #C
    await next(context);    #D
    Console.WriteLine($"  {nameof(FunctionVerboseFilter)} invoked
{context.Function.Name}");    #E
  }
}
```

#A Filter class for auditing function calling
#B Async method triggered when a function is invoked
#C Print to console what function is going to be invoked
#D Call the next middleware filter
#E Print to console what function was invoked

Listing 9.5 Function Invocation Filters - HumanInTheLoopFilter

```
using Microsoft.SemanticKernel;
namespace Filters;

public sealed class HumanInTheLoopFilter : IFunctionInvocationFilter
#A
{
  private const int TimeoutSeconds = 3;    #B

  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #C
  {
    Console.WriteLine($"  Function '{context.Function.Name}' is about to
be invoked. Proceed ([y]/n)?");    #D
    var yesNoResponse = Console.ReadKey(true);    #E
    if (yesNoResponse == ConsoleKey.Y || yesNoResponse ==
ConsoleKey.Enter)    #F
    {
      await next(context);    #G
    }
    else
    {
      var message = "  Command cancelled! Continue.";
      context.Result = new FunctionResult(context.Result, message);    #H
      Console.WriteLine(message);    #I
    }
  }
}
```

#A Filter class for intercepting 'backward' movement
#B TimeoutSeconds constant to keep the elapsing time timeout for reading a key
#C Async method triggered when a function is invoked
#D Ask the user if to proceed or not with function invoking
#E Read the typed key without echo (typed key is not shown in console)
#F Check if key is 'y' or 'Enter'
#G Calls the next middleware filter
#H Set the filter context with a message informing that the function was cancelled
#I Print to console the message informing that the function was cancelled

Listing 9.6 Function Invocation Filters - MissingArgumentFilter

```
using Microsoft.SemanticKernel;
namespace Filters;

public sealed class MissingArgumentFilter : IFunctionInvocationFilter
#A
{
  public async Task OnFunctionInvocationAsync(FunctionInvocationContext
context, Func<FunctionInvocationContext, Task> next)    #B
  {
    if (context.Function.Name.Equals("forward",
StringComparison.InvariantCultureIgnoreCase)
      || context.Function.Name.Equals("backward",
StringComparison.InvariantCultureIgnoreCase))    #C
    {
      if (!context.Arguments.TryGetValue("distance", out var _))    #D
      {
        int distance = 1;
        Console.WriteLine($"  Forcing 'distance' argument to
{distance}");
        context.Arguments["distance"] = distance;    #E
      }
    }

    if (context.Function.Name.Equals("turn_left",
StringComparison.InvariantCultureIgnoreCase)
      || context.Function.Name.Equals("turn_right",
StringComparison.InvariantCultureIgnoreCase))    #F
    {
      if (!context.Arguments.TryGetValue("angle", out var _))    #G
      {
        int angle = 90;
        Console.WriteLine($"  Forcing 'angle' argument to {angle}");
        context.Arguments["angle"] = angle;    #H
      }
    }

    await next(context);    #I
  }
}
```

#A Filter class for providing default values for arguments if they are missing
#B Async method triggered when a function is invoked
#C Check for movement type if it's 'backward' or 'forward'
#D Try getting 'distance' argument value
#E Provide default argument value for 'distance' if they are empty
#F Check for movement type if it's 'turn_left' or 'turn_right'
#G Try getting 'angle' argument value
#H Provide default argument value for 'angle' if they are empty
#I Call the next middleware filter

And now, here is the complete code listing (9.7) that uses the previously declared filters:

Listing 9.7 Working Function Invocation Filters

```
using Microsoft.SemanticKernel;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;
using System.Diagnostics;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Plugins.Native;
using Filters;

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

```
var builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion(
  modelId: configuration["OpenAI:ModelId"]!,
  apiKey: configuration["OpenAI:ApiKey"]!);
builder.Services.AddSingleton<IFunctionInvocationFilter,
BackwardConfirmationFilter>();    #A
builder.Services.AddSingleton<IFunctionInvocationFilter,
HumanInTheLoopFilter>();    #B
builder.Services.AddSingleton<IFunctionInvocationFilter,
MissingArgumentFilter>();    #C
builder.Services.AddSingleton<IFunctionInvocationFilter,
FunctionVerboseFilter>();    #D
var kernel = builder.Build();    #E

kernel.ImportPluginFromType<MotorsPlugin>();    #F

var executionSettings = new OpenAIPromptExecutionSettings
{
  FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};    #G

var history = new ChatHistory();    #H
history.AddSystemMessage("""
  You are an AI assistant controlling a robot car.
  """);    #I
history.AddUserMessage("""
  Your task is to break down complex commands into a sequence of these
basic moves: forward, backward, turn left, turn right, and stop.
  Respond only with the moves, without any additional explanations.
  Use the tools you know to perform the moves.

  Complex command:
  "There is danger in front of you, run away: backward, turn left, turn
right, backward!"
  """);    #J

var chat = kernel.GetRequiredService<IChatCompletionService>();    #K
var response = await chat.GetChatMessageContentAsync(history,
executionSettings, kernel);    #L
Console.WriteLine($"RESPONSE: {response}");    #M
```

#A Register BackwardConfirmationFilter
#B Register HumanInTheLoopFilter
#C Register MissingArgumentFilter
#D Register FunctionVerboseFilter
#E Build the kernel
#F Import a plugin with function kernels to be invoked
#G Set function choice behavior to Required (but can be Auto as well)
#H Initialize chat history
#I Add system message to chat history
#J Add user message to chat history
#K Get chat instance using dependency injection
#L Invoke response with chat history
#M Print to console the response

Output:

```
  Moving backward is cowardly, are you sure ([y]/n)?
  Function 'backward' is about to be invoked. Proceed ([y]/n)?
  Forcing 'distance' argument to 1
  FunctionVerboseFilter invoking backward
```

```
[09:48:10:268] ACTION: Backward: 1m
  FunctionVerboseFilter invoked backward
  Function 'turn_left' is about to be invoked. Proceed ([y]/n)?
  FunctionVerboseFilter invoking turn_left
[09:48:16:293] ACTION: TurnLeft: 90°
  FunctionVerboseFilter invoked turn_left
  Function 'turn_right' is about to be invoked. Proceed ([y]/n)?
  FunctionVerboseFilter invoking turn_right
[09:48:22:326] ACTION: TurnRight: 90°
  FunctionVerboseFilter invoked turn_right
  Moving backward is cowardly, are you sure ([y]/n)?
  Function 'backward' is about to be invoked. Proceed ([y]/n)?
  Forcing 'distance' argument to 1
  FunctionVerboseFilter invoking backward
[09:48:35:183] ACTION: Backward: 1m
  FunctionVerboseFilter invoked backward
RESPONSE: The robot executed the following sequence of moves: moved
backward, turned left, turned right, and moved backward again.
```

We can observe that all functions and all their filters are called with 'y' answer or default ('Enter'), therefore all filters had the chance to run for each function.

Let's run again, responding 'n' for some filters if asked to proceed.

Output:

```
  Moving backward is cowardly, are you sure ([y]/n)?
  Function 'backward' is about to be invoked. Proceed ([y]/n)?
  FunctionVerboseFilter invoking backward
[09:49:53:490] ACTION: Backward: 1m
  FunctionVerboseFilter invoked backward
  Function 'turn_left' is about to be invoked. Proceed ([y]/n)?
  FunctionVerboseFilter invoking turn_left
[09:49:59:511] ACTION: TurnLeft: 90°
  FunctionVerboseFilter invoked turn_left
  Function 'turn_right' is about to be invoked. Proceed ([y]/n)?
  Command cancelled! Continue.
  Moving backward is cowardly, are you sure ([y]/n)?
  Moving backward cancelled! Continue.
RESPONSE: - Move backward for 1 meter
- Turn left 90°
- Turn right 90°
- Move backward for 1 meter
```

An important point: when the user responds negatively ('n') during the `turn_left` function call, the `BackwardConfirmationFilter` intercepts this response. This negative input stops any further middleware filters from running for the current function. However, the kernel can still process any subsequent functions in the pipeline.