

Introduction to Semantic Kernel

Article • 06/24/2024

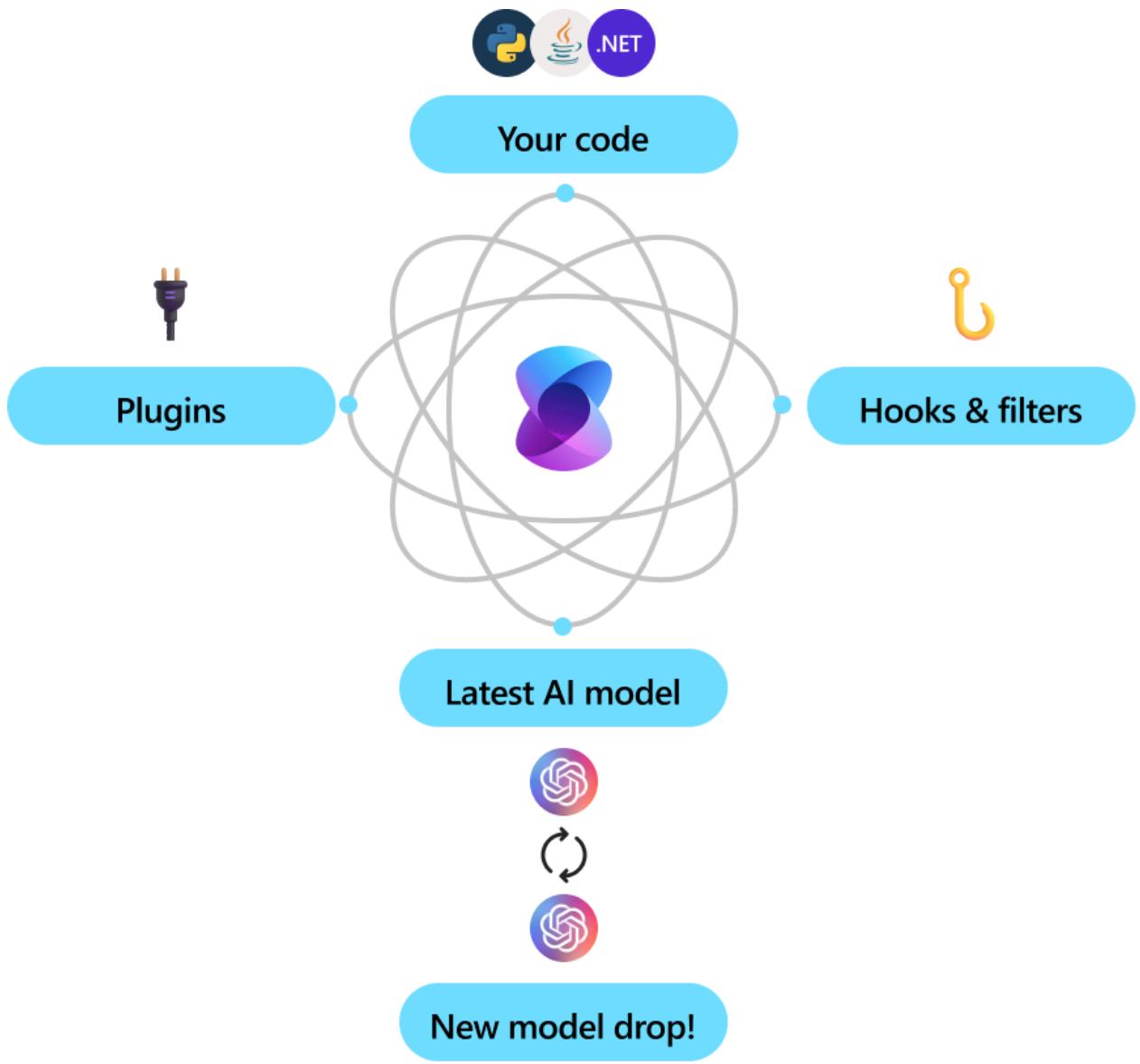
Semantic Kernel is a lightweight, open-source development kit that lets you easily build AI agents and integrate the latest AI models into your C#, Python, or Java codebase. It serves as an efficient middleware that enables rapid delivery of enterprise-grade solutions.

Enterprise ready

Microsoft and other Fortune 500 companies are already leveraging Semantic Kernel because it's flexible, modular, and observable. Backed with security enhancing capabilities like telemetry support, and hooks and filters so you'll feel confident you're delivering responsible AI solutions at scale.

Version 1.0+ support across C#, Python, and Java means it's reliable, committed to non breaking changes. Any existing chat-based APIs are easily expanded to support additional modalities like voice and video.

Semantic Kernel was designed to be future proof, easily connecting your code to the latest AI models evolving with the technology as it advances. When new models are released, you'll simply swap them out without needing to rewrite your entire codebase.

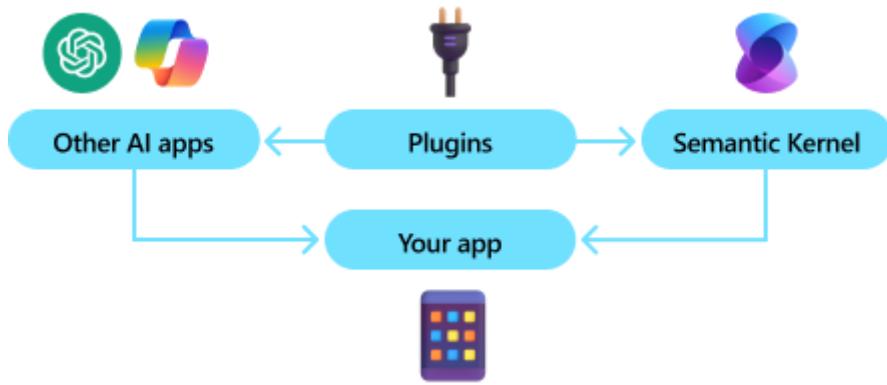


Automating business processes

Semantic Kernel combines prompts with [existing APIs](#) to perform actions. By describing your existing code to AI models, they'll be called to address requests. When a request is made the model calls a function, and Semantic Kernel is the middleware translating the model's request to a function call and passes the results back to the model.

Modular and extensible

By adding your existing code as a plugin, you'll maximize your investment by flexibly integrating AI services through a set of out-of-the-box connectors. Semantic Kernel uses OpenAPI specifications (like Microsoft 365 Copilot) so you can share any extensions with other pro or low-code developers in your company.



Get started

Now that you know what Semantic Kernel is, get started with the quick start guide. You'll build agents that automatically call functions to perform actions faster than any other SDK out there.

[Quickly get started](#)

Getting started with Semantic Kernel

Article • 06/24/2024

In just a few steps, you can build your first AI agent with Semantic Kernel in either Python, .NET, or Java. This guide will show you how to...

- Install the necessary packages
- Create a back-and-forth conversation with an AI
- Give an AI agent the ability to run your code
- Watch the AI create plans on the fly

Installing the SDK

Semantic Kernel has several NuGet packages available. For most scenarios, however, you typically only need `Microsoft.SemanticKernel`.

You can install it using the following command:

Bash

```
dotnet add package Microsoft.SemanticKernel
```

For the full list of Nuget packages, please refer to the [supported languages article](#).

Quickly get started with notebooks

If you're a Python or C# developer, you can quickly get started with our notebooks. These notebooks provide step-by-step guides on how to use Semantic Kernel to build AI agents.

```

!python -m pip install semantic-kernel==0.9.6b1

from services import Service
# Select a service to use for this notebook (available services: OpenAI, AzureOpenAI, HuggingFace)
selectedService = Service.OpenAI

from semantic_kernel import Kernel
from semantic_kernel.connectors.ai.openai import AzureChatCompletion, OpenAIChatCompletion
from semantic_kernel.utils.settings import azure_openai_settings_from_dot_env, openai_settings_from_dot_env

kernel = Kernel()

if selectedService == Service.AzureOpenAI:
    deployment, api_key, endpoint = azure_openai_settings_from_dot_env()
    service_id = "aoai_chat" # used later in the notebook
    azure_chat_service = AzureChatCompletion(
        service_id=service_id, deployment_name="gpt-35-turbo", endpoint=endpoint, api_key=api_key
    ) # set the deployment name to the value of your chat model
    kernel.add_service(azure_chat_service)

# Configure OpenAI service
if selectedService == Service.OpenAI:
    api_key, org_id = openai_settings_from_dot_env()
    service_id = "oai_chat" # used later in the notebook
    oai_chat_service = OpenAIChatCompletion(

```

Writing your first console app

C#

```

// Import packages
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

// Create a kernel with Azure OpenAI chat completion
var builder = Kernel.CreateBuilder().AddAzureOpenAIChatCompletion(modelId,
endpoint, apiKey);

// Add enterprise components
builder.Services.AddLogging(services =>
services.AddConsole().SetMinimumLevel(LogLevel.Trace));

// Build the kernel
Kernel kernel = builder.Build();
var chatCompletionService =
kernel.GetRequiredService<IChatCompletionService>();

// Add a plugin (the LightsPlugin class is defined below)
kernel.Plugins.AddFromType<LightsPlugin>("Lights");

// Enable planning
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};

// Create a history store the conversation
var history = new ChatHistory();

```

```

// Initiate a back-and-forth chat
string? userInput;
do {
    // Collect user input
    Console.Write("User > ");
    userInput = Console.ReadLine();

    // Add user input
    history.AddUserMessage(userInput);

    // Get the response from the AI
    var result = await chatCompletionService.GetChatMessageContentAsync(
        history,
        executionSettings: openAIPromptExecutionSettings,
        kernel: kernel);

    // Print the results
    Console.WriteLine("Assistant > " + result);

    // Add the message from the agent to the chat history
    history.AddMessage(result.Role, result.Content ?? string.Empty);
} while (userInput is not null)

```

The following back-and-forth chat should be similar to what you see in the console. The function calls have been added below to demonstrate how the AI leverages the plugin behind the scenes.

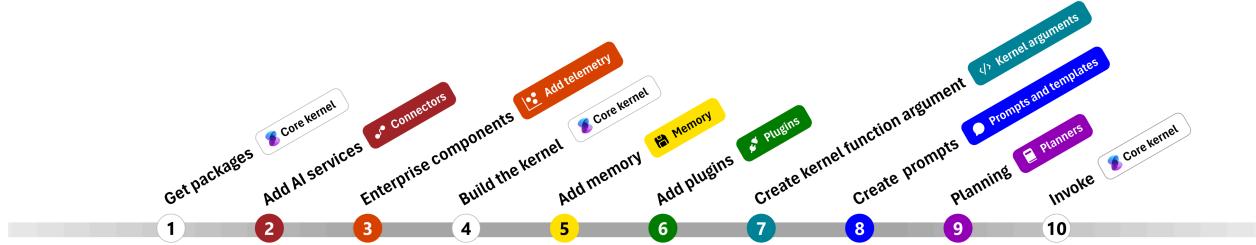
[\[\] Expand table](#)

Role	Message
● User	Please toggle the light
● Assistant (function call)	LightsPlugin.GetState()
● Tool	off
● Assistant (function call)	LightsPlugin.ChangeState(true)
● Tool	on
● Assistant	The light is now on

If you're interested in understanding more about the code above, we'll break it down in the next section.

Understanding the code

To make it easier to get started building enterprise apps with Semantic Kernel, we've created a step-by-step that guides you through the process of creating a kernel and using it to interact with AI services.



In the following sections, we'll unpack the above sample by walking through steps 1, 2, 3, 4, 6, 9, and 10. Everything you need to build a simple agent that is powered by an AI service and can run your code.

1. [Import packages](#)
2. [Add AI services](#)
3. [Enterprise components](#)
4. [Build the kernel](#)
5. Add memory (skipped)
6. [Add plugins](#)
7. Create kernel arguments (skipped)
8. Create prompts (skipped)
9. [Planning](#)
10. [Invoke](#)

1) Import packages

For this sample, we first started by importing the following packages:

```
C#  
  
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.ChatCompletion;  
using Microsoft.SemanticKernel.Connectors.OpenAI;
```

2) Add AI services

Afterwards, we add the most important part of a kernel: the AI services that you want to use. In this example, we added an Azure OpenAI chat completion service to the kernel builder.

(!) Note

In this example, we used Azure OpenAI, but you can use any other chat completion service. To see the full list of supported services, refer to the [supported languages article](#). If you need help creating a different service, refer to the [AI services article](#). There, you'll find guidance on how to use OpenAI or Azure OpenAI models as services.

```
C#
```

```
// Create kernel
var builder = Kernel.CreateBuilder()
builder.AddAzureOpenAIChatCompletion(modelId, endpoint, apiKey);
```

3) Add enterprise services

One of the main benefits of using Semantic Kernel is that it supports enterprise-grade services. In this sample, we added the logging service to the kernel to help debug the AI agent.

```
C#
```

```
builder.Services.AddLogging(services =>
services.AddConsole().SetMinimumLevel(LogLevel.Trace));
```

4) Build the kernel and retrieve services

Once the services have been added, we then build the kernel and retrieve the chat completion service for later use.

```
C#
```

```
Kernel kernel = builder.Build();

// Retrieve the chat completion service
var chatCompletionService =
kernel.Services.GetRequiredService<IChatCompletionService>();
```

6) Add plugins

With plugins, can give your AI agent the ability to run your code to retrieve information from external sources or to perform actions. In the above example, we added a plugin that allows the AI agent to interact with a light bulb. Below, we'll show you how to create this plugin.

Create a native plugin

Below, you can see that creating a native plugin is as simple as creating a new class.

In this example, we've created a plugin that can manipulate a light bulb. While this is a simple example, this plugin quickly demonstrates how you can support both...

1. [Retrieval Augmented Generation \(RAG\)](#) by providing the AI agent with the state of the light bulb
2. And [task automation](#) by allowing the AI agent to turn the light bulb on or off.

In your own code, you can create a plugin that interacts with any external service or API to achieve similar results.

C#

```
using System.ComponentModel;
using Microsoft.SemanticKernel;

public class LightsPlugin
{
    // Mock data for the lights
    private readonly List<LightModel> lights = new()
    {
        new LightModel { Id = 1, Name = "Table Lamp", IsOn = false },
        new LightModel { Id = 2, Name = "Porch light", IsOn = false },
        new LightModel { Id = 3, Name = "Chandelier", IsOn = true }
    };

    [KernelFunction("get_lights")]
    [Description("Gets a list of lights and their current state")]
    [return: Description("An array of lights")]
    public async Task<List<LightModel>> GetLightsAsync()
    {
        return lights
    }

    [KernelFunction("change_state")]
    [Description("Changes the state of the light")]
    [return: Description("The updated state of the light; will return null if the light does not exist")]
    public async Task<LightModel?> ChangeStateAsync(int id, bool isOn)
    {
        var light = lights.FirstOrDefault(light => light.Id == id);
        if (light != null)
            light.IsOn = isOn;
        return light;
    }
}
```

```

        if (light == null)
    {
        return null;
    }

    // Update the light with the new state
    light.IsOn = isOn;

    return light;
}
}

public class LightModel
{
    [JsonPropertyName("id")]
    public int Id { get; set; }

    [JsonPropertyName("name")]
    public string Name { get; set; }

    [JsonPropertyName("is_on")]
    public bool? IsOn { get; set; }
}

```

Add the plugin to the kernel

Once you've created your plugin, you can add it to the kernel so the AI agent can access it. In the sample, we added the `LightsPlugin` class to the kernel.

C#

```

// Add the plugin to the kernel
kernel.Plugins.AddFromType<LightsPlugin>("Lights");

```

9) Planning

Semantic Kernel leverages [function calling](#)—a native feature of most LLMs—to provide [planning](#). With function calling, LLMs can request (or call) a particular function to satisfy a user's request. Semantic Kernel then marshals the request to the appropriate function in your codebase and returns the results back to the LLM so the AI agent can generate a final response.

To enable automatic function calling, we first need to create the appropriate execution settings so that Semantic Kernel knows to automatically invoke the functions in the kernel when the AI agent requests them.

C#

```
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};
```

10) Invoke

Finally, we invoke the AI agent with the plugin. The sample code demonstrates how to generate a [non-streaming response](#), but you can also generate a [streaming response](#) by using the `GetStreamingChatMessageContentAsync` method.

C#

```
// Create chat history
var history = new ChatHistory();

// Get the response from the AI
var result = await chatCompletionService.GetChatMessageContentAsync(
    history,
    executionSettings: openAIPromptExecutionSettings,
    kernel: kernel
);
```

Next steps

In this guide, you learned how to quickly get started with Semantic Kernel by building a simple AI agent that can interact with an AI service and run your code. To see more examples and learn how to build more complex AI agents, check out our [in-depth samples](#).

Deep dive into Semantic Kernel

Article • 06/24/2024

If you want to dive into deeper into Semantic Kernel and learn how to use more advanced functionality not explicitly covered in our Learn documentation, we recommend that you check out our concepts samples that individually demonstrate how to use specific features within the SDK.

Each of the SDKs (Python, C#, and Java) have their own set of samples that walk through the SDK. Each sample is modelled as a test case within our main repo, so you're always guaranteed that the sample will work with the latest nightly version of the SDK! Below are most of the samples you'll find in our concepts project.

[View all C# concept samples on GitHub](#)

-  Example01_NativeFunctions.cs
-  Example02_Pipeline.cs
-  Example03_Variables.cs
-  Example04_CombineLLMPromptsAndNativeCode.cs
-  Example05_InlineFunctionDefinition.cs
-  Example06_TemplateLanguage.cs
-  Example07_BingAndGoogleSkills.cs
-  Example08_RetryHandler.cs

 [Expand table](#)

Category	Sample Name	Link
Agents	ComplexChat_NestedShopper	Link ↗
	Legacy_AgentAuthoring	Link ↗
	Legacy_AgentCharts	Link ↗
	Legacy_AgentCollaboration	Link ↗

Category	Sample Name	Link ↗
	Legacy_AgentDelegation	Link ↗
	Legacy_AgentTools	Link ↗
	Legacy_Agents	Link ↗
	Legacy_ChatCompletionAgent	Link ↗
	MixedChat_Agents	Link ↗
	OpenAIAssistant_ChartMaker	Link ↗
	OpenAIAssistant_CodeInterpreter	Link ↗
	OpenAIAssistant_Retrieval	Link ↗
Audio to text	OpenAI_AudioToText	Link ↗
Automatic function calling	Gemini_FunctionCalling	Link ↗
	OpenAI_FunctionCalling	Link ↗
Semantic caching	SemanticCachingWithFilters	Link ↗
Chat completion	AzureOpenAIWithData_ChatCompletion	Link ↗
	ChatHistoryAuthorName	Link ↗
	ChatHistorySerialization	Link ↗
	Connectors_CustomHttpClient	Link ↗
	Connectors_KernelStreaming	Link ↗
	Connectors_WithMultipleLLMs	Link ↗
	Google_GeminiChatCompletion	Link ↗
	Google_GeminiChatCompletionStreaming	Link ↗
	Google_GeminiGetModelResult	Link ↗
	Google_GeminiVision	Link ↗
	OpenAI_ChatCompletion	Link ↗
	OpenAI_ChatCompletionMultipleChoices	Link ↗
	OpenAI_ChatCompletionStreaming	Link ↗
	OpenAI_ChatCompletionStreamingMultipleChoices	Link ↗

Category	Sample Name	Link ↗
	OpenAI_ChatCompletionWithVision	Link ↗
	OpenAI_CustomAzureOpenAIClient	Link ↗
	OpenAI_UsingLogitBias	Link ↗
	OpenAI_FunctionCalling	Link ↗
	MistralAI_ChatPrompt	Link ↗
	MistralAI_FunctionCalling	Link ↗
	MistralAI_StreamingFunctionCalling	Link ↗
Dependency injection	HttpClient_Registration	Link ↗
	HttpClient_Resiliency	Link ↗
	Kernel_Building	Link ↗
	Kernel_Injecting	Link ↗
Filtering	AutoFunctionInvocationFiltering	Link ↗
	FunctionInvocationFiltering	Link ↗
	Legacy_KernelHooks	Link ↗
	PromptRenderFiltering	Link ↗
	RetryWithFilters	Link ↗
	PIIDetectionWithFilters	Link ↗
Plugin functions	Arguments	Link ↗
	FunctionResult_Metadata	Link ↗
	FunctionResult_StronglyTyped	Link ↗
	MethodFunctions	Link ↗
	MethodFunctions_Advanced	Link ↗
	MethodFunctions_Types	Link ↗
	PromptFunctions_Inline	Link ↗
	PromptFunctions_MultipleArguments	Link ↗
Image to text	HuggingFace_ImageToText	Link ↗

Category	Sample Name	Link ↗
Local models	HuggingFace_ChatCompletionWithTGI	Link ↗
	MultipleProviders_ChatCompletion	Link ↗
Memory	HuggingFace_EMBEDDINGGENERATION	Link ↗
	MemoryStore_CustomReadOnly	Link ↗
Text	SemanticTextMemory_Building	Link ↗
	TextChunkerUsage	Link ↗
Text	TextChunkingAndEmbedding	Link ↗
	TextMemoryPlugin_GeminiEmbeddingGeneration	Link ↗
Text	TextMemoryPlugin_MultipleMemoryStore	Link ↗
	TextMemoryPlugin_TextMemory	Link ↗
Planning	FunctionCallStepwisePlanning	Link ↗
	HandlebarsPlanning	Link ↗
Plugins	ApiManifestBasedPlugins	Link ↗
	ConversationSummaryPlugin	Link ↗
Plugins	CreatePluginFromOpenAI_AzureKeyVault	Link ↗
	CreatePluginFromOpenApiSpec_Github	Link ↗
Plugins	CreatePluginFromOpenApiSpec_Jira	Link ↗
	CustomMutablePlugin	Link ↗
Plugins	DescribeAllPluginsAndFunctions	Link ↗
	GroundednessChecks	Link ↗
Plugins	ImportPluginFromGrpc	Link ↗
	OpenAIPugins	Link ↗
Prompt templates	ChatCompletionPrompts	Link ↗
	ChatWithPrompts	Link ↗
Prompt templates	LiquidPrompts	Link ↗
	MultiplePromptTemplates	Link ↗
Prompt functions	PromptFunctionsWithChatGPT	Link ↗
	TextMemoryPlugin_TextMemory	Link ↗

Category	Sample Name	Link
	TemplateLanguage	Link ↗
	PromptFunction	Link ↗
Retrieval augmented generation	WithFunctionCallingStepwisePlanner	Link ↗
	WithPlugins	Link ↗
Search	BingAndGooglePlugins	Link ↗
	MyAzureAIsearchPlugin	Link ↗
	WebSearchQueriesPlugin	Link ↗
Text generation	Custom_TextGenerationService	Link ↗
	HuggingFace_TextGeneration	Link ↗
	OpenAI_TextGenerationStreaming	Link ↗
Text to audio	OpenAI_TextToAudio	Link ↗
Text to image	OpenAI_TextToImage	Link ↗

Supported Semantic Kernel languages

Article • 06/24/2024

Semantic Kernel plans on providing support to the following languages:

- ✓ C#
- ✓ Python
- ✓ Java

While the overall architecture of the kernel is consistent across all languages, we made sure the SDK for each language follows common paradigms and styles in each language to make it feel native and easy to use.

Available SDK packages

C# packages

In C#, there are several packages to help ensure that you only need to import the functionality that you need for your project. The following table shows the available packages in C#.

[+] Expand table

Package name	Description
<code>Microsoft.SemanticKernel</code>	The main package that includes everything to get started
<code>Microsoft.SemanticKernel.Core</code>	The core package that provides implementations for <code>Microsoft.SemanticKernel.Abstractions</code>
<code>Microsoft.SemanticKernel.Abstractions</code>	The base abstractions for Semantic Kernel
<code>Microsoft.SemanticKernel.Connectors.OpenAI</code>	The connector for OpenAI
<code>Microsoft.SemanticKernel.Connectors.HuggingFace</code>	The connector for Hugging Face models
<code>Microsoft.SemanticKernel.Connectors.Google</code>	The connector for Google models (e.g., Gemini)
<code>Microsoft.SemanticKernel.Connectors.MistralAI</code>	The connector for Mistral AI models

Package name	Description
<code>Microsoft.SemanticKernel.Plugins.OpenApi</code> (Experimental)	Enables loading plugins from OpenAPI specifications
<code>Microsoft.SemanticKernel.PromptTemplates.Handlebars</code>	Enables the use of Handlebars templates for prompts
<code>Microsoft.SemanticKernel.Yaml</code>	Provides support for serializing prompts using YAML files
<code>Microsoft.SemanticKernel.Prompty</code>	Provides support for serializing prompts using Prompty files
<code>Microsoft.SemanticKernel.Agents.Abstractions</code>	Provides abstractions for creating agents
<code>Microsoft.SemanticKernel.Agents.OpenAI</code>	Provides support for Assistant API agents

There are other packages available (e.g., the memory connectors), but they are still experimental and are not yet recommended for production use.

To install any of these packages, you can use the following command:

Bash

```
dotnet add package <package-name>
```

Python packages

In Python, there's a single package that includes everything you need to get started with Semantic Kernel. To install the package, you can use the following command:

Bash

```
pip install semantic-kernel
```

Java packages

For Java, Semantic Kernel has the following packages; all are under the group Id `com.microsoft.semantic-kernel`, and can be imported from maven.

XML

```
<dependency>
    <groupId>com.microsoft.semantic-kernel</groupId>
    <artifactId>semantickernel-api</artifactId>
</dependency>
```

A BOM is provided that can be used to define the versions of all Semantic Kernel packages.

XML

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.microsoft.semantic-kernel</groupId>
            <artifactId>semantickernel-bom</artifactId>
            <version>${semantickernel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- `semantickernel-bom` – A Maven project BOM that can be used to define the versions of all Semantic Kernel packages.
- `semantickernel-api` – Package that defines the core public API for the Semantic Kernel for a Maven project.
- `semantickernel-aiservices-openai` – Provides a connector that can be used to interact with the OpenAI API.

Below is an example POM XML for a simple project that uses OpenAI.

XML

```
<project>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.microsoft.semantic-kernel</groupId>
                <artifactId>semantickernel-bom</artifactId>
                <version>${semantickernel.version}</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.microsoft.semantic-kernel</groupId>
```

```

<artifactId>semantickernel-api</artifactId>
</dependency>
<dependency>
    <groupId>com.microsoft.semantic-kernel</groupId>
        <artifactId>semantickernel-connectors-ai-openai</artifactId>
    </dependency>
</dependencies>
</project>

```

Available features in each SDK

The following tables show which features are available in each language. The  symbol indicates that the feature is partially implemented, please see the associated note column for more details. The  symbol indicates that the feature is not yet available in that language; if you would like to see a feature implemented in a language, please consider [contributing to the project](#) or [opening an issue](#).

Core capabilities

[\[+\] Expand table](#)

Services	C#	Python	Java	Notes
Prompts				To see the full list of supported template and serialization formats, refer to the tables below
Native functions and plugins				
OpenAPI plugins				Java has a sample demonstrating how to load OpenAPI plugins
Automatic function calling				
Open Telemetry logs				
Hooks and filters				

Prompt template formats

When authoring prompts, Semantic Kernel provides a variety of template languages that allow you to embed variables and invoke functions. The following table shows which template languages are supported in each language.

[\[\] Expand table](#)

Formats	C#	Python	Java	Notes
Semantic Kernel template language	✓	✓	✓	
Handlebars	✓	✓	✓	
Liquid	✓	✗	✗	
Jinja2	✗	✓	✗	

Prompt serialization formats

Once you've created a prompt, you can serialize it so that it can be stored or shared across teams. The following table shows which serialization formats are supported in each language.

[\[\] Expand table](#)

Formats	C#	Python	Java	Notes
YAML	✓	✓	✓	
Prompty	✗	✓	✗	

AI Services

[\[\] Expand table](#)

Services	C#	Python	Java	Notes
Text Generation	✓	✓	✓	Example: Text-Davinci-003
Chat Completion	✓	✓	✓	Example: GPT4, Chat-GPT
Text Embeddings (Experimental)	✓	✓	✓	Example: Text-Embeddings-Ada-002
Text to Image (Experimental)	✓	✗	✗	Example: Dall-E
Image to Text (Experimental)	✓	✗	✗	Example: Pix2Struct
Text to Audio (Experimental)	✓	✗	✗	Example: Text-to-speech
Audio to Text (Experimental)	✓	✗	✗	Example: Whisper

AI service endpoints

 Expand table

Endpoints	C#	Python	Java	Notes
OpenAI	✓	✓	✓	
Azure OpenAI	✓	✓	✓	
Other endpoints that support OpenAI APIs	✓	✓	✓	Includes Ollama, LLM Studio, Azure Model-as-a-service, etc.
Hugging Face Inference API	⟳	✗	✗	Coming soon to Python, not all scenarios are covered for .NET

Memory Connectors (Experimental)

Important

All of the existing memory connectors are currently experimental and are undergoing active development to improve the experience of using them. To provide feedback on the latest proposal, please refer to the active [Search](#) and [Memory Connector](#) ADRs.

 Expand table

Memory Connectors	C#	Python	Java	Notes
Azure AI Search	✓	✓	✓	
Chroma	✓	✓	✗	
DuckDB	✓	✗	✗	
Milvus	⟳	✓	✗	
Pinecone	✓	✓	✗	
Postgres	✓	✓	✗	
Qdrant	✓	⟳	✗	
Redis	✓	⟳	✗	
Sqlite	✓	✗	⟳	

Memory Connectors	C#	Python	Java	Notes
Weaviate	✓	✓	✗	

Contributing to Semantic Kernel

Article • 06/21/2023

You can contribute to Semantic Kernel by submitting issues, starting discussions, and submitting pull requests (PRs). Contributing code is greatly appreciated, but simply filing issues for problems you encounter is also a great way to contribute since it helps us focus our efforts.

Reporting issues and feedback

We always welcome bug reports, API proposals, and overall feedback. Since we use GitHub, you can use the [Issues ↗](#) and [Discussions ↗](#) tabs to start a conversation with the team. Below are a few tips when submitting issues and feedback so we can respond to your feedback as quickly as possible.

Reporting issues

New issues for the SDK can be reported in our [list of issues ↗](#), but before you file a new issue, please search the list of issues to make sure it does not already exist. If you have issues with the Semantic Kernel documentation (this site), please file an issue in the [Semantic Kernel documentation repository ↗](#).

If you *do* find an existing issue for what you wanted to report, please include your own feedback in the discussion. We also highly recommend up-voting ( reaction) the original post, as this helps us prioritize popular issues in our backlog.

Writing a Good Bug Report

Good bug reports make it easier for maintainers to verify and root cause the underlying problem. The better a bug report, the faster the problem can be resolved. Ideally, a bug report should contain the following information:

- A high-level description of the problem.
- A *minimal reproduction*, i.e. the smallest size of code/configuration required to reproduce the wrong behavior.
- A description of the *expected behavior*, contrasted with the *actual behavior* observed.
- Information on the environment: OS/distribution, CPU architecture, SDK version, etc.

- Additional information, e.g. Is it a regression from previous versions? Are there any known workarounds?

[Create issue](#)

Submitting feedback

If you have general feedback on Semantic Kernel or ideas on how to make it better, please share it on our [discussions board](#). Before starting a new discussion, please search the list of discussions to make sure it does not already exist.

We recommend using the [ideas category](#) if you have a specific idea you would like to share and the [Q&A category](#) if you have a question about Semantic Kernel.

You can also start discussions (and share any feedback you've created) in the Discord community by joining the [Semantic Kernel Discord server](#).

[Start a discussion](#)

Help us prioritize feedback

We currently use up-votes to help us prioritize issues and features in our backlog, so please up-vote any issues or discussions that you would like to see addressed.

If you think others would benefit from a feature, we also encourage you to ask others to up-vote the issue. This helps us prioritize issues that are impacting the most users. You can ask colleagues, friends, or the [community on Discord](#) to up-vote an issue by sharing the link to the issue or discussion.

Submitting pull requests

We welcome contributions to Semantic Kernel. If you have a bug fix or new feature that you would like to contribute, please follow the steps below to submit a pull request (PR). Afterwards, project maintainers will review code changes and merge them once they've been accepted.

Recommended contribution workflow

We recommend using the following workflow to contribute to Semantic Kernel (this is the same workflow used by the Semantic Kernel team):

1. Create an issue for your work.

- You can skip this step for trivial changes.
- Reuse an existing issue on the topic, if there is one.
- Get agreement from the team and the community that your proposed change is a good one by using the discussion in the issue.
- Clearly state in the issue that you will take on implementation. This allows us to assign the issue to you and ensures that someone else does not accidentally work on it.

2. Create a personal fork of the repository on GitHub (if you don't already have one).

3. In your fork, create a branch off of main (`git checkout -b mybranch`).

- Name the branch so that it clearly communicates your intentions, such as "issue-123" or "githubhandle-issue".

4. Make and commit your changes to your branch.

5. Add new tests corresponding to your change, if applicable.

6. Build the repository with your changes.

- Make sure that the builds are clean.
- Make sure that the tests are all passing, including your new tests.

7. Create a PR against the repository's **main** branch.

- State in the description what issue or improvement your change is addressing.
- Verify that all the Continuous Integration checks are passing.

8. Wait for feedback or approval of your changes from the code maintainers.

9. When area owners have signed off, and all checks are green, your PR will be merged.

Dos and Don'ts while contributing

The following is a list of Dos and Don'ts that we recommend when contributing to Semantic Kernel to help us review and merge your changes as quickly as possible.

Do's:

- **Do** follow the standard [.NET coding style](#) and [Python code style](#) ↗
- **Do** give priority to the current style of the project or file you're changing if it diverges from the general guidelines.

- **Do** include tests when adding new features. When fixing bugs, start with adding a test that highlights how the current behavior is broken.
- **Do** keep the discussions focused. When a new or related topic comes up it's often better to create new issue than to side track the discussion.
- **Do** clearly state on an issue that you are going to take on implementing it.
- **Do** blog and/or tweet about your contributions!

Don'ts:

- **Don't** surprise the team with big pull requests. We want to support contributors, so we recommend filing an issue and starting a discussion so we can agree on a direction before you invest a large amount of time.
- **Don't** commit code that you didn't write. If you find code that you think is a good fit to add to Semantic Kernel, file an issue and start a discussion before proceeding.
- **Don't** submit PRs that alter licensing related files or headers. If you believe there's a problem with them, file an issue and we'll be happy to discuss it.
- **Don't** make new APIs without filing an issue and discussing with the team first. Adding new public surface area to a library is a big deal and we want to make sure we get it right.

Breaking Changes

Contributions must maintain API signature and behavioral compatibility. If you want to make a change that will break existing code, please file an issue to discuss your idea or change if you believe that a breaking change is warranted. Otherwise, contributions that include breaking changes will be rejected.

The continuous integration (CI) process

The continuous integration (CI) system will automatically perform the required builds and run tests (including the ones you should also run locally) for PRs. Builds and test runs must be clean before a PR can be merged.

If the CI build fails for any reason, the PR issue will be updated with a link that can be used to determine the cause of the failure so that it can be addressed.

Contributing to documentation

We also accept contributions to the [Semantic Kernel documentation repository](#). To learn how to make contributions, please start with the Microsoft [docs contributor guide](#).

Running your own Hackathon

Article • 06/24/2024

With these materials you can run your own Semantic Kernel Hackathon, a hands-on event where you can learn and create AI solutions using Semantic Kernel tools and resources.

By participating and running a Semantic Kernel hackathon, you will have the opportunity to:

- Explore the features and capabilities of Semantic Kernel and how it can help you solve problems with AI
- Work in teams to brainstorm and develop your own AI plugins or apps using Semantic Kernel SDK and services
- Present your results and get feedback from other participants
- Have fun!

Download the materials

To run your own hackathon, you will first need to download the materials. You can download the zip file here:

[Download hackathon materials](#)

Once you have unzipped the file, you will find the following resources:

- Hackathon sample agenda
- Hackathon prerequisites
- Hackathon facilitator presentation
- Hackathon team template
- Helpful links

Preparing for the hackathon

Before the hackathon, you and your peers will need to download and install software needed for Semantic Kernel to run. Additionally, you should already have API keys for either OpenAI or Azure OpenAI and access to the Semantic Kernel repo. Please refer to the prerequisites document in the facilitator materials for the complete list of tasks participants should complete before the hackathon.

You should also familiarize yourself with the available documentation and tutorials. This will ensure that you are knowledgeable of core Semantic Kernel concepts and features so that you can help others during the hackathon. The following resources are highly recommended:

- [What is Semantic Kernel?](#)
- [Semantic Kernel LinkedIn training video ↗](#)

Running the hackathon

The hackathon will consist of six main phases: welcome, overview, brainstorming, development, presentation, and feedback.

Here is an approximate agenda and structure for each phase but feel free to modify this based on your team:

 [Expand table](#)

Length (Minutes)	Phase	Description
Day 1		
15	Welcome/Introductions	The hackathon facilitator will welcome the participants, introduce the goals and rules of the hackathon, and answer any questions.
30	Overview of Semantic Kernel	The facilitator will guide you through a live presentation that will give you an overview of AI and why it is important for solving problems in today's world. You will also see demos of how Semantic Kernel can be used for different scenarios.
5	Choose your Track	Review slides in the deck for the specific track you'll pick for the hackathon.
120	Brainstorming	The facilitator will help you form teams based on your interests or skill levels. You will then brainstorm ideas for your own AI plugins or apps using design thinking techniques.
20	Responsible AI	Spend some time reviewing Responsible AI principles and ensure your proposal follows these principles.
60	Break/Lunch	Lunch or Break
360+	Development/Hack	You will use Semantic Kernel SDKs tools, and resources to develop, test, and deploy your projects. This could be

Length (Minutes)	Phase	Description
		for the rest of the day or over multiple days based on the time available and problem to be solved.
Day 2		
5	Welcome Back	Reconnect for Day 2 of the Semantic Kernel Hackathon
20	What did you learn?	Review what you've learned so far in Day 1 of the Hackathon.
120	Hack	You will use Semantic Kernel SDKs tools, and resources to develop, test, and deploy your projects. This could be for the rest of the day or over multiple days based on the time available and problem to be solved.
120	Demo	Each team will present their results using a PowerPoint template provided. You will have about 15 minutes per team to showcase your project, demonstrate how it works, and explain how it solves a problem with AI. You will also receive feedback from other participants.
5	Thank you	The hackathon facilitator will close the hackathon.
30	Feedback	Each team can share their feedback on the hackathon and Semantic Kernel with the group and fill out the Hackathon Exit Survey ↗ .

Following up after the hackathon

We hope you enjoyed running a Semantic Kernel Hackathon and the overall experience! We would love to hear from you about what worked well, what didn't, and what we can improve for future content. Please take a few minutes to fill out the [hackathon facilitator survey ↗](#) and share your feedback and suggestions with us.

If you want to continue developing your AI plugins or projects after the hackathon, you can find more resources and support for Semantic Kernel.

- [Semantic Kernel blog ↗](#)
- [Semantic Kernel GitHub repo ↗](#)

Thank you for your engagement and creativity during the hackathon. We look forward to seeing what you create next with Semantic Kernel!

What are agents?

Article • 06/24/2024

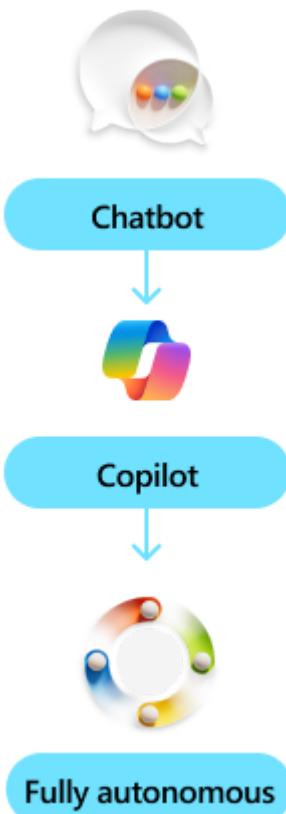
Agents are software based entities that leverage AI models to do works for you. They are built to perform a wide range of tasks and called different names based on the jobs they do.

For example, an agent built to answer questions is referred to as a chatbot due to its chat-based experience. These agents are typically grounded in your own data, like company documents.

An agent built to work along side you is typically called a copilot. These agents help you complete tasks like writing emails or creating other office artifacts by providing recommendations that you choose to either accept or reject.

An agent built to handle repetitive tasks are usually labeled as fully autonomous. They respond to events and perform actions without needing intervention. This is different from a copilot in that it's doing work for you instead of alongside you.

Semantic Kernel provides you with the infrastructure to build any kind of agent you need without being an expert in AI.



Building your first agent

An agent is made up of three core building blocks that harmoniously work together to complete tasks for you. Let's take a closer look by building a copilot that helps you write and send an email.



Plugin



Planner



Persona

A copilot to write and send emails

To create this copilot you'll need to create a plugin that can send emails, a planner that generates a plan to write an email, and a persona that interacts with you to get the information needed to send the email.

 [Expand table](#)

Step	Description
1	Get the user's email address and name
2	Get the email address of the recipient
3	Get the topic of the email
4	Generate the subject and body of the email
5	Review the email with the user
6	Send the email

To enable this scenario, we would need to create a plugin that can send emails, a planner that can generate a plan to write an email, and a persona that can interact with the user to get the necessary information.

The following sections will walk you through the conceptual building blocks and how to put them together to build your first agent. Afterwards, you can refer to the specific guides for each building block to learn more about how to build them.

Plugins: giving your agent skills

To generate the above plan, the copilot first needs the capabilities necessary to perform these steps. This is where plugins come in. Plugins allow you to give your agent skills via code. For example, you could create a plugin that sends emails, retrieves information from a database, asks for help, or even saves and retrieves memories from previous conversations.

In our example, we can build a simple plugin that sends emails using [native code](#). Our plugin just has a single function, `send_email`, that takes in the email address, subject, and body of the email. It would then use this information to send the email.

```
using System.ComponentModel;
using Microsoft.SemanticKernel;

public class EmailPlugin
{
    [KernelFunction("send_email")]
    [Description("Sends an email to a recipient.")]
    public async Task SendEmailAsync(
        Kernel kernel,
        List<string> recipientEmails,
        string subject,
        string body
    )
    {
        // Add logic to send an email using the recipientEmails, subject,
        and body
        // For now, we'll just print out a success message to the console
        Console.WriteLine("Email sent!");
    }
}
```

There are other ways to create plugins. For example, if you have a RestAPI that can send emails, you can automatically create a plugin using its [OpenAPI specification](#). To learn more about other ways to author plugins, see the [plugins section](#).

Planning: completing multi-step tasks

To actually use this plugin (and to wire them up with other steps), the copilot would then need to generate a plan. This is where planning comes in. Planning comes from the built-in ability of LLMs to determine how to iteratively complete a task.

Note

In the past, special prompts were created by AI app developers to guide the AI in generating a plan that could be consumed by an SDK like Semantic Kernel. However, with the advent of LLMs, AIs can now generate plans directly from a conversation with a user with the aid of function calling.

With built-in planning support form LLMs, using Semantic Kernel is now as easy as invoking a chat completion service with auto function calling enabled.

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

// Create kernel with an email plugin
var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<EmailPlugin>();
Kernel kernel = builder.Build();

// Create chat completion service
AzureOpenAIChatCompletionService chatCompletionService = new (
    deploymentName: "gpt-4",
    apiKey: "YOUR_API_KEY",
    endpoint: "YOUR_AZURE_ENDPOINT"
);

// Enable planning
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};

// Create chat history
var history = new ChatHistory();
history.AddUserMessage("Can you help me write an email for my boss?");

// Get the response from the AI
var result = await chatCompletionService.GetChatMessageContentAsync(
    history,
    executionSettings: openAIPromptExecutionSettings,
    kernel: kernel
);
```

To learn more about planning with Semantic Kernel, see the [planning article](#).

Personas: giving your agent a job description

In most cases, using the built-in planning capabilities of LLMs with plugins is sufficient to building an agent, but as you build more domain-specific agents, you may want to add a persona to your agent.

A persona is the instructions that you provide your agent so they can more effectively perform the role you want them to play. At its simplest, the persona could instruct the AI to be polite, ask for clarification when needed, or role-play as a specific job title (e.g., a customer service representative).

For enterprise scenarios, however, you may want to provide more detailed instructions to your agent. For example, you may want to instruct your agent to follow specific rules, ask for approval before taking actions, or escalate to a human if the AI is unable to complete a task.

To provide a persona to your agent, simply pre-pend a system message to the chat history that describes the persona. The AI will then use this persona to guide its interactions with the user.

```
// Create chat history
ChatHistory chatMessages = new ChatHistory("""
    You are a friendly assistant who likes to follow the rules. You will
    complete required steps
        and request approval before taking any consequential actions. If the
        user doesn't provide
            enough information for you to complete a task, you will keep asking
            questions until you have
                enough information to complete the task.
        """
);
chatMessages.AddUserMessage("Can you help me write an email for my boss?");

// Get the response from the AI
var result = await chatCompletionService.GetChatMessageContentAsync(
    history,
    executionSettings: openAIPromptExecutionSettings,
    kernel: kernel
);
```

To learn more about authoring effective personas, refer to the [personas article](#).

Putting the pieces together

Now that we understand the core building blocks of an agent, we can now combine them together to build our first agent. To do so, we'll initialize our `Kernel` object with our plugins, planners, and persona. Afterwards, we'll use the `Kernel` object to generate a plan and then execute that plan.

```
// Create the kernel
var builder = Kernel.CreateBuilder();
builder.Services.AddLogging(c =>
```

```

c.SetMinimumLevel(LogLevel.Trace).AddDebug());
builder.Services.AddChatCompletionService(kernelSettings);
builder.Plugins.AddFromType<EmailPlugin>();
Kernel kernel = builder.Build();

// Retrieve the chat completion service from the kernel
IChatCompletionService chatCompletionService =
kernel.GetRequiredService<IChatCompletionService>();

// Create the chat history
ChatHistory chatMessages = new ChatHistory("""
    You are a friendly assistant who likes to follow the rules. You will
    complete required steps
    and request approval before taking any consequential actions. If the
    user doesn't provide
    enough information for you to complete a task, you will keep asking
    questions until you have
    enough information to complete the task.
""");

// Start the conversation
while (true)
{
    // Get user input
    System.Console.Write("User > ");
    chatMessages.AddUserMessage(Console.ReadLine()!);

    // Get the chat completions
    OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
    {
        FunctionCallBehavior =
FunctionCallBehavior.AutoInvokeKernelFunctions
    };
    var result = chatCompletionService.GetStreamingChatMessageContentsAsync(
        chatMessages,
        executionSettings: openAIPromptExecutionSettings,
        kernel: kernel);

    // Stream the results
    string fullMessage = "";
    await foreach (var content in result)
    {
        if (content.Role.HasValue)
        {
            System.Console.Write("Assistant > ");
        }
        System.Console.Write(content.Content);
        fullMessage += content.Content;
    }
    System.Console.WriteLine();

    // Add the message from the agent to the chat history
    chatMessages.AddAssistantMessage(fullMessage);
}

```

Trying out your agent

Now that we've completed our program, we can give it a test run. Below is a sample conversation with our agent while using gpt-4 model.

[+] Expand table

Role	Message
● User	Can you help me write an email for my boss?
● Assistant	Of course, I'd be happy to help you write an email to your boss. Could you please provide me with more details? Specifically, I need to know: <ol style="list-style-type: none">1. The topic or purpose of the email.2. Your boss's email address.
● User	I want to give her an update on last months sales. We broke a bunch of records that I want to share with her, but we did have a challenge selling the X4321 model.
● Assistant	That sounds like a great topic for an email. Could you please provide me with your boss's email address so I can assist you further?
● User	Sure! It's sarah@contoso.com
● Assistant	Great! Here's a plan for your email: <ol style="list-style-type: none">1. Start with a Clear Subject Line and Greeting: The subject line should be concise and informative, such as "Update on Last Month's Sales & X4321 Model Challenges". Start the email with a professional greeting, like "Dear Sarah,".2. Highlight Achievements and Challenges in Separate Paragraphs: Begin with the positive news about the record-breaking achievements. Use specific data and examples to illustrate the success. Then, transition to the challenges with the X4321 model. Be honest and clear about the issues, but also suggest potential solutions or steps being taken to address these challenges.3. End with a Call to Action and Professional Closing: If you need Sarah to take any actions, state this clearly towards the end of the email. This could be asking for her input, a meeting to discuss further, or any other relevant action. Close the email professionally with a phrase like "Best regards" or "Sincerely", followed by your name. <p>Remember to proofread the email for any grammatical errors or unclear statements before sending it. This will help ensure the message is professional and easy to understand.</p> <p>Would you like me to draft the email for you based on these steps?</p>
● User	Yes please!
● Assistant	Sure, here's a draft based on the steps we discussed: <p>Subject: Update on Last Month's Sales & X4321 Model Challenges</p> <p>Dear Sarah,</p> <p>I hope this email finds you well. I am writing to update you on our sales performance for the past month.</p>

Role	Message
	<p>I am pleased to report that we broke several sales records last month. Our team's hard work and dedication have paid off, resulting in significant growth in our overall sales figures. [Here, you can add specific data or examples of the records broken]</p> <p>However, we did face some challenges with the X4321 model. Despite our best efforts, the sales for this model did not meet our expectations. [Here, you can explain more about the challenges faced]</p> <p>We are currently working on strategies to improve the sales of the X4321 model. [Here, you can mention any specific steps or plans you have to address the issue]</p> <p>I would appreciate your input on this matter. If you would like, we can schedule a meeting to discuss this further.</p> <p>Thank you for your continued support and guidance.</p> <p>Best regards,</p> <p>[Your Name]</p> <p>Please review this draft and let me know if you would like to make any changes before I send it.</p>
 User	<p>Can you add that we increased our revenue by 20%? You can also share that the X4321 didn't sell as well since most customers upgraded to the Y5000 model, but this isn't too bad since it's what helped increase our revenue.</p>
 Assistant	<p>Absolutely, here's the updated draft:</p> <p>Subject: Update on Last Month's Sales & X4321 Model Challenges</p> <p>Dear Sarah,</p> <p>I hope this email finds you well. I am writing to update you on our sales performance for the past month.</p> <p>I am pleased to report that we broke several sales records last month. Our team's hard work and dedication have paid off, resulting in a significant 20% increase in our overall revenue. This growth is a testament to our team's commitment and the quality of our products.</p> <p>However, we did face some challenges with the X4321 model. Despite our best efforts, the sales for this model did not meet our expectations. Interestingly, most customers upgraded to the Y5000 model, which contributed significantly to our increased revenue. While we aim to improve the sales of all our models, this shift in customer preference is not entirely negative.</p> <p>We are currently analyzing this trend and working on strategies to leverage it for future sales. Your insights and suggestions would be greatly appreciated.</p> <p>Thank you for your continued support and guidance.</p> <p>Best regards,</p> <p>Stephen</p> <p>Please review this updated draft and let me know if you would like to make any changes before I send it.</p>
 User	<p>Please sign it with Stephen and then you can go ahead and send it to Sarah</p>
 Tool	<p>Email sent!</p>
 Assistant	<p>The email has been sent to Sarah. If there's anything else you need assistance with,</p>

Role	Message
	feel free to ask!

Next steps

Now that you've built your first agent, you can now deep dive into the different components of Semantic Kernel.

 [Expand table](#)

Section	Description
Kernel	Setting up and using the core Kernel object
AI Services	Integration and usage of various AI services
Plugins	Integrate native code and APIs with Semantic Kernel
Planners	Enabling your AI agents to complete multi-step tasks
Personas	Customize your AI agents to better collaborate with others

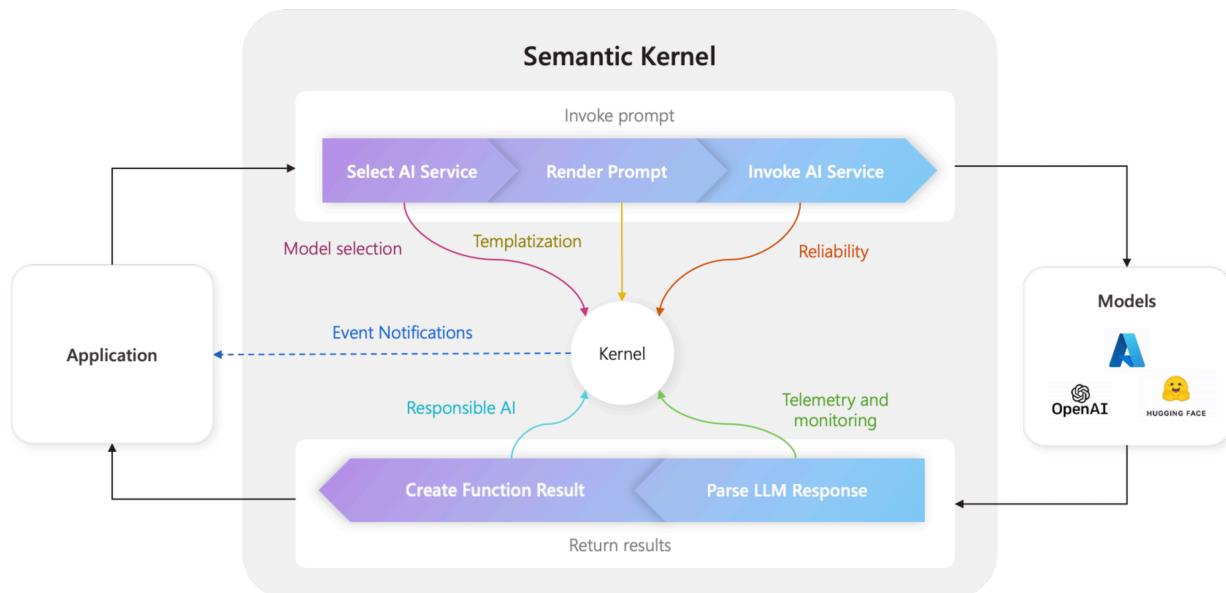
Understanding the kernel

Article • 06/24/2024

The kernel is the central component of Semantic Kernel. At its simplest, the kernel is a Dependency Injection container that manages all of the services and plugins necessary to run your AI application. If you provide all of your services and plugins to the kernel, they will then be seamlessly used by the AI as needed.

The kernel is at the center of your agents

Because the kernel has all of the services and plugins necessary to run both native code and AI services, it is used by nearly every component within the Semantic Kernel SDK to power your agents. This means that if you run any prompt or code in Semantic Kernel, the kernel will always be available to retrieve the necessary services and plugins.



This is extremely powerful, because it means you as a developer have a single place where you can configure, and most importantly monitor, your AI agents. Take for example, when you invoke a prompt from the kernel. When you do so, the kernel will...

1. Select the best AI service to run the prompt.
2. Build the prompt using the provided prompt template.
3. Send the prompt to the AI service.
4. Receive and parse the response.
5. And finally return the response from the LLM to your application.

Throughout this entire process, you can create events and middleware that are triggered at each of these steps. This means you can perform actions like logging, provide status updates to users, and most importantly responsible AI. All from a single place.

Build a kernel with services and plugins

Before building a kernel, you should first understand the two types of components that exist:

[+] Expand table

Components	Description
1 Services	These consist of both AI services (e.g., chat completion) and other services (e.g., logging and HTTP clients) that are necessary to run your application. This was modelled after the Service Provider pattern in .NET so that we could support dependency ingestion across all languages.
2 Plugins	These are the components that are used by your AI services and prompt templates to perform work. AI services, for example, can use plugins to retrieve data from a database or call an external API to perform actions.

To start creating a kernel, import the necessary packages at the top of your file:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;
```

Next, you can add services and plugins. Below is an example of how you can add an Azure OpenAI chat completion, a logger, and a time plugin.

C#

```
// Create a kernel with a logger and Azure OpenAI chat completion service
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(modelId, endpoint, apiKey);
builder.Services.AddLogging(c =>
    c.AddDebug().SetMinimumLevel(LogLevel.Trace));
builder.Plugins.AddFromType<TimePlugin>();
Kernel kernel = builder.Build();
```

Using Dependency Injection

In C#, you can use Dependency Injection to create a kernel. This is done by creating a `ServiceCollection` and adding services and plugins to it. Below is an example of how you can create a kernel using Dependency Injection.

💡 Tip

We recommend that you create a kernel as a transient service so that it is disposed of after each use because the plugin collection is mutable. The kernel is extremely lightweight (since it's just a container for services and plugins), so creating a new kernel for each use is not a performance concern.

C#

```
using Microsoft.SemanticKernel;

var builder = Host.CreateApplicationBuilder(args);

// Add the OpenAI chat completion service as a singleton
builder.Services.AddOpenAIChatCompletion(
    modelId: "gpt-4",
    apiKey: "YOUR_API_KEY",
    orgId: "YOUR_ORG_ID", // Optional; for OpenAI deployment
    serviceId: "YOUR_SERVICE_ID" // Optional; for targeting specific
services within Semantic Kernel
);

// Create singletons of your plugins
builder.Services.AddSingleton(() => new LightsPlugin());
builder.Services.AddSingleton(() => new SpeakerPlugin());

// Create the plugin collection (using the KernelPluginFactory to create
plugins from objects)
builder.Services.AddSingleton<KernelPluginCollection>((serviceProvider) =>
[
    KernelPluginFactory.CreateFromObject(serviceProvider.GetRequiredService<Light
tsPlugin>()),
    KernelPluginFactory.CreateFromObject(serviceProvider.GetRequiredService<Spea
kerPlugin>())
]);

// Finally, create the Kernel service with the service provider and plugin
collection
builder.Services.AddTransient((serviceProvider)=> {
    KernelPluginCollection pluginCollection =
serviceProvider.GetRequiredService<KernelPluginCollection>();

    return new Kernel(serviceProvider, pluginCollection);
});
```

💡 Tip

For more samples on how to use dependency injection in C#, refer to the [concept samples](#).

Next steps

Now that you understand the kernel, you can learn about all the different AI services that you can add to it.

[Learn about AI services](#)

Adding AI services to Semantic Kernel

Article • 06/24/2024

One of the main features of Semantic Kernel is its ability to add different AI services to the kernel. This allows you to easily swap out different AI services to compare their performance and to leverage the best model for your needs. In this section, we will provide sample code for adding different AI services to the kernel.

Within Semantic Kernel, there are interfaces for the most popular AI tasks. In the table below, you can see the services that are supported by each of the SDKs.

[Expand table](#)

Services	C#	Python	Java	Notes
Chat completion	✓	✓	✓	
Text generation	✓	✓	✓	
Embedding generation (Experimental)	✓	✓	✓	
Text-to-image (Experimental)	✓	✗	✗	
Image-to-text (Experimental)	✓	✗	✗	
Text-to-audio (Experimental)	✓	✗	✗	
Audio-to-text (Experimental)	✓	✗	✗	

Tip

In most scenarios, you will only need to add chat completion to your kernel, but to support multi-modal AI, you can add any of the above services to your kernel.

Next steps

To learn more about each of the services, please refer to the specific articles for each service type. In each of the articles we provide sample code for adding the service to the kernel across multiple AI service providers.

[Learn about chat completion](#)

Chat completion

Article • 06/24/2024

With chat completion, you can simulate a back-and-forth conversation with an AI agent. This is of course useful for creating chat bots, but it can also be used for creating autonomous agents that can complete business processes, generate code, and more. As the primary model type provided by OpenAI, Google, Mistral, Facebook, and others, chat completion is the most common AI service that you will add to your Semantic Kernel project.

When picking out a chat completion model, you will need to consider the following:

- What modalities does the model support (e.g., text, image, audio, etc.)?
- Does it support function calling?
- How fast does it receive and generate tokens?
- How much does each token cost?

ⓘ Important

Of all the above questions, the most important is whether the model supports function calling. If it does not, you will not be able to use the model to call your existing code. Most of the latest models from OpenAI, Google, Mistral, and Amazon all support function calling. Support from small language models, however, is still limited.

Installing the necessary packages

Before adding chat completion to your kernel, you will need to install the necessary packages. Below are the packages you will need to install for each AI service provider.

Azure OpenAI

Bash

```
dotnet add package Microsoft.SemanticKernel.Connectors.OpenAI
```

Creating chat completion services

Now that you've installed the necessary packages, you can create chat completion services. Below are the several ways you can create chat completion services using Semantic Kernel.

Adding directly to the kernel

To add a chat completion service, you can use the following code to add it to the kernel's inner service provider.

```
Azure OpenAI

Bash
dotnet add package Microsoft.SemanticKernel.Connectors.OpenAI

C#
using Microsoft.SemanticKernel;

IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddAzureOpenAIChatCompletion(
    deploymentName: "NAME_OF_YOUR_DEPLOYMENT",
    apiKey: "YOUR_API_KEY",
    endpoint: "YOUR_AZURE_ENDPOINT",
    modelId: "gpt-4", // Optional name of the underlying model if the
    deployment name doesn't match the model name
    serviceId: "YOUR_SERVICE_ID", // Optional; for targeting specific
    services within Semantic Kernel
    httpClient: new HttpClient() // Optional; if not provided, the
    HttpClient from the kernel will be used
);
Kernel kernel = kernelBuilder.Build();
```

Using dependency injection

If you're using dependency injection, you'll likely want to add your AI services directly to the service provider. This is helpful if you want to create singletons of your AI services and reuse them in transient kernels.

```
Azure OpenAI

C#
```

```
using Microsoft.SemanticKernel;

var builder = Host.CreateApplicationBuilder(args);

builder.Services.AddAzureOpenAIChatCompletion(
    deploymentName: "NAME_OF_YOUR_DEPLOYMENT",
    apiKey: "YOUR_API_KEY",
    endpoint: "YOUR_AZURE_ENDPOINT",
    modelId: "gpt-4", // Optional name of the underlying model if the
    deployment name doesn't match the model name
    serviceId: "YOUR_SERVICE_ID" // Optional; for targeting specific
    services within Semantic Kernel
);

builder.Services.AddTransient((serviceProvider)=> {
    return new Kernel(serviceProvider);
});
```

Creating standalone instances

Lastly, you can create instances of the service directly so that you can either add them to a kernel later or use them directly in your code without ever injecting them into the kernel or in a service provider.

Azure OpenAI

C#

```
using Microsoft.SemanticKernel.Connectors.OpenAI;

AzureOpenAIChatCompletionService chatCompletionService = new (
    deploymentName: "NAME_OF_YOUR_DEPLOYMENT",
    apiKey: "YOUR_API_KEY",
    endpoint: "YOUR_AZURE_ENDPOINT",
    modelId: "gpt-4", // Optional name of the underlying model if the
    deployment name doesn't match the model name
    httpClient: new HttpClient() // Optional; if not provided, the
    HttpClient from the kernel will be used
);
```

Retrieving chat completion services

Once you've added chat completion services to your kernel, you can retrieve them using the get service method. Below is an example of how you can retrieve a chat completion

service from the kernel.

```
C#
```

```
var chatCompletionService =  
kernel.GetRequiredService<IChatCompletionService>();
```

Using chat completion services

Now that you have a chat completion service, you can use it to generate responses from an AI agent. There are two main ways to use a chat completion service:

- Non-streaming: You wait for the service to generate an entire response before returning it to the user.
- Streaming: Individual chunks of the response are generated and returned to the user as they are created.

Below are the two ways you can use a chat completion service to generate responses.

Non-streaming chat completion

To use non-streaming chat completion, you can use the following code to generate a response from the AI agent.

```
C#
```

```
ChatHistory history = [];  
history.AddUserMessage("Hello, how are you?");  
  
var response = await chatCompletionService.GetChatMessageContentAsync(  
    history,  
    kernel: kernel  
);
```

Streaming chat completion

To use streaming chat completion, you can use the following code to generate a response from the AI agent.

```
C#
```

```
ChatHistory history = [];  
history.AddUserMessage("Hello, how are you?");
```

```
var response = chatCompletionService.GetStreamingChatMessageContentsAsync(
    chatHistory: history,
    kernel: kernel
);

await foreach (var chunk in response)
{
    Console.WriteLine(chunk);
}
```

Next steps

Now that you've added chat completion services to your Semantic Kernel project, you can start creating conversations with your AI agent. To learn more about using a chat completion service, check out the following articles:

- [Using the chat history object](#)
- [Optimizing function calling with chat completion](#)

Chat history

Article • 06/24/2024

The chat history object is used to maintain a record of messages in a chat session. It is used to store messages from different authors, such as users, assistants, tools, or the system. As the primary mechanism for sending and receiving messages, the chat history object is essential for maintaining context and continuity in a conversation.

Creating a chat history object

A chat history object is a list under the hood, making it easy to create and add messages to.

```
using Microsoft.SemanticKernel.ChatCompletion;

// Create a chat history object
ChatHistory chatHistory = [];

chatHistory.AddSystemMessage("You are a helpful assistant.");
chatHistory.AddUserMessage("What's available to order?");
chatHistory.AddAssistantMessage("We have pizza, pasta, and salad available to order. What would you like to order?");
chatHistory.AddUserMessage("I'd like to have the first option, please.");
```

Adding richer messages to a chat history

The easiest way to add messages to a chat history object is to use the methods above. However, you can also add messages manually by creating a new `ChatMessage` object. This allows you to provide additional information, like names and images content.

```
using Microsoft.SemanticKernel.ChatCompletion;

// Add system message
chatHistory.Add(
    new() {
        Role = AuthorRole.System,
        Content = "You are a helpful assistant"
    }
);

// Add user message with an image
chatHistory.Add(
    new() {
        Role = AuthorRole.User,
```

```

        AuthorName = "Laimonis Dumins",
        Items = [
            new TextContent { Text = "What available on this menu" },
            new ImageContent { Uri = new Uri("https://example.com/menu.jpg") }
        ]
    }
);

// Add assistant message
chatHistory.Add(
    new() {
        Role = AuthorRole.Assistant,
        AuthorName = "Restaurant Assistant",
        Content = "We have pizza, pasta, and salad available to order. What would you like to order?"
    }
);

// Add additional message from a different user
chatHistory.Add(
    new() {
        Role = AuthorRole.User,
        AuthorName = "Ema Vargova",
        Content = "I'd like to have the first option, please."
    }
);

```

Simulating function calls

In addition to user, assistant, and system roles, you can also add messages from the tool role to simulate function calls. This is useful for teaching the AI how to use plugins and to provide additional context to the conversation.

For example, to inject information about the current user in the chat history without requiring the user to provide the information or having the LLM waste time asking for it, you can use the tool role to provide the information directly.

Below is an example of how we're able to provide user allergies to the assistant by simulating a function call to the `User` plugin.

💡 Tip

Simulated function calls is particularly helpful for providing details about the current user(s). Today's LLMs have been trained to be particularly sensitive to user information. Even if you provide user details in a system message, the LLM may still

choose to ignore it. If you provide it via a user message, or tool message, the LLM is more likely to use it.

```
// Add a simulated function call from the assistant
chatHistory.Add(
    new() {
        Role = AuthorRole.Assistant,
        Items = [
            new FunctionCallContent(
                functionName: "get_user_allergies",
                pluginName: "User",
                id: "0001",
                arguments: new () { {"username", "laimonisdumins"} }
            ),
            new FunctionCallContent(
                functionName: "get_user_allergies",
                pluginName: "User",
                id: "0002",
                arguments: new () { {"username", "emavargova"} }
            )
        ]
    }
);

// Add a simulated function results from the tool role
chatHistory.Add(
    new() {
        Role = AuthorRole.Tool,
        Items = [
            new FunctionResultContent(
                functionName: "get_user_allergies",
                pluginName: "User",
                id: "0001",
                result: "{ \"allergies\": [\"peanuts\", \"gluten\"] }"
            )
        ]
    }
);
chatHistory.Add(
    new() {
        Role = AuthorRole.Tool,
        Items = [
            new FunctionResultContent(
                functionName: "get_user_allergies",
                pluginName: "User",
                id: "0002",
                result: "{ \"allergies\": [\"dairy\", \"soy\"] }"
            )
        ]
    }
);

```

ⓘ Important

When simulating tool results, you must always provide the `id` of the function call that the result corresponds to. This is important for the AI to understand the context of the result. Some LLMs, like OpenAI, will throw an error if the `id` is missing or if the `id` does not correspond to a function call.

Inspecting a chat history object

Whenever you pass a chat history object to a chat completion service with auto function calling enabled, the chat history object will be manipulated so that it includes the function calls and results. This allows you to avoid having to manually add these messages to the chat history object and also allows you to inspect the chat history object to see the function calls and results.

You must still, however, add the final messages to the chat history object. Below is an example of how you can inspect the chat history object to see the function calls and results.

```
using Microsoft.SemanticKernel.ChatCompletion;

ChatHistory chatHistory = [
    new() {
        Role = AuthorRole.User,
        Content = "Please order me a pizza"
    }
];

// Get the current length of the chat history object
int currentChatHistoryLength = chatHistory.Count;

// Get the chat message content
ChatMessageContent results = await
    chatCompletionService.GetChatMessageContentAsync(
        chatHistory,
        kernel: kernel
);

// Get the new messages added to the chat history object
for (int i = currentChatHistoryLength; i < chatHistory.Count; i++)
{
    Console.WriteLine(chatHistory[i]);
}

// Print the final message
Console.WriteLine(results);
```

```
// Add the final message to the chat history object  
chatHistory.Add(results);
```

Next steps

Now that you know how to create and manage a chat history object, you can learn more about function calling in the [Function calling](#) topic.

[Learn how function calling works](#)

Function calling with chat completion

Article • 06/24/2024

The most powerful feature of chat completion is the ability to call functions from the model. This allows you to create a chat bot that can interact with your existing code, making it possible to automate business processes, create code snippets, and more.

With Semantic Kernel, we simplify the process of using function calling by automatically describing your functions and their parameters to the model and then handling the back-and-forth communication between the model and your code.

When using function calling, however, it's good to understand what's *actually* happening behind the scenes so that you can optimize your code and make the most of this feature.

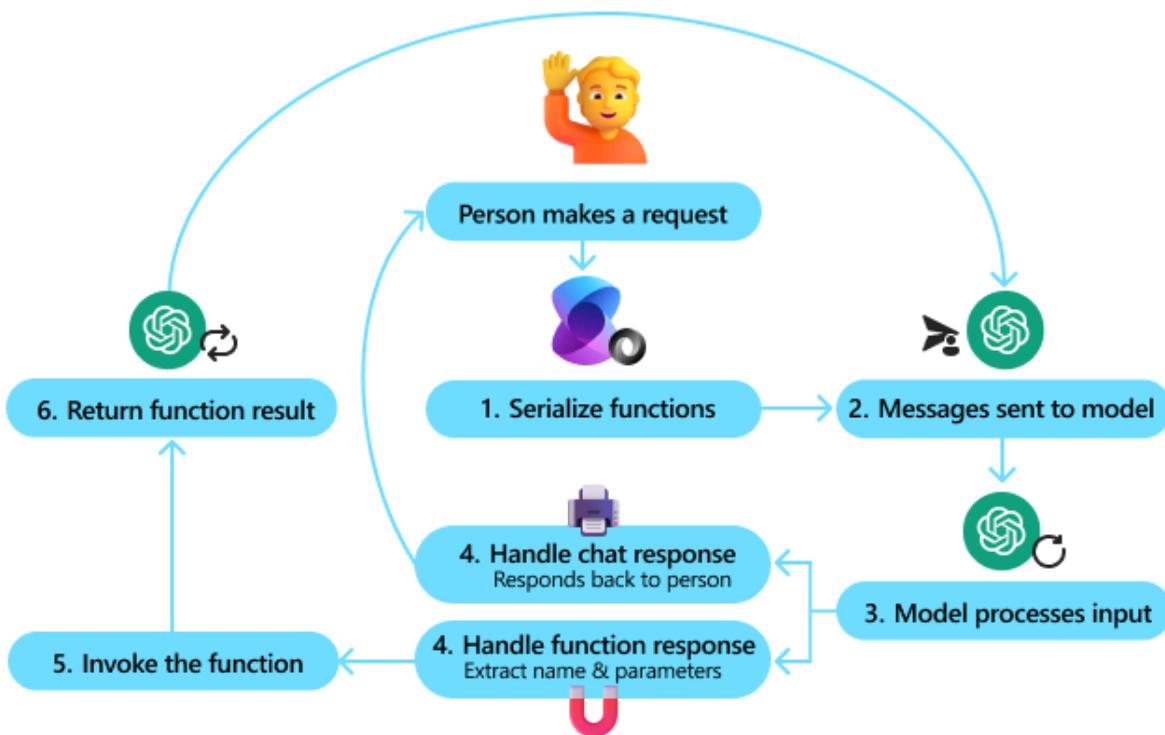
How function calling works

When you make a request to a model with function calling enabled, Semantic Kernel performs the following steps:

[\[+\] Expand table](#)

Step	Description
1 Serialize functions	All of the available functions (and its input parameters) in the kernel are serialized using JSON schema.
2 Send the messages and functions to the model	The serialized functions (and the current chat history) are sent to the model as part of the input.
3 Model processes the input	The model processes the input and generates a response. The response can either be a chat message or a function call
4 Handle the response	If the response is a chat message, it is returned to the developer to print the response to the screen. If the response is a function call, however, Semantic Kernel extracts the function name and its parameters.
5 Invoke the function	The extracted function name and parameters are used to invoke the function in the kernel.
6 Return the function result	The result of the function is then sent back to the model as part of the chat history. Steps 2-6 are then repeated until the model sends a termination signal

The following diagram illustrates the process of function calling:



The following section will use a concrete example to illustrate how function calling works in practice.

Example: Ordering a pizza

Let's assume you have a plugin that allows a user to order a pizza. The plugin has the following functions:

1. `get_pizza_menu`: Returns a list of available pizzas
2. `add_pizza_to_cart`: Adds a pizza to the user's cart
3. `remove_pizza_from_cart`: Removes a pizza from the user's cart
4. `get_pizza_from_cart`: Returns the specific details of a pizza in the user's cart
5. `get_cart`: Returns the user's current cart
6. `checkout`: Checks out the user's cart

In C#, the plugin might look like this:

C#

```
public class OrderPizzaPlugin(  
    IPizzaService pizzaService,  
    IUserContext userContext,  
    IPaymentService paymentService)
```

```

{
    [KernelFunction("get_pizza_menu")]
    public async Task<Menu> GetPizzaMenuAsync()
    {
        return await pizzaService.GetMenu();
    }

    [KernelFunction("add_pizza_to_cart")]
    [Description("Add a pizza to the user's cart; returns the new item and
updated cart")]
    public async Task<CartDelta> AddPizzaToCart(
        PizzaSize size,
        List<PizzaToppings> toppings,
        int quantity = 1,
        string specialInstructions = ""
    )
    {
        Guid cartId = userContext.GetCartId();
        return await pizzaService.AddPizzaToCart(
            cartId: cartId,
            size: size,
            toppings: toppings,
            quantity: quantity,
            specialInstructions: specialInstructions);
    }

    [KernelFunction("remove_pizza_from_cart")]
    public async Task<RemovePizzaResponse> RemovePizzaFromCart(int pizzaId)
    {
        Guid cartId = userContext.GetCartId();
        return await pizzaService.RemovePizzaFromCart(cartId, pizzaId);
    }

    [KernelFunction("get_pizza_from_cart")]
    [Description("Returns the specific details of a pizza in the user's
cart; use this instead of relying on previous messages since the cart may
have changed since then.")]
    public async Task<Pizza> GetPizzaFromCart(int pizzaId)
    {
        Guid cartId = await userContext.GetCartIdAsync();
        return await pizzaService.GetPizzaFromCart(cartId, pizzaId);
    }

    [KernelFunction("get_cart")]
    [Description("Returns the user's current cart, including the total price
and items in the cart.")]
    public async Task<Cart> GetCart()
    {
        Guid cartId = await userContext.GetCartIdAsync();
        return await pizzaService.GetCart(cartId);
    }

    [KernelFunction("checkout")]
    [Description("Checkouts the user's cart; this function will retrieve the
payment from the user and complete the order.")]
}

```

```

public async Task<CheckoutResponse> Checkout()
{
    Guid cartId = await userContext.GetCartIdAsync();
    Guid paymentId = await
paymentService.RequestPaymentFromUserAsync(cartId);

    return await pizzaService.Checkout(cartId, paymentId);
}
}

```

You would then add this plugin to the kernel like so:

```

C#

IKernelBuilder kernelBuilder = new KernelBuilder();
kernelBuilder..AddAzureOpenAIChatCompletion(
    deploymentName: "NAME_OF_YOUR_DEPLOYMENT",
    apiKey: "YOUR_API_KEY",
    endpoint: "YOUR_AZURE_ENDPOINT"
);
kernelBuilder.Plugins.AddFromType<OrderPizzaPlugin>("OrderPizza");
Kernel kernel = kernelBuilder.Build();

```

1) Serializing the functions

When you create a kernel with the `OrderPizzaPlugin`, the kernel will automatically serialize the functions and their parameters. This is necessary so that the model can understand the functions and their inputs.

For the above plugin, the serialized functions would look like this:

JSON

```

[
  {
    "type": "function",
    "function": {
      "name": "OrderPizza-get_pizza_menu",
      "parameters": {
        "type": "object",
        "properties": {},
        "required": []
      }
    }
  },
  {
    "type": "function",
    "function": {
      "name": "OrderPizza-add_pizza_to_cart",
      "parameters": {
        "type": "object",
        "properties": {}
      }
    }
  }
]

```

```
        "description": "Add a pizza to the user's cart; returns the new item and updated cart",
        "parameters": {
            "type": "object",
            "properties": {
                "size": {
                    "type": "string",
                    "enum": ["Small", "Medium", "Large"]
                },
                "toppings": {
                    "type": "array",
                    "items": {
                        "type": "string",
                        "enum": ["Cheese", "Pepperoni", "Mushrooms"]
                    }
                },
                "quantity": {
                    "type": "integer",
                    "default": 1,
                    "description": "Quantity of pizzas"
                },
                "specialInstructions": {
                    "type": "string",
                    "default": "",
                    "description": "Special instructions for the pizza"
                }
            },
            "required": ["size", "toppings"]
        }
    },
    {
        "type": "function",
        "function": {
            "name": "OrderPizza-remove_pizza_from_cart",
            "parameters": {
                "type": "object",
                "properties": {
                    "pizzaId": {
                        "type": "integer"
                    }
                },
                "required": ["pizzaId"]
            }
        }
    },
    {
        "type": "function",
        "function": {
            "name": "OrderPizza-get_pizza_from_cart",
            "description": "Returns the specific details of a pizza in the user's cart; use this instead of relying on previous messages since the cart may have changed since then.",
            "parameters": {
                "type": "object",
                "properties": {
                    "pizzaId": {
                        "type": "integer"
                    }
                }
            }
        }
    }
]
```

```

    "properties": {
      "pizzaId": {
        "type": "integer"
      }
    },
    "required": ["pizzaId"]
  }
},
{
  "type": "function",
  "function": {
    "name": "OrderPizza-get_cart",
    "description": "Returns the user's current cart, including the total price and items in the cart.",
    "parameters": {
      "type": "object",
      "properties": {},
      "required": []
    }
  }
},
{
  "type": "function",
  "function": {
    "name": "OrderPizza-checkout",
    "description": "Checkouts the user's cart; this function will retrieve the payment from the user and complete the order.",
    "parameters": {
      "type": "object",
      "properties": {},
      "required": []
    }
  }
}
]

```

There's a few things to note here which can impact both the performance and the quality of the chat completion:

- Verbosity of function schema** – Serializing functions for the model to use doesn't come for free. The more verbose the schema, the more tokens the model has to process, which can slow down the response time and increase costs.

💡 Tip

Keep your functions as simple as possible. In the above example, you'll notice that not *all* functions have descriptions where the function name is self-explanatory. This is intentional to reduce the number of tokens. The parameters are also kept simple; anything the model shouldn't need to know

(like the `cartId` or `paymentId`) are kept hidden. This information is instead provided by internal services.

⚠ Note

The one thing you don't need to worry about is the complexity of the return types. You'll notice that the return types are not serialized in the schema. This is because the model doesn't need to know the return type to generate a response. In the step 6, however, we'll see how overly verbose return types can impact the quality of the chat completion.

2. **Parameter types** – With the schema, you can specify the type of each parameter. This is important for the model to understand the expected input. In the above example, the `size` parameter is an enum, and the `toppings` parameter is an array of enums. This helps the model generate more accurate responses.

💡 Tip

Avoid, where possible, using `string` as a parameter type. The model can't infer the type of string, which can lead to ambiguous responses. Instead, use enums or other types (e.g., `int`, `float`, and complex types) where possible.

3. **Required parameters** - You can also specify which parameters are required. This is important for the model to understand which parameters are *actually* necessary for the function to work. Later on in step 3, the model will use this information to provide as minimal information as necessary to call the function.

💡 Tip

Only mark parameters as required if they are *actually* required. This helps the model call functions more quickly and accurately.

4. **Function descriptions** – Function descriptions are optional but can help the model generate more accurate responses. In particular, descriptions can tell the model what to expect from the response since the return type is not serialized in the schema. If the model is using functions improperly, you can also add descriptions to provide examples and guidance.

For example, in the `get_pizza_from_cart` function, the description tells the user to use this function instead of relying on previous messages. This is important because the cart may have changed since the last message.

 **Tip**

Before adding a description, ask yourself if the model *needs* this information to generate a response. If not, consider leaving it out to reduce verbosity. You can always add descriptions later if the model is struggling to use the function properly.

5. Plugin name – As you can see in the serialized functions, each function has a `name` property. Semantic Kernel uses the plugin name to namespace the functions. This is important because it allows you to have multiple plugins with functions of the same name. For example, you may have plugins for multiple search services, each with their own `search` function. By namespacing the functions, you can avoid conflicts and make it easier for the model to understand which function to call.

Knowing this, you should choose a plugin name that is unique and descriptive. In the above example, the plugin name is `OrderPizza`. This makes it clear that the functions are related to ordering pizza.

 **Tip**

When choosing a plugin name, we recommend removing superfluous words like "plugin" or "service". This helps reduce verbosity and makes the plugin name easier to understand for the model.

2) Sending the messages and functions to the model

Once the functions are serialized, they are sent to the model along with the current chat history. This allows the model to understand the context of the conversation and the available functions.

In this scenario, we can imagine the user asking the assistant to add a pizza to their cart:

C#

```
ChatHistory chatHistory = [];
chatHistory.AddUserMessage("I'd like to order a pizza!");
```

We can then send this chat history and the serialized functions to the model. The model will use this information to determine the best way to respond.

```
C#  
  
IChatCompletionService chatCompletion =  
kernel.GetRequiredService<IChatCompletionService>();  
  
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()  
{  
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions  
};  
  
ChatResponse response = await chatCompletion.GetChatMessageContentAsync(  
    chatHistory,  
    executionSettings: openAIPromptExecutionSettings,  
    kernel: kernel)
```

3) Model processes the input

With both the chat history and the serialized functions, the model can determine the best way to respond. In this case, the model recognizes that the user wants to order a pizza. The model would likely *want* to call the `add_pizza_to_cart` function, but because we specified the size and toppings as required parameters, the model will ask the user for this information:

```
C#  
  
Console.WriteLine(response);  
chatHistory.AddAssistantMessage(response);  
  
// "Before I can add a pizza to your cart, I need to  
// know the size and toppings. What size pizza would  
// you like? Small, medium, or large?"
```

Since the model wants the user to respond next, a termination signal will be sent to Semantic Kernel to stop automatic function calling until the user responds.

At this point, the user can respond with the size and toppings of the pizza they want to order:

```
C#  
  
chatHistory.AddUserMessage("I'd like a medium pizza with cheese and  
pepperoni, please.");  
  
response = await chatCompletion.GetChatMessageContentAsync()
```

```
chatHistory,  
kernel: kernel)
```

Now that the model has the necessary information, it can now call the `add_pizza_to_cart` function with the user's input. Behind the scenes, it adds a new message to the chat history that looks like this:

C#

```
"tool_calls": [  
  {  
    "id": "call_abc123",  
    "type": "function",  
    "function": {  
      "name": "OrderPizzaPlugin-add_pizza_to_cart",  
      "arguments": "{\n        \"size\": \"Medium\",  
        \"toppings\": [\n          \"Cheese\",  
          \"Pepperoni\"]\n      }  
    }  
]
```

💡 Tip

It's good to remember that every argument you require must be generated by the model. This means spending tokens to generate the response. Avoid arguments that require many tokens (like a GUID). For example, notice that we use an `int` for the `pizzaId`. Asking the model to send a one to two digit number is much easier than asking for a GUID.

ⓘ Important

This step is what makes function calling so powerful. Previously, AI app developers had to create separate processes to extract intent and slot fill functions. With function calling, the model can decide *when* to call a function and *what* information to provide.

4) Handle the response

When Semantic Kernel receives the response from the model, it checks if the response is a function call. If it is, Semantic Kernel extracts the function name and its parameters. In this case, the function name is `OrderPizzaPlugin-add_pizza_to_cart`, and the arguments are the size and toppings of the pizza.

With this information, Semantic Kernel can marshal the inputs into the appropriate types and pass them to the `add_pizza_to_cart` function in the `OrderPizzaPlugin`. In this example, the arguments originate as a JSON string but are deserialized by Semantic Kernel into a `PizzaSize` enum and a `List<PizzaToppings>`.

ⓘ Note

Marshaling the inputs into the correct types is one of the key benefits of using Semantic Kernel. Everything from the model comes in as a JSON object, but Semantic Kernel can automatically deserialize these objects into the correct types for your functions.

After marshalling the inputs, Semantic Kernel can also add the function call to the chat history:

C#

```
chatHistory.Add(
    new() {
        Role = AuthorRole.Assistant,
        Items = [
            new FunctionCallContent(
                functionName: "add_pizza_to_cart",
                pluginName: "OrderPizza",
                id: "call_abc123",
                arguments: new () { {"size", "Medium"}, {"toppings", ["Cheese", "Pepperoni"]} }
            )
        ]
    }
);
```

5) Invoke the function

Once Semantic Kernel has the correct types, it can finally invoke the `add_pizza_to_cart` function. Because the plugin uses dependency injection, the function can interact with external services like `pizzaService` and `userContext` to add the pizza to the user's cart.

Not all functions will succeed, however. If the function fails, Semantic Kernel can handle the error and provide a default response to the model. This allows the model to understand what went wrong and generate a response to the user.

💡 Tip

To ensure a model can self-correct, it's important to provide error messages that clearly communicate what went wrong and how to fix it. This can help the model retry the function call with the correct information.

6) Return the function result

After the function has been invoked, the function result is sent back to the model as part of the chat history. This allows the model to understand the context of the conversation and generate a subsequent response.

Behind the scenes, Semantic Kernel adds a new message to the chat history from the tool role that looks like this:

C#

```
chatHistory.Add(
    new() {
        Role = AuthorRole.Tool,
        Items = [
            new FunctionResultContent(
                functionName: "add_pizza_to_cart",
                pluginName: "OrderPizza",
                id: "0001",
                result: "{ \"new_items\": [ { \"id\": 1, \"size\": \"Medium\", \"toppings\": [\"Cheese\", \"Pepperoni\"] } ] }"
            )
        ]
    }
);
```

Notice that the result is a JSON string that the model then needs to process. As before, the model will need to spend tokens consuming this information. This is why it's important to keep the return types as simple as possible. In this case, the return only includes the new items added to the cart, not the entire cart.

💡 Tip

Be as succinct as possible with your returns. Where possible, only return the information the model needs or summarize the information using another LLM prompt before returning it.

Repeat steps 2-6

After the result is returned to the model, the process repeats. The model processes the latest chat history and generates a response. In this case, the model might ask the user if they want to add another pizza to their cart or if they want to check out.

Parallel function calls

In the above example, we demonstrated how an LLM can call a single function. Often this can be slow if you need to call multiple functions in sequence. To speed up the process, several LLMs support parallel function calls. This allows the LLM to call multiple functions at once, speeding up the process.

For example, if a user wants to order multiple pizzas, the LLM can call the `add_pizza_to_cart` function for each pizza at the same time. This can significantly reduce the number of round trips to the LLM and speed up the ordering process.

Next steps

Now that you understand how function calling works, you can now learn how to actually use function calling in Semantic Kernel by referring to the [planning article](#)

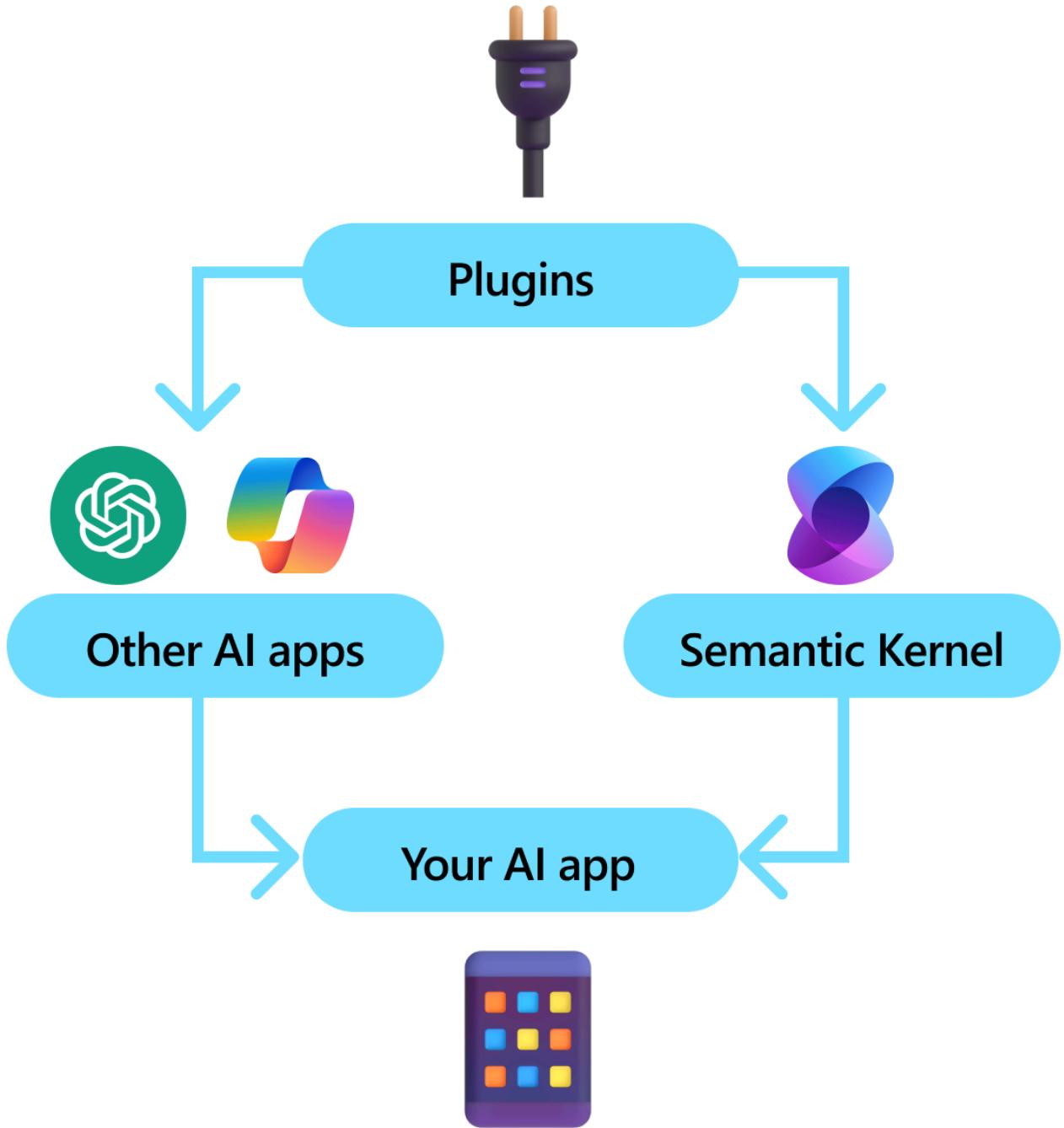
[Learn more about planning](#)

What is a Plugin?

Article • 06/24/2024

Plugins are a key component of Semantic Kernel. If you have already used plugins from ChatGPT or Copilot extensions in Microsoft 365, you're already familiar with them. With plugins, you can encapsulate your existing APIs into a collection that can be used by an AI. This allows you to give your AI the ability to perform actions that it wouldn't be able to do otherwise.

Behind the scenes, Semantic Kernel leverages [function calling](#), a native feature of most of the latest LLMs to allow LLMs, to perform [planning](#) and to invoke your APIs. With function calling, LLMs can request (i.e., call) a particular function. Semantic Kernel then marshals the request to the appropriate function in your codebase and returns the results back to the LLM so the LLM can generate a final response.



Not all AI SDKs have an analogous concept to plugins (most just have functions or tools). In enterprise scenarios, however, plugins are valuable because they encapsulate a set of functionality that mirrors how enterprise developers already develop services and APIs. Plugins also play nicely with dependency injection. Within a plugin's constructor, you can inject services that are necessary to perform the work of the plugin (e.g., database connections, HTTP clients, etc.). This is difficult to accomplish with other SDKs that lack plugins.

Anatomy of a plugin

At a high-level, a plugin is a group of [functions](#) that can be exposed to AI apps and services. The functions within plugins can then be orchestrated by an AI application to

accomplish user requests. Within Semantic Kernel, you can invoke these functions automatically with function calling.

① Note

In other platforms, functions are often referred to as "tools" or "actions". In Semantic Kernel, we use the term "functions" since they are typically defined as native functions in your codebase.

Just providing functions, however, is not enough to make a plugin. To power automatic orchestration with function calling, plugins also need to provide details that semantically describe how they behave. Everything from the function's input, output, and side effects need to be described in a way that the AI can understand, otherwise, the AI will not correctly call the function.

For example, the sample `WriterPlugin` plugin on the right has functions with semantic descriptions that describe what each function does. An LLM can then use these descriptions to choose the best functions to call to fulfill a user's ask.

In the picture on the right, an LLM would likely call the `ShortPoem` and `StoryGen` functions to satisfy the users ask thanks to the provided semantic descriptions.

Writer plugin

Function	Description for model
Brainstorm	Given a goal or topic description generate a list of ideas.
EmailGen	Write an email from the given bullet points.
ShortPoem	Turn a scenario into a short and entertaining poem.
StoryGen	Generate a list of synopsis for a novel or novella with sub-chapters.
Translate	Translate the input into a language of your choice.

Can you write me a short poem about living in Dublin, Ireland and then create a story based on the poem?



Planner

Copilot
Sure! Here's a story based on living along the Grand Canal in Dublin, Ireland...



Importing different types of plugins

There are two primary ways of importing plugins into Semantic Kernel: using [native code](#) or using an [OpenAPI specification](#). The former allows you to author plugins in your existing codebase that can leverage dependencies and services you already have. The latter allows you to import plugins from an OpenAPI specification, which can be shared across different programming languages and platforms.

Below we provide a simple example of importing and using a native plugin. To learn more about how to import these different types of plugins, refer to the following articles:

- [Importing native code](#)
- [Importing an OpenAPI specification](#)

💡 Tip

When getting started, we recommend using native code plugins. As your application matures, and as you work across cross-platform teams, you may want to consider using OpenAPI specifications to share plugins across different programming languages and platforms.

The different types of plugin functions

Within a plugin, you will typically have two different types of functions, those that retrieve data for retrieval augmented generation (RAG) and those that automate tasks. While each type is functionally the same, they are typically used differently within applications that use Semantic Kernel.

For example, with retrieval functions, you may want to use strategies to improve performance (e.g., caching and using cheaper intermediate models for summarization). Whereas with task automation functions, you'll likely want to implement human-in-the-loop approval processes to ensure that tasks are completed correctly.

To learn more about the different types of plugin functions, refer to the following articles:

- [Data retrieval functions](#)
- [Task automation functions](#)

Getting started with plugins

Using plugins within Semantic Kernel is always a three step process:

1. Define your plugin
2. Add the plugin to your kernel
3. And then either invoke the plugin's functions in either a prompt with function calling

Below we'll provide a high-level example of how to use a plugin within Semantic Kernel. Refer to the links above for more detailed information on how to create and use plugins.

1) Define your plugin

The easiest way to create a plugin is by defining a class and annotating its methods with the `KernelFunction` attribute. This let's Semantic Kernel know that this is a function that can be called by an AI or referenced in a prompt.

You can also import plugins from an [OpenAPI specification](#).

Below, we'll create a plugin that can retrieve the state of lights and alter its state.

💡 Tip

Since most LLM have been trained with Python for function calling, its recommended to use snake case for function names and property names even if you're using the C# or Java SDK.

C#

```
using System.ComponentModel;
using Microsoft.SemanticKernel;

public class LightsPlugin
{
    // Mock data for the lights
    private readonly List<LightModel> lights = new()
    {
        new LightModel { Id = 1, Name = "Table Lamp", IsOn = false, Brightness = 100, Hex = "FF0000" },
        new LightModel { Id = 2, Name = "Porch light", IsOn = false, Brightness = 50, Hex = "00FF00" },
        new LightModel { Id = 3, Name = "Chandelier", IsOn = true, Brightness = 75, Hex = "0000FF" }
    };

    [KernelFunction("get_lights")]
    [Description("Gets a list of lights and their current state")]
    [return: Description("An array of lights")]
}
```

```
public async Task<List<LightModel>> GetLightsAsync()
{
    return lights
}

[KernelFunction("get_state")]
[Description("Gets the state of a particular light")]
[return: Description("The state of the light")]
public async Task<LightModel?> GetStateAsync([Description("The ID of the
light")] int id)
{
    // Get the state of the light with the specified ID
    return lights.FirstOrDefault(light => light.Id == id);
}

[KernelFunction("change_state")]
[Description("Changes the state of the light")]
[return: Description("The updated state of the light; will return null if
the light does not exist")]
public async Task<LightModel?> ChangeStateAsync(int id, LightModel
LightModel)
{
    var light = lights.FirstOrDefault(light => light.Id == id);

    if (light == null)
    {
        return null;
    }

    // Update the light with the new state
    light.IsOn = LightModel.IsOn;
    light.Brightness = LightModel.Brightness;
    light.Hex = LightModel.Hex;

    return light;
}
}

public class LightModel
{
    [JsonPropertyName("id")]
    public int Id { get; set; }

    [JsonPropertyName("name")]
    public string Name { get; set; }

    [JsonPropertyName("is_on")]
    public bool? IsOn { get; set; }

    [JsonPropertyName("brightness")]
    public byte? Brightness { get; set; }

    [JsonPropertyName("hex")]
    public string? Hex { get; set; }
}
```

Notice that we provide descriptions for the function, return value, and parameters. This is important for the AI to understand what the function does and how to use it.

💡 Tip

Don't be afraid to provide detailed descriptions for your functions if an AI is having trouble calling them. Few-shot examples, recommendations for when to use (and not use) the function, and guidance on where to get required parameters can all be helpful.

2) Add the plugin to your kernel

Once you've defined your plugin, you can add it to your kernel by creating a new instance of the plugin and adding it to the kernel's plugin collection.

This example demonstrates the easiest way of adding a class as a plugin with the `AddFromType` method. To learn about other ways of adding plugins, refer to the [adding native plugins](#) article.

C#

```
var builder = new KernelBuilder();
builder.Plugins.AddFromType<LightsPlugin>("Lights")
Kernel kernel = builder.Build();
```

3) Invoke the plugin's functions

Finally, you can have the AI invoke your plugin's functions by using function calling. Below is an example that demonstrates how to coax the AI to call the `get_lights` function from the `Lights` plugin before calling the `change_state` function to turn on a light.

C#

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

// Create a kernel with Azure OpenAI chat completion
var builder = Kernel.CreateBuilder().AddAzureOpenAIChatCompletion(modelId,
endpoint, apiKey);
```

```

// Build the kernel
Kernel kernel = builder.Build();
var chatCompletionService =
kernel.GetRequiredService<IChatCompletionService>();

// Add a plugin (the LightsPlugin class is defined below)
kernel.Plugins.AddFromType<LightsPlugin>("Lights");

// Enable planning
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};

// Create a history store the conversation
var history = new ChatHistory();
history.AddUserMessage("Please turn on the lamp");

// Get the response from the AI
var result = await chatCompletionService.GetChatMessageContentAsync(
    history,
    executionSettings: openAIPromptExecutionSettings,
    kernel: kernel);

// Print the results
Console.WriteLine("Assistant > " + result);

// Add the message from the agent to the chat history
history.AddAssistantMessage(result);

```

With the above code, you should get a response that looks like the following:

[\[\] Expand table](#)

Role	Message
>User	Please turn on the lamp
Assistant (function call)	Lights.get_lights()
Tool	[{ "id": 1, "name": "Table Lamp", "isOn": false, "brightness": 100, "hex": "FF0000" }, { "id": 2, "name": "Porch light", "isOn": false, "brightness": 50, "hex": "00FF00" }, { "id": 3, "name": "Chandelier", "isOn": true, "brightness": 75, "hex": "0000FF" }]
Assistant (function call)	Lights.change_state(1, { "isOn": true })
Tool	{ "id": 1, "name": "Table Lamp", "isOn": true, "brightness": 100, "hex": "FF0000" }

Role	Message
Assistant	The lamp is now on

💡 Tip

While you can invoke a plugin function directly, this is not advised because the AI should be the one deciding which functions to call. If you need explicit control over which functions are called, consider using standard methods in your codebase instead of plugins.

Add native code as a plugin

Article • 06/24/2024

The easiest way to provide an AI agent with capabilities that are not natively supported is to wrap native code into a plugin. This allows you to leverage your existing skills as an app developer to extend the capabilities of your AI agents.

Behind the scenes, Semantic Kernel will then use the descriptions you provide, along with reflection, to semantically describe the plugin to the AI agent. This allows the AI agent to understand the capabilities of the plugin and how to interact with it.

Providing the LLM with the right information

When authoring a plugin, you need to provide the AI agent with the right information to understand the capabilities of the plugin and its functions. This includes:

- The name of the plugin
- The names of the functions
- The descriptions of the functions
- The parameters of the functions
- The schema of the parameters

The value of Semantic Kernel is that it can automatically generate most of this information from the code itself. As a developer, this just means that you must provide the semantic descriptions of the functions and parameters so the AI agent can understand them. If you properly comment and annotate your code, however, you likely already have this information on hand.

Below, we'll walk through the two different ways of providing your AI agent with native code and how to provide this semantic information.

Defining a plugin using a class

The easiest way to create a native plugin is to start with a class and then add methods annotated with the `KernelFunction` attribute. It is also recommended to liberally use the `Description` annotation to provide the AI agent with the necessary information to understand the function.

C#

```

public class LightsPlugin
{
    private readonly List<LightModel> _lights;

    public LightsPlugin(LoggerFactory loggerFactory, List<LightModel> lights)
    {
        _lights = lights;
    }

    [KernelFunction("get_lights")]
    [Description("Gets a list of lights and their current state")]
    [return: Description("An array of lights")]
    public async Task<List<LightModel>> GetLightsAsync()
    {
        return _lights;
    }

    [KernelFunction("change_state")]
    [Description("Changes the state of the light")]
    [return: Description("The updated state of the light; will return null if
the light does not exist")]
    public async Task<LightModel?> ChangeStateAsync(LightModel changeState)
    {
        // Find the light to change
        var light = _lights.FirstOrDefault(l => l.Id == changeState.Id);

        // If the light does not exist, return null
        if (light == null)
        {
            return null;
        }

        // Update the light state
        light.IsOn = changeState.IsOn;
        light.Brightness = changeState.Brightness;
        light.Color = changeState.Color;

        return light;
    }
}

```

💡 Tip

Because the LLMs are predominantly trained on Python code, it is recommended to use snake_case for function names and parameters (even if you're using C# or Java). This will help the AI agent better understand the function and its parameters.

If your function has a complex object as an input variable, Semantic Kernel will also generate a schema for that object and pass it to the AI agent. Similar to functions, you

should provide `Description` annotations for properties that are non-obvious to the AI. Below is the definition for the `LightState` class and the `Brightness` enum.

C#

```
using System.Text.Json.Serialization;

public class LightModel
{
    [JsonPropertyName("id")]
    public int Id { get; set; }

    [JsonPropertyName("name")]
    public string? Name { get; set; }

    [JsonPropertyName("is_on")]
    public bool? IsOn { get; set; }

    [JsonPropertyName("brightness")]
    public enum? Brightness { get; set; }

    [JsonPropertyName("color")]
    [Description("The color of the light with a hex code (ensure you include
the # symbol)")]
    public string? Color { get; set; }
}

[JsonConverter(typeof(JsonStringEnumConverter))]
public enum Brightness
{
    Low,
    Medium,
    High
}
```

ⓘ Note

While this is a "fun" example, it does a good job showing just how complex a plugin's parameters can be. In this single case, we have a complex object with *four* different types of properties: an integer, string, boolean, and enum. Semantic Kernel's value is that it can automatically generate the schema for this object and pass it to the AI agent and marshal the parameters generated by the AI agent into the correct object.

Once you're done authoring your plugin class, you can add it to the kernel using the `AddFromType<>` or `AddFromObject` methods.

💡 Tip

When creating a function, always ask yourself "how can I give the AI additional help to use this function?" This can include using specific input types (avoid strings where possible), providing descriptions, and examples.

Adding a plugin using the `AddFromObject` method

The `AddFromObject` method allows you to add an instance of the plugin class directly to the plugin collection in case you want to directly control how the plugin is constructed.

For example, the constructor of the `LightsPlugin` class requires the list of lights. In this case, you can create an instance of the plugin class and add it to the plugin collection.

C#

```
List<LightModel> lights = new()
{
    new LightModel { Id = 1, Name = "Table Lamp", IsOn = false, Brightness
= Brightness.Medium, Color = "#FFFFFF" },
    new LightModel { Id = 2, Name = "Porch light", IsOn = false,
Brightness = Brightness.High, Color = "#FF0000" },
    new LightModel { Id = 3, Name = "Chandelier", IsOn = true, Brightness
= Brightness.Low, Color = "#FFFF00" }
};

kernel.Plugins.AddFromObject(new LightsPlugin(lights));
```

Adding a plugin using the `AddFromType<>` method

When using the `AddFromType<>` method, the kernel will automatically use dependency injection to create an instance of the plugin class and add it to the plugin collection.

This is helpful if your constructor requires services or other dependencies to be injected into the plugin. For example, our `LightsPlugin` class may require a logger and a light service to be injected into it instead of a list of lights.

C#

```
public class LightsPlugin
{
    private readonly Logger _logger;
    private readonly LightService _lightService;
```

```

    public LightsPlugin(LoggerFactory loggerFactory, LightService
lightService)
{
    _logger = loggerFactory.CreateLogger<LightsPlugin>();
    _lightService = lightService;
}

[KernelFunction("get_lights")]
[Description("Gets a list of lights and their current state")]
[return: Description("An array of lights")]
public async Task<List<LightModel>> GetLightsAsync()
{
    _logger.LogInformation("Getting lights");
    return lightService.GetLights();
}

[KernelFunction("change_state")]
[Description("Changes the state of the light")]
[return: Description("The updated state of the light; will return null if
the light does not exist")]
public async Task<LightModel?> ChangeStateAsync(LightModel changeState)
{
    _logger.LogInformation("Changing light state");
    return lightService.ChangeState(changeState);
}
}

```

With Dependency Injection, you can add the required services and plugins to the kernel builder before building the kernel.

C#

```

var builder = Kernel.CreateBuilder();

// Add dependencies for the plugin
builder.Services.AddLogging(loggingBuilder =>
loggingBuilder.AddConsole().SetMinimumLevel(LogLevel.Trace));
builder.Services.AddSingleton<LightService>();

// Add the plugin to the kernel
builder.Plugins.AddFromType<LightsPlugin>("Lights");

// Build the kernel
Kernel kernel = builder.Build();

```

Defining a plugin using a collection of functions

Less common but still useful is defining a plugin using a collection of functions. This is particularly useful if you need to dynamically create a plugin from a set of functions at runtime.

Using this process requires you to use the function factory to create individual functions before adding them to the plugin.

```
C#  
  
kernel.Plugins.AddFromFunctions("time_plugin",  
[  
    KernelFunctionFactory.CreateFromMethod(  
        method: () => DateTime.Now,  
        functionName: "get_time",  
        description: "Get the current time"  
    ),  
    KernelFunctionFactory.CreateFromMethod(  
        method: (DateTime start, DateTime end) => (end -  
start).TotalSeconds,  
        functionName: "diff_time",  
        description: "Get the difference between two times in seconds"  
    )  
]);
```

Additional strategies for adding native code with Dependency Injection

If you're working with Dependency Injection, there are additional strategies you can take to create and add plugins to the kernel. Below are some examples of how you can add a plugin using Dependency Injection.

Inject a plugin collection



We recommend making your plugin collection a transient service so that it is disposed of after each use since the plugin collection is mutable. Creating a new plugin collection for each use is cheap, so it should not be a performance concern.

```
C#  
  
var builder = Host.CreateApplicationBuilder(args);  
  
// Create native plugin collection  
builder.Services.AddTransient((serviceProvider)=>{  
    KernelPluginCollection pluginCollection = [];  
    pluginCollection.AddFromType<LightsPlugin>("Lights");  
  
    return pluginCollection;
```

```
});  
  
// Create the kernel service  
builder.Services.AddTransient<Kernel>((serviceProvider)=> {  
    KernelPluginCollection pluginCollection =  
    serviceProvider.GetRequiredService<KernelPluginCollection>();  
  
    return new Kernel(serviceProvider, pluginCollection);  
});
```

💡 Tip

As mentioned in the [kernel article](#), the kernel is extremely lightweight, so creating a new kernel for each use as a transient is not a performance concern.

Generate your plugins as singletons

Plugins are not mutable, so it's typically safe to create them as singletons. This can be done by using the plugin factory and adding the resulting plugin to your service collection.

C#

```
var builder = Host.CreateApplicationBuilder(args);  
  
// Create singletons of your plugin  
builder.Services.AddKeyedSingleton("LightPlugin", (serviceProvider, key) =>  
{  
    return KernelPluginFactory.CreateFromType<LightsPlugin>();  
});  
  
// Create a kernel service with singleton plugin  
builder.Services.AddTransient((serviceProvider)=> {  
    KernelPluginCollection pluginCollection = [  
        serviceProvider.GetRequiredKeyedService<KernelPlugin>("LightPlugin")  
    ];  
  
    return new Kernel(serviceProvider, pluginCollection);  
});
```

Next steps

Now that you know how to create a plugin, you can now learn how to use them with your AI agent. Depending on the type of functions you've added to your plugins, there are different patterns you should follow. For retrieval functions, refer to the [using](#)

[retrieval functions](#) article. For task automation functions, refer to the [using task automation functions](#) article.

[Learn about using retrieval functions](#)

Add plugins from OpenAPI specifications

Article • 06/24/2024

Often in an enterprise, you already have a set of APIs that perform real work. These could be used by other automation services or power front-end applications that humans interact with. In Semantic Kernel, you can add these exact same APIs as plugins so your agents can also use them.

An example OpenAPI specification

Take for example an API that allows you to alter the state of light bulbs. The OpenAPI specification for this API might look like this:

```
JSON

{
  "openapi": "3.0.1",
  "info": {
    "title": "Light API",
    "version": "v1"
  },
  "paths": {
    "/Light": {
      "get": {
        "tags": [
          "Light"
        ],
        "summary": "Retrieves all lights in the system.",
        "operationId": "get_all_lights",
        "responses": {
          "200": {
            "description": "Returns a list of lights with their current state",
            "application/json": {
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/LightStateModel"
                }
              }
            }
          }
        }
      }
    },
    "/Light/{id}": {
      "put": {
        "tags": [
          "Light"
        ],
        "summary": "Updates a specific light's state",
        "operationId": "update_light_state",
        "parameters": [
          {
            "name": "id",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "The updated light state"
          }
        }
      }
    }
  }
}
```

```

"post": {
    "tags": [
        "Light"
    ],
    "summary": "Changes the state of a light.",
    "operationId": "change_light_state",
    "parameters": [
        {
            "name": "id",
            "in": "path",
            "description": "The ID of the light to change from the get_all_lights tool.",
            "required": true,
            "style": "simple",
            "schema": {
                "type": "string"
            }
        }
    ],
    "requestBody": {
        "description": "The new state of the light and change parameters.",
        "content": {
            "application/json": {
                "schema": {
                    "$ref": "#/components/schemas/ChangeStateRequest"
                }
            }
        }
    },
    "responses": {
        "200": {
            "description": "Returns the updated light state",
            "content": {
                "application/json": {
                    "schema": {
                        "$ref": "#/components/schemas/LightStateModel"
                    }
                }
            }
        },
        "404": {
            "description": "If the light is not found"
        }
    }
},
"components": {
    "schemas": {
        "ChangeStateRequest": {
            "type": "object",
            "properties": {

```

```
"isOn": {
    "type": "boolean",
    "description": "Specifies whether the light is turned
on or off.",
    "nullable": true
},
"hexColor": {
    "type": "string",
    "description": "The hex color code for the light.",
    "nullable": true
},
"brightness": {
    "type": "integer",
    "description": "The brightness level of the light.",
    "format": "int32",
    "nullable": true
},
"fadeDurationInMilliseconds": {
    "type": "integer",
    "description": "Duration for the light to fade to the
new state, in milliseconds.",
    "format": "int32",
    "nullable": true
},
"scheduledTime": {
    "type": "string",
    "description": "Use ScheduledTime to synchronize
lights. It's recommended that you asynchronously create tasks for each light
that's scheduled to avoid blocking the main thread.",
    "format": "date-time",
    "nullable": true
}
},
"additionalProperties": false,
"description": "Represents a request to change the state of
the light."
},
"LightStateModel": {
    "type": "object",
    "properties": {
        "id": {
            "type": "string",
            "nullable": true
        },
        "name": {
            "type": "string",
            "nullable": true
        },
        "on": {
            "type": "boolean",
            "nullable": true
        },
        "brightness": {
            "type": "integer",
            "format": "int32",
            "nullable": true
        }
    }
}
```

```
        "nullable": true
    },
    "hexColor": {
        "type": "string",
        "nullable": true
    }
},
"additionalProperties": false
}
}
}
```

This specification provides everything needed by the AI to understand the API and how to interact with it. The API includes two endpoints: one to get all lights and another to change the state of a light. It also provides the following:

- Semantic descriptions for the endpoints and their parameters
- The types of the parameters
- The expected responses

Since the AI agent can understand this specification, you can add it as a plugin to the agent.

💡 Tip

If you have existing OpenAPI specifications, you may need to make alterations to make them easier for an AI to understand them. For example, you may need to provide guidance in the descriptions. For more tips on how to make your OpenAPI specifications AI-friendly, see [Tips and tricks for adding OpenAPI plugins](#).

Adding the OpenAPI plugin

With a few lines of code, you can add the OpenAPI plugin to your agent. The following code snippet shows how to add the light plugin from the OpenAPI specification above:

C#

```
await kernel.ImportPluginFromOpenApiAsync(
    pluginName: "lights",
    uri: new Uri("https://example.com/v1/swagger.json"),
    executionParameters: new OpenApiFunctionExecutionParameters()
{
    // Determines whether payload parameter names are augmented with
    namespaces.
    // Namespaces prevent naming conflicts by adding the parent parameter
```

```

name
    // as a prefix, separated by dots
    EnablePayloadNamespacing = true
}
);

```

Afterwards, you can use the plugin in your agent as if it were a native plugin.

Tips and tricks for adding OpenAPI plugins

Since OpenAPI specifications are typically designed for humans, you may need to make some alterations to make them easier for an AI to understand. Here are some tips and tricks to help you do that:

[Expand table](#)

Recommendation	Description
Version control your API specifications	Instead of pointing to a live API specification, consider checking-in and versioning your Swagger file. This will allow your AI researchers to test (and alter) the API specification used by the AI agent without affecting the live API and vice versa.
Limit the number of endpoints	Try to limit the number of endpoints in your API. Consolidate similar functionalities into single endpoints with optional parameters to reduce complexity.
Use descriptive names for endpoints and parameters	Ensure that the names of your endpoints and parameters are descriptive and self-explanatory. This helps the AI understand their purpose without needing extensive explanations.
Use consistent naming conventions	Maintain consistent naming conventions throughout your API. This reduces confusion and helps the AI learn and predict the structure of your API more easily.
Simplify your API specifications	Often, OpenAPI specifications are very detailed and include a lot of information that isn't necessary for the AI agent to help a user. The simpler the API, the fewer tokens you need to spend to describe it, and the fewer tokens the AI needs to send requests to it.
Avoid string parameters	When possible, avoid using string parameters in your API. Instead, use more specific types like integers, booleans, or enums. This will help the AI understand the API better.
Provide examples in descriptions	When humans use Swagger files, they typically are able to test the API using the Swagger UI, which includes sample requests and responses. Since the AI agent can't do this, consider providing examples in the descriptions of the parameters.

Recommendation	Description
Reference other endpoints in descriptions	Often, AIs will confuse similar endpoints. To help the AI differentiate between endpoints, consider referencing other endpoints in the descriptions. For example, you could say "This endpoint is similar to the <code>get_all_lights</code> endpoint, but it only returns a single light."
Provide helpful error messages	While not within the OpenAPI specification, consider providing error messages that help the AI self-correct. For example, if a user provides an invalid ID, consider providing an error message that suggests the AI agent get the correct ID from the <code>get_all_lights</code> endpoint.

Next steps

Now that you know how to create a plugin, you can now learn how to use them with your AI agent. Depending on the type of functions you've added to your plugins, there are different patterns you should follow. For retrieval functions, refer to the [using retrieval functions](#) article. For task automation functions, refer to the [using task automation functions](#) article.

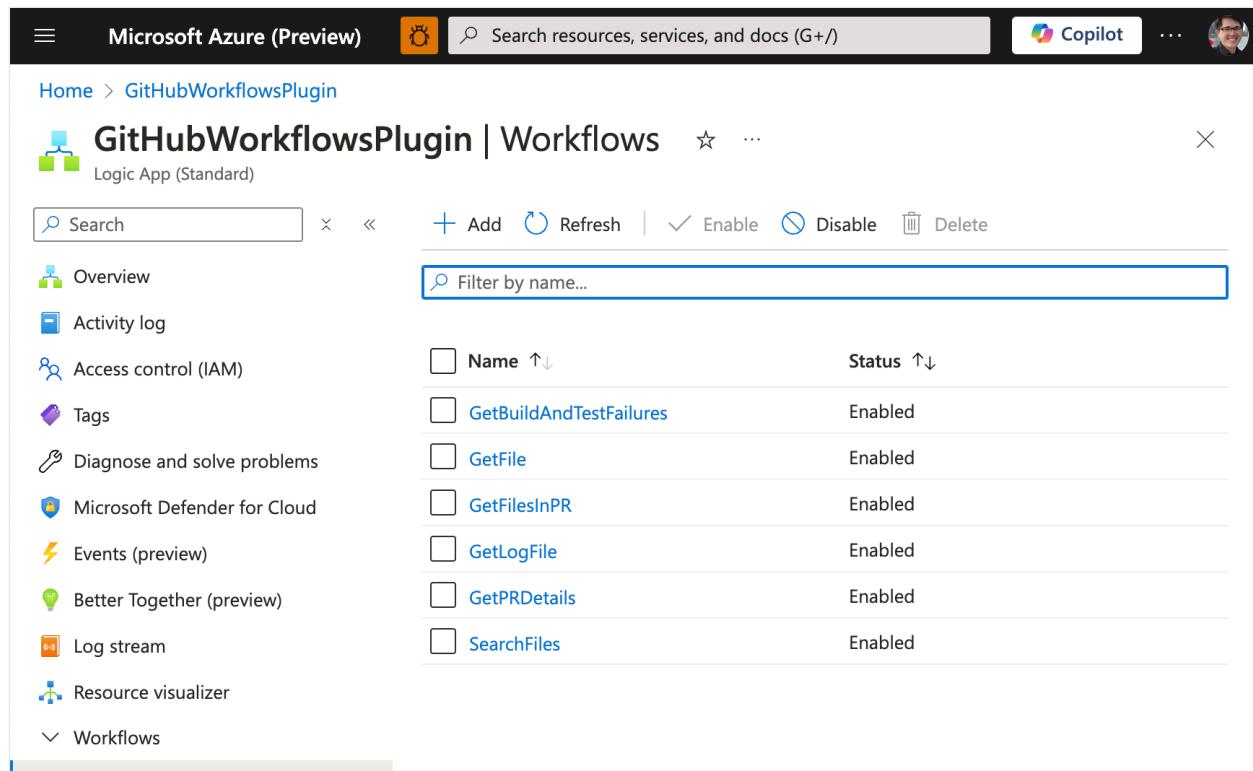
[Learn about using retrieval functions](#)

Add Logic Apps as plugins

Article • 06/24/2024

Often in an enterprise, you already have a set of workflows that perform real work in Logic Apps. These could be used by other automation services or power front-end applications that humans interact with. In Semantic Kernel, you can add these exact same workflows as plugins so your agents can also use them.

Take for example the Logic Apps workflows used by the Semantic Kernel team to answer questions about new PRs. With the following workflows, an agent has everything it needs to retrieve code changes, search for related files, and check failure logs.



The screenshot shows the Microsoft Azure (Preview) portal with the title "GitHubWorkflowsPlugin | Workflows". The page is for a Logic App (Standard). On the left, there's a sidebar with various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Microsoft Defender for Cloud, Events (preview), Better Together (preview), Log stream, Resource visualizer, and Workflows. The main area shows a table of workflows:

Name	Status
GetBuildAndTestFailures	Enabled
GetFile	Enabled
GetFilesInPR	Enabled
GetLogFile	Enabled
GetPRDetails	Enabled
SearchFiles	Enabled

- **Search files** – to find code snippets that are relevant to a given problem
- **Get file** – to retrieve the contents of a file in the GitHub repository
- **Get PR details** – to retrieve the details of a PR (e.g., the PR title, description, and author)
- **Get PR files** – to retrieve the files that were changed in a PR
- **Get build and test failures** – to retrieve the build and test failures for a given GitHub action run
- **Get log file** – to retrieve the log file for a given GitHub action run

Leveraging Logic Apps for Semantic Kernel plugins is also a great way to take advantage of the over [1,400 connectors available in Logic Apps](#). This means you can easily connect to a wide variety of services and systems without writing any code.

ⓘ Important

Today, you can only add standard Logic Apps (also known as single-tenant Logic Apps) as plugins. Consumption Logic Apps are coming soon.

Importing Logic Apps as plugins

To add Logic Apps workflows to Semantic Kernel, you'll use the same methods as loading in an [OpenAPI specifications](#). Below is some sample code.

C#

```
await kernel.ImportPluginFromOpenApiAsync(
    pluginName: "openapi_plugin",
    uri: new Uri("https://example.azurewebsites.net/swagger.json"),
    executionParameters: new OpenApiOperationExecutionParameters()
    {
        // Determines whether payload parameter names are augmented with
        namespaces.
        // Namespaces prevent naming conflicts by adding the parent
        parameter name
        // as a prefix, separated by dots
        EnablePayloadNamespacing = true
    }
);
```

Setting up Logic Apps for Semantic Kernel

Before you can import a Logic App as a plugin, you must first set up the Logic App to be accessible by Semantic Kernel. This involves enabling metadata endpoints and configuring your application for Easy Auth before finally importing the Logic App as a plugin with authentication.

Enable metadata endpoints

For the easiest setup, you can enable unauthenticated access to the metadata endpoints for your Logic App. This will allow you to import your Logic App as a plugin into Semantic Kernel without needing to create a custom HTTP client to handle authentication for the initial import.

The below host.json file will create two unauthenticated endpoints. You can do this in azure portal by [going to kudu console and editing the host.json file located at](#)

C:\home\site\wwwroot\host.json.

```
JSON

{
    "version": "2.0",
    "extensionBundle": {
        "id": "Microsoft.Azure.Functions.ExtensionBundle.Workflows",
        "version": "[1.*, 2.0.0)"
    },
    "extensions": {
        "http": {
            "routePrefix": ""
        },
        "workflow": {
            "MetadataEndpoints": {
                "plugin": {
                    "enable": true,
                    "Authentication": {
                        "Type": "Anonymous"
                    }
                },
                "openapi": {
                    "enable": true,
                    "Authentication": {
                        "Type": "Anonymous"
                    }
                }
            },
            "Settings": {
                "Runtime.Triggers.RequestTriggerDefaultApiVersion": "2020-05-01-preview"
            }
        }
    }
}
```

Configure your application for Easy Auth

You now want to secure your Logic App workflows so only authorized users can access them. You can do this by enabling Easy Auth on your Logic App. This will allow you to use the same authentication mechanism as your other Azure services, making it easier to manage your security policies.

For an in-depth walkthrough on setting up Easy Auth, refer to this tutorial titled [Trigger workflows in Standard logic apps with Easy Auth ↗](#).

For those already familiar with Easy Auth (and already have an Entra client app you want to use), this is the configuration you'll want to post to Azure management.

Bash

```
#!/bin/bash

# Variables
subscription_id="[SUBSCRIPTION_ID]"
resource_group="[RESOURCE_GROUP]"
app_name="[APP_NAME]"
api_version="2022-03-01"
arm_token="[ARM_TOKEN]"
tenant_id="[TENANT_ID]"
aad_client_id="[AAD_CLIENT_ID]"
object_ids=("[OBJECT_ID_FOR_USER1]" "[OBJECT_ID_FOR_USER2]" "[OBJECT_ID_FOR_APP1]")

# Convert the object_ids array to a JSON array
object_ids_json=$(printf '%s\n' "${object_ids[@]}") | jq -R . | jq -s .)

# Request URL
url="https://management.azure.com/subscriptions/$subscription_id/resourceGroups/$resource_group/providers/Microsoft.Web/sites/$app_name/config/authSettingsV2?api-version=$api_version"

# JSON payload
json_payload=$(cat <<EOF
{
    "properties": {
        "platform": {
            "enabled": true,
            "runtimeVersion": "~1"
        },
        "globalValidation": {
            "requireAuthentication": true,
            "unauthenticatedClientAction": "AllowAnonymous"
        },
        "identityProviders": {
            "azureActiveDirectory": {
                "enabled": true,
                "registration": {
                    "openIdIssuer": "https://sts.windows.net/$tenant_id/",
                    "clientId": "$aad_client_id"
                },
                "validation": {
                    "jwtClaimChecks": {},
                    "allowedAudiences": [
                        "api://$aad_client_id"
                    ],
                    "defaultAuthorizationPolicy": {
                        "allowedPrincipals": {
                            "identities": $object_ids_json
                        }
                    }
                }
            }
        }
    }
}
EOF
)
```

```

    "facebook": {
        "enabled": false,
        "registration": {},
        "login": {}
    },
    "gitHub": {
        "enabled": false,
        "registration": {},
        "login": {}
    },
    "google": {
        "enabled": false,
        "registration": {},
        "login": {},
        "validation": {}
    },
    "twitter": {
        "enabled": false,
        "registration": {}
    },
    "legacyMicrosoftAccount": {
        "enabled": false,
        "registration": {},
        "login": {},
        "validation": {}
    },
    "apple": {
        "enabled": false,
        "registration": {},
        "login": {}
    }
}
}

EOF
)

# HTTP PUT request
curl -X PUT "$url" \
-H "Content-Type: application/json" \
-H "Authorization: Bearer $arm_token" \
-d "$json_payload"

```

Use Logic Apps with Semantic Kernel as a plugin

Now that you have your Logic App secured and the metadata endpoints enabled, you've finished all the hard parts. You can now import your Logic App as a plugin into Semantic Kernel using the OpenAPI import method.

When you create your plugin, you'll want to provide a custom HTTP client that can handle the authentication for your Logic App. This will allow you to use the plugin in

your AI agents without needing to worry about the authentication.

Below is an example in C# that leverages interactive auth to acquire a token and authenticate the user for the Logic App.

C#

```
string ClientId = "[AAD_CLIENT_ID]";
string TenantId = "[TENANT_ID]";
string Authority = $"https://login.microsoftonline.com/{TenantId}";
string[] Scopes = new string[] { "api://[AAD_CLIENT_ID]/SKLogicApp" };

var app = PublicClientApplicationBuilder.Create(ClientId)
    .WithAuthority(Authority)
    .WithDefaultRedirectUri() // Uses http://localhost for a console
app
    .Build();

AuthenticationResult authResult = null;
try
{
    authResult = await app.AcquireTokenInteractive(Scopes).ExecuteAsync();
}
catch (MsalException ex)
{
    Console.WriteLine("An error occurred acquiring the token: " +
ex.Message);
}

// Add the plugin to the kernel with a custom HTTP client for authentication
kernel.Plugins.Add(await kernel.ImportPluginFromOpenApiAsync(
    pluginName: "[NAME_OF_PLUGIN]",
    uri: new Uri($"https://[{LOGIC_APP_NAME}].azurewebsites.net/swagger.json"),
    executionParameters: new OpenApiFunctionExecutionParameters()
    {
        HttpClient = new HttpClient()
        {
            DefaultRequestHeaders =
            {
                Authorization = new AuthenticationHeaderValue("Bearer",
authResult.AccessToken)
            }
        },
    }
));
```

Next steps

Now that you know how to create a plugin, you can now learn how to use them with your AI agent. Depending on the type of functions you've added to your plugins, there

are different patterns you should follow. For retrieval functions, refer to the [using retrieval functions](#) article. For task automation functions, refer to the [using task automation functions](#) article.

[Learn about using retrieval functions](#)

Using plugins for Retrieval Augmented Generation (RAG)

Article • 06/24/2024

Often, your AI agents must retrieve data from external sources to generate grounded responses. Without this additional context, your AI agents may hallucinate or provide incorrect information. To address this, you can use plugins to retrieve data from external sources.

When considering plugins for Retrieval Augmented Generation (RAG), you should ask yourself two questions:

1. How will you (or your AI agent) "search" for the required data? Do you need [semantic search](#) or [classic search](#)?
2. Do you already know the data the AI agent needs ahead of time ([pre-fetched data](#)), or does the AI agent need to retrieve the data [dynamically](#)?
3. How will you keep your data secure and [prevent oversharing of sensitive information](#)?

Semantic vs classic search

When developing plugins for Retrieval Augmented Generation (RAG), you can use two types of search: semantic search and classic search.

Semantic Search

Semantic search utilizes vector databases to understand and retrieve information based on the meaning and context of the query rather than just matching keywords. This method allows the search engine to grasp the nuances of language, such as synonyms, related concepts, and the overall intent behind a query.

Semantic search excels in environments where user queries are complex, open-ended, or require a deeper understanding of the content. For example, searching for "best smartphones for photography" would yield results that consider the context of photography features in smartphones, rather than just matching the words "best," "smartphones," and "photography."

When providing an LLM with a semantic search function, you typically only need to define a function with a single search query. The LLM will then use this function to

retrieve the necessary information. Below is an example of a semantic search function that uses Azure AI Search to find documents similar to a given query.

C#

```
using System.ComponentModel;
using System.Text.Json.Serialization;
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Models;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Embeddings;

public class InternalDocumentsPlugin
{
    private readonly ITextEmbeddingGenerationService
    _textEmbeddingGenerationService;
    private readonly SearchIndexClient _indexClient;

    public AzureAIsearchPlugin(ITextEmbeddingGenerationService
    textEmbeddingGenerationService, SearchIndexClient indexClient)
    {
        _textEmbeddingGenerationService = textEmbeddingGenerationService;
        _indexClient = indexClient;
    }

    [KernelFunction("Search")]
    [Description("Search for a document similar to the given query.")]
    public async Task<string> SearchAsync(string query)
    {
        // Convert string query to vector
        ReadOnlyMemory<float> embedding = await
        _textEmbeddingGenerationService.GenerateEmbeddingAsync(query);

        // Get client for search operations
        SearchClient searchClient = _indexClient.GetSearchClient("default-
collection");

        // Configure request parameters
        VectorizedQuery vectorQuery = new(embedding);
        vectorQuery.Fields.Add("vector");

        SearchOptions searchOptions = new() { VectorSearch = new() { Queries
= { vectorQuery } } };

        // Perform search request
        Response<SearchResults<IndexSchema>> response = await
        searchClient.SearchAsync<IndexSchema>(searchOptions);

        // Collect search results
        await foreach ( SearchResult<IndexSchema> result in
        response.Value.GetResultsAsync())
        {
```

```

        return result.Document.Chunk; // Return text from first result
    }

    return string.Empty;
}

private sealed class IndexSchema
{
    [JsonPropertyName("chunk")]
    public string Chunk { get; set; }

    [JsonPropertyName("vector")]
    public ReadOnlyMemory<float> Vector { get; set; }
}
}

```

Classic Search

Classic search, also known as attribute-based or criteria-based search, relies on filtering and matching exact terms or values within a dataset. It is particularly effective for database queries, inventory searches, and any situation where filtering by specific attributes is necessary.

For example, if a user wants to find all orders placed by a particular customer ID or retrieve products within a specific price range and category, classic search provides precise and reliable results. Classic search, however, is limited by its inability to understand context or variations in language.

💡 Tip

In most cases, your existing services already support classic search. Before implementing a semantic search, consider whether your existing services can provide the necessary context for your AI agents.

Take for example, a plugin that retrieves customer information from a CRM system using classic search. Here, the AI simply needs to call the `GetCustomerInfoAsync` function with a customer ID to retrieve the necessary information.

C#

```

using System.ComponentModel;
using Microsoft.SemanticKernel;

public class CRMPlugin
{
    private readonly CRMService _crmService;

```

```

public CRMPlugin(CRMSERVICE crmService)
{
    _crmService = crmService;
}

[KernelFunction("GetCustomerInfo")]
[Description("Retrieve customer information based on the given customer
ID.")]
public async Task<Customer> GetCustomerInfoAsync(string customerId)
{
    return await _crmService.GetCustomerInfoAsync(customerId);
}
}

```

Achieving the same search functionality with semantic search would likely be impossible or impractical due to the non-deterministic nature of semantic queries.

When to Use Each

Choosing between semantic and classic search depends on the nature of the query. It is ideal for content-heavy environments like knowledge bases and customer support where users might ask questions or look for products using natural language. Classic search, on the other hand, should be employed when precision and exact matches are important.

In some scenarios, you may need to combine both approaches to provide comprehensive search capabilities. For instance, a chatbot assisting customers in an e-commerce store might use semantic search to understand user queries and classic search to filter products based on specific attributes like price, brand, or availability.

Below is an example of a plugin that combines semantic and classic search to retrieve product information from an e-commerce database.

C#

```

using System.ComponentModel;
using Microsoft.SemanticKernel;

public class ECommercePlugin
{
    [KernelFunction("search_products")]
    [Description("Search for products based on the given query.")]
    public async Task<IEnumerable<Product>> SearchProductsAsync(string
query, ProductCategories category = null, decimal? minPrice = null, decimal?
maxPrice = null)
    {
        // Perform semantic and classic search with the given parameters
    }
}

```

```
    }  
}
```

Dynamic vs pre-fetched data retrieval

When developing plugins for Retrieval Augmented Generation (RAG), you must also consider whether the data retrieval process is static or dynamic. This allows you to optimize the performance of your AI agents by retrieving data only when necessary.

Dynamic data retrieval

In most cases, the user query will determine the data that the AI agent needs to retrieve. For example, a user might ask for the difference between two different products. The AI agent would then need to dynamically retrieve the product information from a database or API to generate a response using [function calling](#). It would be impractical to pre-fetch all possible product information ahead of time and give it to the AI agent.

Below is an example of a back-and-forth chat between a user and an AI agent where dynamic data retrieval is necessary.

[\[+\] Expand table](#)

Role	Message
● User	Can you tell me about the best mattresses?
● Assistant (function call)	<code>Products.Search("mattresses")</code>
● Tool	<code>[{"id": 25323, "name": "Cloud Nine"}, {"id": 63633, "name": "Best Sleep"}]</code>
● Assistant	Sure! We have both Cloud Nine and Best Sleep
● User	What's the difference between them?
● Assistant (function call)	<code>Products.GetDetails(25323)</code> <code>Products.GetDetails(63633)</code>
● Tool	<code>{ "id": 25323, "name": "Cloud Nine", "price": 1000, "material": "Memory foam" }</code>
● Tool	<code>{ "id": 63633, "name": "Best Sleep", "price": 1200, "material": "Latex" }</code>
● Assistant	Cloud Nine is made of memory foam and costs \$1000. Best Sleep is made of latex and costs \$1200.

Pre-fetched data Retrieval

Static data retrieval involves fetching data from external sources and *always* providing it to the AI agent. This is useful when the data is required for every request or when the data is relatively stable and doesn't change frequently.

Take for example, an agent that always answers questions about the local weather. Assuming you have a `WeatherPlugin`, you can pre-fetch weather data from a weather API and provide it in the chat history. This allows the agent to generate responses about the weather without wasting time requesting the data from the API.

```
C#  
  
using System.Text.Json;  
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.ChatCompletion;  
  
IKernelBuilder builder = Kernel.CreateBuilder();  
builder.AddAzureOpenAIChatCompletion(deploymentName, endpoint, apiKey);  
builder.Plugins.AddFromType<WeatherPlugin>();  
Kernel kernel = builder.Build();  
  
// Get the weather  
var weather = await kernel.Plugins.GetFunction("WeatherPlugin",  
"get_weather").InvokeAsync(kernel);  
  
// Initialize the chat history with the weather  
ChatHistory chatHistory = new ChatHistory("The weather is:\n" +  
JsonSerializer.Serialize(weather));  
  
// Simulate a user message  
chatHistory.AddUserMessage("What is the weather like today?");  
  
// Get the answer from the AI agent  
IChatCompletionService chatCompletionService =  
kernel.GetRequiredService<IChatCompletionService>();  
var result = await  
chatCompletionService.GetChatMessageContentAsync(chatHistory);
```

Keeping data secure

When retrieving data from external sources, it is important to ensure that the data is secure and that sensitive information is not exposed. To prevent oversharing of sensitive information, you can use the following strategies:

[+] Expand table

Strategy	Description
Use the user's auth token	Avoid creating service principals used by the AI agent to retrieve information for users. Doing so makes it difficult to verify that a user has access to the retrieved information.
Avoid recreating search services	Before creating a new search service with a vector DB, check if one already exists for the service that has the required data. By reusing existing services, you can avoid duplicating sensitive content, leverage existing access controls, and use existing filtering mechanisms that only return data the user has access to.
Store reference in vector DBs instead of content	Instead of duplicating sensitive content to vector DBs, you can store references to the actual data. For a user to access this information, their auth token must first be used to retrieve the real data.

Next steps

Now that you now how to ground your AI agents with data from external sources, you can now learn how to use AI agents to automate business processes. To learn more, see [using task automation functions](#).

[Learn about task automation functions](#)

Task automation with agents

Article • 06/24/2024

Most AI agents today simply retrieve data and respond to user queries. AI agents, however, can achieve much more by using plugins to automate tasks on behalf of users. This allows users to delegate tasks to AI agents, freeing up time for more important work.

Once AI Agents start performing actions, however, it's important to ensure that they are acting in the best interest of the user. This is why we provide hooks / filters to allow you to control what actions the AI agent can take.

Requiring user consent

When an AI agent is about to perform an action on behalf of a user, it should first ask for the user's consent. This is especially important when the action involves sensitive data or financial transactions.

In Semantic Kernel, you can use the function invocation filter. This filter is always called whenever a function is invoked from an AI agent. To create a filter, you need to implement the `IFunctionInvocationFilter` interface and then add it as a service to the kernel.

Here's an example of a function invocation filter that requires user consent:

```
C#  
  
public class ApprovalFilterExample() : IFunctionInvocationFilter  
{  
    public async Task OnFunctionInvocationAsync(FunctionInvocationContext  
context, Func<FunctionInvocationContext, Task> next)  
    {  
        if (context.Function.PluginName == "DynamicsPlugin" &&  
context.Function.Name == "create_order")  
        {  
            Console.WriteLine("System > The agent wants to create an  
approval, do you want to proceed? (Y/N)");  
            string shouldProceed = Console.ReadLine();  
  
            if (shouldProceed != "Y")  
            {  
                context.Result = new FunctionResult(context.Result, "The  
order creation was not approved by the user");  
                return;  
            }  
        }  
    }  
}
```

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

You can then add the filter as a service to the kernel:

C#

```
IKernelBuilder builder = Kernel.CreateBuilder();
builder.Services.AddSingleton<IFunctionInvocationFilter,
ApprovalFilterExample>();
Kernel kernel = builder.Build();
```

Now, whenever the AI agent tries to create an order using the `DynamicsPlugin`, the user will be prompted to approve the action.

💡 Tip

Whenever a function is cancelled or fails, you should provide the AI agent with a meaningful error message so it can respond appropriately. For example, if we didn't let the AI agent know that the order creation was not approved, it would assume that the order failed due to a technical issue and would try to create the order again.

Next steps

Now that you've learned how to allow agents to automate tasks, you can learn how to allow agents to automatically create plans to address user needs.

[Automate planning with agents](#)

What is a Planner?

Article • 06/24/2024

Once you have multiple plugins, you then need a way for your AI agent to use them together to solve a user's need. This is where planning comes in.

Early on, Semantic Kernel introduced the concept of planners that used prompts to request the AI to choose which functions to invoke. Since Semantic Kernel was introduced, however, OpenAI introduced a native way for the model to invoke or "call" a function: [function calling](#). Other AI models like Gemini, Claude, and Mistral have since adopted function calling as a core capability, making it a cross-model supported feature.

Because of these advancements, Semantic Kernel has evolved to use function calling as the primary way to plan and execute tasks.

ⓘ Important

Function calling is only available in OpenAI models that are 0613 or newer. If you use an older model (e.g., 0314), this functionality will return an error. We recommend using the latest OpenAI models to take advantage of this feature.

How does function calling create a "plan"?

At its simplest, function calling is merely a way for an AI to invoke a function with the right parameters. Take for example a user wants to turn on a light bulb. Assuming the AI has the right plugin, it can call the function to turn on the light.

 Expand table

Role	Message
● User	Please turn on light #1
● Assistant (function call)	<code>Lights.change_state(1, { "isOn": true })</code>
● Tool	<code>{ "id": 1, "name": "Table Lamp", "isOn": true, "brightness": 100, "hex": "FF0000" }</code>
● Assistant	The lamp is now on

But what if the user doesn't know the ID of the light? Or what if the user wants to turn on all the lights? This is where planning comes in. Today's LLM models are capable of

iteratively calling functions to solve a user's need. This is accomplished by creating a feedback loop where the AI can call a function, check the result, and then decide what to do next.

For example, a user may ask the AI to "toggle" a light bulb. The AI would first need to check the state of the light bulb before deciding whether to turn it on or off.

[+] Expand table

Role	Message
>User	Please toggle all the lights
Assistant (function call)	<code>Lights.get_lights()</code>
Tool	<pre>{ "lights": [{ "id": 1, "name": "Table Lamp", "isOn": true, "brightness": 100, "hex": "FF0000" }, { "id": 2, "name": "Ceiling Light", "isOn": false, "brightness": 0, "hex": "FFFFFF" }] }</pre>
Assistant (function call)	<code>Lights.change_state(1, { "isOn": false })</code> <code>Lights.change_state(2, { "isOn": true })</code>
Tool	<pre>{ "id": 1, "name": "Table Lamp", "isOn": false, "brightness": 0, "hex": "FFFFFF" }</pre>
Tool	<pre>{ "id": 2, "name": "Ceiling Light", "isOn": true, "brightness": 100, "hex": "FF0000" }</pre>
Assistant	The lights have been toggled

ⓘ Note

In this example, you also saw parallel function calling. This is where the AI can call multiple functions at the same time. This is a powerful feature that can help the AI solve complex tasks more quickly. It was added to the OpenAI models in 1106.

The automatic planning loop

Supporting function calling without Semantic Kernel is relatively complex. You would need to write a loop that would accomplish the following:

1. Create JSON schemas for each of your functions
2. Provide the LLM with the previous chat history and function schemas

3. Parse the LLM's response to determine if it wants to reply with a message or call a function
4. If the LLM wants to call a function, you would need to parse the function name and parameters from the LLM's response
5. Invoke the function with the right parameters
6. Return the results of the function so that the LLM can determine what it should do next
7. Repeat steps 2-6 until the LLM decides it has completed the task or needs help from the user

In Semantic Kernel, we make it easy to use function calling by automating this loop for you. This allows you to focus on building the plugins needed to solve your user's needs.

ⓘ Note

Understanding how the function calling loop works is essential for building performant and reliable AI agents. For an in-depth look at how the loop works, see the [function calling](#) article.

Using automatic function calling

To use automatic function calling in Semantic Kernel, you need to do the following:

1. Register the plugin with the kernel
2. Create an execution settings object that tells the AI to automatically call functions
3. Invoke the chat completion service with the chat history and the kernel

```
using System.ComponentModel;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

// 1. Create the kernel with the Lights plugin
var builder = Kernel.CreateBuilder().AddAzureOpenAIChatCompletion(modelId,
endpoint, apiKey);
builder.Plugins.AddFromType<LightsPlugin>("Lights");
Kernel kernel = builder.Build();

var chatCompletionService =
kernel.GetRequiredService<IChatCompletionService>();

// 2. Enable automatic function calling
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
}
```

```
};

var history = new ChatHistory();

string? userInput;
do {
    // Collect user input
    Console.Write("User > ");
    userInput = Console.ReadLine();

    // Add user input
    history.AddUserMessage(userInput);

    // 3. Get the response from the AI with automatic function calling
    var result = await chatCompletionService.GetChatMessageContentAsync(
        history,
        executionSettings: openAIPromptExecutionSettings,
        kernel: kernel);

    // Print the results
    Console.WriteLine("Assistant > " + result);

    // Add the message from the agent to the chat history
    history.AddMessage(result.Role, result.Content ?? string.Empty);
} while (userInput is not null)
```

When you use automatic function calling, all of the steps in the automatic planning loop are handled for you and added to the `ChatHistory` object. After the function calling loop is complete, you can inspect the `ChatHistory` object to see all of the function calls made and results provided by Semantic Kernel.

What about the Function Calling Stepwise and Handlebars planners?

The Stepwise and Handlebars planners are still available in Semantic Kernel. However, we recommend using function calling for most tasks as it is more powerful and easier to use. Both the Stepwise and Handlebars planners will be deprecated in a future release of Semantic Kernel.

Before we deprecate these planners, we will provide guidance on how to migrate your existing planners to function calling. If you have any questions about this process, please reach out to us on the [discussions board](#) in the Semantic Kernel GitHub repository.

⊗ Caution

If you are building a new AI agent, we recommend that you *not* use the Stepwise or Handlebars planners. Instead, use function calling as it is more powerful and easier to use.

Next steps

Now that you understand how planners work in Semantic Kernel, you can learn more about how influence your AI agent so that they best plan and execute tasks on behalf of your users.

[Learn about personas](#)

Personas: Giving your Agent a Role

Article • 06/24/2024

Often called a "meta prompt" or "instruction", the persona is a prompt that is used to influence how the agent responds to stimuli. This allows you to influence how your agents plan tasks, generate responses, and interact with users. For example, you can use the persona to explicitly tell the agent to ask for help if it doesn't know what to do, or to be more verbose when it is explaining something.

In Semantic Kernel, we often describe these prompts as "personas" because they're very similar to persona in user experience design. Just like how designers and UX researchers create personas to represent different types of users and the jobs-to-be-done they have, you can create personas to represent different types of agents and the tasks they are responsible for.

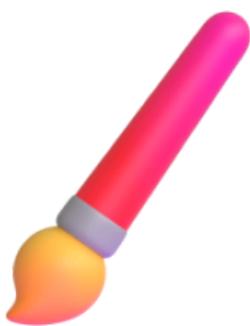
By doing so, you can create agents that are more consistent, more reliable, and more predictable. This is especially important when you're building agents that are responsible for critical tasks or that interact with users in sensitive situations.



Project manager agent



Copywriter agent



Designer agent

The rest of this article will provide tips and best practices when creating personas in Semantic Kernel.

Best practices for creating personas

Creating effective personas involves careful consideration of the agent's intended role, the target audience, and the specific tasks the agent will perform. Here are some best practices to keep in mind:

[] Expand table

Step	Description
1. Define clear objectives	Before creating a persona, clearly define the objectives you want the agent to achieve. This includes understanding the tasks the agent will perform and the type of interactions you expect it to have with users.
2. Reference specific roles or public figures	A well-defined persona should include specific details about the agent's behavior, tone, and style. Avoid vague descriptions by referencing specific roles or public figures so that the LLM can better mimic their behavior.
3. Provide examples and scenarios	To help the agent understand its role better, provide examples and scenarios that illustrate the persona in action. This can include sample conversations, user stories, or use cases that demonstrate how the agent should respond in different situations.
4. Include fallback instructions	Provide instructions on what the agent should do when it encounters an unknown situation or is unable to provide a response. This can include asking for help, providing alternative solutions, or simply acknowledging that it doesn't have an answer.
5. Iterate and Refine	Personas are not static. Continuously test and refine your personas based on user feedback and performance data. This helps in improving the agent's effectiveness over time.

Using system messages to set personas

The most common system messages used within Semantic Kernel are the user, assistant, and tool messages. These messages are used to simulate a conversation between the user and the agent.

A forth message type, the system message, can be used to provide the agent with additional context or instructions that isn't shown to the user. This can be used to set the persona for the agent.

In Semantic Kernel, providing an initial system message for the agent is as simple as adding it in the `ChatHistory` constructor. For example:

```
ChatHistory chatHistory = new("""
    You are a technical support specialist for a software company.
    Your primary task is to assist users with technical issues,
    such as installation problems, software bugs, and feature
    inquiries. Use technical jargon appropriately, but ensure that
    explanations are easy to understand. If a problem is too complex,
    suggest advanced troubleshooting steps or escalate to a higher-level
    support team using the escalate tool.
""")
```

If you need greater control over the persona (for example, you want to change the instructions as the conversation progresses), you can use the `AddSystemMessage` method to add additional system messages to the chat history.

```
chatHistory.AddSystemMessage("Remember to ask for help if you're unsure how
to proceed.")
```

By reminding the LLM of its personas in this way, you can ensure that your agent is consistent, reliable, and predictable in its interactions with users.

Integrations for Semantic Kernel

Article • 06/24/2024

Semantic Kernel provides a wide range of integrations to help you build powerful AI agents. These integrations include AI services, memory connectors. Additionally, Semantic Kernel integrates with other Microsoft services to provide additional functionality via plugins.

Out-of-the-box integrations

With the available AI and memory connectors, developers can easily build AI agents with swappable components. This allows you to experiment with different AI services and memory connectors to find the best combination for your use case.

AI Services

 Expand table

Services	C#	Python	Java	Notes
Text Generation	✓	✓	✓	Example: Text-Davinci-003
Chat Completion	✓	✓	✓	Example: GPT4, Chat-GPT
Text Embeddings (Experimental)	✓	✓	✓	Example: Text-Embeddings-Ada-002
Text to Image (Experimental)	✓	✗	✗	Example: Dall-E
Image to Text (Experimental)	✓	✗	✗	Example: Pix2Struct
Text to Audio (Experimental)	✓	✗	✗	Example: Text-to-speech
Audio to Text (Experimental)	✓	✗	✗	Example: Whisper

Memory Connectors (Experimental)

Vector databases have many use cases across different domains and applications that involve natural language processing (NLP), computer vision (CV), recommendation systems (RS), and other areas that require semantic understanding and matching of data.

One use case for storing information in a vector database is to enable large language models (LLMs) to generate more relevant and coherent text based on an [AI plugin](#).

However, large language models often face challenges such as generating inaccurate or irrelevant information; lacking factual consistency or common sense; repeating or contradicting themselves; being biased or offensive. To overcome these challenges, you can use a vector database to store information about different topics, keywords, facts, opinions, and/or sources related to your desired domain or genre. Then, you can use a large language model and pass information from the vector database with your AI plugin to generate more informative and engaging content that matches your intent and style.

For example, if you want to write a blog post about the latest trends in AI, you can use a vector database to store the latest information about that topic and pass the information along with the ask to a LLM in order to generate a blog post that leverages the latest information.

Available connectors to vector databases

Today, Semantic Kernel offers several connectors to vector databases that you can use to store and retrieve information. These include:

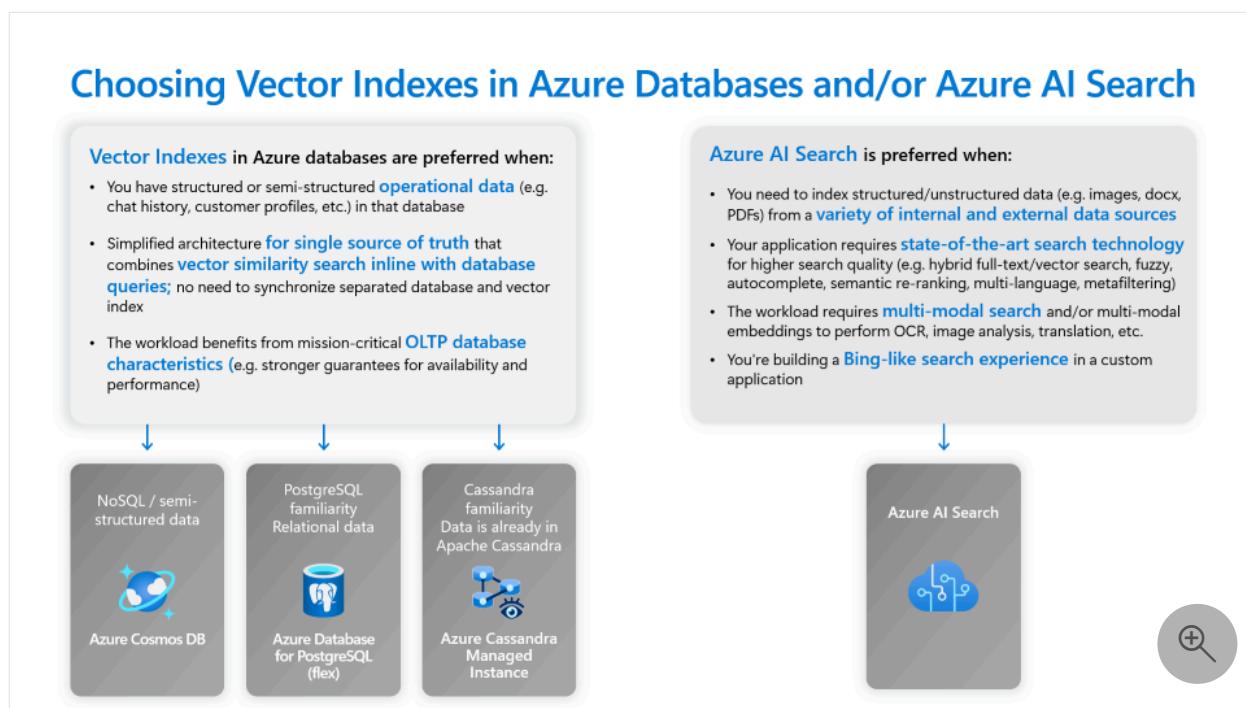
[] [Expand table](#)

Service	C#	Python
Vector Database in Azure Comsmos DB for NoSQL	C# ↗	Python ↗
Vector Database in vCore-based Azure Cosmos DB for MongoDB	C# ↗	Python ↗
Azure AI Search	C# ↗	Python ↗
Azure PostgreSQL Server	C# ↗	
Azure SQL Database	C# ↗	
Chroma	C# ↗	Python ↗
DuckDB	C# ↗	
Milvus	C# ↗	Python ↗
MongoDB Atlas Vector Search	C# ↗	Python ↗
Pinecone	C# ↗	Python ↗
Postgres	C# ↗	Python ↗
Qdrant	C# ↗	

Service	C# ↗	Python ↗
Redis	C# ↗	
Sqlite	C# ↗	
Weaviate	C# ↗	Python ↗

Vector database solutions

- Azure Cosmos DB for NoSQL Integrated Vector Database with DiskANN
- Azure Cosmos DB for MongoDB Integrated Vector Database
- Azure SQL Database
- Azure PostgreSQL Server pgvector Extension
- Azure AI Search
- Open-source vector databases



Additional plugins

If you want to extend the functionality of your AI agent, you can use plugins to integrate with other Microsoft services. Here are some of the plugins that are available for Semantic Kernel:

[+] Expand table

Plugin	C#	Python	Java	Description
Logic Apps	✓	✓	✓	Build workflows within Logic Apps using its available connectors and import them as plugins in Semantic Kernel. Learn more .
Azure Container Apps Dynamic Sessions	✓	✓	✗	With dynamic sessions, you can recreate the Code Interpreter experience from the Assistants API by effortlessly spinning up Python containers where AI agents can execute Python code. Learn more .

Support for Semantic Kernel

Article • 04/18/2024

 Welcome! There are a variety of ways to get supported in the Semantic Kernel (SK) world.

[] Expand table

Your preference	What's available
Read the docs	This learning site is the home of the latest information for developers
Visit the repo	Our open-source GitHub repository is available for perusal and suggestions
Connect with the Semantic Kernel Team	Visit our GitHub Discussions to get supported quickly with our CoC actively enforced
Office Hours	We will be hosting regular office hours; the calendar invites and cadence are located here: Community.MD

More support information

- [Frequently Asked Questions \(FAQs\)](#)
- [Hackathon Materials](#)
- [Code of Conduct](#)

Next step

[Run the samples](#)

Semantic Kernel - .Net V1 Migration Guide

Article • 02/22/2024

ⓘ Note

This document is not final and will get increasingly better!

This guide is intended to help you upgrade from a pre-v1 version of the .NET Semantic Kernel SDK to v1+. The pre-v1 version used as a reference for this document was the `0.26.231009` version which was the last version before the first beta release where the majority of the changes started to happen.

Package Changes

As a result of many packages being redefined, removed and renamed, also considering that we did a good cleanup and namespace simplification many of our old packages needed to be renamed, deprecated and removed. The table below shows the changes in our packages.

All packages that start with `Microsoft.SemanticKernel` were truncated with a `..` prefix for brevity.

 Expand table

Previous Name	V1 Name	Version	Reason
<code>..Connectors.AI.HuggingFace</code>	<code>..Connectors.HuggingFace</code>	preview	
<code>..Connectors.AI.OpenAI</code>	<code>..Connectors.OpenAI</code>	v1	
<code>..Connectors.AI.Oobabooga</code>	<code>MyIA.SemanticKernel.Connectors.AI.Oobabooga</code>	alpha	Community driven connector  Not ready for v1+ yet
<code>..Connectors.Memory.Kusto</code>	<code>..Connectors.Kusto</code>	alpha	
<code>..Connectors.Memory.DuckDB</code>	<code>..Connectors.DuckDB</code>	alpha	
<code>..Connectors.Memory.Pinecone</code>	<code>..Connectors.Pinecone</code>	alpha	
<code>..Connectors.Memory.Redis</code>	<code>..Connectors.Redis</code>	alpha	
<code>..Connectors.Memory.Qdrant</code>	<code>..Connectors.Qdrant</code>	alpha	
--	<code>..Connectors.Postgres</code>	alpha	
<code>..Connectors.Memory.AzureCognitiveSearch</code>	<code>..Connectors.Memory.AzureAISeach</code>	alpha	
<code>..Functions.Semantic</code>	- Removed -		Merged in

Previous Name	V1 Name	Version	Reason
			Core
..Reliability.Basic	- Removed -		Replaced by .NET Dependency Injection
..Reliability.Polly	- Removed -		Replaced by .NET Dependency Injection
..TemplateEngine.Basic	- Removed -		Merged in Core
..Planners.Core	..Planners.OpenAI Planners.Handlebars		preview
--	..Experimental.Agents		alpha
--	..Experimental.Orchestration.Flow	v1	

Reliability Packages - Replaced by .NET Dependency Injection

The Reliability Basic and Polly packages now can be achieved using the .net dependency injection `ConfigureHttpClientDefaults` service collection extension to inject the desired resiliency policies to the `HttpClient` instances.

C#

```
// Before
var retryConfig = new BasicRetryConfig
{
    MaxRetryCount = 3,
    UseExponentialBackoff = true,
};
retryConfig.RetryableStatusCodes.Add(HttpStatusCode.Unauthorized);
var kernel = new KernelBuilder().WithRetryBasic(retryConfig).Build();
```

C#

```
// After
builder.Services.ConfigureHttpClientDefaults(c =>
{
    // Use a standard resiliency policy, augmented to retry on 401 Unauthorized for
    // this example
    c.AddStandardResilienceHandler().Configure(o =>
    {
        o.Retry.ShouldHandle = args =>
ValueTask.FromResult(args.Outcome.Result?.StatusCode is HttpStatusCode.Unauthorized);
    });
});
```

Package Removal and Changes Needed

Ensure that if you use any of the packages below you match the latest version that V1 uses:

[Expand table](#)

Package Name	Version
Microsoft.Extensions.Configuration	8.0.0
Microsoft.Extensions.Configuration.Binder	8.0.0
Microsoft.Extensions.Configuration.EnvironmentVariables	8.0.0
Microsoft.Extensions.Configuration.Json	8.0.0
Microsoft.Extensions.Configuration.UserSecrets	8.0.0
Microsoft.Extensions.DependencyInjection	8.0.0
Microsoft.Extensions.DependencyInjection.Abstractions	8.0.0
Microsoft.Extensions.Http	8.0.0
Microsoft.Extensions.Http.Resilience	8.0.0
Microsoft.Extensions.Logging	8.0.0
Microsoft.Extensions.Logging.Abstractions	8.0.0
Microsoft.Extensions.Logging.Console	8.0.0

Convention Name Changes

Many of our internal naming conventions were changed to better reflect how the AI community names things. As OpenAI started the massive shift and terms like Prompt, Plugins, Models, RAG were taking shape it was clear that we needed to align with those terms to make it easier for the community to understand use the SDK.

[Expand table](#)

Previous Name	V1 Name
Semantic Function	Prompt Function
Native Function	Method Function
Context Variable	Kernel Argument
Request Settings	Prompt Execution Settings
Text Completion	Text Generation
Image Generation	Text to Image

Previous Name	V1 Name
Skill	Plugin

Code Name Changes

Following the convention name changes, many of the code names were also changed to better reflect the new naming conventions. Abbreviations were also removed to make the code more readable.

[Expand table](#)

Previous Name	V1 Name
ContextVariables	KernelArguments
ContextVariables.Set	KernelArguments.Add
IImageGenerationService	ITextToImageService
ITextCompletionService	ITextGenerationService
Kernel.CreateSemanticFunction	Kernel.CreateFunctionFromPrompt
Kernel.ImportFunctions	Kernel.ImportPluginFrom__
Kernel.ImportSemanticFunctionsFromDirectory	Kernel.ImportPluginFromPromptDirectory
Kernel.RunAsync	Kernel.InvokeAsync
NativeFunction	MethodFunction
OpenAIRequestSettings	OpenAIPromptExecutionSettings
RequestSettings	PromptExecutionSettings
SKException	KernelException
SKFunction	KernelFunction
SKFunctionMetadata	KernelFunctionAttribute
SKJsonSchema	KernelJsonSchema
SKParameterMetadata	KernelParameterMetadata
SKPluginCollection	KernelPluginCollection
SKReturnParameterMetadata	KernelReturnParameterMetadata
SemanticFunction	PromptFunction
SKContext	FunctionResult (output)

Namespace Simplifications

The old namespaces before had a deep hierarchy matching 1:1 the directory names in the projects. This is a common practice but did mean that consumers of the Semantic Kernel packages had to add a lot of different `using`'s in their code. We decided to reduce the number of namespaces in the Semantic Kernel packages so the majority of the functionality is in the main `Microsoft.SemanticKernel` namespace. See below for more details.

[Expand table](#)

Previous Name	V1 Name
<code>Microsoft.SemanticKernel.Orchestration</code>	<code>Microsoft.SemanticKernel</code>
<code>Microsoft.SemanticKernel.Connectors.AI.*</code>	<code>Microsoft.SemanticKernel.Connectors.*</code>
<code>Microsoft.SemanticKernel.SemanticFunctions</code>	<code>Microsoft.SemanticKernel</code>
<code>Microsoft.SemanticKernel.Events</code>	<code>Microsoft.SemanticKernel</code>
<code>Microsoft.SemanticKernel.AI.*</code>	<code>Microsoft.SemanticKernel.*</code>
<code>Microsoft.SemanticKernel.Connectors.AI.OpenAI.*</code>	<code>Microsoft.SemanticKernel.Connectors.OpenAI</code>
<code>Microsoft.SemanticKernel.Connectors.AI.HuggingFace.*</code>	<code>Microsoft.SemanticKernel.Connectors.HuggingFace</code>

Kernel

The code to create and use a `Kernel` instance has been simplified. The `IKernel` interface has been eliminated as developers should not need to create their own `Kernel` implementation. The `Kernel` class represents a collection of services and plugins. The current `Kernel` instance is available everywhere which is consistent with the design philosophy behind the Semantic Kernel.

- `IKernel` interface was changed to `Kernel` class.
- `Kernel.ImportFunctions` was removed and replaced by `Kernel.ImportPluginFrom____`, where `____` can be `Functions`, `Object`, `PromptDirectory`, `Type`, `Grp` or `OpenAIAsync`, etc.

C#

```
// Before
var textFunctions = kernel.ImportFunctions(new StaticTextPlugin(), "text");

// After
var textFunctions = kernel.ImportPluginFromObject(new StaticTextPlugin(), "text");
```

- `Kernel.RunAsync` was removed and replaced by `Kernel.InvokeAsync`. Order of parameters shifted, where function is the first.

C#

```
// Before
KernelResult result = kernel.RunAsync(textFunctions["Uppercase"], "Hello World!");
```

```
// After
FunctionResult result = kernel.InvokeAsync(textFunctions["Uppercase"], new() {
    ["input"] = "Hello World!";
});
```

- `Kernel.InvokeAsync` now returns a `FunctionResult` instead of a `KernelResult`.
- `Kernel.InvokeAsync` only targets one function per call as first parameter. Pipelining is not supported, use the [Example 60](#) to achieve a chaining behavior.

 Not supported

C#

```
KernelResult result = await kernel.RunAsync(" Hello World! ",
    textFunctions["TrimStart"],
    textFunctions["TrimEnd"],
    textFunctions["Uppercase"]);
```

 One function per call

C#

```
var trimStartResult = await kernel.InvokeAsync(textFunctions["TrimStart"], new() {
    ["input"] = " Hello World! ";
});
var trimEndResult = await kernel.InvokeAsync(textFunctions["TrimEnd"], new() {
    ["input"] = trimStartResult.GetValue<string>();
});
var finalResult = await kernel.InvokeAsync(textFunctions["Uppercase"], new() {
    ["input"] = trimEndResult.GetValue<string>();
});
```

 Chaining using plugin Kernel injection

C#

```
// Plugin using Kernel injection
public class MyTextPlugin
{
    [KernelFunction]
    public async Task<string> Chain(Kernel kernel, string input)
    {
        var trimStartResult = await kernel.InvokeAsync("textFunctions",
            "TrimStart", new() { ["input"] = input });
        var trimEndResult = await kernel.InvokeAsync("textFunctions", "TrimEnd",
            new() { ["input"] = trimStartResult.GetValue<string>() });
        var finalResult = await kernel.InvokeAsync("textFunctions", "Uppercase",
            new() { ["input"] = trimEndResult.GetValue<string>() });

        return finalResult.GetValue<string>();
    }
}

var plugin = kernel.ImportPluginFromObject(new MyTextPlugin(), "textFunctions");
var finalResult = await kernel.InvokeAsync(plugin["Chain"], new() { ["input"] = "Hello World!" });
```

- `Kernel.InvokeAsync` does not accept string as input anymore, use a `KernelArguments` instance instead. The function now is the first argument and the input argument needs to be provided as a `KernelArguments` instance.

C#

```
// Before
var result = await kernel.RunAsync("I missed the F1 final race", excuseFunction);

// After
var result = await kernel.InvokeAsync(excuseFunction, new() { ["input"] = "I
missed the F1 final race" });
```

- `Kernel.ImportSemanticFunctionsFromDirectory` was removed and replaced by `Kernel.ImportPluginFromPromptDirectory`.
- `Kernel.CreateSemanticFunction` was removed and replaced by `Kernel.CreateFunctionFromPrompt`.
 - Arguments: `OpenAIRequestSettings` is now `OpenAIPromptExecutionSettings`

Context Variables

`ContextVariables` was redefined as `KernelArguments` and is now a dictionary, where the key is the name of the argument and the value is the value of the argument. Methods like `Set` and `Get` were removed and the common dictionary Add or the indexer `[]` to set and get values should be used instead.

C#

```
// Before
var variables = new ContextVariables("Today is: ");
variables.Set("day", DateTimeOffset.Now.ToString("dddd", CultureInfo.CurrentCulture));

// After
var arguments = new KernelArguments() {
    ["input"] = "Today is: ",
    ["day"] = DateTimeOffset.Now.ToString("dddd", CultureInfo.CurrentCulture)
};

// Initialize directly or use the dictionary indexer below
arguments["day"] = DateTimeOffset.Now.ToString("dddd", CultureInfo.CurrentCulture);
```

Kernel Builder

Many changes were made to our `KernelBuilder` to make it more intuitive and easier to use, as well as to make it simpler and more aligned with the .NET builders approach.

- Creating a `KernelBuilder` can now be only created using the `Kernel.CreateBuilder()` method.

This change make it simpler and easier to use the KernelBuilder in any code-base ensuring one main way of using the builder instead of multiple ways that adds complexity and maintenance overhead.

```
C#  
  
// Before  
IKernel kernel = new KernelBuilder().Build();  
  
// After  
var builder = Kernel.CreateBuilder().Build();
```

- `KernelBuilder.With...` was renamed to `KernelBuilder.Add...`
 - `WithOpenAIChatCompletionService` was renamed to `AddOpenAIChatCompletionService`
 - `WithAIService<ITextCompletion>`
- `KernelBuilder.WithLoggerFactory` is not more used, instead use dependency injection approach to add the logger factory.

```
C#  
  
IKernelBuilder builder = Kernel.CreateBuilder();  
builder.Services.AddLogging(c =>  
    c.AddConsole().SetMinimumLevel(LogLevel.Information));
```

- `WithAIService<T>` Dependency Injection

Previously the `KernelBuilder` had a method `WithAIService<T>` that was removed and a new `ServiceCollection Services` property is exposed to allow the developer to add services to the dependency injection container. i.e.:

```
C#  
  
builder.Services.AddSingleton<ITextGenerationService>()
```

Kernel Result

As the Kernel became just a container for the plugins and now executes just one function there was not more need to have a `KernelResult` entity and all function invocations from Kernel now return a `FunctionResult`.

SKContext

After a lot of discussions and feedback internally and from the community, to simplify the API and make it more intuitive, the `SKContext` concept was dilluted in different entities: `KernelArguments` for function inputs and `FunctionResult` for function outputs.

With the important decision to make `Kernel` a required argument of a function calling, the `SKContext` was removed and the `KernelArguments` and `FunctionResult` were introduced.

`KernelArguments` is a dictionary that holds the input arguments for the function invocation that were previously held in the `SKContext.Variables` property.

`FunctionResult` is the output of the `Kernel.InvokeAsync` method and holds the result of the function invocation that was previously held in the `SKContext.Result` property.

New Plugin Abstractions

- **KernelPlugin Entity:** Before V1 there was no concept of a plugin centric entity. This changed in V1 and for any function you add to a Kernel you will get a Plugin that it belongs to.

Plugins Immutability

Plugins are created by default as immutable by our out-of-the-box `DefaultKernelPlugin` implementation, which means that they cannot be modified or changed after creation.

Also attempting to import the plugins that share the same name in the kernel will give you a key violation exception.

The addition of the `KernelPlugin` abstraction allows dynamic implementations that may support mutability and we provided an example on how to implement a mutable plugin in the [Example 69 ↗](#).

Combining multiple plugins into one

Attempting to create a plugin from directory and adding Method functions afterwards for the same plugin will not work unless you use another approach like creating both plugins separately and then combining them into a single plugin iterating over its functions to aggregate into the final plugin using `kernel.ImportPluginFromFunctions("myAggregatePlugin", myAggregatedFunctions)` extension.

Usage of Experimental Attribute Feature.

This features was introduced to mark some functionalities in V1 that we can possibly change or completely remove.

For mode details one the list of current released experimental features [check here ↗](#).

Prompt Configuration Files

Major changes were introduced to the Prompt Configuration files including default and multiple service/model configurations.

Other naming changes to note:

- `completion` was renamed to `execution_settings`
- `input` was renamed to `input_variables`
- `defaultValue` was renamed to `default`
- `parameters` was renamed to `input_variables`
- Each property name in the `execution_settings` once matched to the `service_id` will be used to configure the service/model execution settings. i.e.:

C#

```
// The "service1" execution settings will be used to configure the
OpenAIChatCompletion service
Kernel kernel = Kernel.CreateBuilder()
    .AddOpenAIChatCompletion(serviceId: "service1", modelId: "gpt-4")
```

Before

JSON

```
{
  "schema": 1,
  "description": "Given a text input, continue it with additional text.",
  "type": "completion",
  "completion": {
    "max_tokens": 4000,
    "temperature": 0.3,
    "top_p": 0.5,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0
  },
  "input": {
    "parameters": [
      {
        "name": "input",
        "description": "The text to continue.",
        "defaultValue": ""
      }
    ]
  }
}
```

After

JSON

```
{
  "schema": 1,
  "description": "Given a text input, continue it with additional text.",
  "execution_settings": {
    "default": {
      "max_tokens": 4000,
      "temperature": 0.3,
      "top_p": 0.5,
      "presence_penalty": 0.0,
      "frequency_penalty": 0.0
    },
  }
}
```

```
"service1": {
    "model_id": "gpt-4",
    "max_tokens": 200,
    "temperature": 0.2,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0,
    "stop_sequences": ["Human", "AI"]
},
"service2": {
    "model_id": "gpt-3.5_turbo",
    "max_tokens": 256,
    "temperature": 0.3,
    "top_p": 0.0,
    "presence_penalty": 0.0,
    "frequency_penalty": 0.0,
    "stop_sequences": ["Human", "AI"]
}
},
"input_variables": [
{
    "name": "input",
    "description": "The text to continue.",
    "default": ""
}
]
}
```

Glossary for Semantic Kernel

Article • 06/24/2024

 Hello! We've included a Glossary below with key terminology.

[Expand table](#)

Term/Word	Definition
Agent	An agent is an artificial intelligence that can answer questions and automate processes for users. There's a wide spectrum of agents that can be built, ranging from simple chat bots to fully automated AI assistants. With Semantic Kernel, we provide you with the tools to build increasingly more sophisticated agents that don't require you to be an AI expert.
API	Application Programming Interface. A set of rules and specifications that allow software components to communicate and exchange data.
Autonomous	Agents that can respond to stimuli with minimal human intervention.
Chatbot	A simple back-and-forth chat with a user and AI Agent.
Connectors	Connectors allow you to integrate existing APIs (Application Programming Interface) with LLMs (Large Language Models). For example, a Microsoft Graph connector can be used to automatically send the output of a request in an email, or to build a description of relationships in an organization chart.
Copilot	Agents that work side-by-side with a user to complete a task.
Kernel	Similar to operating system, the kernel is responsible for managing resources that are necessary to run "code" in an AI application. This includes managing the AI models, services, and plugins that are necessary for both native code and AI services to run together. Because the kernel has all the services and plugins necessary to run both native code and AI services, it is used by nearly every component within the Semantic Kernel SDK. This means that if you run any prompt or code in Semantic Kernel, it will always go through a kernel.
LLM	Large Language Models are Artificial Intelligence tools that can summarize, read or generate text in the form of sentences similar to how humans talk and write. LLMs can be incorporate into various products at Microsoft to unearth richer user value.
Memory	Memories are a powerful way to provide broader context for your ask. Historically, we've always called upon memory as a core component for how computers work: think the RAM in your laptop. For with just a CPU that can crunch numbers, the computer isn't that useful unless it knows what numbers you care about. Memories are what make computation relevant to the task at hand.

Term/Word	Definition
Plugins	To generate this plan, the copilot would first need the capabilities necessary to perform these steps. This is where plugins come in. Plugins allow you to give your agent skills via code. For example, you could create a plugin that sends emails, retrieves information from a database, asks for help, or even saves and retrieves memories from previous conversations.
Planners	To use a plugin (and to wire them up with other steps), the copilot would need to first generate a plan. This is where planners come in. Planners are special prompts that allow an agent to generate a plan to complete a task. The simplest planners are just a single prompt that helps the agent use function calling to complete a task.
Prompts	Prompts play a crucial role in communicating and directing the behavior of Large Language Models (LLMs) AI. They serve as inputs or queries that users can provide to elicit specific responses from a model.
Prompt Engineering	Because of the amount of control that exists, prompt engineering is a critical skill for anyone working with LLM AI models. It's also a skill that's in high demand as more organizations adopt LLM AI models to automate tasks and improve productivity. A good prompt engineer can help organizations get the most out of their LLM AI models by designing prompts that produce the desired outputs.
RAG	Retrieval Augmented Generation - a term that refers to the process of retrieving additional data to provide as context to an LLM to use when generating a response (completion) to a user's question (prompt).

More support information

- [Frequently Asked Questions \(FAQs\)](#)
- [Hackathon Materials](#)
- [Code of Conduct](#)

Microsoft.SemanticKernel Namespace

Reference

ⓘ Important

Some information relates to prerelease product that may be substantially modified before it's released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

Classes

[] Expand table

AggregatorPromptTemplateFactory	Provides a IPromptTemplateFactory which aggregates multiple prompt template factories.
AudioContent	Represents audio content.
AutoFunctionInvocationContext	Class with data related to automatic function invocation.
BinaryContent	Provides access to binary content.
CancelKernelEventArgs	Provides an EventArgs for cancelable operations related to Kernel -based operations.
ChatHistoryExtensions	Chat history extensions.
ChatMessageContent	Represents chat message content return from a IChatCompletionService service.
EchoPromptTemplateFactory	Provides an implementation of IPromptTemplateFactory which creates no operation instances of IPromptTemplate .
FileReferenceContent	Content type to support file references.
FromKernelServicesAttribute	Specifies that an argument to a KernelFunction should be supplied from the associated Kernel 's Services rather than from KernelArguments .
FunctionCallContent	Represents a function call requested by LLM.
FunctionCallContentBuilder	A builder class for creating FunctionCallContent objects from incremental function call updates represented by StreamingFunctionCallUpdateContent .
FunctionFilterContext	Base class with data related to function invocation.

FunctionInvocationContext	Class with data related to function invocation.
FunctionInvokedContext	Class with data related to function after invocation.
FunctionInvokedEventArgs	Provides a CancelKernelEventArgs used in events just after a function is invoked.
FunctionInvokingContext	Class with data related to function before invocation.
FunctionInvokingEventArgs	Provides a CancelKernelEventArgs used in events just before a function is invoked.
FunctionResult	Represents the result of a KernelFunction invocation.
FunctionResultContent	Represents the result of a function call.
GoogleAIKernelBuilderExtensions	Extensions for adding GoogleAI generation services to the application.
GoogleAIMemoryBuilderExtensions	Provides extension methods for the MemoryBuilder class to configure GoogleAI connector.
GoogleAIServiceCollectionExtensions	Extensions for adding GoogleAI generation services to the application.
HandlebarsKernelExtensions	Provides Kernel extensions methods for Handlebars functionality.
HttpOperationException	Represents an exception specific to HTTP operations.
HuggingFaceKernelBuilderExtensions	Provides extension methods for the IKernelBuilder class to configure Hugging Face connectors.
HuggingFaceServiceCollectionExtensions	Provides extension methods for the IServiceCollection interface to configure Hugging Face connectors.
ImageContent	Represents image content.
InputVariable	Represents an input variable for prompt functions.
Kernel	Provides state for use throughout a Semantic Kernel workload.
KernelArguments	Provides a collection of arguments for operations such as KernelFunction 's <code>InvokeAsync</code> and IPromptTemplate 's <code>RenderAsync</code> .
KernelContent	Base class for all AI non-streaming results

KernelEventArgs	Provides an EventArgs for operations related to Kernel -based operations.
KernelException	Represents the base exception from which all Semantic Kernel exceptions derive.
KernelExtensions	Provides extension methods for interacting with Kernel and related types.
KernelFunction	Represents a function that can be invoked as part of a Semantic Kernel workload.
KernelFunction Attribute	Specifies that a method on a class imported as a plugin should be included as a KernelFunction in the resulting KernelPlugin .
KernelFunction CanceledException	Provides an OperationCanceledException -derived exception type that's thrown from a KernelFunction invocation when a Kernel function filter (e.g. FunctionInvocationFilters) requests cancellation.
KernelFunctionFactory	Provides factory methods for creating commonly-used implementations of KernelFunction , such as those backed by a prompt to be submitted to an LLM or those backed by a .NET method.
KernelFunctionFrom MethodOptions	Optional options that can be provided when creating a KernelFunction from a method.
KernelFunction Markdown	Factory methods for creating instances.
KernelFunction Metadata	Provides read-only metadata for a KernelFunction .
KernelFunctionYaml	Factory methods for creating instances.
KernelJsonSchema	Represents JSON Schema for describing types used in KernelFunctions .
KernelJsonSchema. JsonConverter	Converter for reading/writing the schema.
KernelParameter Metadata	Provides read-only metadata for a KernelFunction parameter.
KernelPlugin	Represents a plugin that may be registered with a Kernel .
KernelPlugin Collection	Provides a collection of KernelPlugins .
KernelPlugin Extensions	Provides extension methods for working with KernelPlugins and collections of them.
KernelPluginFactory	Provides static factory methods for creating commonly-used plugin implementations.
KernelPrompt	Provides an implementation of IPromptTemplateFactory for the

TemplateFactory	SemanticKernelTemplateFormat template format.
KernelReturnParameterMetadata	Provides read-only metadata for a KernelFunction 's return parameter.
MarkdownKernelExtensions	Class for extensions methods to define functions using prompt markdown format.
MistralAIKernelBuilderExtensions	Provides extension methods for the IKernelBuilder class to configure Mistral connectors.
MistralAIServiceCollectionExtensions	Provides extension methods for the IServiceCollection interface to configure Mistral connectors.
OnnxKernelBuilderExtensions	Provides extension methods for the IKernelBuilder class to configure ONNX connectors.
OnnxServiceCollectionExtensions	Provides extension methods for the IServiceCollection interface to configure ONNX connectors.
OpenAIServiceCollectionExtensions	Provides extension methods for IServiceCollection and related classes to configure OpenAI and Azure OpenAI connectors.
OutputVariable	Represents an output variable returned from a prompt function.
PromptExecutionSettings	Provides execution settings for an AI request.
PromptFilterContext	Base class with data related to prompt rendering.
PromptRenderContext	Class with data related to prompt rendering.
PromptRenderedContext	Class with data related to prompt after rendering.
PromptRenderedEventArgs	Provides a CancelKernelEventArgs used in events raised just after a prompt has been rendered.
PromptRenderingContext	Class with data related to prompt before rendering.
PromptRenderingEventArgs	Provides a KernelEventArgs used in events raised just before a prompt is rendered.
PromptTemplateConfig	Provides the configuration information necessary to create a prompt template.
PromptTemplateFactoryExtensions	Provides extension methods for operating on IPromptTemplateFactory instances.
PromptYamlKernelExtensions	Class for extensions methods to define functions using prompt YAML format.

PromptyKernelExtensions	Provides extension methods for creating KernelFunctions from the Prompty template format.
RestApiOperationResponse	The REST API operation response.
RestApiOperationResponseConverter	Converts a object of RestApiOperationResponse type to string type.
StreamingChatMessageContent	Abstraction of chat message content chunks when using streaming from IChatCompletionService interface.
StreamingFunctionCallUpdateContent	Represents a function streaming call requested by LLM.
StreamingKernelContent	Represents a single update to a streaming content.
StreamingMethodContent	Represents a manufactured streaming content from a single function result.
StreamingTextContent	Abstraction of text content chunks when using streaming from ITextGenerationService interface.
TextContent	Represents text content return from a ITextGenerationService service.
VertexAIKernelBuilderExtensions	Extensions for adding VertexAI generation services to the application.
VertexAIMemoryBuilderExtensions	Provides extension methods for the MemoryBuilder class to configure VertexAI connector.
VertexAIServiceCollectionExtensions	Extensions for adding VertexAI generation services to the application.

Interfaces

[\[\]](#) Expand table

IAIServiceSelector	Represents a selector which will return a tuple containing instances of IAIService and PromptExecutionSettings from the specified provider based on the model settings.
IAutoFunctionInvocationFilter	Interface for filtering actions during automatic function invocation.
IFunctionInvocationFilter	Interface for filtering actions during function invocation.

IKernelBuilder	Provides a builder for constructing instances of Kernel .
IKernelBuilder Plugins	Provides a builder for adding plugins as singletons to a service collection.
IPromptRenderFilter	Interface for filtering actions during prompt rendering.
IPromptTemplate	Represents a prompt template that can be rendered to a string.
IPromptTemplate Factory	Represents a factory for prompt templates for one or more prompt template formats.
 IReadOnlyKernel PluginCollection	Provides a read-only collection of KernelPlugins .

Kernel Class

Reference

The Kernel class is the main entry point for the Semantic Kernel. It provides the ability to run semantic/native functions, and manage plugins, memory, and AI services.

Initialize a new instance of the Kernel class.

Inheritance [KernelFilterExtension](#) → Kernel
[KernelFunctionExtension](#) → Kernel
[KernelServicesExtension](#) → Kernel
[KernelReliabilityExtension](#) → Kernel

Constructor

Python

```
Kernel(plugins: KernelPlugin | dict[str,  
semantic_kernel.functions.kernel_plugin.KernelPlugin] |  
list[semantic_kernel.functions.kernel_plugin.KernelPlugin] | None = None,  
services: AI_SERVICE_CLIENT_TYPE | list[AI_SERVICE_CLIENT_TYPE] | dict[str,  
AI_SERVICE_CLIENT_TYPE] | None = None, ai_service_selector:  
AIServiceSelector | None = None, *, retry_mechanism: RetryMechanismBase =  
None, function_invocation_filters: list[tuple[int,  
collections.abc.Callable[[FILTER_CONTEXT_TYPE,  
collections.abc.Callable[[FILTER_CONTEXT_TYPE], None\]\], None\\]\\]\\] = None,  
prompt\\_rendering\\_filters: list\\[tuple\\[int,  
collections.abc.Callable\\[\\[FILTER\\_CONTEXT\\_TYPE,  
collections.abc.Callable\\[\\[FILTER\\_CONTEXT\\_TYPE\\], None\\\]\\\], None\\\\]\\\\]\\\\] = None,  
auto\\\\_function\\\\_invocation\\\\_filters: list\\\\[tuple\\\\[int,  
collections.abc.Callable\\\\[\\\\[FILTER\\\\_CONTEXT\\\\_TYPE,  
collections.abc.Callable\\\\[\\\\[FILTER\\\\_CONTEXT\\\\_TYPE\\\\], None\\\\\]\\\\\], None\\\\\\]\\\\\\]\\\\\\] = None\\\\\\)
```

Parameters

[] [Expand table](#)

Name	Description
plugins	<xref:KernelPlugin dict> [str ,<xref: KernelPlugin>]<xref:> list>[<xref:KernelPlugin>]<xref:> None The plugins to be used by the kernel, will be rewritten to a dict with plugin name as key

Name	Description
	default value: None
dict[str Required*	str (<xref:services> (AIServiceClientBase list[AIServiceClientBase]) The services to be used by the kernel, will be rewritten to a dict with service_id as key
None Required*	AIServiceClientBase] The services to be used by the kernel, will be rewritten to a dict with service_id as key
ai_service_selector	<xref:<xref:semantic_kernel.kernel.AIServiceSelector None>> The AI service selector to be used by the kernel, default is based on order of execution settings. default value: None
**kwargs Required*	Any Additional fields to be passed to the Kernel model, these are limited to retry_mechanism and function_invoking_handlers and function_invoked_handlers, the best way to add function_invoking_handlers and function_invoked_handlers is to use the add_function_invoking_handler and add_function_invoked_handler methods.
services	default value: None

Keyword-Only Parameters

[+] [Expand table](#)

Name	Description
retry_mechanism Required*	
function_invocation_filters Required*	
prompt_rendering_filters Required*	
auto_function_invocation_filters Required*	

Methods

invoke	<p>Execute one or more functions.</p> <p>When multiple functions are passed the FunctionResult of each is put into a list.</p> <pre>:param :param if this is none: :param function_name and plugin_name are used and cannot be None.: :param arguments: The arguments to pass to the function(s), optional :type arguments: <xref:semantic_kernel.kernel.KernelArguments> :param function_name: The name of the function to execute :type function_name: <xref:semantic_kernel.kernel.str None> :param plugin_name: The name of the plugin to execute :type plugin_name: <xref:semantic_kernel.kernel.str None> :param metadata: The metadata to pass to the function(s) :type metadata: <xref:semantic_kernel.kernel.dict[str, Any]> :param kwargs: arguments that can be used instead of supplying KernelArguments :type kwargs: <xref:semantic_kernel.kernel.dict[str, Any]></pre>
invoke_prompt	Invoke a function from the provided prompt
invoke_prompt_stream	Invoke a function from the provided prompt and stream the results
invoke_stream	<p>Execute one or more stream functions.</p> <p>This will execute the functions in the order they are provided, if a list of functions is provided. When multiple functions are provided only the last one is streamed, the rest is executed as a pipeline.</p> <pre>:param :param if this is none: :param function_name and plugin_name are used and cannot be None.: :param arguments: The arguments to pass to the function(s), optional :type arguments: <xref:semantic_kernel.kernel.KernelArguments> :param function_name: The name of the function to execute :type function_name: <xref:semantic_kernel.kernel.str None> :param plugin_name: The name of the plugin to execute :type plugin_name: <xref:semantic_kernel.kernel.str None> :param metadata: The metadata to pass to the function(s) :type metadata: <xref:semantic_kernel.kernel.dict[str, Any]> :param return_function_results: If True, the function results are yielded as a list[FunctionResult] :type return_function_results: bool :param in addition to the streaming content: :param otherwise only the streaming content is yielded.: :param kwargs: arguments that can be used instead of supplying KernelArguments :type kwargs: <xref:semantic_kernel.kernel.dict[str, Any]></pre>

invoke

Execute one or more functions.

When multiple functions are passed the FunctionResult of each is put into a list.

:param :param if this is none: :param function_name and plugin_name are used and cannot be None.: :param arguments: The arguments to pass to the function(s), optional :type arguments: <xref:semantic_kernel.kernel.KernelArguments> :param function_name: The name of the function to execute :type function_name: <xref:semantic_kernel.kernel.str | None> :param plugin_name: The name of the plugin to execute :type plugin_name: <xref:semantic_kernel.kernel.str | None> :param metadata: The metadata to pass to the function(s) :type metadata: <xref:semantic_kernel.kernel.dict[str, Any]> :param kwargs: arguments that can be used instead of supplying KernelArguments :type kwargs: <xref:semantic_kernel.kernel.dict[str, Any]>

Python

```
async invoke(function: KernelFunction | None = None, arguments: KernelArguments | None = None, function_name: str | None = None, plugin_name: str | None = None, metadata: dict[str, Any] = {}, **kwargs: Any) -> FunctionResult | None
```

Parameters

[] Expand table

Name	Description
function	default value: None
arguments	default value: None
function_name	default value: None
plugin_name	default value: None
metadata	default value: {}

Returns

[] Expand table

Type	Description
FunctionResult list[FunctionResult] None	The result of the function(s)

invoke_prompt

Invoke a function from the provided prompt

Python

```
async invoke_prompt(function_name: str, plugin_name: str, prompt: str,
arguments: KernelArguments | None = None, template_format:
Literal['semantic-kernel', 'handlebars', 'jinja2'] = 'semantic-kernel',
**kwargs: Any) -> FunctionResult | None
```

Parameters

[] Expand table

Name	Description
function_name Required*	str The name of the function
plugin_name Required*	str The name of the plugin
prompt Required*	str The prompt to use
arguments	<xref:<xref:semantic_kernel.kernel.KernelArguments None>> The arguments to pass to the function(s), optional default value: None
template_format	<xref:<xref:semantic_kernel.kernel.str None>> The format of the prompt template default value: semantic-kernel
kwargs Required*	dict[str,<xref: Any>] arguments that can be used instead of supplying KernelArguments

Returns

[] Expand table

Type	Description
FunctionResult list[FunctionResult] None	The result of the function(s)

invoke_prompt_stream

Invoke a function from the provided prompt and stream the results

Python

```
async invoke_prompt_stream(function_name: str, plugin_name: str, prompt: str, arguments: KernelArguments | None = None, template_format: Literal['semantic-kernel', 'handlebars', 'jinja2'] = 'semantic-kernel', return_function_results: bool | None = False, **kwargs: Any) -> AsyncIterable[list[semantic_kernel.contents.streaming_content_mixin.StreamingContentMixin] | FunctionResult | list[semantic_kernel.functions.function_result.FunctionResult]]
```

Parameters

[] Expand table

Name	Description
function_name Required*	str The name of the function
plugin_name Required*	str The name of the plugin
prompt Required*	str The prompt to use
arguments	<xref:<xref:semantic_kernel.kernel.KernelArguments None>> The arguments to pass to the function(s), optional default value: None
template_format	<xref:<xref:semantic_kernel.kernel.str None>> The format of the prompt template default value: semantic-kernel
kwargs Required*	dict[str,<xref: Any>] arguments that can be used instead of supplying KernelArguments
return_function_results	default value: False

Returns

[+] Expand table

Type	Description
<code>Asynclitable[StreamingContentMixin]</code>	The content of the stream of the last function provided.

invoke_stream

Execute one or more stream functions.

This will execute the functions in the order they are provided, if a list of functions is provided. When multiple functions are provided only the last one is streamed, the rest is executed as a pipeline.

:param :param if this is none: :param function_name and plugin_name are used and cannot be None.: :param arguments: The arguments to pass to the function(s), optional :type arguments: <xref:semantic_kernel.kernel.KernelArguments> :param function_name: The name of the function to execute :type function_name: <xref:semantic_kernel.kernel.str | None> :param plugin_name: The name of the plugin to execute :type plugin_name: <xref:semantic_kernel.kernel.str | None> :param metadata: The metadata to pass to the function(s) :type metadata: <xref:semantic_kernel.kernel.dict[str, Any]> :param return_function_results: If True, the function results are yielded as a list[FunctionResult] :type return_function_results: bool :param in addition to the streaming content: :param otherwise only the streaming content is yielded.: :param kwargs: arguments that can be used instead of supplying KernelArguments :type kwargs: <xref:semantic_kernel.kernel.dict[str, Any]>

Python

```
async invoke_stream(function: KernelFunction | None = None, arguments: KernelArguments | None = None, function_name: str | None = None, plugin_name: str | None = None, metadata: dict[str, Any] = {}, return_function_results: bool = False, **kwargs: Any) -> AsyncGenerator[list['StreamingContentMixin'] | FunctionResult | list[semantic_kernel.functions.function_result.FunctionResult], Any]
```

Parameters

[+] Expand table

Name	Description
<code>function</code>	default value: None
<code>arguments</code>	default value: None
<code>function_name</code>	default value: None
<code>plugin_name</code>	default value: None
<code>metadata</code>	default value: {}
<code>return_function_results</code>	default value: False

Attributes

`model_computed_fields`

A dictionary of computed field names and their corresponding `ComputedFieldInfo` objects.

Python

```
model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}
```

`model_config`

Configuration for the model, should be a dictionary conforming to [`ConfigDict`] [`pydantic.config.ConfigDict`].

Python

```
model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True,
'populate_by_name': True, 'validate_assignment': True}
```

`model_fields`

Metadata about the fields defined on the model, mapping of field names to [`FieldInfo`][`pydantic.fields.FieldInfo`].

This replaces `Model.fields` from Pydantic V1.

Python

```
model_fields: ClassVar[dict[str, FieldInfo]] = {'ai_service_selector':  
    FieldInfo(annotation=AIServiceSelector, required=False,  
    default_factory=AIServiceSelector), 'auto_function_invocation_filters':  
    FieldInfo(annotation=list[tuple[int, Callable[list, NoneType]]],  
    required=False, default_factory=list), 'function_invocation_filters':  
    FieldInfo(annotation=list[tuple[int, Callable[list, NoneType]]],  
    required=False, default_factory=list), 'plugins':  
    FieldInfo(annotation=dict[str, KernelPlugin], required=False,  
    default_factory=dict), 'prompt_rendering_filters':  
    FieldInfo(annotation=list[tuple[int, Callable[list, NoneType]]],  
    required=False, default_factory=list), 'retry_mechanism':  
    FieldInfo(annotation=RetryMechanismBase, required=False,  
    default_factory=PassThroughWithoutRetry), 'services':  
    FieldInfo(annotation=dict[str, AIServiceClientBase], required=False,  
    default_factory=dict)}
```

plugins

The plugins to be used by the kernel

Python

```
plugins: dict[str, KernelPlugin]
```

services

The services to be used by the kernel

Python

```
services: dict[str, AIServiceClientBase]
```

ai_service_selector

The AI service selector to be used by the kernel

Python

```
ai_service_selector: AIServiceSelector
```

retry_mechanism

The retry mechanism to be used by the kernel

Python

```
retry_mechanism: RetryMechanismBase
```

auto_function_invocation_filters

Python

```
auto_function_invocation_filters: list[tuple[int, CALLABLE_FILTER_TYPE]]
```

function_invocation_filters

Python

```
function_invocation_filters: list[tuple[int, CALLABLE_FILTER_TYPE]]
```

prompt_rendering_filters

Python

```
prompt_rendering_filters: list[tuple[int, CALLABLE_FILTER_TYPE]]
```