

Introduction to Semantic Kernel

Article • 06/24/2024

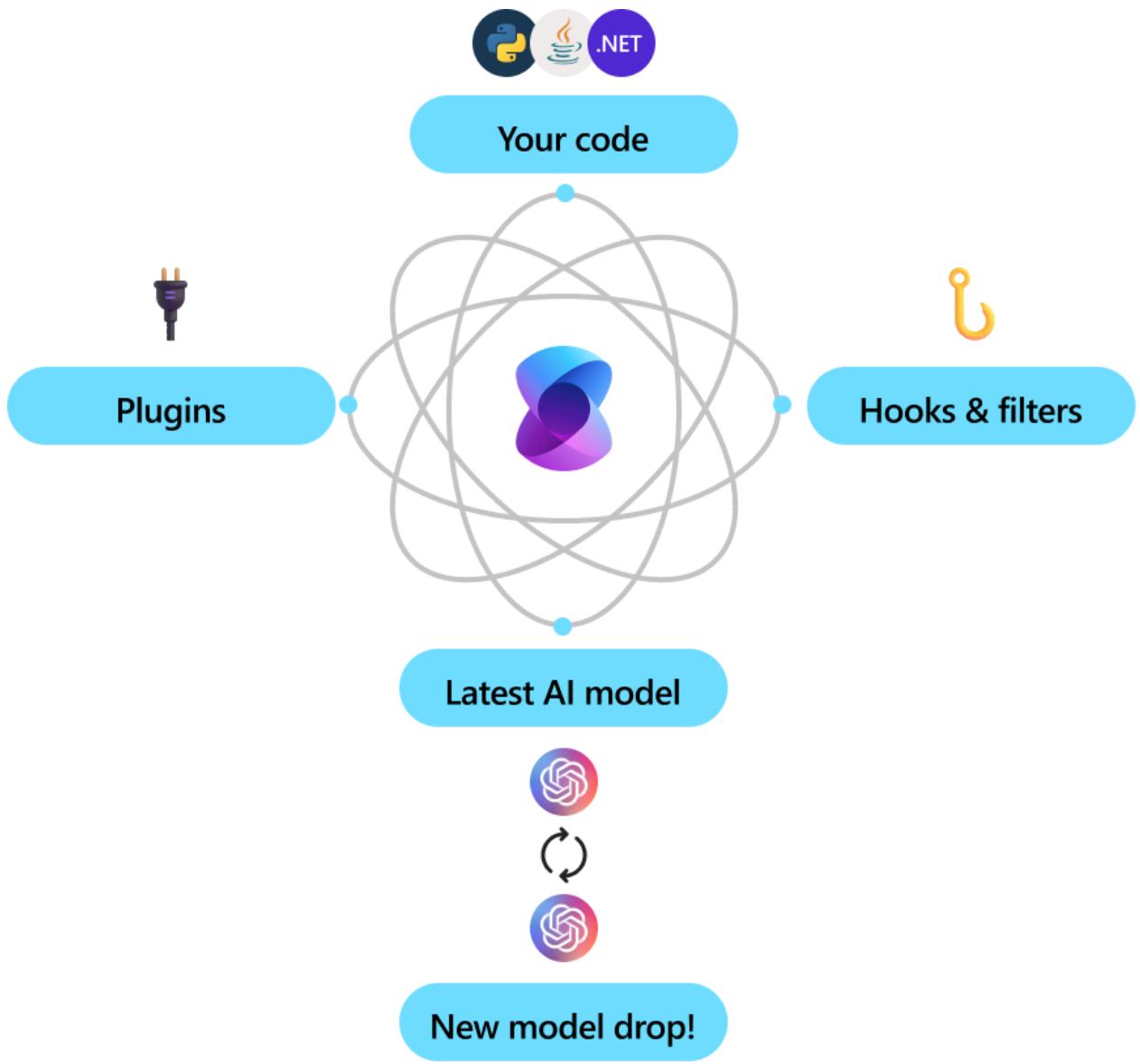
Semantic Kernel is a lightweight, open-source development kit that lets you easily build AI agents and integrate the latest AI models into your C#, Python, or Java codebase. It serves as an efficient middleware that enables rapid delivery of enterprise-grade solutions.

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.

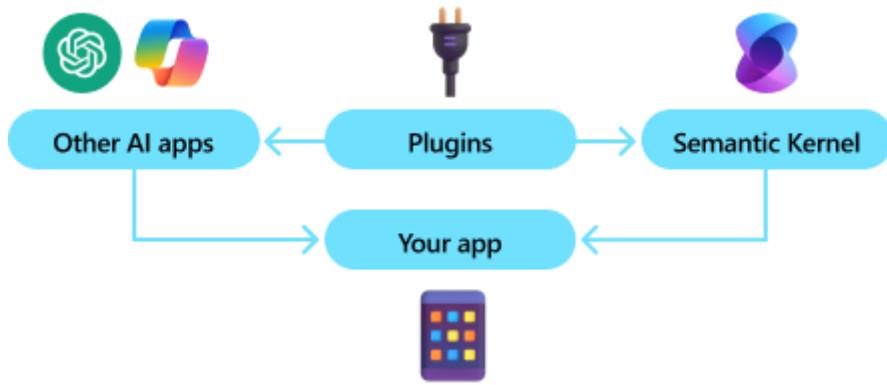


Automating business processes

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.

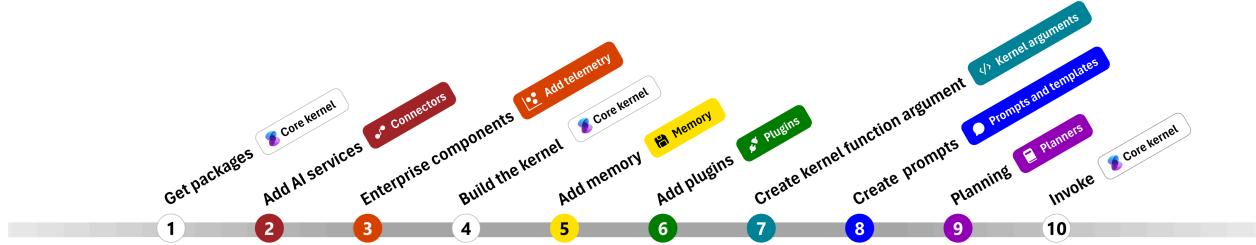
[\[\] Expand table](#)

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 • 10/03/2024

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

Supported Semantic Kernel languages

Article • 09/09/2024

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
```

On [PyPI](#) under `Provides-Extra` the additional extras you can install are also listed and when used that will install the packages needed for using SK with that specific connector or service, you can install those with the square bracket syntax for instance:

Bash

```
pip install semantic-kernel[azure]
```

This will install Semantic Kernel, as well as specific tested versions of: `azure-ai-inference`, `azure-search-documents`, `azure-core`, `azure-identity`, `azure-cosmos` and `msgraph-sdk` (and any dependencies of those packages). Similarly the extra `hugging_face` will install `transformers` and `sentence-transformers`.

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

Services	C#	Python	Java	Notes
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 Modalities

[\[+\] 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 Connectors

[\[+\] 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 will be replaced by Vector Store connectors. These will provide more functionality via an updated abstraction layer.

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

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

Vector Store Connectors (Experimental)

ⓘ **Important**

All of the existing Vector Store 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.

For the list of out of the box vector store connectors and the language support for each, refer to [out of the box connectors](#).

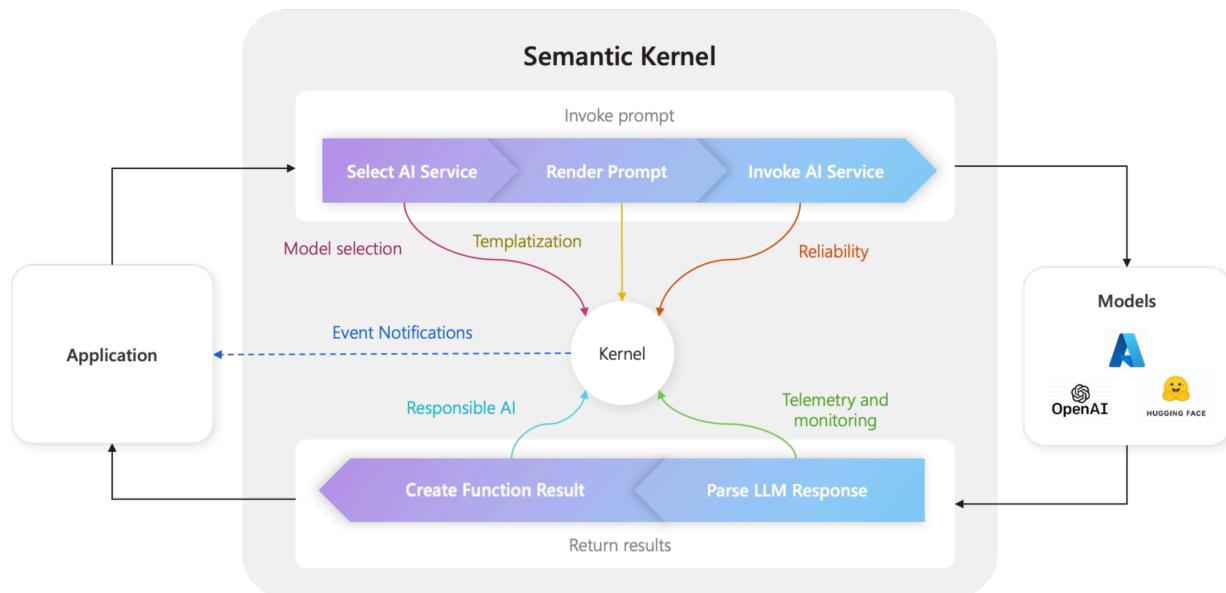
Understanding the kernel

Article • 07/25/2024

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 • 09/11/2024

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 • 09/17/2024

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.

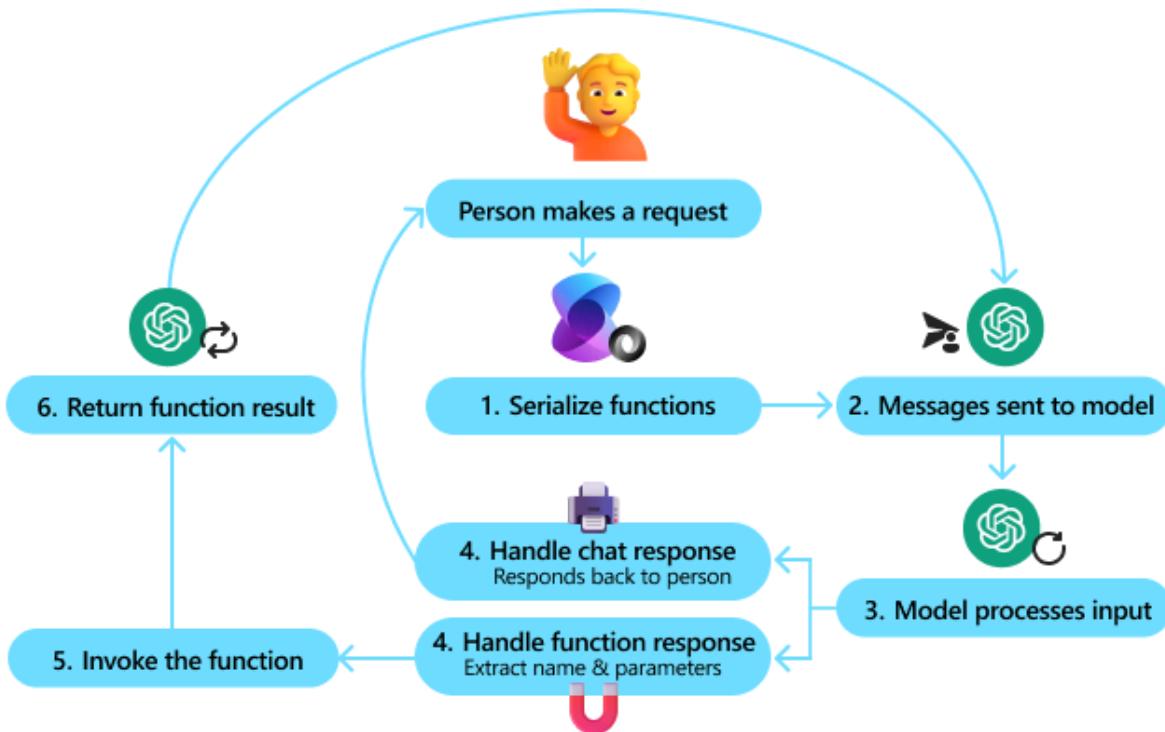
How function calling works

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()  
{  
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()  
};  
  
ChatResponse response = await chatCompletion.GetChatMessageContentAsync(  
    chatHistory,  
    executionSettings: openAIPromptExecutionSettings,  
    kernel: kernel)
```

① Note

This example uses the `FunctionChoiceBehavior.Auto()` behavior, one of the few available ones. For more information about other function choice behaviors, check out the [function choice behaviors article](#).

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        ]\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.

ⓘ Note

Semantic Kernel automatically invokes functions by default. However, if you prefer to manage function invocation manually, you can enable manual function invocation mode. For more details on how to do this, please refer to the [function invocation article](#).

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 proceed to learn how to configure various aspects of function calling that better correspond to your specific scenarios by referring to the [function choice behavior article](#)

[Function Choice Behavior](#)

Function Choice Behaviors

Article • 09/18/2024

Function choice behaviors are bits of configuration that allows a developer to configure:

1. Which functions are advertised to AI models.
2. How the models should choose them for invocation.
3. How Semantic Kernel might invoke those functions.

As of today, the function choice behaviors are represented by three static methods of the `FunctionChoiceBehavior` class:

- **Auto**: Allows the AI model to decide to choose from zero or more of the provided function(s) for invocation.
- **Required**: Forces the AI model to choose provided function(s).
- **None**: Instructs the AI model not to choose any function(s).

⚠ Warning

The function-calling capability is experimental and subject to change. It is expected to reach general availability (GA) by mid-November 2024. Please refer to the [migration guide](#) to migrate your code to the latest function-calling capabilities.

ⓘ Note

The function-calling capabilities is only supported by a few AI connectors so far, see the [Supported AI Connectors](#) section below for more details.

Function Advertising

Function advertising is the process of providing functions to AI models for further calling and invocation. All three function choice behaviors accept a list of functions to advertise as a `functions` parameter. By default, it is null, which means all functions from plugins registered on the Kernel are provided to the AI model.

C#

```
using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
```

```
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

// All functions from the DateTimeUtils and WeatherForecastUtils plugins
// will be sent to AI model together with the prompt.
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };

await kernel.InvokePromptAsync("Given the current time of day and weather,
what is the likely color of the sky in Boston?", new(settings));
```

If a list of functions is provided, only those functions are sent to the AI model:

```
C#

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

KernelFunction getWeatherForCity =
kernel.Plugins.GetFunction("WeatherForecastUtils", "GetWeatherForCity");
KernelFunction getCurrentTime = kernel.Plugins.GetFunction("DateTimeUtils",
"GetCurrentUtcDateTime");

// Only the specified getWeatherForCity and getCurrentTime functions will be
// sent to AI model alongside the prompt.
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto(functions: [getWeatherForCity, getCurrentTime]) };

await kernel.InvokePromptAsync("Given the current time of day and weather,
what is the likely color of the sky in Boston?", new(settings));
```

An empty list of functions means no functions are provided to the AI model, which is equivalent to disabling function calling.

```
C#

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();
```

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

// Disables function calling. Equivalent to var settings = new() {
FunctionChoiceBehavior = null } or var settings = new() { };
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto(functions: [] ) };

await kernel.InvokePromptAsync("Given the current time of day and weather,
what is the likely color of the sky in Boston?", new(settings));
```

Using Auto Function Choice Behavior

The `Auto` function choice behavior instructs the AI model to decide to choose from zero or more of the provided function(s) for invocation.

In this example, all the functions from the `DateTimeUtils` and `WeatherForecastUtils` plugins will be provided to the AI model alongside the prompt. The model will first choose `GetCurrentTime` function for invocation to obtain the current date and time, as this information is needed as input for the `GetWeatherForCity` function. Next, it will choose `GetWeatherForCity` function for invocation to get the weather forecast for the city of Boston using the obtained date and time. With this information, the model will be able to determine the likely color of the sky in Boston.

C#

```
using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

// All functions from the DateTimeUtils and WeatherForecastUtils plugins
// will be provided to AI model alongside the prompt.
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };

await kernel.InvokePromptAsync("Given the current time of day and weather,
what is the likely color of the sky in Boston?", new(settings));
```

The same example can be easily modeled in a YAML prompt template configuration:

C#

```

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

string promptTemplateConfig = """
    template_format: semantic-kernel
    template: Given the current time of day and weather, what is the likely
color of the sky in Boston?
    execution_settings:
        default:
            function_choice_behavior:
                type: auto
    """;

KernelFunction promptFunction =
KernelFunctionYaml.FromPromptYaml(promptTemplateConfig);

Console.WriteLine(await kernel.InvokeAsync(promptFunction));

```

Using Required Function Choice Behavior

The `Required` behavior forces the model to choose the provided function(s) for invocation. This is useful for scenarios when the AI model must obtain required information from the specified functions rather than from its own knowledge.

Note

The behavior advertises functions in the first request to the AI model only and stops sending them in subsequent requests to prevent an infinite loop where the model keeps choosing the same functions for invocation repeatedly.

Here, we specify that the AI model must choose the `GetWeatherForCity` function for invocation to obtain the weather forecast for the city of Boston, rather than guessing it based on its own knowledge. The model will first choose the `GetWeatherForCity` function for invocation to retrieve the weather forecast. With this information, the model can then determine the likely color of the sky in Boston using the response from the call to `GetWeatherForCity`.

```

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();

Kernel kernel = builder.Build();

KernelFunction getWeatherForCity =
kernel.Plugins.GetFunction("WeatherForecastUtils", "GetWeatherForCity");

PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Required(functions: [getWeatherFunction]) };

await kernel.InvokePromptAsync("Given that it is now the 10th of September
2024, 11:29 AM, what is the likely color of the sky in Boston?",
new(settings));

```

An identical example in a YAML template configuration:

C#

```

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();

Kernel kernel = builder.Build();

string promptTemplateConfig = """
template_format: semantic-kernel
template: Given that it is now the 10th of September 2024, 11:29 AM,
what is the likely color of the sky in Boston?
execution_settings:
  default:
    function_choice_behavior:
      type: auto
      functions:
        - WeatherForecastUtils.GetWeatherForCity
""";

KernelFunction promptFunction =
KernelFunctionYaml.FromPromptYaml(promptTemplateConfig);

Console.WriteLine(await kernel.InvokeAsync(promptFunction));

```

Alternatively, all functions registered in the kernel can be provided to the AI model as required. However, only the ones chosen by the AI model as a result of the first request

will be invoked by the Semantic Kernel. The functions will not be sent to the AI model in subsequent requests to prevent an infinite loop, as mentioned above.

C#

```
using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();

Kernel kernel = builder.Build();

PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Required() };

await kernel.InvokePromptAsync("Given that it is now the 10th of September
2024, 11:29 AM, what is the likely color of the sky in Boston?",  
new(settings));
```

Using None Function Choice Behavior

The `None` behavior instructs the AI model to use the provided function(s) without choosing any of them for invocation to generate a response. This is useful for dry runs when the caller may want to see which functions the model would choose without actually invoking them. It is also useful when you want the AI model to extract information from a user ask e.g. in the sample below the AI model correctly worked out that Boston was the city name.

Here, we advertise all functions from the `DateTimeUtils` and `WeatherForecastUtils` plugins to the AI model but instruct it not to choose any of them. Instead, the model will provide a response describing which functions it would choose to determine the color of the sky in Boston on a specified date.

C#

```
using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

KernelFunction getWeatherForCity =
kernel.Plugins.GetFunction("WeatherForecastUtils", "GetWeatherForCity");
```

```

PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.None() };

await kernel.InvokePromptAsync("Specify which provided functions are needed
to determine the color of the sky in Boston on a specified date.",
new(settings))

// Sample response: To determine the color of the sky in Boston on a
// specified date, first call the DateTimeUtils-GetCurrentUtcDateTime function
// to obtain the
// current date and time in UTC. Next, use the WeatherForecastUtils-
// GetWeatherForCity function, providing 'Boston' as the city name and the
// retrieved UTC date and time.
// These functions do not directly provide the sky's color, but the
// GetWeatherForCity function offers weather data, which can be used to infer
// the general sky condition (e.g., clear, cloudy, rainy).

```

A corresponding example in a YAML prompt template configuration:

```

C#

using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

string promptTemplateConfig = """
    template_format: semantic-kernel
    template: Specify which provided functions are needed to determine the
color of the sky in Boston on a specified date.
    execution_settings:
        default:
            function_choice_behavior:
                type: none
    """;

KernelFunction promptFunction =
KernelFunctionYaml.FromPromptYaml(promptTemplateConfig);

Console.WriteLine(await kernel.InvokeAsync(promptFunction));

```

Function Invocation

Function invocation is the process whereby Sematic Kernel invokes functions chosen by the AI model. For more details on function invocation see [function invocation article](#).

Supported AI Connectors

As of today, the following AI connectors in Semantic Kernel support the function calling model:

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

AI Connector	FunctionChoiceBehavior	ToolCallBehavior
Anthropic	Planned	✗
AzureAllInference	Coming soon	✗
AzureOpenAI	✓	✓
Gemini	Planned	✓
HuggingFace	Planned	✗
Mistral	Planned	✓
Ollama	Coming soon	✗
Onnx	Coming soon	✗
OpenAI	✓	✓

Function Invocation Modes

Article • 09/17/2024

When the AI model receives a prompt containing a list of functions, it may choose one or more of them for invocation to complete the prompt. When a function is chosen by the model, it needs to be **invoked** by Semantic Kernel.

The function calling subsystem in Semantic Kernel has two modes of function invocation: **auto** and **manual**.

Depending on the invocation mode, Semantic Kernel either does end-to-end function invocation or gives the caller control over the function invocation process.

Auto Function Invocation

Auto function invocation is the default mode of the Semantic Kernel function-calling subsystem. When the AI model chooses one or more functions, Semantic Kernel automatically invokes the chosen functions. The results of these function invocations are added to the chat history and sent to the model automatically in subsequent requests. The model then reasons about the chat history, chooses additional functions if needed, or generates the final response. This approach is fully automated and requires no manual intervention from the caller.

This example demonstrates how to use the auto function invocation in Semantic Kernel. AI model decides which functions to call to complete the prompt and Semantic Kernel does the rest and invokes them automatically.

C#

```
using Microsoft.SemanticKernel;

IKernelBuilder builder = Kernel.CreateBuilder();
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");
builder.Plugins.AddFromType<WeatherForecastUtils>();
builder.Plugins.AddFromType<DateTimeUtils>();

Kernel kernel = builder.Build();

// By default, functions are set to be automatically invoked.
// If you want to explicitly enable this behavior, you can do so with the
// following code:
// PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
// FunctionChoiceBehavior.Auto(autoInvoke: true) };
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };
```

```
await kernel.InvokePromptAsync("Given the current time of day and weather,  
what is the likely color of the sky in Boston?", new(settings));
```

Manual Function Invocation

In cases when the caller wants to have more control over the function invocation process, manual function invocation can be used.

When manual function invocation is enabled, Semantic Kernel does not automatically invoke the functions chosen by the AI model. Instead, it returns a list of chosen functions to the caller, who can then decide which functions to invoke, invoke them sequentially or in parallel, handle exceptions, and so on. The function invocation results need to be added to the chat history and returned to the model, which reasons about them and decides whether to choose additional functions or generate the final response.

The example below demonstrates how to use manual function invocation.

C#

```
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.ChatCompletion;  
  
IKernelBuilder builder = Kernel.CreateBuilder();  
builder.AddOpenAIChatCompletion("<model-id>", "<api-key>");  
builder.Plugins.AddFromType<WeatherForecastUtils>();  
builder.Plugins.AddFromType<DateTimeUtils>();  
  
Kernel kernel = builder.Build();  
  
IChatCompletionService chatCompletionService =  
kernel.GetRequiredService<IChatCompletionService>();  
  
// Manual function invocation needs to be enabled explicitly by setting  
autoInvoke to false.  
PromptExecutionSettings settings = new() { FunctionChoiceBehavior =  
Microsoft.SemanticKernel.FunctionChoiceBehavior.Auto(autoInvoke: false) };  
  
ChatHistory chatHistory = [];  
chatHistory.AddUserMessage("Given the current time of day and weather, what  
is the likely color of the sky in Boston?");  
  
while (true)  
{  
    ChatMessageContent result = await  
chatCompletionService.GetChatMessageContentAsync(chatHistory, settings,  
kernel);
```

```

    // Check if the AI model has generated a response.
    if (result.Content is not null)
    {
        Console.WriteLine(result.Content);
        // Sample output: "Considering the current weather conditions in
        Boston with a tornado watch in effect resulting in potential severe
        thunderstorms,
        // the sky color is likely unusual such as green, yellow, or dark
        gray. Please stay safe and follow instructions from local authorities."
        break;
    }

    // Adding AI model response containing chosen functions to chat history
    // as it's required by the models to preserve the context.
    chatHistory.Add(result);

    // Check if the AI model has chosen any function for invocation.
    IEnumerable<FunctionCallContent> functionCalls =
    FunctionCallContent.GetFunctionCalls(result);
    if (!functionCalls.Any())
    {
        break;
    }

    // Sequentially iterating over each chosen function, invoke it, and add
    // the result to the chat history.
    foreach (FunctionCallContent functionCall in functionCalls)
    {
        try
        {
            // Invoking the function
            FunctionResultContent resultContent = await
            functionCall.InvokeAsync(kernel);

            // Adding the function result to the chat history
            chatHistory.Add(resultContent.ToChatMessage());
        }
        catch (Exception ex)
        {
            // Adding function exception to the chat history.
            chatHistory.Add(new FunctionResultContent(functionCall,
            ex).ToChatMessage());
            // or
            //chatHistory.Add(new FunctionResultContent(functionCall, "Error
            details that the AI model can reason about.").ToChatMessage());
        }
    }
}

```

 **Note**

The FunctionCallContent and FunctionResultContent classes are used to represent AI model function calls and Semantic Kernel function invocation results, respectively. They contain information about chosen function, such as the function ID, name, and arguments, and function invocation results, such as function call ID and result.

AI Integrations for Semantic Kernel

Article • 10/16/2024

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

Out-of-the-box integrations

With the available AI connectors, developers can easily build AI agents with swappable components. This allows you to experiment with different AI services 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

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 .

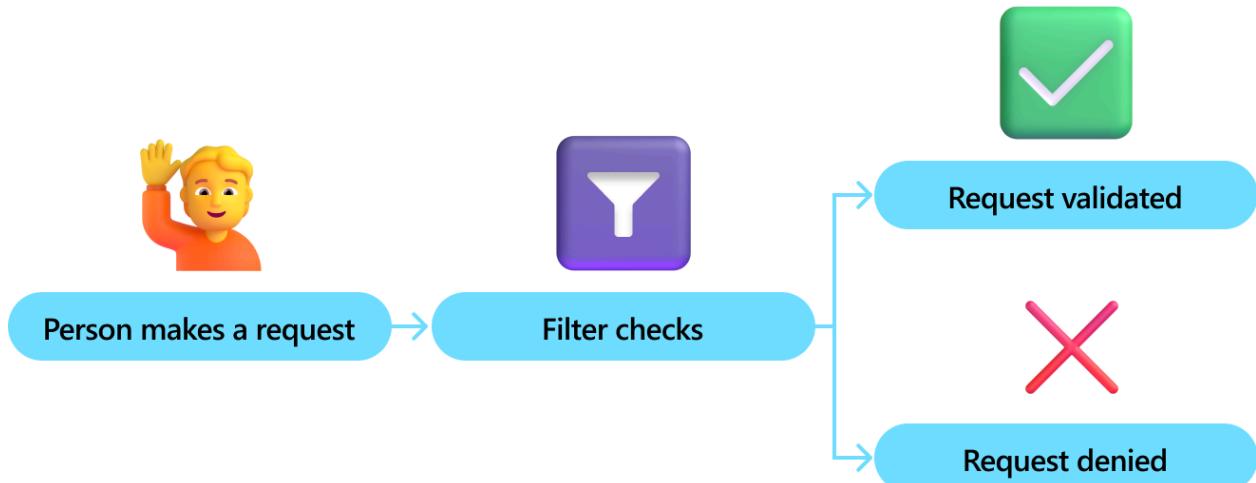
What are Filters?

Article • 09/24/2024

Filters enhance security by providing control and visibility over how and when functions run. This is needed to instill responsible AI principles into your work so that you feel confident your solution is enterprise ready.

For example, filters are leveraged to validate permissions before an approval flow begins. The `IFunctionInvocationFilter` is run to check the permissions of the person that's looking to submit an approval. This means that only a select group of people will be able to kick off the process.

A good example of filters is provided [here](#) in our detailed Semantic Kernel blog post on Filters.



There are 3 types of filters:

- Function invocation filter - it's executed every time `KernelFunction` is invoked. Allows to get information about function which is going to be executed, its arguments, catch an exception during function execution, override function result, retry function execution in case of failure (can be used to [switch to other AI model](#)).
- Prompt render filter - it's executed before prompt rendering operation. Allows to see what prompt is going to be sent to AI, modify prompt (e.g. RAG, [PII redaction](#) scenarios) and prevent the prompt from being sent to AI with function result override (can be used for [Semantic Caching](#)).
- Auto function invocation filter - similar to function invocation filter, but it is executed in a scope of `automatic function calling` operation, so it has more information available in a context, including chat history, list of all functions that will be executed and request iteration counters. It also allows to terminate auto

function calling process (e.g. there are 3 functions to execute, but there is already the desired result from the second function).

Each filter has `context` object that contains all information related to function execution or prompt rendering. Together with context, there is also a `next` delegate/callback, which executes next filter in pipeline or function itself. This provides more control, and it is useful in case there are some reasons to avoid function execution (e.g. malicious prompt or function arguments). It is possible to register multiple filters of the same type, where each filter will have different responsibility.

Example of function invocation filter to perform logging before and after function invocation:

C#

```
public sealed class LoggingFilter(ILogger logger) :  
    IFunctionInvocationFilter  
{  
    public async Task OnFunctionInvocationAsync(FunctionInvocationContext  
        context, Func<FunctionInvocationContext, Task> next)  
    {  
        logger.LogInformation("FunctionInvoking - {PluginName}.  
        {FunctionName}", context.Function.PluginName, context.Function.Name);  
  
        await next(context);  
  
        logger.LogInformation("FunctionInvoked - {PluginName}.  
        {FunctionName}", context.Function.PluginName, context.Function.Name);  
    }  
}
```

Example of prompt render filter which overrides rendered prompt before sending it to AI:

C#

```
public class SafePromptFilter : IPromptRenderFilter  
{  
    public async Task OnPromptRenderAsync(PromptRenderContext context,  
        Func<PromptRenderContext, Task> next)  
    {  
        // Example: get function information  
        var functionName = context.Function.Name;  
  
        await next(context);  
  
        // Example: override rendered prompt before sending it to AI  
        context.RenderedPrompt = "Safe prompt";  
    }  
}
```

```
    }  
}
```

Example of auto function invocation filter which terminates function calling process as soon as we have the desired result:

C#

```
public sealed class EarlyTerminationFilter : IAutoFunctionInvocationFilter  
{  
    public async Task  
OnAutoFunctionInvocationAsync(AutoFunctionInvocationContext context,  
Func<AutoFunctionInvocationContext, Task> next)  
    {  
        await next(context);  
  
        var result = context.Result.GetValue<string>();  
  
        if (result == "desired result")  
        {  
            context.Terminate = true;  
        }  
    }  
}
```

More information

C#/.NET:

- [Function invocation filter examples ↗](#)
- [Prompt render filter examples ↗](#)
- [Auto function invocation filter examples ↗](#)
- [PII detection and redaction with filters ↗](#)
- [Semantic Caching with filters ↗](#)
- [Content Safety with filters ↗](#)
- [Text summarization and translation quality check with filters ↗](#)

Observability in Semantic Kernel

Article • 09/24/2024

Brief introduction to observability

When you build AI solutions, you want to be able to observe the behavior of your services. Observability is the ability to monitor and analyze the internal state of components within a distributed system. It is a key requirement for building enterprise-ready AI solutions.

Observability is typically achieved through logging, metrics, and tracing. They are often referred to as the three pillars of observability. You will also hear the term "telemetry" used to describe the data collected by these three pillars. Unlike debugging, observability provides an ongoing overview of the system's health and performance.

Useful materials for further reading:

- [Observability defined by Cloud Native Computing Foundation ↗](#)
- [Distributed tracing](#)
- [Observability in .Net](#)
- [OpenTelemetry ↗](#)

Observability in Semantic Kernel

Semantic Kernel is designed to be observable. It emits logs, metrics, and traces that are compatible to the OpenTelemetry standard. You can use your favorite observability tools to monitor and analyze the behavior of your services built on Semantic Kernel.

Specifically, Semantic Kernel provides the following observability features:

- **Logging:** Semantic Kernel logs meaningful events and errors from the kernel, kernel plugins and functions, as well as the AI connectors. 

Important

[Traces in Application Insights](#) represent traditional log entries and [OpenTelemetry span events ↗](#). They are not the same as distributed traces.

- **Metrics:** Semantic Kernel emits metrics from kernel functions and AI connectors. You will be able to monitor metrics such as the kernel function execution time, the token consumption of AI connectors, etc. 

- **Tracing:** Semantic Kernel supports distributed tracing. You can track activities across different services and within Semantic Kernel.

 Complete end-to-end transaction of a request

Expand table

Telemetry	Description
Log	Logs are recorded throughout the Kernel. For more information on Logging in .Net, please refer to this document . Sensitive data, such as kernel function arguments and results, are logged at the trace level. Please refer to this table for more information on log levels.
Activity	Each kernel function execution and each call to an AI model are recorded as an activity. All activities are generated by an activity source named "Microsoft.SemanticKernel".
Metric	Semantic Kernel captures the following metrics from kernel functions: <ul style="list-style-type: none"> • <code>semantic_kernel.function.invocation.duration</code> (Histogram) - function execution time (in seconds) • <code>semantic_kernel.function.streaming.duration</code> (Histogram) - function streaming execution time (in seconds) • <code>semantic_kernel.function.invocation.token_usage.prompt</code> (Histogram) - number of prompt token usage (only for <code>KernelFunctionFromPrompt</code>) • <code>semantic_kernel.function.invocation.token_usage.completion</code> (Histogram) - number of completion token usage (only for <code>KernelFunctionFromPrompt</code>)

OpenTelemetry Semantic Convention

Semantic Kernel follows the [OpenTelemetry Semantic Convention](#) for Observability. This means that the logs, metrics, and traces emitted by Semantic Kernel are structured and follow a common schema. This ensures that you can more effectively analyze the telemetry data emitted by Semantic Kernel.

Note

Currently, the [Semantic Conventions for Generative AI](#) are in experimental status. Semantic Kernel strives to follow the OpenTelemetry Semantic Convention as closely as possible, and provide a consistent and meaningful observability experience for AI solutions.

Next steps

Now that you have a basic understanding of observability in Semantic Kernel, you can learn more about how to output telemetry data to the console or use APM tools to visualize and analyze telemetry data.

[Console](#)

[Application Insights](#)

[Aspire Dashboard](#)

Inspection of telemetry data with the console

Article • 09/24/2024

Although the console is not a recommended way to inspect telemetry data, it is a simple and quick way to get started. This article shows you how to output telemetry data to the console for inspection with a minimal Kernel setup.

Exporter

Exporters are responsible for sending telemetry data to a destination. Read more about exporters [here](#). In this example, we use the console exporter to output telemetry data to the console.

Prerequisites

- An Azure OpenAI chat completion deployment.
- The latest [.Net SDK](#) for your operating system.

Setup

Create a new console application

In a terminal, run the following command to create a new console application in C#:

```
Console  
dotnet new console -n TelemetryConsoleQuickstart
```

Navigate to the newly created project directory after the command completes.

Install required packages

- Semantic Kernel

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

- OpenTelemetry Console Exporter

Console

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

Create a simple application with Semantic Kernel

From the project directory, open the `Program.cs` file with your favorite editor. We are going to create a simple application that uses Semantic Kernel to send a prompt to a chat completion model. Replace the existing content with the following code and fill in the required values for `deploymentName`, `endpoint`, and `apiKey`:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel;
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

namespace TelemetryConsoleQuickstart
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // Telemetry setup code goes here

            IKernelBuilder builder = Kernel.CreateBuilder();
            // builder.Services.AddSingleton(loggerFactory);
            builder.AddAzureOpenAIChatCompletion(
                deploymentName: "your-deployment-name",
                endpoint: "your-azure-openai-endpoint",
                apiKey: "your-azure-openai-api-key"
            );

            Kernel kernel = builder.Build();

            var answer = await kernel.InvokePromptAsync(
                "Why is the sky blue in one sentence?"
            );

            Console.WriteLine(answer);
        }
    }
}
```

```
    }  
}
```

Add telemetry

If you run the console app now, you should expect to see a sentence explaining why the sky is blue. To observe the kernel via telemetry, replace the `// Telemetry setup code goes here` comment with the following code:

C#

```
var resourceBuilder = ResourceBuilder  
.CreateDefault()  
.AddService("TelemetryConsoleQuickstart");  
  
// Enable model diagnostics with sensitive data.  
AppContext.SetSwitch("Microsoft.SemanticKernel.Experimental.GenAI.EnableOTel  
DiagnosticsSensitive", true);  
  
using var traceProvider = Sdk.CreateTracerProviderBuilder()  
.SetResourceBuilder(resourceBuilder)  
.AddSource("Microsoft.SemanticKernel*")  
.AddConsoleExporter()  
.Build();  
  
using var meterProvider = Sdk.CreateMeterProviderBuilder()  
.SetResourceBuilder(resourceBuilder)  
.AddMeter("Microsoft.SemanticKernel*")  
.AddConsoleExporter()  
.Build();  
  
using var loggerFactory = LoggerFactory.Create(builder =>  
{  
    // Add OpenTelemetry as a logging provider  
    builder.AddOpenTelemetry(options =>  
    {  
        options.SetResourceBuilder(resourceBuilder);  
        options.AddConsoleExporter();  
        // Format log messages. This is default to false.  
        options.IncludeFormattedMessage = true;  
        options.IncludeScopes = true;  
    });  
    builder.SetMinimumLevel(LogLevel.Information);  
});
```

Finally Uncomment the line `// builder.Services.AddSingleton(loggerFactory);` to add the logger factory to the builder.

In the above code snippet, we first create a resource builder for building resource instances. A resource represents the entity that produces telemetry data. You can read more about resources [here](#). The resource builder to the providers is optional. If not provided, the default resource with default attributes is used.

Next, we turn on diagnostics with sensitive data. This is an experimental feature that allows you to enable diagnostics for the AI services in the Semantic Kernel. With this turned on, you will see additional telemetry data such as the prompts sent to and the responses received from the AI models, which are considered sensitive data. If you don't want to include sensitive data in your telemetry, you can use another switch `Microsoft.SemanticKernel.Experimental.GenAI.EnableOTelDiagnostics` to enable diagnostics with non-sensitive data, such as the model name, the operation name, and token usage, etc.

Then, we create a tracer provider builder and a meter provider builder. A provider is responsible for processing telemetry data and piping it to exporters. We subscribe to the `Microsoft.SemanticKernel*` source to receive telemetry data from the Semantic Kernel namespaces. We add a console exporter to both the tracer provider and the meter provider. The console exporter sends telemetry data to the console.

Finally, we create a logger factory and add OpenTelemetry as a logging provider that sends log data to the console. We set the minimum log level to `Information` and include formatted messages and scopes in the log output. The logger factory is then added to the builder.

Important

A provider should be a singleton and should be alive for the entire application lifetime. The provider should be disposed of when the application is shutting down.

Run

Run the console application with the following command:

```
Console
```

```
dotnet run
```

Inspect telemetry data

Log records

You should see multiple log records in the console output. They look similar to the following:

```
Console

LogRecord.Timestamp: 2024-09-12T21:48:35.2295938Z
LogRecord.TraceId: 159d3f07664838f6abdad7af6a892cfa
LogRecord.SpanId: ac79a006da8a6215
LogRecord.TraceFlags: Recorded
LogRecord.CategoryName: Microsoft.SemanticKernel.KernelFunction
LogRecord.Severity: Info
LogRecord.SeverityText: Information
LogRecord.FormattedMessage: Function
InvokePromptAsync_290eb9bece084b00aea46b569174feae invoking.
LogRecord.Body: Function {FunctionName} invoking.
LogRecord.Attributes (Key:Value):
  FunctionName: InvokePromptAsync_290eb9bece084b00aea46b569174feae
  OriginalFormat (a.k.a Body): Function {FunctionName} invoking.

Resource associated with LogRecord:
service.name: TelemetryConsoleQuickstart
service.instance.id: a637dfc9-0e83-4435-9534-fb89902e64f8
telemetry.sdk.name: opentelemetry
telemetry.sdk.language: dotnet
telemetry.sdk.version: 1.9.0
```

There are two parts to each log record:

- The log record itself: contains the timestamp and namespace at which the log record was generated, the severity and body of the log record, and any attributes associated with the log record.
- The resource associated with the log record: contains information about the service, instance, and SDK used to generate the log record.

Activities

Note

Activities in .Net are similar to spans in OpenTelemetry. They are used to represent a unit of work in the application.

You should see multiple activities in the console output. They look similar to the following:

Console

```
Activity.TraceId:          159d3f07664838f6abdad7af6a892cfa
Activity.SpanId:           8c7c79bc1036eab3
Activity.TraceFlags:        Recorded
Activity.ParentSpanId:     ac79a006da8a6215
Activity.ActivitySourceName: Microsoft.SemanticKernel.Diagnostics
Activity.DisplayName:       chat.completions gpt-4o
Activity.Kind:              Client
Activity.StartTime:         2024-09-12T21:48:35.5717463Z
Activity.Duration:          00:00:02.3992014
Activity.Tags:
    gen_ai.operation.name: chat.completions
    gen_ai.system: openai
    gen_ai.request.model: gpt-4o
    gen_ai.response.prompt_tokens: 16
    gen_ai.response.completion_tokens: 29
    gen_ai.response.finish_reason: Stop
    gen_ai.response.id: chatcmpl-A6lxz14rKuQpQibmiCpzmye6z9rxC
Activity.Events:
    gen_ai.content.prompt [9/12/2024 9:48:35 PM +00:00]
        gen_ai.prompt: [{"role": "user", "content": "Why is the sky blue in one sentence?"}]
    gen_ai.content.completion [9/12/2024 9:48:37 PM +00:00]
        gen_ai.completion: [{"role": "Assistant", "content": "The sky appears blue because shorter blue wavelengths of sunlight are scattered in all directions by the gases and particles in the Earth\u0027s atmosphere more than other colors."}]
Resource associated with Activity:
    service.name: TelemetryConsoleQuickstart
    service.instance.id: a637dfc9-0e83-4435-9534-fb89902e64f8
    telemetry.sdk.name: opentelemetry
    telemetry.sdk.language: dotnet
    telemetry.sdk.version: 1.9.0
```

There are two parts to each activity:

- The activity itself: contains the span ID and parent span ID that APM tools use to build the traces, the duration of the activity, and any tags and events associated with the activity.
- The resource associated with the activity: contains information about the service, instance, and SDK used to generate the activity.

ⓘ Important

The attributes to pay extra attention to are the ones that start with `gen_ai`. These are the attributes specified in the [GenAI Semantic Conventions](#).

Metrics

You should see multiple metric records in the console output. They look similar to the following:

Console

```
Metric Name: semantic_kernel.connectors.openai.tokens.prompt, Number of
prompt tokens used, Unit: {token}, Meter:
Microsoft.SemanticKernel.Connectors.OpenAI
(2024-09-12T21:48:37.9531072Z, 2024-09-12T21:48:38.0966737Z] LongSum
Value: 16
```

Here you can see the name, the description, the unit, the time range, the type, the value of the metric, and the meter that the metric belongs to.

ⓘ Note

The above metric is a Counter metric. For a full list of metric types, see [here](#). Depending on the type of metric, the output may vary.

Next steps

Now that you have successfully output telemetry data to the console, you can learn more about how to use APM tools to visualize and analyze telemetry data.

[Application Insights](#)

[Aspire Dashboard](#)

Inspection of telemetry data with Application Insights

Article • 09/24/2024

[Application Insights](#) is part of [Azure Monitor](#), which is a comprehensive solution for collecting, analyzing, and acting on telemetry data from your cloud and on-premises environments. With Application Insights, you can monitor your application's performance, detect issues, and diagnose problems.

In this example, we will learn how to export telemetry data to Application Insights, and inspect the data in the Application Insights portal.

Exporter

Exporters are responsible for sending telemetry data to a destination. Read more about exporters [here](#). In this example, we use the Azure Monitor exporter to output telemetry data to an Application Insights instance.

Prerequisites

- An Azure OpenAI chat completion deployment.
- An Application Insights instance. Follow the [instructions](#) here to create a resource if you don't have one. Copy the [connection string](#) for later use.
- The latest [.Net SDK](#) for your operating system.

Setup

Create a new console application

In a terminal, run the following command to create a new console application in C#:

```
Console
```

```
dotnet new console -n TelemetryApplicationInsightsQuickstart
```

Navigate to the newly created project directory after the command completes.

Install required packages

- Semantic Kernel

```
Console
```

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

- OpenTelemetry Console Exporter

```
Console
```

```
dotnet add package Azure.Monitor.OpenTelemetry.Exporter
```

Create a simple application with Semantic Kernel

From the project directory, open the `Program.cs` file with your favorite editor. We are going to create a simple application that uses Semantic Kernel to send a prompt to a chat completion model. Replace the existing content with the following code and fill in the required values for `deploymentName`, `endpoint`, and `apiKey`:

```
C#
```

```
using Azure.Monitor.OpenTelemetry.Exporter;

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel;
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

namespace TelemetryApplicationInsightsQuickstart
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // Telemetry setup code goes here

            IKernelBuilder builder = Kernel.CreateBuilder();
            // builder.Services.AddSingleton(loggerFactory);
            builder.AddAzureOpenAIChatCompletion(
                deploymentName: "your-deployment-name",
                endpoint: "your-azure-openai-endpoint",
                apiKey: "your-azure-openai-api-key"
            );
        }
    }
}
```

```

        Kernel kernel = builder.Build();

        var answer = await kernel.InvokePromptAsync(
            "Why is the sky blue in one sentence?"
        );

        Console.WriteLine(answer);
    }
}

```

Add telemetry

If you run the console app now, you should expect to see a sentence explaining why the sky is blue. To observe the kernel via telemetry, replace the `// Telemetry setup code goes here` comment with the following code:

C#

```

// Replace the connection string with your Application Insights connection
string
var connectionString = "your-application-insights-connection-string";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService("TelemetryApplicationInsightsQuickstart");

// Enable model diagnostics with sensitive data.
AppContext.SetSwitch("Microsoft.SemanticKernel.Experimental.GenAI.EnableOTel
DiagnosticsSensitive", true);

using var traceProvider = Sdk.CreateTracerProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddSource("Microsoft.SemanticKernel*")
    .AddAzureMonitorTraceExporter(options => options.ConnectionString =
connectionString)
    .Build();

using var meterProvider = Sdk.CreateMeterProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddMeter("Microsoft.SemanticKernel*")
    .AddAzureMonitorMetricExporter(options => options.ConnectionString =
connectionString)
    .Build();

using var loggerFactory = LoggerFactory.Create(builder =>
{
    // Add OpenTelemetry as a logging provider
    builder.AddOpenTelemetry(options =>
    {
        options.SetResourceBuilder(resourceBuilder);
    });
}

```

```
        options.AddAzureMonitorLogExporter(options =>
    options.ConnectionString = connectionString);
    // Format log messages. This is default to false.
    options.IncludeFormattedMessage = true;
    options.IncludeScopes = true;
});
builder.SetMinimumLevel(LogLevel.Information);
});
```

Finally Uncomment the line `// builder.Services.AddSingleton(loggerFactory);` to add the logger factory to the builder.

Please refer to this [article](#) for more information on the telemetry setup code. The only difference here is that we are using `AddAzureMonitor[Trace|Metric|Log]Exporter` to export telemetry data to Application Insights.

Run

Run the console application with the following command:

```
Console
```

```
dotnet run
```

Inspect telemetry data

After running the application, head over to the Application Insights portal to inspect the telemetry data. It may take a few minutes for the data to appear in the portal.

Transaction search

Navigate to the **Transaction search** tab to view the transactions that have been recorded.



Hit refresh to see the latest transactions. When results appear, click on one of them to see more details.



Toggle between the **View all** and **View timeline** button to see all traces and dependencies of the transaction in different views.

Important

[Traces](#) represent traditional log entries and [OpenTelemetry span events](#). They are not the same as distributed traces. Dependencies represent the calls to (internal and external) components. Please refer to this [article](#) for more information on the data model in Application Insights.

For this particular example, you should see two dependencies and multiple traces. The first dependency represents a kernel function that is created from the prompt. The second dependency represents the call to the Azure OpenAI chat completion model. When you expand the `chat.completion {your-deployment-name}` dependency, you should see the details of the call. A set of `gen_ai` attributes are attached to the dependency, which provides additional context about the call.



If you have the switch

```
Microsoft.SemanticKernel.Experimental.GenAI.EnableOTelDiagnosticsSensitive
```

 set to `true`, you will also see two traces that carry the sensitive data of the prompt and the completion result.

Click on them and you will see the prompt and the completion result under the custom properties section.

Log analytics

Transaction search is not the only way to inspect telemetry data. You can also use [Log analytics](#) to query and analyze the data. Navigate to the [Logs](#) under [Monitoring](#) to start.

Follow this [document](#) to start exploring the log analytics interface.

Below are some sample queries you can use for this example:

Kusto

```
// Retrieves the total number of completion and prompt tokens used for the
model if you run the application multiple times.
dependencies
| where namestartswith "chat"
| project model = customDimensions["gen_ai.request.model"], completion_token
= toint(customDimensions["gen_ai.response.completion_tokens"]), prompt_token
= toint(customDimensions["gen_ai.response.prompt_tokens"])
```

```
| where model == "gpt-4o"
| project completion_token, prompt_token
| summarize total_completion_tokens = sum(completion_token),
total_prompt_tokens = sum(prompt_token)
```

Kusto

```
// Retrieves all the prompts and completions and their corresponding token
usage.
dependencies
| where name startswith "chat"
| project timestamp, operation_Id, name, completion_token =
customDimensions["gen_ai.response.completion_tokens"], prompt_token =
customDimensions["gen_ai.response.prompt_tokens"]
| join traces on operation_Id
| where message startswith "gen_ai"
| project timestamp, messages = customDimensions, token=iff(customDimensions
contains "gen_ai.prompt", prompt_token, completion_token)
```



Next steps

Now that you have successfully output telemetry data to Application Insights, you can explore more features of Semantic Kernel that can help you monitor and diagnose your application:

[Advanced telemetry with Semantic Kernel](#)

Inspection of telemetry data with Aspire Dashboard

Article • 09/24/2024

Aspire Dashboard is part of the [.NET Aspire](#) offering. The dashboard allows developers to monitor and inspect their distributed applications.

In this example, we will use the [standalone mode](#) and learn how to export telemetry data to Aspire Dashboard, and inspect the data there.

Exporter

Exporters are responsible for sending telemetry data to a destination. Read more about exporters [here](#). In this example, we use the [OpenTelemetry Protocol \(OTLP\)](#) exporter to send telemetry data to Aspire Dashboard.

Prerequisites

- An Azure OpenAI chat completion deployment.
- Docker
- The latest [.Net SDK](#) for your operating system.

Setup

Create a new console application

In a terminal, run the following command to create a new console application in C#:

```
Console
```

```
dotnet new console -n TelemetryAspireDashboardQuickstart
```

Navigate to the newly created project directory after the command completes.

Install required packages

- Semantic Kernel

```
Console
```

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

- OpenTelemetry Console Exporter

```
Console
```

```
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
```

Create a simple application with Semantic Kernel

From the project directory, open the `Program.cs` file with your favorite editor. We are going to create a simple application that uses Semantic Kernel to send a prompt to a chat completion model. Replace the existing content with the following code and fill in the required values for `deploymentName`, `endpoint`, and `apiKey`:

```
C#
```

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel;
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

namespace TelemetryAspireDashboardQuickstart
{
    class Program
    {
        static async Task Main(string[] args)
        {
            // Telemetry setup code goes here

            IKernelBuilder builder = Kernel.CreateBuilder();
            // builder.Services.AddSingleton(loggerFactory);
            builder.AddAzureOpenAIChatCompletion(
                deploymentName: "your-deployment-name",
                endpoint: "your-azure-openai-endpoint",
                apiKey: "your-azure-openai-api-key"
            );

            Kernel kernel = builder.Build();

            var answer = await kernel.InvokePromptAsync(
                "Why is the sky blue in one sentence?"
            );
        }
    }
}
```

```
        Console.WriteLine(answer);
    }
}
}
```

Add telemetry

If you run the console app now, you should expect to see a sentence explaining why the sky is blue. To observe the kernel via telemetry, replace the `// Telemetry setup code goes here` comment with the following code:

C#

```
// Endpoint to the Aspire Dashboard
var endpoint = "http://localhost:4317";

var resourceBuilder = ResourceBuilder
    .CreateDefault()
    .AddService("TelemetryAspireDashboardQuickstart");

// Enable model diagnostics with sensitive data.
AppContext.SetSwitch("Microsoft.SemanticKernel.Experimental.GenAI.EnableOTel
DiagnosticsSensitive", true);

using var traceProvider = Sdk.CreateTracerProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddSource("Microsoft.SemanticKernel*")
    .AddOtlpExporter(options => options.Endpoint = new Uri(endpoint))
    .Build();

using var meterProvider = Sdk.CreateMeterProviderBuilder()
    .SetResourceBuilder(resourceBuilder)
    .AddMeter("Microsoft.SemanticKernel*")
    .AddOtlpExporter(options => options.Endpoint = new Uri(endpoint))
    .Build();

using var loggerFactory = LoggerFactory.Create(builder =>
{
    // Add OpenTelemetry as a logging provider
    builder.AddOpenTelemetry(options =>
    {
        options.SetResourceBuilder(resourceBuilder);
        options.AddOtlpExporter(options => options.Endpoint = new
Uri(endpoint));
        // Format log messages. This is default to false.
        options.IncludeFormattedMessage = true;
        options.IncludeScopes = true;
    });
});
```

```
    builder.SetMinimumLevel(LogLevel.Information);  
});
```

Finally Uncomment the line `// builder.Services.AddSingleton(loggerFactory);` to add the logger factory to the builder.

Please refer to this [article](#) for more information on the telemetry setup code. The only difference here is that we are using `AddOtlpExporter` to export telemetry data to Aspire Dashboard.

Start the Aspire Dashboard

Follow the instructions [here](#) to start the dashboard. Once the dashboard is running, open a browser and navigate to `http://localhost:18888` to access the dashboard.

Run

Run the console application with the following command:

```
Console  
dotnet run
```

Inspect telemetry data

After running the application, head over to the dashboard to inspect the telemetry data.

💡 Tip

Follow this [guide](#) to explore the Aspire Dashboard interface.

Traces

If this is your first time running the application after starting the dashboard, you should see a one trace is the `Traces` tab. Click on the trace to view more details.



In the trace details, you can see the span that represents the prompt function and the span that represents the chat completion model. Click on the chat completion span to

see details about the request and response.

💡 Tip

You can filter the attributes of the spans to find the one you are interested in.



Logs

Head over to the `Structured` tab to view the logs emitted by the application. Please refer to this [guide](#) on how to work with structured logs in the dashboard.

Next steps

Now that you have successfully output telemetry data to Aspire Dashboard, you can explore more features of Semantic Kernel that can help you monitor and diagnose your application:

[Advanced telemetry with Semantic Kernel](#)

More advanced scenarios for telemetry

Article • 09/24/2024

ⓘ Note

This article will use [Aspire Dashboard](#) for illustration. If you prefer to use other tools, please refer to the documentation of the tool you are using on setup instructions.

Auto Function Calling

Auto Function Calling is a Semantic Kernel feature that allows the kernel to automatically execute functions when the model responds with function calls, and provide the results back to the model. This feature is useful for scenarios where a query requires multiple iterations of function calls to get a final natural language response. For more details, please see these GitHub [samples](#).

ⓘ Note

Function calling is not supported by all models.

ⓘ Tip

You will hear the term "tools" and "tool calling" sometimes used interchangeably with "functions" and "function calling".

Prerequisites

- An Azure OpenAI chat completion deployment that supports function calling.
- Docker
- The latest [.Net SDK](#) for your operating system.

Setup

Create a new console application

In a terminal, run the following command to create a new console application in C#:

Console

```
dotnet new console -n TelemetryAutoFunctionCallingQuickstart
```

Navigate to the newly created project directory after the command completes.

Install required packages

- Semantic Kernel

Console

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

- OpenTelemetry Console Exporter

Console

```
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
```

Create a simple application with Semantic Kernel

From the project directory, open the `Program.cs` file with your favorite editor. We are going to create a simple application that uses Semantic Kernel to send a prompt to a chat completion model. Replace the existing content with the following code and fill in the required values for `deploymentName`, `endpoint`, and `apiKey`:

C#

```
using System.ComponentModel;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using OpenTelemetry;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;

namespace TelemetryAutoFunctionCallingQuickstart
{
    class BookingPlugin
    {
        [KernelFunction("FindAvailableRooms")]
    }
}
```

```

    [Description("Finds available conference rooms for today.")]
    public async Task<List<string>> FindAvailableRoomsAsync()
    {
        // Simulate a remote call to a booking system.
        await Task.Delay(1000);
        return ["Room 101", "Room 201", "Room 301"];
    }

    [KernelFunction("BookRoom")]
    [Description("Books a conference room.")]
    public async Task<string> BookRoomAsync(string room)
    {
        // Simulate a remote call to a booking system.
        await Task.Delay(1000);
        return $"Room {room} booked.";
    }
}

class Program
{
    static async Task Main(string[] args)
    {
        // Endpoint to the Aspire Dashboard
        var endpoint = "http://localhost:4317";

        var resourceBuilder = ResourceBuilder
            .CreateDefault()
            .AddService("TelemetryAspireDashboardQuickstart");

        // Enable model diagnostics with sensitive data.

        ApplicationContext.SetSwitch("Microsoft.SemanticKernel.Experimental.GenAI.EnableOTel
DiagnosticsSensitive", true);

        using var traceProvider = Sdk.CreateTracerProviderBuilder()
            .SetResourceBuilder(resourceBuilder)
            .AddSource("Microsoft.SemanticKernel*")
            .AddOtlpExporter(options => options.Endpoint = new
Uri(endpoint))
            .Build();

        using var meterProvider = Sdk.CreateMeterProviderBuilder()
            .SetResourceBuilder(resourceBuilder)
            .AddMeter("Microsoft.SemanticKernel*")
            .AddOtlpExporter(options => options.Endpoint = new
Uri(endpoint))
            .Build();

        using var loggerFactory = LoggerFactory.Create(builder =>
{
    // Add OpenTelemetry as a logging provider
    builder.AddOpenTelemetry(options =>
{
    options.SetResourceBuilder(resourceBuilder);
    options.AddOtlpExporter(options => options.Endpoint =

```

```

new Uri(endpoint));
        // Format log messages. This is default to false.
        options.IncludeFormattedMessage = true;
        options.IncludeScopes = true;
    });
    builder.SetMinimumLevel(LogLevel.Information);
});

IKernelBuilder builder = Kernel.CreateBuilder();
builder.Services.AddSingleton(loggerFactory);
builder.AddAzureOpenAIChatCompletion(
    deploymentName: "your-deployment-name",
    endpoint: "your-azure-openai-endpoint",
    apiKey: "your-azure-openai-api-key"
);
builder.Plugins.AddFromType<BookingPlugin>();

Kernel kernel = builder.Build();

var answer = await kernel.InvokePromptAsync(
    "Reserve a conference room for me today.",
    new KernelArguments(
        new OpenAIPromptExecutionSettings {
            ToolCallBehavior =
                ToolCallBehavior.AutoInvokeKernelFunctions
        }
    )
);

Console.WriteLine(answer);
}
}
}

```

In the code above, we first define a mock conference room booking plugin with two functions: `FindAvailableRoomsAsync` and `BookRoomAsync`. We then create a simple console application that registers the plugin to the kernel, and ask the kernel to automatically call the functions when needed.

Start the Aspire Dashboard

Follow the instructions [here](#) to start the dashboard. Once the dashboard is running, open a browser and navigate to `http://localhost:18888` to access the dashboard.

Run

Run the console application with the following command:

Console

```
dotnet run
```

You should see an output similar to the following:

Console

```
Room 101 has been successfully booked for you today.
```

Inspect telemetry data

After running the application, head over to the dashboard to inspect the telemetry data.

Find the trace for the application in the **Traces** tab. You should five spans in the trace:



These 5 spans represent the internal operations of the kernel with auto function calling enabled. It first invokes the model, which requests a function call. Then the kernel automatically executes the function `FindAvailableRoomsAsync` and returns the result to the model. The model then requests another function call to make a reservation, and the kernel automatically executes the function `BookRoomAsync` and returns the result to the model. Finally, the model returns a natural language response to the user.

And if you click on the last span, and look for the prompt in the `gen_ai.content.prompt` event, you should see something similar to the following:

JSON

```
[  
  { "role": "user", "content": "Reserve a conference room for me today." },  
  {  
    "role": "Assistant",  
    "content": null,  
    "tool_calls": [  
      {  
        "id": "call_NtKi00g011j1StLk0mJU8cP",  
        "function": { "arguments": {}, "name": "FindAvailableRooms" },  
        "type": "function"  
      }  
    ]  
  },  
  {  
    "role": "tool",  
    "content": "[\u0022Room 101\u0022,\u0022Room 201\u0022,\u0022Room  
301\u0022]"  
  },
```

```
{
  "role": "Assistant",
  "content": null,
  "tool_calls": [
    {
      "id": "call_mjQfnZXLbqp4Wb3F2xySds7q",
      "function": { "arguments": { "room": "Room 101" }, "name": "BookRoom" },
      "type": "function"
    }
  ],
  { "role": "tool", "content": "Room Room 101 booked." }
]
```

This is the chat history that gets built up as the model and the kernel interact with each other. This is sent to the model in the last iteration to get a natural language response.

Error handling

If an error occurs during the execution of a function, the kernel will automatically catch the error and return an error message to the model. The model can then use this error message to provide a natural language response to the user.

Modify the `BookRoomAsync` function in the C# code to simulate an error:

```
C#
[KernelFunction("BookRoom")]
[Description("Books a conference room.")]
public async Task<string> BookRoomAsync(string room)
{
    // Simulate a remote call to a booking system.
    await Task.Delay(1000);

    throw new Exception("Room is not available.");
}
```

Run the application again and observe the trace in the dashboard. You should see the span representing the kernel function call with an error:



➊ Note

It is very likely that the model responses to the error may vary each time you run the application, because the model is stochastic. You may see the model reserving

all three rooms at the same time, or reserving one the first time then reserving the other two the second time, etc.

Next steps and further reading

In production, your services may get a large number of requests. Semantic Kernel will generate a large amount of telemetry data, some of which may not be useful for your use case and will introduce unnecessary costs to store the data. You can use the [sampling](#) feature to reduce the amount of telemetry data that is collected.

Observability in Semantic Kernel is constantly improving. You can find the latest updates and new features in the [GitHub repository](#).

Security

Article • 09/24/2024

Microsoft takes the security of our software products and services seriously, which includes all source code repositories managed through our GitHub organizations, which include [Microsoft](#), [Azure](#), [DotNet](#), [AspNet](#), [Xamarin](#), and [our GitHub organizations](#).

If you believe you have found a security vulnerability in any Microsoft-owned repository that meets [Microsoft's definition of a security vulnerability](#), please report it to us as described below.

Reporting Security Issues

Please do not report security vulnerabilities through public GitHub issues.

Instead, please report them to the Microsoft Security Response Center (MSRC) at <https://msrc.microsoft.com/create-report>.

If you prefer to submit without logging in, send email to secure@microsoft.com. If possible, encrypt your message with our PGP key; please download it from the [Microsoft Security Response Center PGP Key page](#).

You should receive a response within 24 hours. If for some reason you do not, please follow up via email to ensure we received your original message. Additional information can be found at microsoft.com/msrc.

Please include the requested information listed below (as much as you can provide) to help us better understand the nature and scope of the possible issue:

- Type of issue (e.g. buffer overflow, SQL injection, cross-site scripting, etc.)
- Full paths of source file(s) related to the manifestation of the issue
- The location of the affected source code (tag/branch/commit or direct URL)
- Any special configuration required to reproduce the issue
- Step-by-step instructions to reproduce the issue
- Proof-of-concept or exploit code (if possible)
- Impact of the issue, including how an attacker might exploit the issue

This information will help us triage your report more quickly.

If you are reporting for a bug bounty, more complete reports can contribute to a higher bounty award. Please visit our [Microsoft Bug Bounty Program](#) page for more details about our active programs.

Preferred Languages

We prefer all communications to be in English.

Policy

Microsoft follows the principle of [Coordinated Vulnerability Disclosure](#).

What are Semantic Kernel Vector Store connectors? (Preview)

Article • 10/16/2024

⚠ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

💡 Tip

If you are looking for information about the legacy Memory Store connectors, refer to the [Memory Stores page](#).

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 responses. 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 help 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. The vector database allows you to efficiently find the subset of information related to a specific question or topic. You can then pass information from the vector database with your prompt to your large language model to generate more accurate and relevant content.

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.

Semantic Kernel and .net provides an abstraction for interacting with Vector Stores and a list of out-of-the-box connectors that implement these abstractions. Features include creating, listing and deleting collections of records, and uploading, retrieving and deleting records. The abstraction makes it easy to experiment with a free or locally hosted Vector Store and then switch to a service when needing to scale up.

The Vector Store Abstraction

The main interfaces in the Vector Store abstraction are the following.

Microsoft.Extensions.VectorData.IVectorStore

`IVectorStore` contains operations that spans across all collections in the vector store, e.g. `ListCollectionNames`. It also provides the ability to get `IVectorStoreRecordCollection<TKey, TRecord>` instances.

Microsoft.Extensions.VectorData.IVectorStoreRecordCollection<TKey, TRecord>

`IVectorStoreRecordCollection<TKey, TRecord>` represents a collection. This collection may or may not exist, and the interface provides methods to check if the collection exists, create it or delete it. The interface also provides methods to upsert, get and delete records. Finally, the interface inherits from `IVectorizedSearch<TRecord>` providing vector search capabilities.

Microsoft.Extensions.VectorData.IVectorizedSearch<TRecord>

`IVectorizedSearch<TRecord>` contains a method for doing vector searches.

`IVectorStoreRecordCollection<TKey, TRecord>` inherits from `IVectorizedSearch<TRecord>` making it possible to use `IVectorizedSearch<TRecord>` on its own in cases where only search is needed and no record or collection management is needed.

IVectorizableTextSearch<TRecord>

`IVectorizableTextSearch<TRecord>` contains a method for doing vector searches where the vector database has the ability to generate embeddings automatically. E.g. you can call this method with a text string and the database will generate the embedding for you and search against a vector field. This is not supported by all vector databases and is therefore only implemented by select connectors.

Getting started with Vector Store connectors

Import the necessary nuget packages

All the vector store interfaces and any abstraction related classes are available in the `Microsoft.Extensions.VectorData.Abstractions` nuget package. Each vector store implementation is available in its own nuget package. For a list of known implementations, see the [Out-of-the-box connectors page](#).

The abstractions package can be added like this.

.NET CLI

```
dotnet add package Microsoft.Extensions.VectorData.Abstractions --prerelease
```

⚠ Warning

From version 1.23.0 of Semantic Kernel, the Vector Store abstractions have been removed from `Microsoft.SemanticKernel.Abstractions` and are available in the new dedicated `Microsoft.Extensions.VectorData.Abstractions` package.

Note that from version 1.23.0, `Microsoft.SemanticKernel.Abstractions` has a dependency on `Microsoft.Extensions.VectorData.Abstractions`, therefore there is no need to reference additional packages. The abstractions will however now be in the new `Microsoft.Extensions.VectorData` namespace.

When upgrading from 1.22.0 or earlier to 1.23.0 or later, you will need to add an additional `using Microsoft.Extensions.VectorData;` clause in files where any of the Vector Store abstraction types are used e.g. `IVectorStore`, `IVectorStoreRecordCollection`, `VectorStoreRecordDataAttribute`, `VectorStoreRecordKeyProperty`, etc.

This change has been made to support vector store providers when creating their own implementations. A provider only has to reference the `Microsoft.Extensions.VectorData.Abstractions` package. This reduces potential version conflicts and allows Semantic Kernel to continue to evolve fast without impacting vector store providers.

Define your data model

The Semantic Kernel Vector Store connectors use a model first approach to interacting with databases. This means that the first step is to define a data model that maps to the storage schema. To help the connectors create collections of records and map to the storage schema, the model can be annotated to indicate the function of each property.

C#

```
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true)]
    public string Description { get; set; }

    [VectorStoreRecordVector(Dimensions: 4, DistanceFunction.CosineDistance,
    IndexKind.Hnsw)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
```

```
    public string[] Tags { get; set; }  
}
```

💡 Tip

For more information on how to annotate your data model, refer to [defining your data model](#).

💡 Tip

For an alternative to annotating your data model, refer to [defining your schema with a record definition](#).

Connect to your database and select a collection

Once you have defined your data model, the next step is to create a VectorStore instance for the database of your choice and select a collection of records.

In this example, we'll use Qdrant. You will therefore need to import the Qdrant nuget package.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.Qdrant --prerelease
```

Since databases support many different types of keys and records, we allow you to specify the type of the key and record for your collection using generics. In our case, the type of record will be the `Hotel` class we already defined, and the type of key will be `ulong`, since the `HotelId` property is a `ulong` and Qdrant only supports `Guid` or `ulong` keys.

C#

```
using Microsoft.SemanticKernel.Connectors.Qdrant;  
using Qdrant.Client;  
  
// Create a Qdrant VectorStore object  
var vectorStore = new QdrantVectorStore(new QdrantClient("localhost"));  
  
// Choose a collection from the database and specify the type of key and record  
// stored in it via Generic parameters.  
var collection = vectorStore.GetCollection<ulong, Hotel>("skhotels");
```

💡 Tip

For more information on what key and field types each Vector Store connector supports, refer to [the documentation for each connector](#).

Create the collection and add records

C#

```
// Create the collection if it doesn't exist yet.  
await collection.CreateCollectionIfNotExistsAsync();  
  
// Upsert a record.  
string descriptionText = "A place where everyone can be happy.";  
ulong hotelId = 1;  
  
// Create a record and generate a vector for the description using your chosen  
embedding generation implementation.  
// Just showing a placeholder embedding generation method here for brevity.  
await collection.UpsertAsync(new Hotel  
{  
    HotelId = hotelId,  
    HotelName = "Hotel Happy",  
    Description = descriptionText,  
    DescriptionEmbedding = await GenerateEmbeddingAsync(descriptionText),  
    Tags = new[] { "luxury", "pool" }  
});  
  
// Retrieve the upserted record.  
Hotel? retrievedHotel = await collection.GetAsync(hotelId);
```

💡 Tip

For more information on how to generate embeddings see [embedding generation](#).

Do a vector search

C#

```
// Generate a vector for your search text, using your chosen embedding generation  
implementation.  
// Just showing a placeholder method here for brevity.  
var searchVector = await GenerateEmbeddingAsync("I'm looking for a hotel where  
customer happiness is the priority.");  
// Do the search.  
var searchResult = await collection.VectorizedSearchAsync(searchVector, new() { Top =  
1 }).Results.ToListAsync()  
  
// Inspect the returned hotels.  
Hotel hotel = searchResult.First().Record;  
Console.WriteLine("Found hotel description: " + hotel.Description);
```

💡 Tip

For more information on how to generate embeddings see [embedding generation](#).

Next steps

[Learn about the Vector Store data architecture](#)

[How to ingest data into a Vector Store](#)

The Semantic Kernel Vector Store data architecture (Preview)

Article • 10/16/2024

⚠ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Vector Store abstractions in Semantic Kernel are based on three main components: **vector stores**, **collections** and **records**. **Records** are contained by **collections**, and **collections** are contained by **vector stores**.

- A **vector store** maps to an instance of a database
- A **collection** is a collection of **records** including any index required to query or filter those **records**
- A **record** is an individual data entry in the database

Collections in different databases

The underlying implementation of what a collection is, will vary by connector and is influenced by how each database groups and indexes records. Most databases have a concept of a collection of records and there is a natural mapping between this concept and the Vector Store abstraction collection. Note that this concept may not always be referred to as a `collection` in the underlying database.

💡 Tip

For more information on what the underlying implementation of a collection is per connector, refer to [the documentation for each connector](#).

Defining your data model (Preview)

Article • 10/18/2024

⚠ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Semantic Kernel Vector Store connectors use a model first approach to interacting with databases.

All methods to upsert or get records use strongly typed model classes. The properties on these classes are decorated with attributes that indicate the purpose of each property.

💡 Tip

For an alternative to using attributes, refer to [defining your schema with a record definition](#).

💡 Tip

For an alternative to defining your own data model, refer to [using Vector Store abstractions without defining your own data model](#).

Here is an example of a model that is decorated with these attributes.

C#

```
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
    public string HotelName { get; set; }
```

```

[VectorStoreRecordData(IsFullTextSearchable = true)]
public string Description { get; set; }

[VectorStoreRecordVector(4, DistanceFunction.CosineDistance,
IndexKind.Hnsw)]
public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }

[VectorStoreRecordData(IsFilterable = true)]
public string[] Tags { get; set; }
}

```

Attributes

VectorStoreRecordKeyAttribute

Use this attribute to indicate that your property is the key of the record.

C#

```

[VectorStoreRecordKey]
public ulong HotelId { get; set; }

```

VectorStoreRecordKeyAttribute parameters

[] Expand table

Parameter	Required	Description
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where alternatives like <code>JsonPropertyNameAttribute</code> is supported.

💡 Tip

For more information on which connectors support `StoragePropertyName` and what alternatives are available, refer to [the documentation for each connector](#).

VectorStoreRecordDataAttribute

Use this attribute to indicate that your property contains general data that is not a key or a vector.

C#

```
[VectorStoreRecordData(IsFilterable = true)]
public string HotelName { get; set; }
```

VectorStoreRecordDataAttribute parameters

[] Expand table

Parameter	Required	Description
IsFilterable	No	Indicates whether the property should be indexed for filtering in cases where a database requires opting in to indexing per property. Default is false.
IsFullTextSearchable	No	Indicates whether the property should be indexed for full text search for databases that support full text search. Default is false.
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where alternatives like <code>JsonPropertyNameAttribute</code> is supported.

Tip

For more information on which connectors support StoragePropertyName and what alternatives are available, refer to [the documentation for each connector](#).

VectorStoreRecordVectorAttribute

Use this attribute to indicate that your property contains a vector.

C#

```
[VectorStoreRecordVector(Dimensions: 4, DistanceFunction.CosineDistance,
IndexKind.Hnsw)]
public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
```

VectorStoreRecordVectorAttribute parameters

[] Expand table

Parameter	Required	Description
Dimensions	Yes for collection create, optional otherwise	The number of dimensions that the vector has. This is typically required when creating a vector index for a collection.
IndexKind	No	The type of index to index the vector with. Default varies by vector store type.
DistanceFunction	No	The type of distance function to use when doing vector comparison during vector search over this vector. Default varies by vector store type.
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where alternatives like <code>JsonPropertyNameAttribute</code> is supported.

Common index kinds and distance function types are supplied as static values on the `Microsoft.SemanticKernel.Data.IndexKind` and `Microsoft.SemanticKernel.Data.DistanceFunction` classes. Individual Vector Store implementations may also use their own index kinds and distance functions, where the database supports unusual types.

💡 Tip

For more information on which connectors support `StoragePropertyName` and what alternatives are available, refer to [the documentation for each connector](#).

Defining your storage schema using a record definition (Experimental)

Article • 08/20/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is experimental, still in development and is subject to change.

Overview

The Semantic Kernel Vector Store connectors use a model first approach to interacting with databases and allows annotating data models with information that is needed for creating indexes or mapping data to the database schema.

Another way of providing this information is via record definitions, that can be defined and supplied separately to the data model. This can be useful in multiple scenarios:

- There may be a case where a developer wants to use the same data model with more than one configuration.
- There may be a case where the developer wants to store data using a very different schema to the model and wants to supply a custom mapper for converting between the data model and storage schema.
- There may be a case where a developer wants to use a built-in type, like a dict, or a optimized format like a dataframe and still wants to leverage the vector store functionality.

Here is an example of how to create a record definition.

C#

```
using Microsoft.SemanticKernel;

var hotelDefinition = new VectorStoreRecordDefinition
{
    Properties = new List<VectorStoreRecordProperty>
    {
        new VectorStoreRecordKeyProperty("HotelId", typeof(ulong)),
        new VectorStoreRecordDataProperty("HotelName", typeof(string)) {
            IsFilterable = true },
        new VectorStoreRecordDataProperty("Description", typeof(string)) {
            IsFullTextSearchable = true },
    }
}
```

```

        new VectorStoreRecordVectorProperty("DescriptionEmbedding",
typeof(float)) { Dimensions = 4, IndexKind = IndexKind.Hnsw,
DistanceFunction = DistanceFunction.CosineDistance },
    }
};

```

When creating a definition you always have to provide a name and type for each property in your schema, since this is required for index creation and data mapping.

To use the definition, pass it to the `GetCollection` method.

C#

```

var collection = vectorStore.GetCollection<ulong, Glossary>("skhotels",
hotelDefinition);

```

Record Property configuration classes

VectorStoreRecordKeyProperty

Use this class to indicate that your property is the key of the record.

C#

```

new VectorStoreRecordKeyProperty("HotelId", typeof(ulong)),

```

VectorStoreRecordKeyProperty configuration settings

[] Expand table

Parameter	Required	Description
DataModelPropertyName	Yes	The name of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
.PropertyType	Yes	The type of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where alternatives like <code>JsonPropertyNameAttribute</code> is supported.

💡 Tip

For more information on which connectors support StoragePropertyName and what alternatives are available, refer to [the documentation for each connector](#).

VectorStoreRecordDataProperty

Use this class to indicate that your property contains general data that is not a key or a vector.

C#

```
new VectorStoreRecordDataProperty("HotelName", typeof(string)) {  
    IsFilterable = true },
```

VectorStoreRecordDataProperty configuration settings

[+] Expand table

Parameter	Required	Description
DataModelPropertyName	Yes	The name of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
.PropertyType	Yes	The type of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
IsFilterable	No	Indicates whether the property should be indexed for filtering in cases where a database requires opting in to indexing per property. Default is false.
IsFullTextSearchable	No	Indicates whether the property should be indexed for full text search for databases that support full text search. Default is false.
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where alternatives like <code>JsonPropertyNameAttribute</code> is supported.

💡 Tip

For more information on which connectors support StoragePropertyName and what alternatives are available, refer to [the documentation for each connector](#).

VectorStoreRecordVectorProperty

Use this class to indicate that your property contains a vector.

C#

```
new VectorStoreRecordVectorProperty("DescriptionEmbedding", typeof(float)) {  
    Dimensions = 4, IndexKind = IndexKind.Hnsw, DistanceFunction =  
    DistanceFunction.CosineDistance },
```

VectorStoreRecordVectorProperty configuration settings

[+] Expand table

Parameter	Required	Description
DataModelPropertyName	Yes	The name of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
.PropertyType	Yes	The type of the property on the data model. Used by the built in mappers to automatically map between the storage schema and data model and for creating indexes.
Dimensions	Yes for collection create, optional otherwise	The number of dimensions that the vector has. This is typically required when creating a vector index for a collection.
IndexKind	No	The type of index to index the vector with. Default varies by vector store type.
DistanceFunction	No	The type of distance function to use when doing vector comparison during vector search over this vector. Default varies by vector store type.
StoragePropertyName	No	Can be used to supply an alternative name for the property in the database. Note that this parameter is not supported by all connectors, e.g. where

Parameter	Required	Description
		alternatives like <code>JsonPropertyNameAttribute</code> is supported.

💡 Tip

For more information on which connectors support `StoragePropertyName` and what alternatives are available, refer to [the documentation for each connector](#).

Using Vector Store abstractions without defining your own data model (Preview)

Article • 10/18/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Semantic Kernel Vector Store connectors use a model first approach to interacting with databases. This makes using the connectors easy and simple, since your data model reflects the schema of your database records and to add any additional schema information required, you can simply add attributes to your data model properties.

There are cases though where it is not desirable or possible to define your own data model. E.g. let's say that you do not know at compile time what your database schema looks like, and the schema is only provided via configuration. Creating a data model that reflects the schema would be impossible in this case.

To cater for this scenario, we provide a generic data model.

Generic Data Model

The generic data model is a class named `VectorStoreGenericDataModel` and is available in the `Microsoft.Extensions.VectorData.Abstractions` package.

To support any type of database, the type of the key of the `VectorStoreGenericDataModel` is specified via a generic parameter.

All other properties are divided into `Data` and `Vector` properties. Any property that is not a vector or a key is considered a data property. `Data` and `Vector` property sets are stored as string-keyed dictionaries of objects.

Supplying schema information when using the Generic Data Model

When using the generic data model, connectors still need to know what the database schema looks like. Without the schema information the connector would not be able to create a collection, or know how to map to and from the storage representation that each database uses.

A record definition can be used to provide the schema information. Unlike a data model, a record definition can be created from configuration at runtime, providing a solution for when schema information is not known at compile time.

💡 Tip

To see how to create a record definition, refer to [defining your schema with a record definition](#).

Example

To use the generic data model with a connector, simply specify it as your data model when creating a collection, and simultaneously provide a record definition.

C#

```
// Create the definition to define the schema.
VectorStoreRecordDefinition vectorStoreRecordDefinition = new()
{
    Properties = new List<VectorStoreRecordProperty>
    {
        new VectorStoreRecordKeyProperty("Key", typeof(string)),
        new VectorStoreRecordDataProperty("Term", typeof(string)),
        new VectorStoreRecordDataProperty("Definition", typeof(string)),
        new VectorStoreRecordVectorProperty("DefinitionEmbedding",
typeof(ReadOnlyMemory<float>)) { Dimensions = 1536 }
    }
};

// When getting your collection instance from a vector store instance
// specify the generic data model, using the appropriate key type for your
// database
// and also pass your record definition.
var genericDataModelCollection = vectorStore.GetCollection<string,
VectorStoreGenericDataModel<string>>(
    "glossary",
    vectorStoreRecordDefinition);

// Since we have schema information available from the record definition
// it's possible to create a collection with the right vectors, dimensions,
// indexes and distance functions.
await genericDataModelCollection.CreateCollectionIfNotExistsAsync();
```

```
// When retrieving a record from the collection, data and vectors can
// now be accessed via the Data and Vector dictionaries respectively.
var record = await genericDataModelCollection.GetAsync("SK");
Console.WriteLine(record.Data["Definition"])
```

When constructing a collection instance directly, the record definition is passed as an option. E.g. here is an example of constructing an Azure AI Search collection instance with the generic data model.

C#

```
new
AzureAISeachVectorStoreRecordCollection<VectorStoreGenericDataModel<string>
>(
    searchIndexClient,
    "glossary",
    new() { VectorStoreRecordDefinition = vectorStoreRecordDefinition });
```

Generating embeddings for Semantic Kernel Vector Store connectors

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Semantic Kernel supports generating embeddings using many popular AI services out of the box.

These services can be constructed directly or added to a dependency injection container and resolved from there.

Constructing an embedding generator

You can construct instances of the text embedding services provided by Semantic Kernel directly. They all implement the `ITextEmbeddingGenerationService` interface.

C#

```
// Constructing an Azure Open AI embedding generation service directly.  
ITextEmbeddingGenerationService azureOpenAITES = new  
AzureOpenAITextEmbeddingGenerationService(  
    "text-embedding-ada-002",  
    "https://myservice.openai.azure.com/",  
    "apikey");  
  
// Constructing an Ollama embedding generation service directly.  
ITextEmbeddingGenerationService olamaTES = new  
OllamaTextEmbeddingGenerationService(  
    "mxbai-embed-large",  
    new Uri("http://localhost:11434"));
```

You can also use helpers to register them with a dependency injection container.

C#

```
// Registering Google AI embedding generation service with a service collection.  
var services = new ServiceCollection();  
services.AddGoogleAIEmbeddingGeneration("text-embedding-004", "apiKey");
```

```
// Registering Mistral AI embedding generation service with the dependency
// injection container on
// the kernel builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddMistralTextEmbeddingGeneration("mistral-embed", "apiKey");
```

Generating embeddings

To use the `ITextEmbeddingGenerationService` you created, just call the `GenerateEmbeddingAsync` method on it.

Here is an example of generating embeddings when uploading records.

C#

```
public async Task GenerateEmbeddingsAndUpsertAsync(
    ITextEmbeddingGenerationService textEmbeddingGenerationService,
    IVectorStoreRecordCollection<ulong, Hotel> collection)
{
    // Upsert a record.
    string descriptionText = "A place where everyone can be happy.";
    ulong hotelId = 1;

    // Generate the embedding.
    ReadOnlyMemory<float> embedding =
        await
    textEmbeddingGenerationService.GenerateEmbeddingAsync(descriptionText);

    // Create a record and upsert with the already generated embedding.
    await collection.UpsertAsync(new Hotel
    {
        HotelId = hotelId,
        HotelName = "Hotel Happy",
        Description = descriptionText,
        DescriptionEmbedding = embedding,
        Tags = new[] { "luxury", "pool" }
    });
}
```

Here is an example of generating embeddings when searching.

C#

```
public async Task GenerateEmbeddingsAndSearchAsync(
    ITextEmbeddingGenerationService textEmbeddingGenerationService,
    IVectorStoreRecordCollection<ulong, Hotel> collection)
{
```

```

// Upsert a record.
string descriptionText = "Find me a hotel with happiness in mind.";

// Generate the embedding.
ReadOnlyMemory<float> searchEmbedding =
    await
textEmbeddingGenerationService.GenerateEmbeddingAsync(descriptionText);

// Search using the already generated embedding.
List<VectorSearchResult<Hotel>> searchResult = await
collection.VectorizedSearchAsync(searchEmbedding).ToListAsync();

// Print the first search result.
Console.WriteLine("Score for first result: " +
searchResult.FirstOrDefault()?.Score);
Console.WriteLine("Hotel description for first result: " +
searchResult.FirstOrDefault()?.Record.Description);
}

```

Embedding dimensions

Vector databases typically require you to specify the number of dimensions that each vector has when creating the collection. Different embedding models typically support generating vectors with different dimension sizes. E.g. Open AI `text-embedding-ada-002` generates vectors with 1536 dimensions. Some models also allow a developer to choose the number of dimensions they want in the output vector, e.g. Google `text-embedding-004` produces vectors with 768 dimension by default, but allows a developer to choose any number of dimensions between 1 and 768.

It is important to ensure that the vectors generated by the embedding model have the same number of dimensions as the matching vector in the database.

If creating a collection using the Semantic Kernel Vector Store abstractions, you need to specify the number of dimensions required for each vector property either via annotations or via the record definition. Here are examples of both setting the number of dimensions to 1536.

C#

```

[VectorStoreRecordVector(Dimensions: 1536)]
public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }

```

C#

```

new VectorStoreRecordVectorProperty("DescriptionEmbedding", typeof(float)) {

```

```
Dimensions = 1536 }
```

💡 Tip

For more information on how to annotate your data model, refer to [defining your data model](#).

💡 Tip

For more information on creating a record definition, refer to [defining your schema with a record definition](#).

Vector search using Semantic Kernel Vector Store connectors (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Semantic Kernel provides vector search capabilities as part of its Vector Store abstractions. This supports filtering and many other options, which this article will explain in more detail.

Vector Search

The `VectorizedSearchAsync` method allows searching using data that has already been vectorized. This method takes a vector and an optional `VectorSearchOptions` class as input. This method is available on the following interfaces:

1. `IVectorizedSearch<TRecord>`
2. `IVectorStoreRecordCollection< TKey, TRecord >`

Note that `IVectorStoreRecordCollection< TKey, TRecord >` inherits from `IVectorizedSearch<TRecord>`.

Assuming you have a collection that already contains data, you can easily search it. Here is an example using Qdrant.

C#

```
using Microsoft.SemanticKernel.Connectors.Qdrant;
using Microsoft.Extensions.VectorData;
using Qdrant.Client;

// Create a Qdrant VectorStore object and choose an existing collection that
already contains records.
IVectorStore vectorStore = new QdrantVectorStore(new
QdrantClient("localhost"));
IVectorStoreRecordCollection<ulong, Hotel> collection =
vectorStore.GetCollection<ulong, Hotel>("skhotels");

// Generate a vector for your search text, using your chosen embedding
```

```
generation implementation.  
// Just showing a placeholder method here for brevity.  
var searchVector = await GenerateEmbeddingAsync("I'm looking for a hotel  
where customer happiness is the priority.");  
  
// Do the search, passing an options object with a Top value to limit  
// resultst to the single top match.  
var searchResult = await collection.VectorizedSearchAsync(searchVector,  
new() { Top = 1 }).Results.ToListAsync();  
  
// Inspect the returned hotel.  
Hotel hotel = searchResult.First().Record;  
Console.WriteLine("Found hotel description: " + hotel.Description);
```

💡 Tip

For more information on how to generate embeddings see [embedding generation](#).

Supported Vector Types

`VectorizedSearchAsync` takes a generic type as the vector parameter. The types of vectors supported by each data store vary. See [the documentation for each connector](#) for the list of supported vector types.

It is also important for the search vector type to match the target vector that is being searched, e.g. if you have two vectors on the same record with different vector types, make sure that the search vector you supply matches the type of the specific vector you are targeting. See [VectorPropertyName](#) for how to pick a target vector if you have more than one per record.

Vector Search Options

The following options can be provided using the `VectorSearchOptions` class.

VectorPropertyName

The `VectorPropertyName` option can be used to specify the name of the vector property to target during the search. If none is provided, the first vector found on the data model or specified in the record definition will be used.

Note that when specifying the `VectorPropertyName`, use the name of the property as defined on the data model or in the record definition. Use this property name even if

the property may be stored under a different name in the vector store. The storage name may e.g. be different because of custom serialization settings.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.Connectors.Memory.InMemory;

var vectorStore = new InMemoryVectorStore();
var collection = vectorStore.GetCollection<int, Product>("skproducts");

// Create the vector search options and indicate that we want to search the
FeatureListEmbedding property.
var vectorSearchOptions = new VectorSearchOptions
{
    VectorPropertyName = nameof(Product.FeatureListEmbedding)
};

// This snippet assumes searchVector is already provided, having been
created using the embedding model of your choice.
var searchResult = await collection.VectorizedSearchAsync(searchVector,
vectorSearchOptions).Results.ToListAsync();

public sealed class Product
{
    [VectorStoreRecordKey]
    public int Key { get; set; }

    [VectorStoreRecordData]
    public string Description { get; set; }

    [VectorStoreRecordData]
    public List<string> FeatureList { get; set; }

    [VectorStoreRecordVector(1536)]
    public ReadOnlyMemory<float> DescriptionEmbedding { get; set; }

    [VectorStoreRecordVector(1536)]
    public ReadOnlyMemory<float> FeatureListEmbedding { get; set; }
}
```

Top and Skip

The `Top` and `Skip` options allow you to limit the number of results to the Top n results and to skip a number of results from the top of the resultset. Top and Skip can be used to do paging if you wish to retrieve a large number of results using separate calls.

C#

```
// Create the vector search options and indicate that we want to skip the
// first 40 results and then get the next 20.
var vectorSearchOptions = new VectorSearchOptions
{
    Top = 20,
    Skip = 40
};

// This snippet assumes searchVector is already provided, having been
// created using the embedding model of your choice.
var searchResult = await collection.VectorizedSearchAsync(searchVector,
vectorSearchOptions).Results.ToListAsync();
```

The default values for `Top` is 3 and `Skip` is 0.

IncludeVectors

The `IncludeVectors` option allows you to specify whether you wish to return vectors in the search results. If `false`, the vector properties on the returned model will be left null. Using `false` can significantly reduce the amount of data retrieved from the vector store during search, making searches more efficient.

The default value for `IncludeVectors` is `false`.

C#

```
// Create the vector search options and indicate that we want to include
// vectors in the search results.
var vectorSearchOptions = new VectorSearchOptions
{
    IncludeVectors = true
};
// This snippet assumes searchVector is already provided, having been
// created using the embedding model of your choice.
var searchResult = await collection.VectorizedSearchAsync(searchVector,
vectorSearchOptions).Results.ToListAsync()
```

VectorSearchFilter

The `VectorSearchFilter` option can be used to provide a filter for filtering the records in the chosen collection before applying the vector search.

This has multiple benefits:

- Reduce latency and processing cost, since only records remaining after filtering need to be compared with the search vector and therefore fewer vector comparisons have to be done.
- Limit the resultset for e.g. access control purposes, by excluding data that the user shouldn't have access to.

Note that in order for fields to be used for filtering, many vector stores require those fields to be indexed first. Some vector stores will allow filtering using any field, but may optionally allow indexing to improve filtering performance.

If creating a collection via the Semantic Kernel vector store abstractions and you wish to enable filtering on a field, set the `IsFilterable` property to true when defining your data model or when creating your record definition.

Tip

For more information on how to set the `IsFilterable` property, refer to [VectorStoreRecordDataAttribute parameters](#) or [VectorStoreRecordDataProperty configuration settings](#).

To create a filter use the `VectorSearchFilter` class. You can combine multiple filter clauses together in one `VectorSearchFilter`. All filter clauses are combined with `and`. Note that when providing a property name when constructing the filter, use the name of the property as defined on the data model or in the record definition. Use this property name even if the property may be stored under a different name in the vector store. The storage name may e.g. be different because of custom serialization settings.

C#

```
// Filter where Category == 'External Definitions' and Tags contain
// 'memory'.
var filter = new VectorSearchFilter()
    .EqualTo(nameof(Glossary.Category), "External Definitions")
    .AnyTagEqualTo(nameof(Glossary.Tags), "memory");

// Create the vector search options and set the filter on the options.
var vectorSearchOptions = new VectorSearchOptions
{
    Filter = filter
};

// This snippet assumes searchVector is already provided, having been
// created using the embedding model of your choice.
searchResult = await collection.VectorizedSearchAsync(searchVector,
vectorSearchOptions).Results.ToListAsync();
```

```
private sealed class Glossary
{
    [VectorStoreRecordKey]
    public ulong Key { get; set; }

    // Category is marked as filterable, since we want to filter using this
    // property.
    [VectorStoreRecordData(IsFilterable = true)]
    public string Category { get; set; }

    // Tags is marked as filterable, since we want to filter using this
    // property.
    [VectorStoreRecordData(IsFilterable = true)]
    public List<string> Tags { get; set; }

    [VectorStoreRecordData]
    public string Term { get; set; }

    [VectorStoreRecordData]
    public string Definition { get; set; }

    [VectorStoreRecordVector(1536)]
    public ReadOnlyMemory<float> DefinitionEmbedding { get; set; }
}
```

EqualTo filter clause

Use `EqualTo` for a direct comparison between property and value.

AnyTagEqualTo filter clause

Use `AnyTagEqualTo` to check if any of the strings, stored in a tag property in the vector store, contains a provided value. For a property to be considered a tag property, it needs to be a List, array or other enumerable of string.

Serialization of your data model to and from different stores (Experimental)

Article • 08/20/2024

Coming soon

More info coming soon.

Legacy Semantic Kernel Memory Stores

Article • 10/21/2024

Tip

We recommend using the Vector Store abstractions instead of the legacy Memory Stores. For more information on how to use the Vector Store abstractions start [here](#).

Semantic Kernel provides a set of Memory Store abstractions where the primary interface is `Microsoft.SemanticKernel.Memory.IMemoryStore`.

Memory Store vs Vector Store abstractions

As part of an effort to evolve and expand the vector storage and search capabilities of Semantic Kernel, we have released a new set of abstractions to replace the Memory Store abstractions. We are calling the replacement abstractions Vector Store abstractions. The purpose of both are similar, but their interfaces differ and the Vector Store abstractions provide expanded functionality.

[+] Expand table

Characteristic	Legacy Memory Stores	Vector Stores
Main Interface	<code>IMemoryStore</code>	<code>IVectorStore</code>
Abstractions nuget package	<code>Microsoft.SemanticKernel.Abstractions</code>	<code>Microsoft.Extensions.VectorData.Abstractions</code>
Naming Convention	{Provider}MemoryStore, e.g. <code>RedisMemoryStore</code>	{Provider}VectorStore, e.g. <code>RedisVectorStore</code>
Supports record upsert, get and delete	Yes	Yes
Supports collection create and delete	Yes	Yes

Characteristic	Legacy Memory Stores	Vector Stores
Supports vector search	Yes	Yes
Supports choosing your preferred vector search index and distance function	No	Yes
Supports multiple vectors per record	No	Yes
Supports custom schemas	No	Yes
Supports metadata pre-filtering for vector search	No	Yes
Supports vector search on non-vector databases by downloading the entire dataset onto the client and doing a local vector search	Yes	No

Available Memory Store connectors

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

[Expand table](#)

Service	C#	Python
Vector Database in Azure Cosmos DB for NoSQL	C# ↗	Python ↗

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

Migrating from Memory Stores to Vector Stores

If you wanted to migrate from using the Memory Store abstractions to the Vector Store abstractions there are various ways in which you can do this.

Use the existing collection with the Vector Store abstractions

The simplest way in many cases could be to just use the Vector Store abstractions to access a collection that was created using the Memory Store abstractions. In many cases this is possible, since the Vector Store abstraction allows you to choose the schema that you would like to use. The main requirement is to create a data model that matches the schema that the legacy Memory Store implementation used.

E.g. to access a collection created by the Azure AI Search Memory Store, you can use the following Vector Store data model.

C#

```
using Microsoft.Extensions.VectorData;

class VectorStoreRecord
{
    [VectorStoreRecordKey]
    public string Id { get; set; }

    [VectorStoreRecordData]
    public string Description { get; set; }

    [VectorStoreRecordData]
    public string Text { get; set; }

    [VectorStoreRecordData]
    public bool IsReference { get; set; }

    [VectorStoreRecordData]
    public string ExternalSourceName { get; set; }

    [VectorStoreRecordVector(VectorSize)]
    public ReadOnlyMemory<float> Embedding { get; set; }
}
```

💡 Tip

For more detailed examples on how to use the Vector Store abstractions to access collections created using a Memory Store, see [here ↗](#).

Create a new collection

In some cases migrating to a new collection may be preferable than using the existing collection directly. The schema that was chosen by the Memory Store may not match your requirements, especially with regards to filtering.

E.g. The Redis Memory store uses a schema with three fields:

- string metadata
- long timestamp
- float[] embedding

All data other than the embedding or timestamp is stored as a serialized json string in the Metadata field. This means that it is not possible to index the individual values and

filter on them. E.g. perhaps you may want to filter using the ExternalSourceName, but this is not possible while it is inside a json string.

In this case, it may be better to migrate the data to a new collection with a flat schema. There are two options here. You could create a new collection from your source data or simply map and copy the data from the old to the new. The first option may be more costly as you will need to regenerate the embeddings from the source data.

💡 Tip

For an example using Redis showing how to copy data from a collection created using the Memory Store abstractions to one created using the Vector Store abstractions see [here ↴](#).

Semantic Kernel Vector Store code samples (Preview)

Article • 10/18/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

End to end RAG sample with Vector Stores

This example is a standalone console application that demonstrates RAG using Semantic Kernel. The sample has the following characteristics:

1. Allows a choice of chat and embedding services
 2. Allows a choice of vector databases
 3. Reads the contents of one or more PDF files and creates a chunks for each section
 4. Generates embeddings for each text chunk and upserts it to the chosen vector database
 5. Registers the Vector Store as a Text Search plugin with the kernel
 6. Invokes the plugin to augment the prompt provided to the AI model with more context
- [End to end RAG demo ↗](#)

Simple Data Ingestion and Vector Search

For two very simple examples of how to do data ingestion into a vector store and do vector search, check out these two examples, which use Qdrant and InMemory vector stores to demonstrate their usage.

- [Simple Vector Search ↗](#)
- [Simple Data Ingestion ↗](#)

Common code with multiple stores

Vector stores may different in certain aspects, e.g. with regards to the types of their keys or the types of fields each support. Even so, it is possible to write code that is agnostic

to these differences.

For a data ingestion sample that demonstrates this, see:

- [MultiStore Data Ingestion ↗](#)

For a vector search sample demonstrating the same concept see the following samples.

Each of these samples are referencing the same common code, and just differ on the type of vector store they create to use with the common code.

- [Azure AI Search vector search with common code ↗](#)
- [InMemory vector search with common code ↗](#)
- [Qdrant vector search with common code ↗](#)
- [Redis vector search with common code ↗](#)

Supporting multiple vectors in the same record

The Vector Store abstractions support multiple vectors in the same record, for vector databases that support this. The following sample shows how to create some records with multiple vectors, and pick the desired target vector when doing a vector search.

- [Choosing a vector for search on a record with multiple vectors ↗](#)

Vector search with paging

When doing vector search with the Vector Store abstractions it's possible to use Top and Skip parameters to support paging, where e.g. you need to build a service that responds with a small set of results per request.

- [Vector search with paging ↗](#)

Warning

Not all vector databases support Skip functionality natively for vector searches, so some connectors may have to fetch Skip + Top records and skip on the client side to simulate this behavior.

Using the generic data model vs using a custom data model

It's possible to use the Vector Store abstractions without defining a data model and defining your schema via a record definition instead. This example shows how you can create a vector store using a custom model and read using the generic data model or vice versa.

- [Generic data model interop ↗](#)

 **Tip**

For more information about using the generic data model, refer to [using Vector Store abstractions without defining your own data model](#).

Out-of-the-box Vector Store connectors (Preview)

Article • 10/16/2024

⚠ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Semantic Kernel provides a number of out-of-the-box Vector Store integrations making it easy to get started with using Vector Stores. It also allows you to experiment with a free or locally hosted Vector Store and then easily switch to a service when scale requires it.

ⓘ Important

Some connectors are using SDKs that are not officially supported by Microsoft or by the Database provider. The *Officially supported SDK* column lists which are using officially supported SDKs and which are not.

[+] Expand table

Vector Store Connectors	C#	Python	Java	Officially supported SDK
Azure AI Search	✓	✓	In Development	✓
Cosmos DB	✓		In Development	✓
MongoDB		In Development	In Development	
Cosmos DB No SQL	✓	In Development	In Development	✓
In-Memory	✓	In Development	In Development	N/A
Pinecone	✓	In Development	In Development	C#: ✘ Python: ✓
Qdrant	✓	✓	In Development	✓

Vector Store Connectors	C#	Python	Java	Officially supported SDK
Redis	✓	✓	In Development	✓
Volatile (In-Memory)	Deprecated (use In-Memory)	✓	In Development	N/A
Weaviate	✓	In Development	In Development	N/A

Using the Azure AI Search Vector Store connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Azure AI Search Vector Store connector can be used to access and manage data in Azure AI Search. The connector has the following characteristics.

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

Feature Area	Support
Collection maps to	Azure AI Search Index
Supported key property types	string
Supported data property types	<ul style="list-style-type: none">• string• int• long• double• float• bool• DateTimeOffset• <i>and enumerables of each of these types</i>
Supported vector property types	ReadOnlyMemory<float>
Supported index types	<ul style="list-style-type: none">• Hnsw• Flat
Supported distance functions	<ul style="list-style-type: none">• CosineSimilarity• DotProductSimilarity• EuclideanDistance

Feature Area	Support
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	No, use <code>JsonSerializerOptions</code> and <code>JsonPropertyNameAttribute</code> instead. See here for more info.

Limitations

Notable Azure AI Search connector functionality limitations.

[] Expand table

Feature Area	Workaround
Configuring full text search analyzers during collection creation is not supported.	Use the Azure AI Search Client SDK directly for collection creation

Getting started

Add the Azure AI Search Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.AzureAISSearch --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Azure;
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
```

```
.AddAzureAIVectorStore(new Uri(azureAIUri), new  
AzureKeyCredential(secret));
```

C#

```
using Azure;  
using Microsoft.SemanticKernel;  
  
// Using IServiceCollection with ASP.NET Core.  
var builder = WebApplication.CreateBuilder(args);  
builder.Services.AddAzureAIVectorStore(new Uri(azureAIUri), new  
AzureKeyCredential(secret));
```

Extension methods that take no parameters are also provided. These require an instance of the Azure AI Search `SearchIndexClient` to be separately registered with the dependency injection container.

C#

```
using Azure;  
using Azure.Search.Documents.Indexes;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.SemanticKernel;  
  
// Using Kernel Builder.  
var kernelBuilder = Kernel.CreateBuilder();  
kernelBuilder.Services.AddSingleton<SearchIndexClient>(  
    sp => new SearchIndexClient(  
        new Uri(azureAIUri),  
        new AzureKeyCredential(secret)));  
kernelBuilder.AddAzureAIVectorStore();
```

C#

```
using Azure;  
using Azure.Search.Documents.Indexes;  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.SemanticKernel;  
  
// Using IServiceCollection with ASP.NET Core.  
var builder = WebApplication.CreateBuilder(args);  
builder.Services.AddSingleton<SearchIndexClient>(  
    sp => new SearchIndexClient(  
        new Uri(azureAIUri),  
        new AzureKeyCredential(secret)));  
builder.Services.AddAzureAIVectorStore();
```

You can construct an Azure AI Search Vector Store instance directly.

```
C#
```

```
using Azure;
using Azure.Search.Documents.Indexes;
using Microsoft.SemanticKernel.Connectors.AzureAISearch;

var vectorStore = new AzureAISeachVectorStore(
    new SearchIndexClient(
        new Uri(azureAISeachUri),
        new AzureKeyCredential(secret)));
```

It is possible to construct a direct reference to a named collection.

```
C#
```

```
using Azure;
using Azure.Search.Documents.Indexes;
using Microsoft.SemanticKernel.Connectors.AzureAISeach;

var collection = new AzureAISeachVectorStoreRecordCollection<Hotel>(
    new SearchIndexClient(new Uri(azureAISeachUri), new
AzureKeyCredential(secret)),
    "skhotels");
```

Data mapping

The default mapper used by the Azure AI Search connector when mapping data from the data model to storage is the one provided by the Azure AI Search SDK.

This mapper does a direct conversion of the list of properties on the data model to the fields in Azure AI Search and uses `System.Text.Json.JsonSerializer` to convert to the storage schema. This means that usage of the `JsonPropertyNameAttribute` is supported if a different storage name to the data model property name is required.

It is also possible to use a custom `JsonSerializerOptions` instance with a customized property naming policy. To enable this, the `JsonSerializerOptions` must be passed to both the `SearchIndexClient` and the `AzureAISeachVectorStoreRecordCollection` on construction.

```
C#
```

```
var jsonSerializerOptions = new JsonSerializerOptions { PropertyNamingPolicy
= JsonNamingPolicy.SnakeCaseUpper };
var collection = new AzureAISeachVectorStoreRecordCollection<Hotel>(
    new SearchIndexClient(
        new Uri(azureAISeachUri),
```

```
    new AzureKeyCredential(secret),
    new() { Serializer = new JsonObjectSerializer(jsonSerializerOptions)
}),
"skhotels",
new() { JsonSerializerOptions = jsonSerializerOptions });
```

Using the Azure CosmosDB MongoDB Vector Store connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Azure CosmosDB MongoDB Vector Store connector can be used to access and manage data in Azure CosmosDB MongoDB. The connector has the following characteristics.

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

Feature Area	Support
Collection maps to	Azure Cosmos DB MongoDB Collection + Index
Supported key property types	string
Supported data property types	<ul style="list-style-type: none">• string• int• long• double• float• decimal• bool• DateTime• <i>and enumerables of each of these types</i>
Supported vector property types	<ul style="list-style-type: none">• <code>ReadOnlyMemory<float></code>• <code>ReadOnlyMemory<double></code>
Supported index types	<ul style="list-style-type: none">• Hnsw• IvfFlat
Supported distance functions	<ul style="list-style-type: none">• CosineDistance• DotProductSimilarity

Feature Area	Support
	<ul style="list-style-type: none"> • EuclideanDistance
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable supported?	No
StoragePropertyName supported?	No, use <code>BsonElementAttribute</code> instead. See here for more info.

Getting started

Add the Azure CosmosDB MongoDB Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.AzureCosmosDBMongoDB
--prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddAzureCosmosDBMongoDBVectorStore(connectionString, databaseName);
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddAzureCosmosDBMongoDBVectorStore(connectionString,
databaseName);
```

Extension methods that take no parameters are also provided. These require an instance of `MongoDB.Driver.IMongoDatabase` to be separately registered with the dependency injection container.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using MongoDB.Driver;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Services.AddSingleton<IMongoDatabase>(
    sp =>
{
    var mongoClient = new MongoClient(connectionString);
    return mongoClient.GetDatabase(databaseName);
});
kernelBuilder.AddAzureCosmosDBMongoDBVectorStore();
```

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using MongoDB.Driver;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IMongoDatabase>(
    sp =>
{
    var mongoClient = new MongoClient(connectionString);
    return mongoClient.GetDatabase(databaseName);
});
builder.Services.AddAzureCosmosDBMongoDBVectorStore();
```

You can construct an Azure CosmosDB MongoDB Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Connectors.AzureCosmosDBMongoDB;
using MongoDB.Driver;

var mongoClient = new MongoClient(connectionString);
var database = mongoClient.GetDatabase(databaseName);
var vectorStore = new AzureCosmosDBMongoDBVectorStore(database);
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.SemanticKernel.Connectors.AzureCosmosDBMongoDB;
using MongoDB.Driver;

var mongoClient = new MongoClient(connectionString);
var database = mongoClient.GetDatabase(databaseName);
var collection = new AzureCosmosDBMongoDBVectorStoreRecordCollection<Hotel>(
    database,
    "skhotels");
```

Data mapping

The Azure CosmosDB MognoDB Vector Store connector provides a default mapper when mapping data from the data model to storage.

This mapper does a direct conversion of the list of properties on the data model to the fields in Azure CosmosDB MongoDB and uses `MongoDB.Bson.Serialization` to convert to the storage schema. This means that usage of the

`MongoDB.Bson.Serialization.Attributes.BsonElement` is supported if a different storage name to the data model property name is required. The only exception is the key of the record which is mapped to a database field named `_id`, since all CosmosDB MongoDB records must use this name for ids.

Property name override

For data properties and vector properties, you can provide override field names to use in storage that is different to the property names on the data model. This is not supported for keys, since a key has a fixed name in MongoDB.

The property name override is done by setting the `BsonElement` attribute on the data model properties.

Here is an example of a data model with `BsonElement` set.

C#

```
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [BsonElement("hotel_name")]
}
```

```
[VectorStoreRecordData(IsFilterable = true)]
public string HotelName { get; set; }

[BsonElement("hotel_description")]
[VectorStoreRecordData(IsFullTextSearchable = true)]
public string Description { get; set; }

[BsonElement("hotel_description_embedding")]
[VectorStoreRecordVector(4, DistanceFunction.CosineDistance,
IndexKind.Hnsw)]
public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

Using the Azure CosmosDB NoSQL Vector Store connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Azure CosmosDB NoSQL Vector Store connector can be used to access and manage data in Azure CosmosDB NoSQL. The connector has the following characteristics.

[+] [Expand table](#)

Feature Area	Support
Collection maps to	Azure Cosmos DB NoSQL Container
Supported key property types	<ul style="list-style-type: none">• string• AzureCosmosDBNoSQLCompositeKey
Supported data property types	<ul style="list-style-type: none">• string• int• long• double• float• bool• DateTimeOffset• <i>and enumerables of each of these types</i>
Supported vector property types	<ul style="list-style-type: none">• ReadOnlyMemory<float>• ReadOnlyMemory<byte>• ReadOnlyMemory<sbyte>• ReadOnlyMemory<Half>
Supported index types	<ul style="list-style-type: none">• Flat• QuantizedFlat• DiskAnn

Feature Area	Support
Supported distance functions	<ul style="list-style-type: none"> • CosineSimilarity • DotProductSimilarity • EuclideanDistance
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	No, use <code>JsonSerializerOptions</code> and <code>JsonPropertyNameAttribute</code> instead. See here for more info.

Getting started

Add the Azure CosmosDB NoSQL Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddAzureCosmosDBNoSQLVectorStore(connectionString, databaseName);
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddAzureCosmosDBNoSQLVectorStore(connectionString,
databaseName);
```

Extension methods that take no parameters are also provided. These require an instance of `Microsoft.Azure.Cosmos.Database` to be separately registered with the dependency injection container.

C#

```
using Microsoft.Azure.Cosmos;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Services.AddSingleton<Database>(
    sp =>
{
    var cosmosClient = new CosmosClient(connectionString);
    return cosmosClient.GetDatabase(databaseName);
});
kernelBuilder.AddAzureCosmosDBNoSQLVectorStore();
```

C#

```
using Microsoft.Azure.Cosmos;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<Database>(
    sp =>
{
    var cosmosClient = new CosmosClient(connectionString);
    return cosmosClient.GetDatabase(databaseName);
});
builder.Services.AddAzureCosmosDBNoSQLVectorStore();
```

You can construct an Azure CosmosDB NoSQL Vector Store instance directly.

C#

```
using Microsoft.Azure.Cosmos;
using Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL;

var cosmosClient = new CosmosClient(connectionString);
var database = cosmosClient.GetDatabase(databaseName);
var vectorStore = new AzureCosmosDBNoSQLVectorStore(database);
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.Azure.Cosmos;
using Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL;

var cosmosClient = new CosmosClient(connectionString);
var database = cosmosClient.GetDatabase(databaseName);
var collection = new AzureCosmosDBNoSQLVectorStoreRecordCollection<Hotel>(
    database,
    "skhotels");
```

Data mapping

The Azure CosmosDB NoSQL Vector Store connector provides a default mapper when mapping from the data model to storage.

This mapper does a direct conversion of the list of properties on the data model to the fields in Azure CosmosDB NoSQL and uses `System.Text.Json.JsonSerializer` to convert to the storage schema. This means that usage of the `JsonPropertyNameAttribute` is supported if a different storage name to the data model property name is required. The only exception is the key of the record which is mapped to a database field named `id`, since all CosmosDB NoSQL records must use this name for ids.

It is also possible to use a custom `JsonSerializerOptions` instance with a customized property naming policy. To enable this, the `JsonSerializerOptions` must be passed to the `AzureCosmosDBNoSQLVectorStoreRecordCollection` on construction.

C#

```
using System.Text.Json;
using Microsoft.Azure.Cosmos;
using Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL;

var jsonSerializerOptions = new JsonSerializerOptions { PropertyNamingPolicy =
    JsonNamingPolicy.SnakeCaseUpper };

var cosmosClient = new CosmosClient(connectionString);

var database = cosmosClient.GetDatabase(databaseName);
var collection = new AzureCosmosDBNoSQLVectorStoreRecordCollection<Hotel>(
    database,
    "skhotels",
    new() { JsonSerializerOptions = jsonSerializerOptions });
```

Using the above custom `JsonSerializerOptions` which is using `SnakeCaseUpper`, the following data model will be mapped to the below json.

C#

```
using System.Text.Json.Serialization;
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true)]
    public string Description { get; set; }

    [JsonPropertyName("HOTEL_DESCRIPTION_EMBEDDING")]
    [VectorStoreRecordVector(4, DistanceFunction.EuclideanDistance,
    IndexKind.QuantizedFlat)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

JSON

```
{
    "id": 1,
    "HOTEL_NAME": "Hotel Happy",
    "DESCRIPTION": "A place where everyone can be happy.",
    "HOTEL_DESCRIPTION_EMBEDDING": [0.9, 0.1, 0.1, 0.1],
}
```

Using the In-Memory connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The In-Memory Vector Store connector is a Vector Store implementation provided by Semantic Kernel that uses no external database and stores data in memory. This Vector Store is useful for prototyping scenarios or where high-speed in-memory operations are required.

The connector has the following characteristics.

[] [Expand table](#)

Feature Area	Support
Collection maps to	In-memory dictionary
Supported key property types	Any type that can be compared
Supported data property types	Any type
Supported vector property types	ReadOnlyMemory<float>
Supported index types	N/A
Supported distance functions	N/A
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable	Yes

Feature Area	Support
supported?	
StoragePropertyName supported?	No, since storage is in-memory and data reuse is therefore not possible, custom naming is not applicable.

Getting started

Add the Semantic Kernel Core nuget package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.InMemory --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddInMemoryVectorStore();
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddInMemoryVectorStore();
```

You can construct an InMemory Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Connectors.InMemory;

var vectorStore = new InMemoryVectorStore();
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.SemanticKernel.Connectors.InMemory;

var collection = new InMemoryVectorStoreRecordCollection<string, Hotel>
("skhotels");
```

Using the Pinecone connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Pinecone Vector Store connector can be used to access and manage data in Pinecone. The connector has the following characteristics.

 Expand table

Feature Area	Support
Collection maps to	Pinecone serverless Index
Supported key property types	string
Supported data property types	<ul style="list-style-type: none">• string• int• long• double• float• bool• decimal• <i>enumerables of type string</i>
Supported vector property types	ReadOnlyMemory<float>
Supported index types	PGA (Pinecone Graph Algorithm)
Supported distance functions	<ul style="list-style-type: none">• CosineSimilarity• DotProductSimilarity• EuclideanDistance
Supports multiple vectors in a record	No
IsFilterable supported?	Yes
IsFullTextSearchable supported?	No

Feature Area	Support
StoragePropertyName supported?	Yes

Limitations

Notable Pinecone connector functionality limitations.

[Expand table](#)

Feature Area	Workaround
Vector Search is not yet implemented	No workaround at this stage, implementation to follow soon

Getting started

Add the Pinecone Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.Pinecone --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddPineconeVectorStore(pineconeApiKey);
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddPineconeVectorStore(pineconeApiKey);
```

Extension methods that take no parameters are also provided. These require an instance of the `PineconeClient` to be separately registered with the dependency injection container.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using PineconeClient = Pinecone.PineconeClient;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Services.AddSingleton<PineconeClient>(
    sp => new PineconeClient(pineconeApiKey));
kernelBuilder.AddPineconeVectorStore();
```

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using PineconeClient = Pinecone.PineconeClient;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<PineconeClient>(
    sp => new PineconeClient(pineconeApiKey));
builder.Services.AddPineconeVectorStore();
```

You can construct a Pinecone Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Connectors.Pinecone;
using PineconeClient = Pinecone.PineconeClient;

var vectorStore = new PineconeVectorStore(
    new PineconeClient(pineconeApiKey));
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.SemanticKernel.Connectors.Pinecone;
using PineconeClient = Pinecone.PineconeClient;

var collection = new PineconeVectorStoreRecordCollection<Hotel>(
    new PineconeClient(pineconeApiKey),
    "skhotels");
```

Index Namespace

The Vector Store abstraction does not support a multi tiered record grouping mechanism. Collections in the abstraction map to a Pinecone serverless index and no second level exists in the abstraction. Pinecone does support a second level of grouping called namespaces.

By default the Pinecone connector will pass null as the namespace for all operations. However it is possible to pass a single namespace to the Pinecone collection when constructing it and use this instead for all operations.

C#

```
using Microsoft.SemanticKernel.Connectors.Pinecone;
using PineconeClient = Pinecone.PineconeClient;

var collection = new PineconeVectorStoreRecordCollection<Hotel>(
    new PineconeClient(pineconeApiKey),
    "skhotels",
    new() { IndexNamespace = "seasidehotels" });
```

Data mapping

The Pinecone connector provides a default mapper when mapping data from the data model to storage. Pinecone requires properties to be mapped into id, metadata and values groupings. The default mapper uses the model annotations or record definition to determine the type of each property and to do this mapping.

- The data model property annotated as a key will be mapped to the Pinecone id property.
- The data model properties annotated as data will be mapped to the Pinecone metadata object.
- The data model property annotated as a vector will be mapped to the Pinecone vector property.

Property name override

For data properties, you can provide override field names to use in storage that is different to the property names on the data model. This is not supported for keys, since a key has a fixed name in Pinecone. It is also not supported for vectors, since the vector is stored under a fixed name `values`. The property name override is done by setting the `StoragePropertyName` option via the data model attributes or record definition.

Here is an example of a data model with `StoragePropertyName` set on its attributes and how that will be represented in Pinecone.

C#

```
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true, StoragePropertyName =
    "hotel_name")]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true, StoragePropertyName =
    = "hotel_description")]
    public string Description { get; set; }

    [VectorStoreRecordVector(Dimensions: 4, DistanceFunction.CosineDistance,
    IndexKind.Hnsw)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

JSON

```
{
  "id": "h1",
  "values": [0.9, 0.1, 0.1, 0.1],
  "metadata": { "hotel_name": "Hotel Happy", "hotel_description": "A place
where everyone can be happy." }
}
```

Using the Qdrant connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Qdrant Vector Store connector can be used to access and manage data in Qdrant. The connector has the following characteristics.

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

Feature Area	Support
Collection maps to	Qdrant collection with payload indices for filterable data fields
Supported key property types	<ul style="list-style-type: none">ulongGuid
Supported data property types	<ul style="list-style-type: none">stringintlongdoublefloatbool<i>and enumerables of each of these types</i>
Supported vector property types	ReadOnlyMemory<float>
Supported index types	Hnsw
Supported distance functions	<ul style="list-style-type: none">CosineSimilarityDotProductSimilarityEuclideanDistanceManhattanDistance
Supports multiple vectors in a record	Yes (configurable)

Feature Area	Support
IsFilterable supported?	Yes
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	Yes

Getting started

Add the Qdrant Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.Qdrant --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddQdrantVectorStore("localhost");
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddQdrantVectorStore("localhost");
```

Extension methods that take no parameters are also provided. These require an instance of the `Qdrant.Client.QdrantClient` class to be separately registered with the dependency injection container.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
```

```
using Qdrant.Client;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Services.AddSingleton<QdrantClient>(sp => new
QdrantClient("localhost"));
kernelBuilder.AddQdrantVectorStore();
```

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using Qdrant.Client;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<QdrantClient>(sp => new
QdrantClient("localhost"));
builder.Services.AddQdrantVectorStore();
```

You can construct a Qdrant Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Connectors.Qdrant;
using Qdrant.Client;

var vectorStore = new QdrantVectorStore(new QdrantClient("localhost"));
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.SemanticKernel.Connectors.Qdrant;
using Qdrant.Client;

var collection = new QdrantVectorStoreRecordCollection<Hotel>(
    new QdrantClient("localhost"),
    "skhotels");
```

Data mapping

The Qdrant connector provides a default mapper when mapping data from the data model to storage. Qdrant requires properties to be mapped into id, payload and vector(s) groupings. The default mapper uses the model annotations or record definition to determine the type of each property and to do this mapping.

- The data model property annotated as a key will be mapped to the Qdrant point id.
- The data model properties annotated as data will be mapped to the Qdrant point payload object.
- The data model properties annotated as vectors will be mapped to the Qdrant point vector object.

Property name override

For data properties and vector properties (if using named vectors mode), you can provide override field names to use in storage that is different to the property names on the data model. This is not supported for keys, since a key has a fixed name in Qdrant. It is also not supported for vectors in *single unnamed vector* mode, since the vector is stored under a fixed name.

The property name override is done by setting the `StoragePropertyName` option via the data model attributes or record definition.

Here is an example of a data model with `StoragePropertyName` set on its attributes and how that will be represented in Qdrant.

C#

```
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true, StoragePropertyName =
"hotel_name")]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true, StoragePropertyName =
= "hotel_description")]
    public string Description { get; set; }

    [VectorStoreRecordVector(4, DistanceFunction.CosineDistance,
IndexKind.Hnsw, StoragePropertyName = "hotel_description_embedding")]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

JSON

```
{
  "id": 1,
```

```
    "payload": { "hotel_name": "Hotel Happy", "hotel_description": "A place where everyone can be happy." },
    "vector": {
        "hotel_description_embedding": [0.9, 0.1, 0.1, 0.1],
    }
}
```

Qdrant vector modes

Qdrant supports two modes for vector storage and the Qdrant Connector with default mapper supports both modes. The default mode is *single unnamed vector*.

Single unnamed vector

With this option a collection may only contain a single vector and it will be unnamed in the storage model in Qdrant. Here is an example of how an object is represented in Qdrant when using *single unnamed vector* mode:

C#

```
new Hotel
{
    HotelId = 1,
    HotelName = "Hotel Happy",
    Description = "A place where everyone can be happy.",
    DescriptionEmbedding = new float[4] { 0.9f, 0.1f, 0.1f, 0.1f }
};
```

JSON

```
{
    "id": 1,
    "payload": { "HotelName": "Hotel Happy", "Description": "A place where everyone can be happy." },
    "vector": [0.9, 0.1, 0.1, 0.1]
}
```

Named vectors

If using the named vectors mode, it means that each point in a collection may contain more than one vector, and each will be named. Here is an example of how an object is represented in Qdrant when using *named vectors* mode:

C#

```
new Hotel
{
    HotelId = 1,
    HotelName = "Hotel Happy",
    Description = "A place where everyone can be happy.",
    HotelNameEmbedding = new float[4] { 0.9f, 0.5f, 0.5f, 0.5f }
    DescriptionEmbedding = new float[4] { 0.9f, 0.1f, 0.1f, 0.1f }
};
```

JSON

```
{
    "id": 1,
    "payload": { "HotelName": "Hotel Happy", "Description": "A place where
everyone can be happy." },
    "vector": {
        "HotelNameEmbedding": [0.9, 0.5, 0.5, 0.5],
        "DescriptionEmbedding": [0.9, 0.1, 0.1, 0.1],
    }
}
```

To enable named vectors mode, pass this as an option when constructing a Vector Store or collection. The same options can also be passed to any of the provided dependency injection container extension methods.

C#

```
using Microsoft.SemanticKernel.Connectors.Qdrant;
using Qdrant.Client;

var vectorStore = new QdrantVectorStore(
    new QdrantClient("localhost"),
    new() { HasNamedVectors = true });

var collection = new QdrantVectorStoreRecordCollection<Hotel>(
    new QdrantClient("localhost"),
    "skhotels",
    new() { HasNamedVectors = true });
```

Using the Redis connector (Experimental)

Article • 08/20/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is experimental, still in development and is subject to change.

Overview

The Redis Vector Store connector can be used to access and manage data in Redis. The connector supports both Hashes and JSON modes and which mode you pick will determine what other features are supported.

The connector has the following characteristics.

[] Expand table

Feature Area	Support
Collection maps to	Redis index with prefix set to <collectionname>:
Supported key property types	string
Supported data property types	<p>When using Hashes:</p> <ul style="list-style-type: none">• string• int• uint• long• ulong• double• float• bool <p>When using JSON:</p> <p>Any types serializable to JSON</p>
Supported vector property types	<ul style="list-style-type: none">• <code>ReadOnlyMemory<float></code>• <code>ReadOnlyMemory<double></code>

Feature Area	Support
Supported index types	<ul style="list-style-type: none"> • Hnsw • Flat
Supported distance functions	<ul style="list-style-type: none"> • CosineSimilarity • DotProductSimilarity • EuclideanDistance
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	<p>When using Hashes: Yes</p> <p>When using JSON: No, use <code>JsonSerializerOptions</code> and <code>JsonPropertyNameAttribute</code> instead. See here for more info.</p>

Getting started

Add the Redis Vector Store connector nuget package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.Redis --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddRedisVectorStore("localhost:6379");
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddRedisVectorStore("localhost:6379");
```

Extension methods that take no parameters are also provided. These require an instance of the Redis `IDatabase` to be separately registered with the dependency injection container.

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using StackExchange.Redis;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.Services.AddSingleton<IDatabase>(sp =>
ConnectionMultiplexer.Connect("localhost:6379").GetDatabase());
kernelBuilder.AddRedisVectorStore();
```

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using StackExchange.Redis;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IDatabase>(sp =>
ConnectionMultiplexer.Connect("localhost:6379").GetDatabase());
builder.Services.AddRedisVectorStore();
```

You can construct a Redis Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Connectors.Redis;
using StackExchange.Redis;

var vectorStore = new
RedisVectorStore(ConnectionMultiplexer.Connect("localhost:6379").GetDatabase
());
```

It is possible to construct a direct reference to a named collection. When doing so, you have to choose between the JSON or Hashes instance depending on how you wish to

store data in Redis.

C#

```
using Microsoft.SemanticKernel.Connectors.Redis;
using StackExchange.Redis;

// Using Hashes.
var hashesCollection = new RedisHashSetVectorStoreRecordCollection<Hotel>(
    ConnectionMultiplexer.Connect("localhost:6379").GetDatabase(),
    "skhotelshashes");
```

C#

```
using Microsoft.SemanticKernel.Connectors.Redis;
using StackExchange.Redis;

// Using JSON.
var jsonCollection = new RedisJsonVectorStoreRecordCollection<Hotel>(
    ConnectionMultiplexer.Connect("localhost:6379").GetDatabase(),
    "skhotelsjson");
```

When constructing a `RedisVectorStore` or registering it with the dependency injection container, it's possible to pass a `RedisVectorStoreOptions` instance that configures the preferred storage type / mode used: Hashes or JSON. If not specified, the default is JSON.

C#

```
using Microsoft.SemanticKernel.Connectors.Redis;
using StackExchange.Redis;

var vectorStore = new RedisVectorStore(
    ConnectionMultiplexer.Connect("localhost:6379").GetDatabase(),
    new() { StorageType = RedisStorageType.HashSet });
```

Index prefixes

Redis uses a system of key prefixing to associate a record with an index. When creating an index you can specify one or more prefixes to use with that index. If you want to associate a record with that index, you have to add the prefix to the key of that record.

E.g. If you create a index called `skhotelsjson` with a prefix of `skhotelsjson:`, when setting a record with key `h1`, the record key will need to be prefixed like this `skhotelsjson:h1` to be added to the index.

When creating a new collection using the Redis connector, the connector will create an index in Redis with a prefix consisting of the collection name and a colon, like this `<collectionname>:`. By default, the connector will also prefix all keys with the this prefix when doing record operations like Get, Upsert, and Delete.

If you didn't want to use a prefix consisting of the collection name and a colon, it is possible to switch off the prefixing behavior and pass in the fully prefixed key to the record operations.

```
C#
```

```
using Microsoft.SemanticKernel.Connectors.Redis;
using StackExchange.Redis;

var collection = new RedisJsonVectorStoreRecordCollection<Hotel>(
    ConnectionMultiplexer.Connect("localhost:6379").GetDatabase(),
    "skhotelsjson",
    new() { PrefixCollectionNameToKeyNames = false });

await collection.GetAsync("myprefix_h1");
```

Data mapping

Redis supports two modes for storing data: JSON and Hashes. The Redis connector supports both storage types, and mapping differs depending on the chosen storage type.

Data mapping when using the JSON storage type

When using the JSON storage type, the Redis connector will use `System.Text.Json.JsonSerializer` to do mapping. Since Redis stores records with a separate key and value, the mapper will serialize all properties except for the key to a JSON object and use that as the value.

Usage of the `JsonPropertyNameAttribute` is supported if a different storage name to the data model property name is required. It is also possible to use a custom `JsonSerializerOptions` instance with a customized property naming policy. To enable this, the `JsonSerializerOptions` must be passed to the `RedisJsonVectorStoreRecordCollection` on construction.

```
C#
```

```

var jsonSerializerOptions = new JsonSerializerOptions { PropertyNamingPolicy
= JsonNamingPolicy.SnakeCaseUpper };
var collection = new RedisJsonVectorStoreRecordCollection<Hotel>(
    ConnectionMultiplexer.Connect("localhost:6379").GetDatabase(),
    "skhotelsjson",
    new() { JsonSerializerOptions = jsonSerializerOptions });

```

Since a naming policy of snake case upper was chosen, here is an example of how this data type will be set in Redis. Also note the use of `JsonPropertyNameAttribute` on the `Description` property to further customize the storage naming.

C#

```

using System.Text.Json.Serialization;
using Microsoft.SemanticKernel.Data;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
    public string HotelName { get; set; }

    [JsonPropertyName("HOTEL_DESCRIPTION")]
    [VectorStoreRecordData(IsFullTextSearchable = true)]
    public string Description { get; set; }

    [VectorStoreRecordVector(Dimensions: 4, IndexKind.Hnsw,
    DistanceFunction.CosineDistance)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}

```

redis

```

JSON.SET skhotelsjson:h1 $ '{ "HOTEL_NAME": "Hotel Happy",
"HOTEL_DESCRIPTION": "A place where everyone can be happy.",
"DESCRIPTION_EMBEDDING": [0.9, 0.1, 0.1, 0.1] }'

```

Data mapping when using the Hashes storage type

When using the Hashes storage type, the Redis connector provides its own mapper to do mapping. This mapper will map each property to a field-value pair as supported by the Redis `HSET` command.

For data properties and vector properties, you can provide override field names to use in storage that is different to the property names on the data model. This is not supported for keys, since keys cannot be named in Redis.

Property name overriding is done by setting the `StoragePropertyName` option via the data model attributes or record definition.

Here is an example of a data model with `StoragePropertyName` set on its attributes and how these are set in Redis.

C#

```
using Microsoft.SemanticKernel.Data;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true, StoragePropertyName =
"hotel_name")]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true, StoragePropertyName =
= "hotel_description")]
    public string Description { get; set; }

    [VectorStoreRecordVector(Dimensions: 4, IndexKind.Hnsw,
DistanceFunction.CosineDistance, StoragePropertyName =
"hotel_description_embedding")]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

redis

```
HSET skhotelshashes:h1 hotel_name "Hotel Happy" hotel_description 'A place
where everyone can be happy.' hotel_description_embedding <vector_bytes>
```

Using the Volatile (In-Memory) connector (Preview)

Article • 10/16/2024

⚠ Warning

The C# VolatileVectorStore is obsolete and has been replaced with a new package.

See [InMemory Connector](#)

Overview

The Volatile Vector Store connector is a Vector Store implementation provided by Semantic Kernel that uses no external database and stores data in memory. This Vector Store is useful for prototyping scenarios or where high-speed in-memory operations are required.

The connector has the following characteristics.

[] Expand table

Feature Area	Support
Collection maps to	In-memory dictionary
Supported key property types	Any type that can be compared
Supported data property types	Any type
Supported vector property types	ReadOnlyMemory<float>
Supported index types	N/A
Supported distance functions	N/A
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes

Feature Area	Support
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	No, since storage is volatile and data reuse is therefore not possible, custom naming is not useful and not supported.

Getting started

Add the Semantic Kernel Core nuget package to your project.

.NET CLI

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

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddVolatileVectorStore();
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddVolatileVectorStore();
```

You can construct a Volatile Vector Store instance directly.

C#

```
using Microsoft.SemanticKernel.Data;

var vectorStore = new VolatileVectorStore();
```

It is possible to construct a direct reference to a named collection.

C#

```
using Microsoft.SemanticKernel.Data;

var collection = new VolatileVectorStoreRecordCollection<string, Hotel>
("skhotels");
```

Using the Weaviate Vector Store connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Weaviate Vector Store connector can be used to access and manage data in Weaviate. The connector has the following characteristics.

↔ [Expand table](#)

Feature Area	Support
Collection maps to	Weaviate Collection
Supported key property types	Guid
Supported data property types	<ul style="list-style-type: none">• string• byte• short• int• long• double• float• decimal• bool• DateTime• DateTimeOffset• Guid• <i>and enumerables of each of these types</i>
Supported vector property types	<ul style="list-style-type: none">• <code>ReadOnlyMemory<float></code>• <code>ReadOnlyMemory<double></code>
Supported index types	<ul style="list-style-type: none">• Hnsw• Flat

Feature Area	Support
	<ul style="list-style-type: none"> • Dynamic
Supported distance functions	<ul style="list-style-type: none"> • CosineDistance • DotProductSimilarity • EuclideanSquaredDistance • Hamming • ManhattanDistance
Supports multiple vectors in a record	Yes
IsFilterable supported?	Yes
IsFullTextSearchable supported?	Yes
StoragePropertyName supported?	No, use <code>JsonSerializerOptions</code> and <code>JsonPropertyNameAttribute</code> instead. See here for more info.

Limitations

Notable Weaviate connector functionality limitations.

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

Feature Area	Workaround
Using the 'vector' property for single vector objects is not supported	Use of the 'vectors' property is supported instead.

Getting started

Add the Weaviate Vector Store connector NuGet package to your project.

.NET CLI

```
dotnet add package Microsoft.SemanticKernel.Connectors.Weaviate --prerelease
```

You can add the vector store to the dependency injection container available on the `KernelBuilder` or to the `IServiceCollection` dependency injection container using extension methods provided by Semantic Kernel. The Weaviate vector store uses an `HttpClient` to communicate with the Weaviate service. There are two options for

providing the URL/endpoint for the Weaviate service. It can be provided via options or by setting the base address of the `HttpClient`.

This first example shows how to set the service URL via options. Also note that these methods will retrieve an `HttpClient` instance for making calls to the Weaviate service from the dependency injection service provider.

C#

```
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel
    .CreateBuilder()
    .AddWeaviateVectorStore(options: new() { Endpoint = new
Uri("http://localhost:8080/v1/") });
```

C#

```
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddWeaviateVectorStore(options: new() { Endpoint = new
Uri("http://localhost:8080/v1/") });
```

Overloads where you can specify your own `HttpClient` are also provided. In this case it's possible to set the service url via the `HttpClient BaseAddress` option.

C#

```
using System.Net.Http;
using Microsoft.SemanticKernel;

// Using Kernel Builder.
var kernelBuilder = Kernel.CreateBuilder();
using HttpClient client = new HttpClient { BaseAddress = new
Uri("http://localhost:8080/v1/") };
kernelBuilder.AddWeaviateVectorStore(client);
```

C#

```
using System.Net.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;

// Using IServiceCollection with ASP.NET Core.
var builder = WebApplication.CreateBuilder(args);
```

```
using HttpClient client = new HttpClient { BaseAddress = new Uri("http://localhost:8080/v1/") };
builder.Services.AddWeaviateVectorStore(client);
```

You can construct a Weaviate Vector Store instance directly as well.

C#

```
using System.Net.Http;
using Microsoft.SemanticKernel.Connectors.Weaviate;

var vectorStore = new WeaviateVectorStore(
    new HttpClient { BaseAddress = new Uri("http://localhost:8080/v1/") });
```

It is possible to construct a direct reference to a named collection.

C#

```
using System.Net.Http;
using Microsoft.SemanticKernel.Connectors.Weaviate;

var collection = new WeaviateVectorStoreRecordCollection<Hotel>(
    new HttpClient { BaseAddress = new Uri("http://localhost:8080/v1/") },
    "skhotels");
```

If needed, it is possible to pass an Api Key, as an option, when using any of the above mentioned mechanisms, e.g.

C#

```
using Microsoft.SemanticKernel;

var kernelBuilder = Kernel
    .CreateBuilder()
    .AddWeaviateVectorStore(options: new() { Endpoint = new Uri("http://localhost:8080/v1/"), ApiKey = secretVar });
```

Data mapping

The Weaviate Vector Store connector provides a default mapper when mapping from the data model to storage. Weaviate requires properties to be mapped into id, payload and vectors groupings. The default mapper uses the model annotations or record definition to determine the type of each property and to do this mapping.

- The data model property annotated as a key will be mapped to the Weaviate `id` property.
- The data model properties annotated as data will be mapped to the Weaviate `properties` object.
- The data model properties annotated as vectors will be mapped to the Weaviate `vectors` object.

The default mapper uses `System.Text.Json.JsonSerializer` to convert to the storage schema. This means that usage of the `JsonPropertyNameAttribute` is supported if a different storage name to the data model property name is required.

Here is an example of a data model with `JsonPropertyNameAttribute` set and how that will be represented in Weaviate.

C#

```
using System.Text.Json.Serialization;
using Microsoft.Extensions.VectorData;

public class Hotel
{
    [VectorStoreRecordKey]
    public ulong HotelId { get; set; }

    [VectorStoreRecordData(IsFilterable = true)]
    public string HotelName { get; set; }

    [VectorStoreRecordData(IsFullTextSearchable = true)]
    public string Description { get; set; }

    [JsonPropertyName("HOTEL_DESCRIPTION_EMBEDDING")]
    [VectorStoreRecordVector(4, DistanceFunction.EuclideanDistance,
    IndexKind.QuantizedFlat)]
    public ReadOnlyMemory<float>? DescriptionEmbedding { get; set; }
}
```

JSON

```
{
  "id": 1,
  "properties": { "HotelName": "Hotel Happy", "Description": "A place
where everyone can be happy." },
  "vectors": {
    "HOTEL_DESCRIPTION_EMBEDDING": [0.9, 0.1, 0.1, 0.1],
  }
}
```

How to ingest data into a Vector Store using Semantic Kernel (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

This article will demonstrate how to create an application to

1. Take text from each paragraph in a Microsoft Word document
2. Generate an embedding for each paragraph
3. Upsert the text, embedding and a reference to the original location into a Redis instance.

Prerequisites

For this sample you will need

1. An embedding generation model hosted in Azure or another provider of your choice.
2. An instance of Redis or Docker Desktop so that you can run Redis locally.
3. A Word document to parse and load. Here is a zip containing a sample Word document you can download and use: [vector-store-data-ingestion-input.zip](#).

Setup Redis

If you already have a Redis instance you can use that. If you prefer to test your project locally you can easily start a Redis container using docker.

```
docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest
```

To verify that it is running successfully, visit <http://localhost:8001/redis-stack/browser> in your browser.

The rest of these instructions will assume that you are using this container using the above settings.

Create your project

Create a new project and add nuget package references for the Redis connector from Semantic Kernel, the open xml package to read the word document with and the OpenAI connector from Semantic Kernel for generating embeddings.

.NET CLI

```
dotnet new console --framework net8.0 --name SKVectorIngest
cd SKVectorIngest
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI
dotnet add package Microsoft.SemanticKernel.Connectors.Redis --prerelease
dotnet add package DocumentFormat.OpenXml
```

Add a data model

To upload data we need to first describe what format the data should have in the database. We can do this by creating a data model with attributes that describe the function of each property.

Add a new file to the project called `TextParagraph.cs` and add the following model to it.

C#

```
using Microsoft.Extensions.VectorData;

namespace SKVectorIngest;

internal class TextParagraph
{
    /// <summary>A unique key for the text paragraph.</summary>
    [VectorStoreRecordKey]
    public required string Key { get; init; }

    /// <summary>A uri that points at the original location of the document
    containing the text.</summary>
    [VectorStoreRecordData]
    public required string DocumentUri { get; init; }

    /// <summary>The id of the paragraph from the document containing the
    text.</summary>
    [VectorStoreRecordData]
    public required string ParagraphId { get; init; }
```

```
/// <summary>The text of the paragraph.</summary>
[VectorStoreRecordData]
public required string Text { get; init; }

/// <summary>The embedding generated from the Text.</summary>
[VectorStoreRecordVector(1536)]
public ReadOnlyMemory<float> TextEmbedding { get; set; }
}
```

Note that we are passing the value `1536` to the `VectorStoreRecordVectorAttribute`. This is the dimension size of the vector and has to match the size of vector that your chosen embedding generator produces.

💡 Tip

For more information on how to annotate your data model and what additional options are available for each attribute, refer to [defining your data model](#).

Read the paragraphs in the document

We need some code to read the word document and find the text of each paragraph in it.

Add a new file to the project called `DocumentReader.cs` and add the following class to read the paragraphs from a document.

C#

```
using System.Text;
using System.Xml;
using DocumentFormat.OpenXml.Packaging;

namespace SKVectorIngest;

internal class DocumentReader
{
    public static IEnumerable<TextParagraph> ReadParagraphs(Stream
documentContents, string documentUri)
    {
        // Open the document.
        using WordprocessingDocument wordDoc =
WordprocessingDocument.Open(documentContents, false);
        if (wordDoc.MainDocumentPart == null)
        {
            yield break;
        }
```

```

        // Create an XmlDocument to hold the document contents and load the
        // document contents into the XmlDocument.
        XmlDocument xmlDoc = new XmlDocument();
        XmlNamespaceManager nsManager = new
        XmlNamespaceManager(xmlDoc.NameTable);
        nsManager.AddNamespace("w",
        "http://schemas.openxmlformats.org/wordprocessingml/2006/main");
        nsManager.AddNamespace("w14",
        "http://schemas.microsoft.com/office/word/2010/wordml");

        xmlDoc.Load(wordDoc.MainDocumentPart.GetStream());

        // Select all paragraphs in the document and break if none found.
        XmlNodeList? paragraphs = xmlDoc.SelectNodes("//w:p", nsManager);
        if (paragraphs == null)
        {
            yield break;
        }

        // Iterate over each paragraph.
        foreach (XmlNode paragraph in paragraphs)
        {
            // Select all text nodes in the paragraph and continue if none
            // found.
            XmlNodeList? texts = paragraph.SelectNodes("./w:t", nsManager);
            if (texts == null)
            {
                continue;
            }

            // Combine all non-empty text nodes into a single string.
            var textBuilder = new StringBuilder();
            foreach (XmlNode text in texts)
            {
                if (!string.IsNullOrWhiteSpace(text.InnerText))
                {
                    textBuilder.Append(text.InnerText);
                }
            }

            // Yield a new TextParagraph if the combined text is not empty.
            var combinedText = textBuilder.ToString();
            if (!string.IsNullOrWhiteSpace(combinedText))
            {
                Console.WriteLine("Found paragraph:");
                Console.WriteLine(combinedText);
                Console.WriteLine();

                yield return new TextParagraph
                {
                    Key = Guid.NewGuid().ToString(),
                    DocumentUri = documentUri,
                    ParagraphId = paragraph.Attributes?["w14:paraId"]?.Value
                };
            }
        }
    }
}

```

```
        };
    }
}
}
```

Generate embeddings and upload the data

We will need some code to generate embeddings and upload the paragraphs to Redis. Let's do this in a separate class.

Add a new file called `DataUploader.cs` and add the following class to it.

C#

```
#pragma warning disable SKEXP0001 // Type is for evaluation purposes only
and is subject to change or removal in future updates. Suppress this
diagnostic to proceed.

using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Embeddings;

namespace SKVectorIngest;

internal class DataUploader(IVectorStore vectorStore,
ITextEmbeddingGenerationService textEmbeddingGenerationService)
{
    /// <summary>
    /// Generate an embedding for each text paragraph and upload it to the
    /// specified collection.
    /// </summary>
    /// <param name="collectionName">The name of the collection to upload
    /// the text paragraphs to.</param>
    /// <param name="textParagraphs">The text paragraphs to upload.</param>
    /// <returns>An async task.</returns>
    public async Task GenerateEmbeddingsAndUpload(string collectionName,
IEnumerable<TextParagraph> textParagraphs)
    {
        var collection = vectorStore.GetCollection<string, TextParagraph>
(collectionName);
        await collection.CreateCollectionIfNotExistsAsync();

        foreach (var paragraph in textParagraphs)
        {
            // Generate the text embedding.
            Console.WriteLine($"Generating embedding for paragraph:
{paragraph.ParagraphId}");
            paragraph.TextEmbedding = await
textEmbeddingGenerationService.GenerateEmbeddingAsync(paragraph.Text);

            // Upload the text paragraph.
        }
    }
}
```

```

        Console.WriteLine($"Upserting paragraph:
{paragraph.ParagraphId}");
        await collection.UpsertAsync(paragraph);

        Console.WriteLine();
    }
}
}

```

Put it all together

Finally, we need to put together the different pieces. In this example, we will use the Semantic Kernel dependency injection container but it is also possible to use any `IServiceCollection` based container.

Add the following code to your `Program.cs` file to create the container, register the Redis vector store and register the embedding service. Make sure to replace the text embedding generation settings with your own values.

C#

```

#pragma warning disable SKEXP0010 // Type is for evaluation purposes only
and is subject to change or removal in future updates. Suppress this
diagnostic to proceed.
#pragma warning disable SKEXP0020 // Type is for evaluation purposes only
and is subject to change or removal in future updates. Suppress this
diagnostic to proceed.

using Microsoft.Extensions.DependencyInjection;
using Microsoft.SemanticKernel;
using SKVectorIngest;

// Replace with your values.
var deploymentName = "text-embedding-ada-002";
var endpoint = "https://sksample.openai.azure.com/";
var apiKey = "your-api-key";

// Register Azure Open AI text embedding generation service and Redis vector
// store.
var builder = Kernel.CreateBuilder()
    .AddAzureOpenAITextEmbeddingGeneration(deploymentName, endpoint, apiKey)
    .AddRedisVectorStore("localhost:6379");

// Register the data uploader.
builder.Services.AddSingleton<DataUploader>();

// Build the kernel and get the data uploader.
var kernel = builder.Build();
var dataUploader = kernel.Services.GetRequiredService<DataUploader>();

```

As a last step, we want to read the paragraphs from our word document, and call the data uploader to generate the embeddings and upload the paragraphs.

C#

```
// Load the data.  
var textParagraphs = DocumentReader.ReadParagraphs(  
    new FileStream(  
        "vector-store-data-ingestion-input.docx",  
        FileMode.Open),  
    "file:///c:/vector-store-data-ingestion-input.docx");  
  
await dataUploader.GenerateEmbeddingsAndUpload(  
    "sk-documentation",  
    textParagraphs);
```

See your data in Redis

Navigate to the Redis stack browser, e.g. <http://localhost:8001/redis-stack/browser> where you should now be able to see your uploaded paragraphs. Here is an example of what you should see for one of the uploaded paragraphs.

JSON

```
{  
    "DocumentUri" : "file:///c:/vector-store-data-ingestion-input.docx",  
    "ParagraphId" : "14CA7304",  
    "Text" : "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.",  
    "TextEmbedding" : [...]  
}
```

How to build a custom mapper for a Vector Store connector (Preview)

Article • 10/16/2024

⚠️ Warning

The Semantic Kernel Vector Store functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

In this how to, we will show how you can replace the default mapper for a vector store record collection with your own mapper.

We will use Qdrant to demonstrate this functionality, but the concepts will be similar for other connectors.

Background

Each Vector Store connector includes a default mapper that can map from the provided data model to the storage schema supported by the underlying store. Some stores allow a lot of freedom with regards to how data is stored while other stores require a more structured approach, e.g. where all vectors have to be added to a dictionary of vectors and all non-vector fields to a dictionary of data fields. Therefore, mapping is an important part of abstracting away the differences of each data store implementation.

In some cases, the developer may want to replace the default mapper if e.g.

1. they want to use a data model that differs from the storage schema.
2. they want to build a performance optimized mapper for their scenario.
3. the default mapper doesn't support a storage structure that the developer requires.

All Vector Store connector implementations allow you to provide a custom mapper.

Differences by vector store type

The underlying data stores of each Vector Store connector have different ways of storing data. Therefore what you are mapping to on the storage side may differ for each connector.

E.g. if using the Qdrant connector, the storage type is a `PointStruct` class provided by the Qdrant SDK. If using the Redis JSON connector, the storage type is a `string` key and a `JsonNode`, while if using a JSON HashSet connector, the storage type is a `string` key and a `HashEntry` array.

If you want to do custom mapping, and you want to use multiple connector types, you will therefore need to implement a mapper for each connector type.

Creating the data model

Our first step is to create a data model. In this case we will not annotate the data model with attributes, since we will provide a separate record definition that describes what the database schema will look like.

Also note that this model is complex, with separate classes for vectors and additional product info.

```
C#  
  
public class Product  
{  
    public ulong Id { get; set; }  
    public string Name { get; set; }  
    public string Description { get; set; }  
    public ProductVectors Vectors { get; set; }  
    public ProductInfo ProductInfo { get; set; }  
}  
  
public class ProductInfo  
{  
    public double Price { get; set; }  
    public string SupplierId { get; set; }  
}  
  
public class ProductVectors  
{  
    public ReadOnlyMemory<float> NameEmbedding { get; set; }  
    public ReadOnlyMemory<float> DescriptionEmbedding { get; set; }  
}
```

Creating the record definition

We need to create a record definition instance to define what the database schema will look like. Normally a connector will require this information to do mapping when using the default mapper. Since we are creating a custom mapper, this is not required for

mapping, however, the connector will still require this information for creating collections in the data store.

Note that the definition here is different to the data model above. To store `ProductInfo` we have a string property called `ProductInfoJson`, and the two vectors are defined at the same level as the `Id`, `Name` and `Description` fields.

C#

```
using Microsoft.Extensions.VectorData;

var productDefinition = new VectorStoreRecordDefinition
{
    Properties = new List<VectorStoreRecordProperty>
    {
        new VectorStoreRecordKeyProperty("Id", typeof(ulong)),
        new VectorStoreRecordDataProperty("Name", typeof(string)) {
            IsFilterable = true },
        new VectorStoreRecordDataProperty("Description", typeof(string)),
        new VectorStoreRecordDataProperty("ProductInfoJson",
            typeof(string)),
        new VectorStoreRecordVectorProperty("NameEmbedding",
            typeof(ReadOnlyMemory<float>)) { Dimensions = 1536 },
        new VectorStoreRecordVectorProperty("DescriptionEmbedding",
            typeof(ReadOnlyMemory<float>)) { Dimensions = 1536 }
    }
};
```

ⓘ Important

For this scenario, it would not be possible to use attributes instead of a record definition since the storage schema does not resemble the data model.

Creating the custom mapper

All mappers implement the generic interface

`Microsoft.SemanticKernel.Data.IVectorStoreRecordMapper<TRecordDataModel,`
`TStorageModel>`. `TRecordDataModel` will differ depending on what data model the developer wants to use, and `TStorageModel` will be determined by the type of Vector Store.

For Qdrant `TStorageModel` is `Qdrant.Client.Grpc.PointStruct`.

We therefore have to implement a mapper that will map between our `Product` data model and a Qdrant `PointStruct`.

C#

```
using Microsoft.Extensions.VectorData;
using Qdrant.Client.Grpc;

public class ProductMapper : IVectorStoreRecordMapper<Product, PointStruct>
{
    public PointStruct MapFromDataToStorageModel(Product dataModel)
    {
        // Create a Qdrant PointStruct to map our data to.
        var pointStruct = new PointStruct
        {
            Id = new PointId { Num = dataModel.Id },
            Vectors = new Vectors(),
            Payload = { },
        };

        // Add the data fields to the payload dictionary and serialize the
        product info into a json string.
        pointStruct.Payload.Add("Name", dataModel.Name);
        pointStruct.Payload.Add("Description", dataModel.Description);
        pointStruct.Payload.Add("ProductInfoJson",
        JsonSerializer.Serialize(dataModel.ProductInfo));

        // Add the vector fields to the vector dictionary.
        var namedVectors = new NamedVectors();
        namedVectors.Vectors.Add("NameEmbedding",
        dataModel.Vectors.NameEmbedding.ToArray());
        namedVectors.Vectors.Add("DescriptionEmbedding",
        dataModel.Vectors.DescriptionEmbedding.ToArray());
        pointStruct.Vectors_.Vectors_ = namedVectors;

        return pointStruct;
    }

    public Product MapFromStorageToDataModel(PointStruct storageModel,
    StorageToDataModelMapperOptions options)
    {
        var product = new Product
        {
            Id = storageModel.Id.Num,

            // Retrieve the data fields from the payload dictionary and
            deserialize the product info from the json string that it was stored as.
            Name = storageModel.Payload["Name"].StringValue,
            Description = storageModel.Payload["Description"].StringValue,
            ProductInfo = JsonSerializer.Deserialize<ProductInfo>
            (storageModel.Payload["ProductInfoJson"].StringValue)!

            // Retrieve the vector fields from the vector dictionary.
        };
    }
}
```

```

        Vectors = new ProductVectors
    {
        NameEmbedding = new ReadOnlyMemory<float>
(storageModel.Vectors.Vectors_.Vectors[ "NameEmbedding" ].Data.ToArray()),
        DescriptionEmbedding = new ReadOnlyMemory<float>
(storageModel.Vectors.Vectors_.Vectors[ "DescriptionEmbedding" ].Data.ToArray())
    }
};

return product;
}
}

```

Using your custom mapper with a record collection

To use the custom mapper that we have created, we need to pass it to the record collection at construction time. We also need to pass the record definition that we created earlier, so that collections are created in the data store using the right schema. One more setting that is important here, is Qdrant's named vectors mode, since we have more than one vector. Without this mode switched on, only one vector is supported.

C#

```

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.Qdrant;
using Qdrant.Client;

var productMapper = new ProductMapper();
var collection = new QdrantVectorStoreRecordCollection<Product>(
    new QdrantClient("localhost"),
    "skproducts",
    new()
{
    HasNamedVectors = true,
    PointStructCustomMapper = productMapper,
    VectorStoreRecordDefinition = productDefinition
});

```

Using a custom mapper with IVectorStore

When using `IVectorStore` to get `IVectorStoreRecordCollection` object instances, it is not possible to provide a custom mapper directly to the `GetCollection` method. This is

because custom mappers differ for each Vector Store type, and would make it impossible to use `IVectorStore` to communicate with any vector store implementation.

It is however possible to provide a factory when constructing a Vector Store implementation. This can be used to customize `IVectorStoreRecordCollection` instances as they are created.

Here is an example of such a factory, which checks if `CreateCollection` was called with the product definition and data type, and if so injects the custom mapper and switches on named vectors mode.

C#

```
public class QdrantCollectionFactory(VectorStoreRecordDefinition
productDefinition) : IQdrantVectorStoreRecordCollectionFactory
{
    public IVectorStoreRecordCollection<TKey, TRecord>
CreateVectorStoreRecordCollection<TKey, TRecord>(QdrantClient qdrantClient,
string name, VectorStoreRecordDefinition? vectorStoreRecordDefinition)
    where TKey : notnull
    where TRecord : class
    {
        // If the record definition is the product definition and the record
        // type is the product data
        // model, inject the custom mapper into the collection options.
        if (vectorStoreRecordDefinition == productDefinition &&
typeof(TRecord) == typeof(Product))
        {
            var customCollection = new
QdrantVectorStoreRecordCollection<Product>(
                qdrantClient,
                name,
                new()
                {
                    HasNamedVectors = true,
                    PointStructCustomMapper = new ProductMapper(),
                    VectorStoreRecordDefinition =
vectorStoreRecordDefinition
                } as IVectorStoreRecordCollection<TKey, TRecord>;
            return customCollection!;
        }

        // Otherwise, just create a standard collection with the default
        // mapper.
        var collection = new QdrantVectorStoreRecordCollection<TRecord>(
            qdrantClient,
            name,
            new()
            {
                VectorStoreRecordDefinition = vectorStoreRecordDefinition
            } as IVectorStoreRecordCollection<TKey, TRecord>;
        return collection!;
    }
}
```

```
    }  
}
```

To use the collection factory, pass it to the Vector Store when constructing it, or when registering it with the dependency injection container.

C#

```
// When registering with the dependency injection container on the kernel  
builder.  
kernelBuilder.AddQdrantVectorStore(  
    "localhost",  
    options: new()  
    {  
        VectorStoreCollectionFactory = new  
QdrantCollectionFactory(productDefinition)  
    });
```

C#

```
// When constructing the Vector Store instance directly.  
var vectorStore = new QdrantVectorStore(  
    new QdrantClient("localhost"),  
    new()  
    {  
        VectorStoreCollectionFactory = new  
QdrantCollectionFactory(productDefinition)  
    });
```

Now you can use the vector store as normal to get a collection.

C#

```
var collection = vectorStore.GetCollection<ulong, Product>("skproducts",  
productDefinition);
```

What are prompts?

Article • 09/27/2024

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.

The subtleties of prompting

Effective prompt design is essential to achieving desired outcomes with LLM AI models. Prompt engineering, also known as prompt design, is an emerging field that requires creativity and attention to detail. It involves selecting the right words, phrases, symbols, and formats that guide the model in generating high-quality and relevant texts.

If you've already experimented with ChatGPT, you can see how the model's behavior changes dramatically based on the inputs you provide. For example, the following prompts produce very different outputs:

Prompt
Please give me the history of humans.

Prompt
Please give me the history of humans in 3 sentences.

The first prompt produces a long report, while the second prompt produces a concise response. If you were building a UI with limited space, the second prompt would be more suitable for your needs. Further refined behavior can be achieved by adding even more details to the prompt, but it's possible to go too far and produce irrelevant outputs. As a prompt engineer, you must find the right balance between specificity and relevance.

When you work directly with LLM models, you can also use other controls to influence the model's behavior. For example, you can use the `temperature` parameter to control the randomness of the model's output. Other parameters like top-k, top-p, frequency penalty, and presence penalty also influence the model's behavior.

Prompt engineering: a new career

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.

Becoming a great prompt engineer with Semantic Kernel

Semantic Kernel is a valuable tool for prompt engineering because it allows you to experiment with different prompts and parameters across multiple different models using a common interface. This allows you to quickly compare the outputs of different models and parameters, and iterate on prompts to achieve the desired results.

Once you've become familiar with prompt engineering, you can also use Semantic Kernel to apply your skills to real-world scenarios. By combining your prompts with native functions and connectors, you can build powerful AI-powered applications.

Lastly, by deeply integrating with Visual Studio Code, Semantic Kernel also makes it easy for you to integrate prompt engineering into your existing development processes.

- ✓ Create prompts directly in your preferred code editor.
- ✓ Write tests for them using your existing testing frameworks.
- ✓ And deploy them to production using your existing CI/CD pipelines.

Additional tips for prompt engineering

Becoming a skilled prompt engineer requires a combination of technical knowledge, creativity, and experimentation. Here are some tips to excel in prompt engineering:

- **Understand LLM AI models:** Gain a deep understanding of how LLM AI models work, including their architecture, training processes, and behavior.
- **Domain knowledge:** Acquire domain-specific knowledge to design prompts that align with the desired outputs and tasks.
- **Experimentation:** Explore different parameters and settings to fine-tune prompts and optimize the model's behavior for specific tasks or domains.
- **Feedback and iteration:** Continuously analyze the outputs generated by the model and iterate on prompts based on user feedback to improve their quality and relevance.
- **Stay updated:** Keep up with the latest advancements in prompt engineering techniques, research, and best practices to enhance your skills and stay ahead in the field.

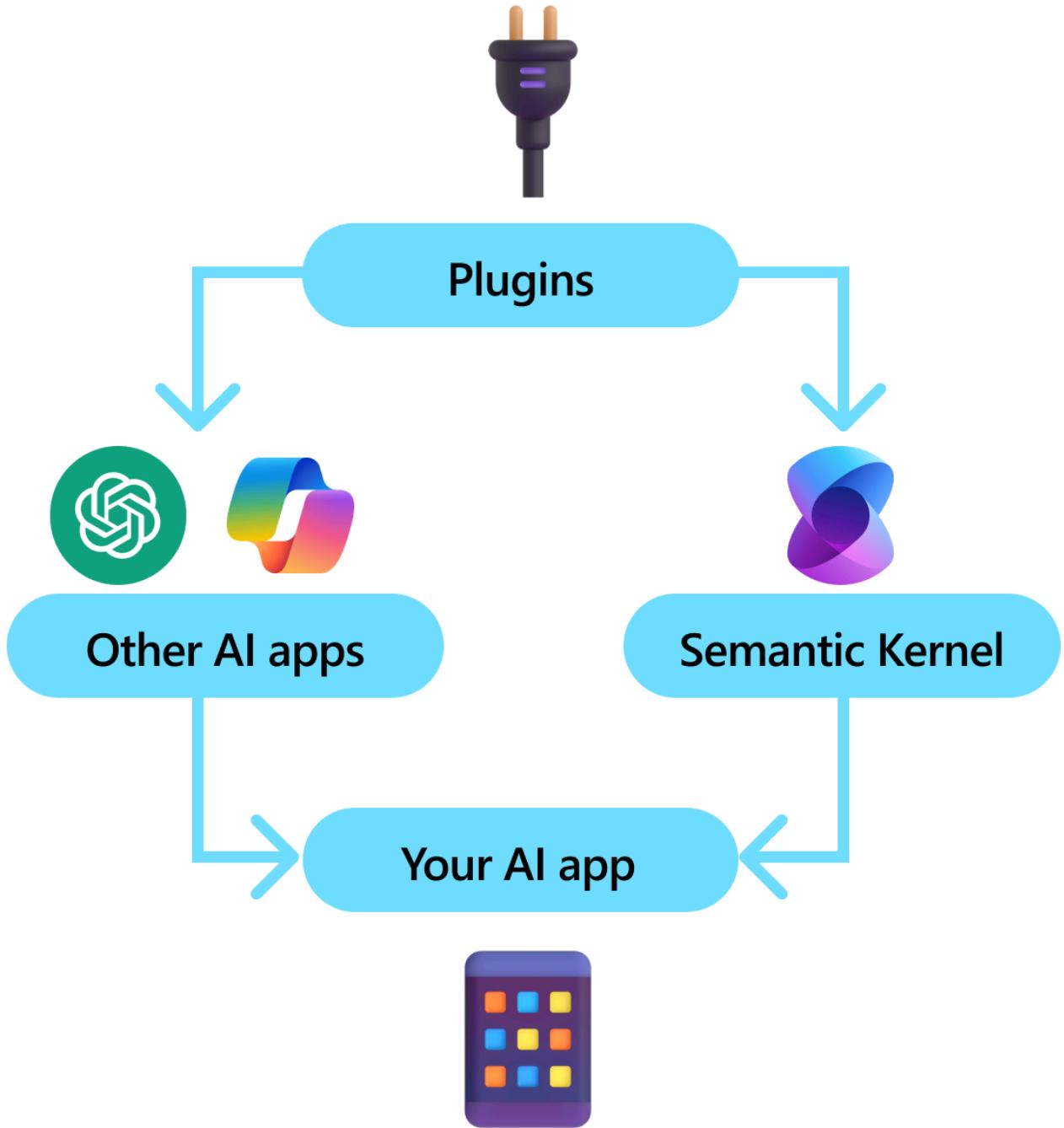
Prompt engineering is a dynamic and evolving field, and skilled prompt engineers play a crucial role in harnessing the capabilities of LLM AI models effectively.

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

Tip

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.

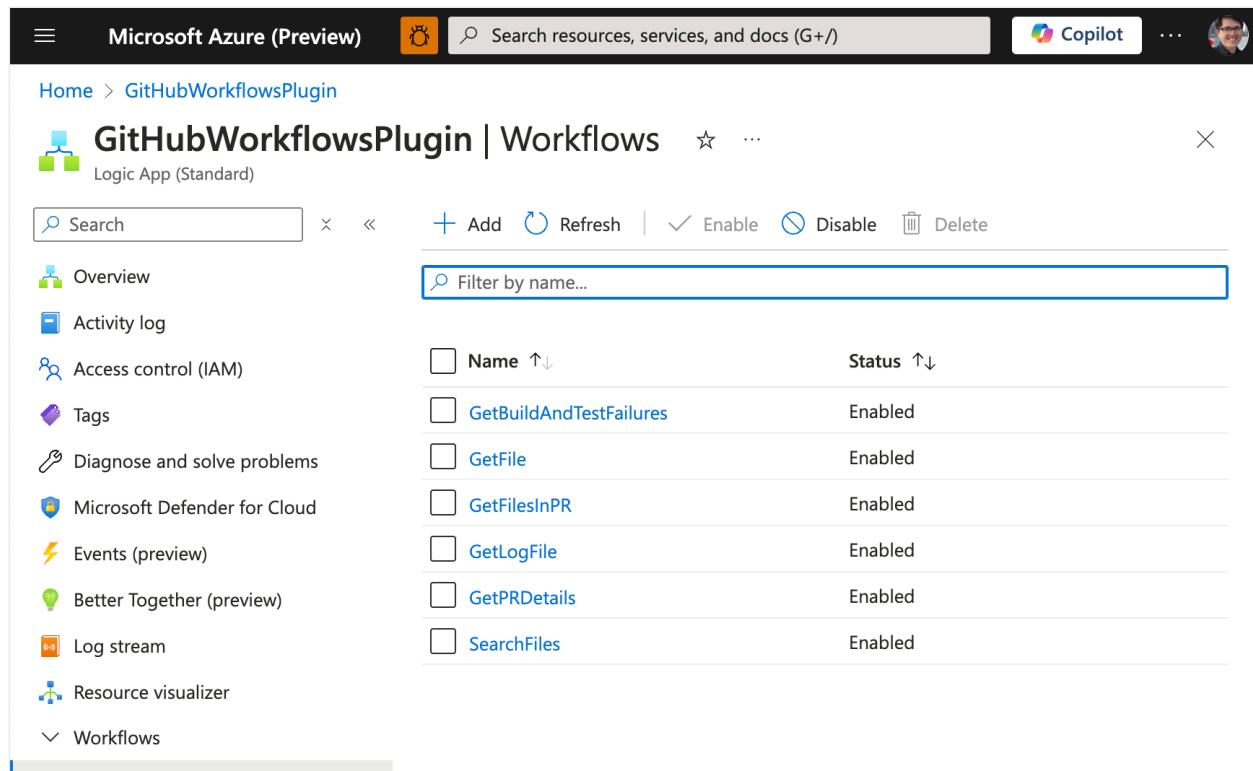
[Learn about using retrieval functions](#)

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 links 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 (which is expanded). The main area has a search bar, a filter bar ("Filter by name..."), and a table with columns for Name and Status. The table contains the following data:

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 • 09/09/2024

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 Semantic Kernel Text Search?

Article • 10/16/2024

⚠ Warning

The Semantic Kernel Text Search functionality is preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Semantic Kernel provides capabilities that allow developers to integrate search when calling a Large Language Model (LLM). This is important because LLM's are trained on fixed data sets and may need access to additional data to accurately respond to a user ask.

The process of providing additional context when prompting a LLM is called Retrieval-Augmented Generation (RAG). RAG typically involves retrieving additional data that is relevant to the current user ask and augmenting the prompt sent to the LLM with this data. The LLM can use its training plus the additional context to provide a more accurate response.

A simple example of when this becomes important is when the user's ask is related to up-to-date information not included in the LLM's training data set. By performing an appropriate text search and including the results with the user's ask, more accurate responses will be achieved.

Semantic Kernel provides a set of Text Search capabilities that allow developers to perform searches using Web Search or Vector Databases and easily add RAG to their applications.

Implementing RAG using web text search

In the following sample code you can choose between using Bing or Google to perform web search operations.

💡 Tip

To run the samples shown on this page go to
[GettingStartedWithTextSearch/Step1_Web_Search.cs](#).

Create text search instance

Each sample creates a text search instance and then performs a search operation to get results for the provided query. The search results will contain a snippet of text from the webpage that describes its contents. This provides only a limited context i.e., a subset of the web page contents and no link to the source of the information. Later samples show how to address these limitations.

💡 Tip

The following sample code uses the Semantic Kernel OpenAI connector and Web plugins, install using the following commands:

```
dotnet add package Microsoft.SemanticKernel  
dotnet add package Microsoft.SemanticKernel.Plugins.Web
```

Bing web search

C#

```
using Microsoft.SemanticKernel.Data;  
using Microsoft.SemanticKernel.Plugins.Web.Bing;  
  
// Create an ITextSearch instance using Bing search  
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");  
  
var query = "What is the Semantic Kernel?";  
  
// Search and return results  
KernelSearchResults<string> searchResults = await  
textSearch.SearchAsync(query, new() { Top = 4 });  
await foreach (string result in searchResults.Results)  
{  
    Console.WriteLine(result);  
}
```

Google web search

C#

```
using Microsoft.SemanticKernel.Data;  
using Microsoft.SemanticKernel.Plugins.Web.Google;  
  
// Create an ITextSearch instance using Google search  
var textSearch = new GoogleTextSearch(  
    searchEngineId: "<Your Google Search Engine Id>",  
    apiKey: "<Your Google API Key>");
```

```
var query = "What is the Semantic Kernel?";

// Search and return results
KernelSearchResults<string> searchResults = await
textSearch.SearchAsync(query, new() { Top = 4 });
await foreach (string result in searchResults.Results)
{
    Console.WriteLine(result);
}
```

💡 Tip

For more information on what types of search results can be retrieved, refer to [the documentation on Text Search Plugins](#).

Use text search results to augment a prompt

Next steps are to create a Plugin from the web text search and invoke the Plugin to add the search results to the prompt.

The sample code below shows how to achieve this:

1. Create a `Kernel` that has an OpenAI service registered. This will be used to call the `gpt-4o` model with the prompt.
2. Create a text search instance.
3. Create a Search Plugin from the text search instance.
4. Create a prompt template that will invoke the Search Plugin with the query and include search results in the prompt along with the original query.
5. Invoke the prompt and display the response.

The model will provide a response that is grounded in the latest information available from a web search.

Bing web search

C#

```
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
```

```

        apiKey: "<Your OpenAI API Key>");

Kernel kernel = kernelBuilder.Build();

// Create a text search using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel
var searchPlugin = textSearch.CreateWithSearch("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
var prompt = "{{{SearchPlugin.Search $query}}}. {{$query}}";
KernelArguments arguments = new() { { "query", query } };
Console.WriteLine(await kernel.InvokePromptAsync(prompt, arguments));

```

Google web search

C#

```

using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Google;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
Kernel kernel = kernelBuilder.Build();

// Create an ITextSearch instance using Google search
var textSearch = new GoogleTextSearch(
    searchEngineId: "<Your Google Search Engine Id>",
    apiKey: "<Your Google API Key>");

// Build a text search plugin with Google search and add to the kernel
var searchPlugin = textSearch.CreateWithSearch("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
var prompt = "{{{SearchPlugin.Search $query}}}. {{$query}}";
KernelArguments arguments = new() { { "query", query } };
Console.WriteLine(await kernel.InvokePromptAsync(prompt, arguments));

```

There are a number of issues with the above sample:

1. The response does not include citations showing the web pages that were used to provide grounding context.

2. The response will include data from any web site, it would be better to limit this to trusted sites.
3. Only a snippet of each web page is being used to provide grounding context to the model, the snippet may not contain the data required to provide an accurate response.

See the page which describes [Text Search Plugins](#) for solutions to these issues.

Next we recommend looking at [Text Search Abstractions](#).

Next steps

[Text Search Abstractions](#)

[Text Search Plugins](#)

[Text Search Function Calling](#)

[Text Search with Vector Stores](#)

Why are Text Search abstractions needed?

Article • 10/16/2024

When dealing with text prompts or text content in chat history a common requirement is to provide additional relevant information related to this text. This provides the AI model with relevant context which helps it to provide more accurate responses. To meet this requirement the Semantic Kernel provides a Text Search abstraction which allows using text inputs from various sources, e.g. Web search engines, vector stores, etc., and provide results in a few standardized formats.

ⓘ Note

Search for image content or audio content is not currently supported.

Text search abstraction

The Semantic Kernel text search abstractions provides three methods:

1. `Search`
2. `GetSearchResults`
3. `GetTextSearchResults`

Search

Performs a search for content related to the specified query and returns string values representing the search results. `Search` can be used in the most basic use cases e.g., when augmenting a `semantic-kernel` format prompt template with search results.

`Search` always returns just a single string value per search result so is not suitable if citations are required.

GetSearchResults

Performs a search for content related to the specified query and returns search results in the format defined by the implementation. `GetSearchResults` returns the full search result as defined by the underlying search service. This provides the most versatility at the cost of tying your code to a specific search service implementation.

GetTextSearchResults

Performs a search for content related to the specified query and returns a normalized data model representing the search results. This normalized data model includes a string value and optionally a name and link. `GetTextSearchResults` allows your code to be isolated from the a specific search service implementation, so the same prompt can be used with multiple different search services.

Tip

To run the samples shown on this page go to
[GettingStartedWithTextSearch/Step1_Web_Search.cs](#).

The sample code below shows each of the text search methods in action.

C#

```
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;

// Create an ITextSearch instance using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

var query = "What is the Semantic Kernel?";

// Search and return results
KernelSearchResults<string> searchResults = await
textSearch.SearchAsync(query, new() { Top = 4 });
await foreach (string result in searchResults.Results)
{
    Console.WriteLine(result);
}

// Search and return results as BingWebPage items
KernelSearchResults<object> webPages = await
textSearch.GetSearchResultsAsync(query, new() { Top = 4 });
await foreach (BingWebPage webPage in webPages.Results)
{
    Console.WriteLine($"Name: {webPage.Name}");
    Console.WriteLine($"Snippet: {webPage.Snippet}");
    Console.WriteLine($"Url: {webPage.Url}");
    Console.WriteLine($"DisplayUrl: {webPage.DisplayUrl}");
    Console.WriteLine($"DateLastCrawled: {webPage.DateLastCrawled}");
}

// Search and return results as TextSearchResult items
KernelSearchResults<TextSearchResult> textResults = await
textSearch.GetTextSearchResultsAsync(query, new() { Top = 4 });
await foreach (TextSearchResult result in textResults.Results)
```

```
{  
    Console.WriteLine($"Name: {result.Name}");  
    Console.WriteLine($"Value: {result.Value}");  
    Console.WriteLine($"Link: {result.Link}");  
}
```

Next steps

[Text Search Plugins](#)

[Text Search Function Calling](#)

[Text Search with Vector Stores](#)

What are Semantic Kernel Text Search plugins?

Article • 10/16/2024

Semantic Kernel uses [Plugins](#) to connect existing APIs with AI. These Plugins have functions that can be used to add relevant data or examples to prompts, or to allow the AI to perform actions automatically.

To integrate Text Search with Semantic Kernel, we need to turn it into a Plugin. Once we have a Text Search plugin, we can use it to add relevant information to prompts or to retrieve information as needed. Creating a plugin from Text Search is a simple process, which we will explain below.

💡 Tip

To run the samples shown on this page go to

[GettingStartedWithTextSearch/Step2_Search_For_RAG.cs](#).

Basic search plugin

Semantic Kernel provides a default template implementation that supports variable substitution and function calling. By including an expression such as `{{MyPlugin.Function $arg1}}` in a prompt template, the specified function i.e., `MyPlugin.Function` will be invoked with the provided argument `arg1` (which is resolved from `KernelArguments`). The return value from the function invocation is inserted into the prompt. This technique can be used to inject relevant information into a prompt.

The sample below shows how to create a plugin named `SearchPlugin` from an instance of `BingTextSearch`. Using `CreateWithSearch` creates a new plugin with a single `Search` function that calls the underlying text search implementation. The `SearchPlugin` is added to the `Kernel` which makes it available to be called during prompt rendering. The prompt template includes a call to `{{SearchPlugin.Search $query}}` which will invoke the `SearchPlugin` to retrieve results related to the current query. The results are then inserted into the rendered prompt before it is sent to the AI model.

C#

```
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;
```

```

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
Kernel kernel = kernelBuilder.Build();

// Create a text search using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel
var searchPlugin = textSearch.CreateWithSearch("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
var prompt = "{{SearchPlugin.Search $query}}. {{$query}}";
KernelArguments arguments = new() { { "query", query } };
Console.WriteLine(await kernel.InvokePromptAsync(prompt, arguments));

```

Search plugin with citations

The sample below repeats the pattern described in the previous section with a few notable changes:

1. `CreateWithGetTextSearchResults` is used to create a `SearchPlugin` which calls the `GetTextSearchResults` method from the underlying text search implementation.
2. The prompt template uses Handlebars syntax. This allows the template to iterate over the search results and render the name, value and link for each result.
3. The prompt includes an instruction to include citations, so the AI model will do the work of adding citations to the response.

C#

```

using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
Kernel kernel = kernelBuilder.Build();

// Create a text search using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel

```

```

var searchPlugin =
textSearch.CreateWithGetTextSearchResults("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
string promptTemplate = """
{{#with (SearchPlugin-GetTextSearchResults query)}}
  {{#each this}}
    Name: {{Name}}
    Value: {{Value}}
    Link: {{Link}}
  -----
  {{/each}}
{{/with}}

{{query}}
```

Include citations to the relevant information where it is referenced in the response.

```

""";
KernelArguments arguments = new() { { "query", query } };
HandlebarsPromptTemplateFactory promptTemplateFactory = new();
Console.WriteLine(await kernel.InvokePromptAsync(
  promptTemplate,
  arguments,
  templateFormat:
HandlebarsPromptTemplateFactory.HandlebarsTemplateFormat,
  promptTemplateFactory: promptTemplateFactory
));

```

Search plugin with a filter

The samples shown so far will use the top ranked web search results to provide the grounding data. To provide more reliability in the data the web search can be restricted to only return results from a specified site.

The sample below builds on the previous one to add filtering of the search results. A `TextSearchFilter` with an equality clause is used to specify that only results from the Microsoft Developer Blogs site (`site == 'devblogs.microsoft.com'`) are to be included in the search results.

The sample uses `KernelPluginFactory.CreateFromFunctions` to create the `SearchPlugin`. A custom description is provided for the plugin. The `ITextSearch.CreateGetTextSearchResults` extension method is used to create the `KernelFunction` which invokes the text search service.

💡 Tip

The `site` filter is Bing specific. For Google web search use `siteSearch`.

C#

```
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
Kernel kernel = kernelBuilder.Build();

// Create a text search using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Create a filter to search only the Microsoft Developer Blogs site
var filter = new TextSearchFilter().Equality("site",
"devblogs.microsoft.com");
var searchOptions = new TextSearchOptions() { Filter = filter };

// Build a text search plugin with Bing search and add to the kernel
var searchPlugin = KernelPluginFactory.CreateFromFunctions(
    "SearchPlugin", "Search Microsoft Developer Blogs site only",
    [textSearch.CreateGetTextSearchResults(searchOptions: searchOptions)]);
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
string promptTemplate = """
{{#with (SearchPlugin-GetTextSearchResults query)}}
{{#each this}}
Name: {{Name}}
Value: {{Value}}
Link: {{Link}}
-----
{{/each}}
{{/with}}

{{query}}
```

Include citations to the relevant information where it is referenced in the response.

""";

```
KernelArguments arguments = new() { { "query", query } };
HandlebarsPromptTemplateFactory promptTemplateFactory = new();
Console.WriteLine(await kernel.InvokePromptAsync(
    promptTemplate,
    arguments,
```

```
    templateFormat:  
    HandlebarsPromptTemplateFactory.HandlebarsTemplateFormat,  
    promptTemplateFactory: promptTemplateFactory  
);
```

💡 Tip

Follow the link for more information on how to [filter the answers that Bing returns](#).

Custom search plugin

In the previous sample a static site filter was applied to the search operations. What if you need this filter to be dynamic?

The next sample shows how you can perform more customization of the `SearchPlugin` so that the filter value can be dynamic. The sample uses

`KernelFunctionFromMethodOptions` to specify the following for the `SearchPlugin`:

- `FunctionName`: The search function is named `GetSiteResults` because it will apply a site filter if the query includes a domain.
- `Description`: The description describes how this specialized search function works.
- `Parameters`: The parameters include an additional optional parameter for the `site` so the domain can be specified.

Customizing the search function is required if you want to provide multiple specialized search functions. In prompts you can use the function names to make the template more readable. If you use function calling then the model will use the function name and description to select the best search function to invoke.

When this sample is executed, the response will use techcommunity.microsoft.com as the source for relevant data.

C#

```
using Microsoft.SemanticKernel.Data;  
using Microsoft.SemanticKernel.Plugins.Web.Bing;  
  
// Create a kernel with OpenAI chat completion  
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();  
kernelBuilder.AddOpenAIChatCompletion(  
    modelId: "gpt-4o",  
    apiKey: "<Your OpenAI API Key>");  
Kernel kernel = kernelBuilder.Build();
```

```

// Create a text search using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel
var options = new KernelFunctionFromMethodOptions()
{
    FunctionName = "GetSiteResults",
    Description = "Perform a search for content related to the specified query and optionally from the specified domain.",
    Parameters =
    [
        new KernelParameterMetadata("query") { Description = "What to search for", IsRequired = true },
        new KernelParameterMetadata("top") { Description = "Number of results", IsRequired = false, DefaultValue = 5 },
        new KernelParameterMetadata("skip") { Description = "Number of results to skip", IsRequired = false, DefaultValue = 0 },
        new KernelParameterMetadata("site") { Description = "Only return results from this domain", IsRequired = false },
    ],
    ReturnParameter = new() { ParameterType =
typeof(KernelSearchResults<string>) },
};

var searchPlugin = KernelPluginFactory.CreateFromFunctions("SearchPlugin",
"Search specified site", [textSearch.CreateGetTextSearchResults(options)]);
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
string promptTemplate = """
{{#with (SearchPlugin-GetSiteResults query)}}
{{#each this}}
Name: {{Name}}
Value: {{Value}}
Link: {{Link}}
-----
{{/each}}
{{/with}}
{{query}}

Only include results from techcommunity.microsoft.com.
Include citations to the relevant information where it is referenced in the response.
""";
KernelArguments arguments = new() { { "query", query } };
HandlebarsPromptTemplateFactory promptTemplateFactory = new();
Console.WriteLine(await kernel.InvokePromptAsync(
    promptTemplate,
    arguments,
    templateFormat:
HandlebarsPromptTemplateFactory.HandlebarsTemplateFormat,
    promptTemplateFactory: promptTemplateFactory
));

```

Next steps

[Text Search Function Calling](#)

[Text Search with Vector Stores](#)

Why use function calling with Semantic Kernel Text Search?

Article • 10/16/2024

In the previous Retrieval-Augmented Generation (RAG) based samples the user ask has been used as the search query when retrieving relevant information. The user ask could be long and may span multiple topics or there may be multiple different search implementations available which provide specialized results. For either of these scenarios it can be useful to allow the AI model to extract the search query or queries from the user ask and use function calling to retrieve the relevant information it needs.

💡 Tip

To run the samples shown on this page go to

[GettingStartedWithTextSearch/Step3_Search_With_FunctionCalling.cs](#).

Function calling with Bing text search

💡 Tip

The samples in this section use an `IFunctionInvocationFilter` filter to log the function that the model calls and what parameters it sends. It is interesting to see what the model uses as a search query when calling the `SearchPlugin`.

Here is the `IFunctionInvocationFilter` filter implementation.

C#

```
private sealed class FunctionInvocationFilter(TextWriter output) :  
    IFunctionInvocationFilter  
{  
    public async Task OnFunctionInvocationAsync(InvocationContext context, Func<InvocationContext, Task> next)  
    {  
        if (context.Function.PluginName == "SearchPlugin")  
        {  
            output.WriteLine($"{context.Function.Name}:  
{JsonSerializer.Serialize(context.Arguments)}\n");  
        }  
        await next(context);  
    }  
}
```

```
    }  
}
```

The sample below creates a `SearchPlugin` using Bing web search. This plugin will be advertised to the AI model for use with automatic function calling, using the `FunctionChoiceBehavior` in the prompt execution settings. When you run this sample check the console output to see what the model used as the search query.

C#

```
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.Connectors.OpenAI;  
using Microsoft.SemanticKernel.Data;  
using Microsoft.SemanticKernel.Plugins.Web.Bing;  
  
// Create a kernel with OpenAI chat completion  
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();  
kernelBuilder.AddOpenAIChatCompletion(  
    modelId: "gpt-4o",  
    apiKey: "<Your OpenAI API Key>");  
kernelBuilder.Services.AddSingleton<ITestOutputHelper>(output);  
kernelBuilder.Services.AddSingleton<IFunctionInvocationFilter,  
FunctionInvocationFilter>();  
Kernel kernel = kernelBuilder.Build();  
  
// Create a search service with Bing search  
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");  
  
// Build a text search plugin with Bing search and add to the kernel  
var searchPlugin = textSearch.CreateWithSearch("SearchPlugin");  
kernel.Plugins.Add(searchPlugin);  
  
// Invoke prompt and use text search plugin to provide grounding information  
OpenAIPromptExecutionSettings settings = new() { FunctionChoiceBehavior =  
FunctionChoiceBehavior.Auto() };  
KernelArguments arguments = new(settings);  
Console.WriteLine(await kernel.InvokePromptAsync("What is the Semantic  
Kernel?", arguments));
```

Function calling with Bing text search and citations

The sample below includes the required changes to include citations:

1. Use `CreateWithGetTextSearchResults` to create the `SearchPlugin`, this will include the link to the original source of the information.
2. Modify the prompt to instruct the model to include citations in its response.

C#

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
kernelBuilder.Services.AddSingleton<ITestOutputHelper>(output);
kernelBuilder.Services.AddSingleton<IFunctionInvocationFilter,
FunctionInvocationFilter>();
Kernel kernel = kernelBuilder.Build();

// Create a search service with Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel
var searchPlugin =
textSearch.CreateWithGetTextSearchResults("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
OpenAIPromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };
KernelArguments arguments = new(settings);
Console.WriteLine(await kernel.InvokePromptAsync("What is the Semantic
Kernel? Include citations to the relevant information where it is referenced
in the response.", arguments));
```

Function calling with Bing text search and filtering

The final sample in this section shows how to use a filter with function calling. For this sample only search results from the Microsoft Developer Blogs site will be included. An instance of `TextSearchFilter` is created and an equality clause is added to match the `devblogs.microsoft.com` site. This filter will be used when the function is invoked in response to a function calling request from the model.

C#

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;
```

```
// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: "gpt-4o",
    apiKey: "<Your OpenAI API Key>");
kernelBuilder.Services.AddSingleton<ITestOutputHelper>(output);
kernelBuilder.Services.AddSingleton<IFunctionInvocationFilter,
FunctionInvocationFilter>();
Kernel kernel = kernelBuilder.Build();

// Create a search service with Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

// Build a text search plugin with Bing search and add to the kernel
var filter = new TextSearchFilter().Equality("site",
"devblogs.microsoft.com");
var searchOptions = new TextSearchOptions() { Filter = filter };
var searchPlugin = KernelPluginFactory.CreateFromFunctions(
    "SearchPlugin", "Search Microsoft Developer Blogs site only",
    [textSearch.CreateGetTextSearchResults(searchOptions: searchOptions)]);
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
OpenAIPromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };
KernelArguments arguments = new(settings);
Console.WriteLine(await kernel.InvokePromptAsync("What is the Semantic
Kernel? Include citations to the relevant information where it is referenced
in the response.", arguments));
```

Next steps

[Text Search with Vector Stores](#)

How to use Vector Stores with Semantic Kernel Text Search

Article • 10/16/2024

All of the Vector Store [connectors](#) can be used for text search.

1. Use the Vector Store connector to retrieve the record collection you want to search.
2. Wrap the record collection with `VectorStoreTextSearch`.
3. Convert to a plugin for use in RAG and/or function calling scenarios.

It's very likely that you will want to customize the plugin search function so that its description reflects the type of data available in the record collection. For example if the record collection contains information about hotels the plugin search function description should mention this. This will allow you to register multiple plugins e.g., one to search for hotels, another for restaurants and another for things to do.

The [text search abstractions](#) include a function to return a normalized search result i.e., an instance of `TextSearchResult`. This normalized search result contains a value and optionally a name and link. The [text search abstractions](#) include a function to return a string value e.g., one of the data model properties will be returned as the search result. For text search to work correctly you need to provide a way to map from the Vector Store data model to an instance of `TextSearchResult`. The next section describes the two options you can use to perform this mapping.

💡 Tip

To run the samples shown on this page go to

[GettingStartedWithTextSearch/Step4_Search_With_VectorStore.cs](#) ↗

Using a vector store model with text search

The mapping from a Vector Store data model to a `TextSearchResult` can be done declaratively using attributes.

1. `[TextSearchResultValue]` - Add this attribute to the property of the data model which will be the value of the `TextSearchResult`, e.g. the textual data that the AI model will use to answer questions.

2. `[TextSearchResultName]` - Add this attribute to the property of the data model which will be the name of the `TextSearchResult`.
3. `[TextSearchResultLink]` - Add this attribute to the property of the data model which will be the link to the `TextSearchResult`.

The following sample shows an data model which has the text search result attributes applied.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Data;

public sealed class DataModel
{
    [VectorStoreRecordKey]
    [TextSearchResultName]
    public Guid Key { get; init; }

    [VectorStoreRecordData]
    [TextSearchResultValue]
    public string Text { get; init; }

    [VectorStoreRecordData]
    [TextSearchResultLink]
    public string Link { get; init; }

    [VectorStoreRecordData(IsFilterable = true)]
    public required string Tag { get; init; }

    [VectorStoreRecordVector(1536)]
    public ReadOnlyMemory<float> Embedding { get; init; }
}
```

The mapping from a Vector Store data model to a `string` or a `TextSearchResult` can also be done by providing implementations of `ITextSearchStringMapper` and `ITextSearchResultMapper` respectively.

You may decide to create custom mappers for the following scenarios:

1. Multiple properties from the data model need to be combined together e.g., if multiple properties need to be combined to provide the value.
2. Additional logic is required to generate one of the properties e.g., if the link property needs to be computed from the data model properties.

The following sample shows a data model and two example mapper implementations that can be used with the data model.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Data;

protected sealed class DataModel
{
    [VectorStoreRecordKey]
    public Guid Key { get; init; }

    [VectorStoreRecordData]
    public required string Text { get; init; }

    [VectorStoreRecordData]
    public required string Link { get; init; }

    [VectorStoreRecordData(IsFilterable = true)]
    public required string Tag { get; init; }

    [VectorStoreRecordVector(1536)]
    public ReadOnlyMemory<float> Embedding { get; init; }
}

/// <summary>
/// String mapper which converts a DataModel to a string.
/// </summary>
protected sealed class DataModelTextSearchStringMapper : ITextSearchStringMapper
{
    /// <inheritdoc />
    public string MapFromResultToString(object result)
    {
        if (result is DataModel dataModel)
        {
            return dataModel.Text;
        }
        throw new ArgumentException("Invalid result type.");
    }
}

/// <summary>
/// Result mapper which converts a DataModel to a TextSearchResult.
/// </summary>
protected sealed class DataModelTextSearchResultMapper : ITextSearchResultMapper
{
    /// <inheritdoc />
    public TextSearchResult MapFromResultToTextSearchResult(object result)
    {
        if (result is DataModel dataModel)
        {
            return new TextSearchResult(value: dataModel.Text) { Name = dataModel.Key.ToString(), Link = dataModel.Link };
        }
    }
}
```

```
        throw new ArgumentException("Invalid result type.");
    }
}
```

The mapper implementations can be provided as parameters when creating the `VectorStoreTextSearch` as shown below:

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Data;

// Create custom mapper to map a <see cref="DataModel"/> to a <see
// cref="string"/>
var stringMapper = new DataModelTextSearchStringMapper();

// Create custom mapper to map a <see cref="DataModel"/> to a <see
// cref="TextSearchResult"/>
var resultMapper = new DataModelTextSearchResultMapper();

// Add code to create instances of IVectorStoreRecordCollection and
// ITextEmbeddingGenerationService

// Create a text search instance using the vector store record collection.
var result = new VectorStoreTextSearch<DataModel>
(vectorStoreRecordCollection, textEmbeddingGeneration, stringMapper,
resultMapper);
```

Using a vector store with text search

The sample below shows how to create an instance of `VectorStoreTextSearch` using a Vector Store record collection.

Tip

The following samples require instances of `IVectorStoreRecordCollection` and `ITextEmbeddingGenerationService`. To create an instance of `IVectorStoreRecordCollection` refer to [the documentation for each connector](#). To create an instance of `ITextEmbeddingGenerationService` select the service you wish to use e.g., Azure OpenAI, OpenAI, ... or use a local model ONNX, Ollama, ... and create an instance of the corresponding `ITextEmbeddingGenerationService` implementation.

Tip

A `VectorStoreTextSearch` can also be constructed from an instance of `IVectorizableTextSearch`. In this case no `ITextEmbeddingGenerationService` is needed.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.PromptTemplates.Handlebars;

// Add code to create instances of IVectorStoreRecordCollection and
ITextEmbeddingGenerationService

// Create a text search instance using the vector store record collection.
var textSearch = new VectorStoreTextSearch<DataModel>
(vectorStoreRecordCollection, textEmbeddingGeneration);

// Search and return results as TextSearchResult items
var query = "What is the Semantic Kernel?";
KernelSearchResults<TextSearchResult> textResults = await
textSearch.GetTextSearchResultsAsync(query, new() { Top = 2, Skip = 0 });
Console.WriteLine("\n--- Text Search Results ---\n");
await foreach (TextSearchResult result in textResults.Results)
{
    Console.WriteLine($"Name: {result.Name}");
    Console.WriteLine($"Value: {result.Value}");
    Console.WriteLine($"Link: {result.Link}");
}
```

Creating a search plugin from a vector store

The sample below shows how to create a plugin named `SearchPlugin` from an instance of `VectorStoreTextSearch`. Using `CreateWithGetTextSearchResults` creates a new plugin with a single `GetTextSearchResults` function that calls the underlying Vector Store record collection search implementation. The `SearchPlugin` is added to the `Kernel` which makes it available to be called during prompt rendering. The prompt template includes a call to `{{SearchPlugin.Search $query}}` which will invoke the `SearchPlugin` to retrieve results related to the current query. The results are then inserted into the rendered prompt before it is sent to the model.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel;
```

```

using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.PromptTemplates.Handlebars;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: TestConfiguration.OpenAI.ChatModelId,
    apiKey: TestConfiguration.OpenAI.ApiKey);
Kernel kernel = kernelBuilder.Build();

// Add code to create instances of IVectorStoreRecordCollection and
ITextEmbeddingGenerationService

// Create a text search instance using the vector store record collection.
var textSearch = new VectorStoreTextSearch<DataModel>
(vectorStoreRecordCollection, textEmbeddingGeneration);

// Build a text search plugin with vector store search and add to the kernel
var searchPlugin =
textSearch.CreateWithGetTextSearchResults("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
var query = "What is the Semantic Kernel?";
string promptTemplate = """
{{#with (SearchPlugin-GetTextSearchResults query)}}
```

```

{{#each this}}
Name: {{Name}}
Value: {{Value}}
Link: {{Link}}
-----
{{/each}}
```

```

{{/with}}
```

```

{{query}}
```

Include citations to the relevant information where it is referenced in
the response.

```

""";
```

```

KernelArguments arguments = new() { { "query", query } };
HandlebarsPromptTemplateFactory promptTemplateFactory = new();
Console.WriteLine(await kernel.InvokePromptAsync(
    promptTemplate,
    arguments,
    templateFormat:
HandlebarsPromptTemplateFactory.HandlebarsTemplateFormat,
    promptTemplateFactory: promptTemplateFactory
));
```

Using a vector store with function calling

The sample below also creates a `SearchPlugin` from an instance of `VectorStoreTextSearch`. This plugin will be advertised to the model for use with automatic function calling using the `FunctionChoiceBehavior` in the prompt execution settings. When you run this sample the model will invoke the search function to retrieve additional information to respond to the question. It will likely just search for "Semantic Kernel" rather than the entire query.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.PromptTemplates.Handlebars;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: TestConfiguration.OpenAI.ChatModelId,
    apiKey: TestConfiguration.OpenAI.ApiKey);
Kernel kernel = kernelBuilder.Build();

// Add code to create instances of IVectorStoreRecordCollection and
ITextEmbeddingGenerationService

// Create a text search instance using the vector store record collection.
var textSearch = new VectorStoreTextSearch<DataModel>
(vectorStoreRecordCollection, textEmbeddingGeneration);

// Build a text search plugin with vector store search and add to the kernel
var searchPlugin =
textSearch.CreateWithGetTextSearchResults("SearchPlugin");
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
OpenAIPromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };
KernelArguments arguments = new(settings);
Console.WriteLine(await kernel.InvokePromptAsync("What is the Semantic
Kernel?", arguments));
```

Customizing the search function

The sample below shows how to customize the description of the search function that is added to the `SearchPlugin`. Some things you might want to do are:

1. Change the name of the search function to reflect what is in the associated record collection e.g., you might want to name the function `SearchForHotels` if the record

collection contains hotel information.

2. Change the description of the function. An accurate function description helps the AI model to select the best function to call. This is especially important if you are adding multiple search functions.
3. Add an additional parameter to the search function. If the record collection contain hotel information and one of the properties is the city name you could add a property to the search function to specify the city. A filter will be automatically added and it will filter search results by city.

💡 Tip

The sample below uses the default implementation of search. You can opt to provide your own implementation which calls the underlying Vector Store record collection with additional options to fine tune your searches.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.PromptTemplates.Handlebars;

// Create a kernel with OpenAI chat completion
IKernelBuilder kernelBuilder = Kernel.CreateBuilder();
kernelBuilder.AddOpenAIChatCompletion(
    modelId: TestConfiguration.OpenAI.ChatModelId,
    apiKey: TestConfiguration.OpenAI.ApiKey);
Kernel kernel = kernelBuilder.Build();

// Add code to create instances of IVectorStoreRecordCollection and
ITextEmbeddingGenerationService

// Create a text search instance using the vector store record collection.
var textSearch = new VectorStoreTextSearch<DataModel>
(vectorStoreRecordCollection, textEmbeddingGeneration);

// Create options to describe the function I want to register.
var options = new KernelFunctionFromMethodOptions()
{
    FunctionName = "Search",
    Description = "Perform a search for content related to the specified
query from a record collection.",
    Parameters =
    [
        new KernelParameterMetadata("query") { Description = "What to search
for", IsRequired = true },
        new KernelParameterMetadata("top") { Description = "Number of
results", IsRequired = false, DefaultValue = 2 },
    ]
}
```

```
        new KernelParameterMetadata("skip") { Description = "Number of results to skip", IsRequired = false, DefaultValue = 0 },
    ],
    ReturnParameter = new() { ParameterType =
typeof(KernelSearchResults<string>) },
};

// Build a text search plugin with vector store search and add to the kernel
var searchPlugin = textSearch.CreateWithGetTextSearchResults("SearchPlugin",
"Search a record collection", [textSearch.CreateSearch(options)]);
kernel.Plugins.Add(searchPlugin);

// Invoke prompt and use text search plugin to provide grounding information
OpenAIPromptExecutionSettings settings = new() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() };
KernelArguments arguments = new(settings);
Console.WriteLine(await kernel.InvokePromptAsync("What is the Semantic
Kernel?", arguments));
```

Out-of-the-box Text Search (Preview)

Article • 10/21/2024

⚠ Warning

The Semantic Kernel Text Search functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Semantic Kernel provides a number of out-of-the-box Text Search integrations making it easy to get started with using Text Search.

[+] Expand table

Text Search	C#	Python	Java
Bing	✓	In Development	In Development
Google	✓	In Development	In Development
Vector Store	✓	In Development	In Development

Using the Bing Text Search (Preview)

Article • 10/21/2024

⚠ Warning

The Semantic Kernel Text Search functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Bing Text Search implementation uses the [Bing Web Search API](#) to retrieve search results. You must provide your own Bing Search Api Key to use this component.

Limitations

[] Expand table

Feature Area	Support
Search API	Bing Web Search API only.
Supported filter clauses	Only "equal to" filter clauses are supported.
Supported filter keys	The responseFilter query parameter and advanced search keywords are supported.

💡 Tip

Follow this link for more information on how to [filter the answers that Bing returns](#). Follow this link for more information on using [advanced search keywords](#).

Getting started

The sample below shows how to create a `BingTextSearch` and use it to perform a text search.

```
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Bing;
```

```

// Create an ITextSearch instance using Bing search
var textSearch = new BingTextSearch(apiKey: "<Your Bing API Key>");

var query = "What is the Semantic Kernel?";

// Search and return results as a string items
KernelSearchResults<string> stringResults = await
textSearch.SearchAsync(query, new() { Top = 4, Skip = 0 });
Console.WriteLine("--- String Results ---\n");
await foreach (string result in stringResults.Results)
{
    Console.WriteLine(result);
}

// Search and return results as TextSearchResult items
KernelSearchResults<TextSearchResult> textResults = await
textSearch.GetTextSearchResultsAsync(query, new() { Top = 4, Skip = 4 });
Console.WriteLine("\n--- Text Search Results ---\n");
await foreach (TextSearchResult result in textResults.Results)
{
    Console.WriteLine($"Name: {result.Name}");
    Console.WriteLine($"Value: {result.Value}");
    Console.WriteLine($"Link: {result.Link}");
}

// Search and return s results as BingWebPage items
KernelSearchResults<object> fullResults = await
textSearch.GetSearchResultsAsync(query, new() { Top = 4, Skip = 8 });
Console.WriteLine("\n--- Bing Web Page Results ---\n");
await foreach (BingWebResponse result in fullResults.Results)
{
    Console.WriteLine($"Name: {result.Name}");
    Console.WriteLine($"Snippet: {result.Snippet}");
    Console.WriteLine($"Url: {result.Url}");
    Console.WriteLine($"DisplayUrl: {result.DisplayUrl}");
    Console.WriteLine($"DateLastCrawled: {result.DateLastCrawled}");
}

```

Next steps

The following sections of the documentation show you how to:

1. Create a [plugin](#) and use it for Retrieval Augmented Generation (RAG).
2. Use text search together with [function calling](#).
3. Learn more about using [vector stores](#) for text search.

[Text Search Abstractions](#)

[Text Search Plugins](#)

[Text Search Function Calling](#)

[Text Search with Vector Stores](#)

Using the Google Text Search (Preview)

Article • 10/21/2024

⚠ Warning

The Semantic Kernel Text Search functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Google Text Search implementation uses [Google Custom Search](#) to retrieve search results. You must provide your own Google Search Api Key and Search Engine Id to use this component.

Limitations

[] Expand table

Feature Area	Support
Search API	Google Custom Search API only.
Supported filter clauses	Only "equal to" filter clauses are supported.
Supported filter keys	Following parameters are supported: "cr", "dateRestrict", "exactTerms", "excludeTerms", "filter", "gl", "hl", "linkSite", "lr", "orTerms", "rights", "siteSearch". For more information see parameters .

💡 Tip

Follow this link for more information on how [search is performed](#)

Getting started

The sample below shows how to create a `GoogleTextSearch` and use it to perform a text search.

C#

```

using Google.Apis.Http;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Plugins.Web.Google;

// Create an ITextSearch instance using Google search
var textSearch = new GoogleTextSearch(
    initializer: new() { ApiKey = "<Your Google API Key>", HttpClientFactory
= new CustomHttpClientFactory(this.Output) },
    searchEngineId: "<Your Google Search Engine Id>");

var query = "What is the Semantic Kernel?";

// Search and return results as string items
KernelSearchResults<string> stringResults = await
textSearch.SearchAsync(query, new() { Top = 4, Skip = 0 });
Console.WriteLine("— String Results —\n");
await foreach (string result in stringResults.Results)
{
    Console.WriteLine(result);
}

// Search and return results as TextSearchResult items
KernelSearchResults<TextSearchResult> textResults = await
textSearch.GetTextSearchResultsAsync(query, new() { Top = 4, Skip = 4 });
Console.WriteLine("\n— Text Search Results —\n");
await foreach (TextSearchResult result in textResults.Results)
{
    Console.WriteLine($"Name: {result.Name}");
    Console.WriteLine($"Value: {result.Value}");
    Console.WriteLine($"Link: {result.Link}");
}

// Search and return results as Google.Apis.CustomSearchAPI.v1.Data.Result
// items
KernelSearchResults<object> fullResults = await
textSearch.GetSearchResultsAsync(query, new() { Top = 4, Skip = 8 });
Console.WriteLine("\n— Google Web Page Results —\n");
await foreach (Google.Apis.CustomSearchAPI.v1.Data.Result result in
fullResults.Results)
{
    Console.WriteLine($"Title: {result.Title}");
    Console.WriteLine($"Snippet: {result.Snippet}");
    Console.WriteLine($"Link: {result.Link}");
    Console.WriteLine($"DisplayLink: {result.DisplayLink}");
    Console.WriteLine($"Kind: {result.Kind}");
}

```

Next steps

The following sections of the documentation show you how to:

1. Create a [plugin](#) and use it for Retrieval Augmented Generation (RAG).
2. Use text search together with [function calling](#).
3. Learn more about using [vector stores](#) for text search.

[Text Search Abstractions](#)

[Text Search Plugins](#)

[Text Search Function Calling](#)

[Text Search with Vector Stores](#)

Using the Vector Store Text Search (Preview)

Article • 10/21/2024

⚠️ Warning

The Semantic Kernel Text Search functionality is in preview, and improvements that require breaking changes may still occur in limited circumstances before release.

Overview

The Vector Store Text Search implementation uses the [Vector Store Connectors](#) to retrieve search results. This means you can use Vector Store Text Search with any Vector Store which Semantic Kernel supports and any implementation of [Microsoft.Extensions.VectorData.Abstractions](#).

Limitations

See the limitations listed for the [Vector Store connector](#) you are using.

Getting started

The sample below shows how to use an in-memory vector store to create a `VectorStoreTextSearch` and use it to perform a text search.

C#

```
using Microsoft.Extensions.VectorData;
using Microsoft.SemanticKernel.Connectors.InMemory;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Embeddings;

// Create an embedding generation service.
var textEmbeddingGeneration = new OpenAITextEmbeddingGenerationService(
    modelId: TestConfiguration.OpenAI.EmbeddingModelId,
    apiKey: TestConfiguration.OpenAI.ApiKey);

// Construct an InMemory vector store.
var vectorStore = new InMemoryVectorStore();
var collectionName = "records";
```

```

// Get and create collection if it doesn't exist.
var recordCollection = vectorStore.GetCollection< TKey, TRecord>
(collectionName);
await
recordCollection.CreateCollectionIfNotExistsAsync().ConfigureAwait(false);

// TODO populate the record collection with your test data
// Example https://github.com/microsoft/semantic-
kernel/blob/main/dotnet/samples/Concepts/Search/VectorStore_TextSearch.cs

// Create a text search instance using the InMemory vector store.
var textSearch = new VectorStoreTextSearch<DataModel>(recordCollection,
textEmbeddingGeneration);

// Search and return results as TextSearchResult items
var query = "What is the Semantic Kernel?";
KernelSearchResults<TextSearchResult> textResults = await
textSearch.GetTextSearchResultsAsync(query, new() { Top = 2, Skip = 0 });
Console.WriteLine("\n--- Text Search Results ---\n");
await foreach (TextSearchResult result in textResults.Results)
{
    Console.WriteLine($"Name: {result.Name}");
    Console.WriteLine($"Value: {result.Value}");
    Console.WriteLine($"Link: {result.Link}");
}

```

Next steps

The following sections of the documentation show you how to:

1. Create a [plugin](#) and use it for Retrieval Augmented Generation (RAG).
2. Use text search together with [function calling](#).
3. Learn more about using [vector stores](#) for text search.

[Text Search Abstractions](#)
[Text Search Plugins](#)
[Text Search Function Calling](#)
[Text Search with Vector Stores](#)

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](#)

Semantic Kernel Agent Framework (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

The *Semantic Kernel Agent Framework* provides a platform within the Semantic Kernel eco-system that allow for the creation of AI **agents** and the ability to incorporate **agentic patterns** into any application based on the same patterns and features that exist in the core *Semantic Kernel* framework.

What is an AI agent?

An **AI agent** is a software entity designed to perform tasks autonomously or semi-autonomously by receiving input, processing information, and taking actions to achieve specific goals.

Agents can send and receive messages, generating responses using a combination of models, tools, human inputs, or other customizable components.

Agents are designed to work collaboratively, enabling complex workflows by interacting with each other. The *Agent Framework* allows for the creation of both simple and sophisticated agents, enhancing modularity and ease of maintenance

What problems do AI agents solve?

AI agents offer several advantages for application development, particularly by enabling the creation of modular AI components that are able to collaborate to reduce manual intervention in complex tasks. AI agents can operate autonomously or semi-autonomously, making them powerful tools for a range of applications.

Here are some of the key benefits:

- **Modular Components:** Allows developers to define various types of agents for specific tasks (e.g., data scraping, API interaction, or natural language processing).

This makes it easier to adapt the application as requirements evolve or new technologies emerge.

- **Collaboration:** Multiple agents may "collaborate" on tasks. For example, one agent might handle data collection while another analyzes it and yet another uses the results to make decisions, creating a more sophisticated system with distributed intelligence.
- **Human-Agent Collaboration:** Human-in-the-loop interactions allow agents to work alongside humans to augment decision-making processes. For instance, agents might prepare data analyses that humans can review and fine-tune, thus improving productivity.
- **Process Orchestration:** Agents can coordinate different tasks across systems, tools, and APIs, helping to automate end-to-end processes like application deployments, cloud orchestration, or even creative processes like writing and design.

When to use an AI agent?

Using an agent framework for application development provides advantages that are especially beneficial for certain types of applications. While traditional AI models are often used as tools to perform specific tasks (e.g., classification, prediction, or recognition), agents introduce more autonomy, flexibility, and interactivity into the development process.

- **Autonomy and Decision-Making:** If your application requires entities that can make independent decisions and adapt to changing conditions (e.g., robotic systems, autonomous vehicles, smart environments), an agent framework is preferable.
- **Multi-Agent Collaboration:** If your application involves complex systems that require multiple independent components to work together (e.g., supply chain management, distributed computing, or swarm robotics), agents provide built-in mechanisms for coordination and communication.
- **Interactive and Goal-Oriented:** If your application involves goal-driven behavior (e.g., completing tasks autonomously or interacting with users to achieve specific objectives), agent-based frameworks are a better choice. Examples include virtual assistants, game AI, and task planners.

How do I install the *Semantic Kernel Agent Framework*?

Installing the *Agent Framework SDK* is specific to the distribution channel associated with your programming language.

For .NET SDK, several NuGet packages are available.

Note: The core *Semantic Kernel SDK* is required in addition to any agent packages.

[+] Expand table

Package	Description
Microsoft.SemanticKernel	This contains the core <i>Semantic Kernel</i> libraries for getting started with the <i>Agent Framework</i> . This must be explicitly referenced by your application.
Microsoft.SemanticKernel.Agents.Abstractions	Defines the core agent abstractions for the <i>Agent Framework</i> . Generally not required to be specified as it is included in both the Microsoft.SemanticKernel.Agents.Core and Microsoft.SemanticKernel.Agents.OpenAI packages.
Microsoft.SemanticKernel.Agents.Core	Includes the ChatCompletionAgent and AgentGroupChat classes.
Microsoft.SemanticKernel.Agents.OpenAI	Provides ability to use the Open AI Assistant API via the OpenAIAssistantAgent .

[Agent Architecture](#)

An Overview of the Agent Architecture (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

This article covers key concepts in the architecture of the Agent Framework, including foundational principles, design objectives, and strategic goals.

Goals

The *Agent Framework* was developed with the following key priorities in mind:

- The *Semantic Kernel* framework serves as the core foundation for implementing agent functionalities.
- Multiple agents can collaborate within a single conversation, while integrating human input.
- An agent can engage in and manage multiple concurrent conversations simultaneously.
- Different types of agents can participate in the same conversation, each contributing their unique capabilities.

Agent

The abstract *Agent* class serves as the core abstraction for all types of agents, providing a foundational structure that can be extended to create more specialized agents. One key subclass is *Kernel Agent*, which establishes a direct association with a *Kernel* object. This relationship forms the basis for more specific agent implementations, such as the *Chat Completion Agent* and the *Open AI Assistant Agent*, both of which leverage the Kernel's capabilities to execute their respective functions.

- [Agent](#)
- [KernelAgent](#)

Agents can either be invoked directly to perform tasks or orchestrated within an *Agent Chat*, where multiple agents may collaborate or interact dynamically with user inputs.

This flexible structure allows agents to adapt to various conversational or task-driven scenarios, providing developers with robust tools for building intelligent, multi-agent systems.

Deep Dive:

- [ChatCompletionAgent](#)
- [OpenAIAssistantAgent](#)

Agent Chat

The [*Agent Chat*](#) class serves as the foundational component that enables agents of any type to engage in a specific conversation. This class provides the essential capabilities for managing agent interactions within a chat environment. Building on this, the [*Agent Group Chat*](#) class extends these capabilities by offering a strategy-based container, which allows multiple agents to collaborate across numerous interactions within the same conversation.

This structure facilitates more complex, multi-agent scenarios where different agents can work together, share information, and dynamically respond to evolving conversations, making it an ideal solution for advanced use cases such as customer support, multi-faceted task management, or collaborative problem-solving environments.

Deep Dive:

- [AgentChat](#)

Agent Channel

The [*Agent Channel*](#) class enables agents of various types to participate in an [*Agent Chat*](#). This functionality is completely hidden from users of the [*Agent Framework*](#) and only needs to be considered by developers creating a custom [*Agent*](#).

- [AgentChannel](#)

Agent Alignment with *Semantic Kernel* Features

The [*Agent Framework*](#) is built on the foundational concepts and features that many developers have come to know within the [*Semantic Kernel*](#) ecosystem. These core

principles serve as the building blocks for the Agent Framework's design. By leveraging the familiar structure and capabilities of the *Semantic Kernel*, the Agent Framework extends its functionality to enable more advanced, autonomous agent behaviors, while maintaining consistency with the broader *Semantic Kernel* architecture. This ensures a smooth transition for developers, allowing them to apply their existing knowledge to create intelligent, adaptable agents within the framework.

The *Kernel*

At the heart of the *Semantic Kernel* ecosystem is the *Kernel*, which serves as the core object that drives AI operations and interactions. To create any agent within this framework, a *Kernel instance* is required as it provides the foundational context and capabilities for the agent's functionality. The *Kernel* acts as the engine for processing instructions, managing state, and invoking the necessary AI services that power the agent's behavior.

The [Chat Completion Agent](#) and [Open AI Assistant Agent](#) articles provide specific details on how to create each type of agent. These resources offer step-by-step instructions and highlight the key configurations needed to tailor the agents to different conversational or task-based applications, demonstrating how the *Kernel* enables dynamic and intelligent agent behaviors across diverse use cases.

Related API's:

- [IKernelBuilder](#)
- [Kernel](#)
- [KernelBuilderExtensions](#)
- [KernelExtensions](#)

Plugins and Function Calling

Plugins are a fundamental aspect of the *Semantic Kernel*, enabling developers to integrate custom functionalities and extend the capabilities of an AI application. These plugins offer a flexible way to incorporate specialized features or business-specific logic into the core AI workflows. Additionally, agent capabilities within the framework can be significantly enhanced by utilizing [Plugins](#) and leveraging [Function Calling](#). This allows agents to dynamically interact with external services or execute complex tasks, further expanding the scope and versatility of the AI system within diverse applications.

Example:

- How-To: [Chat Completion Agent](#)

Related API's:

- [KernelFunctionFactory](#)
- [KernelFunction](#)
- [KernelPluginFactory](#)
- [KernelPlugin](#)
- [Kernel.Plugins](#)

Agent Messages

Agent messaging, including both input and response, is built upon the core content types of the *Semantic Kernel*, providing a unified structure for communication. This design choice simplifies the process of transitioning from traditional chat-completion patterns to more advanced agent-driven patterns in your application development. By leveraging familiar *Semantic Kernel* content types, developers can seamlessly integrate agent capabilities into their applications without needing to overhaul existing systems. This streamlining ensures that as you evolve from basic conversational AI to more autonomous, task-oriented agents, the underlying framework remains consistent, making development faster and more efficient.

Note: The [Open AI Assistant Agent](#) introduced content types specific to its usage for *File References* and *Content Annotation*:

Related API's:

- [ChatHistory](#)
- [ChatMessageContent](#)
- [KernelContent](#)
- [StreamingKernelContent](#)
- [FileReferenceContent](#)
- [AnnotationContent](#)

Templating

An agent's role is primarily shaped by the instructions it receives, which dictate its behavior and actions. Similar to invoking a *Kernel prompt*, an agent's instructions can include templated parameters—both values and functions—that are dynamically

substituted during execution. This enables flexible, context-aware responses, allowing the agent to adjust its output based on real-time input.

Additionally, an agent can be configured directly using a *Prompt Template Configuration*, providing developers with a structured and reusable way to define its behavior. This approach offers a powerful tool for standardizing and customizing agent instructions, ensuring consistency across various use cases while still maintaining dynamic adaptability.

Example:

- [How-To: Chat Completion Agent](#)

Related API's:

- [PromptTemplateConfig](#)
- [KernelFunctionYaml.FromPromptYaml](#)
- [IPromptTemplateFactory](#)
- [KernelPromptTemplateFactory](#)
- [Handlebars](#)
- [Prompty](#)
- [Liquid](#)

Chat Completion

The *Chat Completion Agent* is designed around any *Semantic Kernel AI service*, offering a flexible and convenient persona encapsulation that can be seamlessly integrated into a wide range of applications. This agent allows developers to easily bring conversational AI capabilities into their systems without having to deal with complex implementation details. It mirrors the features and patterns found in the underlying *AI service*, ensuring that all functionalities—such as natural language processing, dialogue management, and contextual understanding—are fully supported within the *Chat Completion Agent*, making it a powerful tool for building conversational interfaces.

Related API's:

- [IChatCompletionService](#)
- [Microsoft.SemanticKernel.Connectors.AzureOpenAI](#)
- [Microsoft.SemanticKernel.Connectors.OpenAI](#)
- [Microsoft.SemanticKernel.Connectors.Google](#)
- [Microsoft.SemanticKernel.Connectors.HuggingFace](#)

- Microsoft.SemanticKernel.Connectors.MistralAI
- Microsoft.SemanticKernel.Connectors.Onnx

Exploring Chat Completion Agent

Exploring the *Semantic Kernel* Chat Completion Agent (Experimental)

Article • 10/09/2024

⚠ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Detailed API documentation related to this discussion is available at:

- [ChatCompletionAgent](#)
- [Microsoft.SemanticKernel.Agents](#)
- [IChatCompletionService](#)
- [Microsoft.SemanticKernel.ChatCompletion](#)

Chat Completion in *Semantic Kernel*

Chat Completion is fundamentally a protocol for a chat-based interaction with an AI model where the chat-history maintained and presented to the model with each request. *Semantic Kernel AI services* offer a unified framework for integrating the chat-completion capabilities of various AI models.

A *chat completion agent* can leverage any of these [AI services](#) to generate responses, whether directed to a user or another agent.

For .NET, *chat-completion* AI Services are based on the [IChatCompletionService](#) interface.

For .NET, some of AI services that support models with chat-completion include:

[+] Expand table

Model	<i>Semantic Kernel</i> AI Service
Azure Open AI	Microsoft.SemanticKernel.Connectors.AzureOpenAI
Gemini	Microsoft.SemanticKernel.Connectors.Google
HuggingFace	Microsoft.SemanticKernel.Connectors.HuggingFace

Model	Semantic Kernel AI Service
Mistral	Microsoft.SemanticKernel.Connectors.MistralAI
OpenAI	Microsoft.SemanticKernel.Connectors.OpenAI
Onnx	Microsoft.SemanticKernel.Connectors.Onnx

Creating a Chat Completion Agent

A *chat completion agent* is fundamentally based on an [AI services](#). As such, creating an *chat completion agent* starts with creating a [Kernel](#) instance that contains one or more chat-completion services and then instantiating the agent with a reference to that [Kernel](#) instance.

```
// Initialize a Kernel with a chat-completion service
IKernelBuilder builder = Kernel.CreateBuilder();

builder.AddAzureOpenAIChatCompletion(/*<...configuration parameters>*/);

Kernel kernel = builder.Build();

// Create the agent
ChatCompletionAgent agent =
    new()
{
    Name = "SummarizationAgent",
    Instructions = "Summarize user input",
    Kernel = kernel
};
```

AI Service Selection

No different from using *Semantic Kernel AI services* directly, a *chat completion agent* support the specification of a *service-selector*. A *service-selector* indentifies which [AI service](#) to target when the [Kernel](#) contains more than one.

Note: If multiple [AI services](#) are present and no *service-selector* is provided, the same *default* logic is applied for the agent that you'd find when using an [AI services](#) outside of the *Agent Framework*

```
IKernelBuilder builder = Kernel.CreateBuilder();

// Initialize multiple chat-completion services.
```

```

builder.AddAzureOpenAIChatCompletion(/*<...service configuration>*/,
serviceId: "service-1");
builder.AddAzureOpenAIChatCompletion(/*<...service configuration>*/,
serviceId: "service-2");

Kernel kernel = builder.Build();

ChatCompletionAgent agent =
    new()
{
    Name = "<agent name>",
    Instructions = "<agent instructions>",
    Kernel = kernel,
    Arguments = // Specify the service-identifier via the
KernelArguments
    new KernelArguments(
        new OpenAIPromptExecutionSettings()
    {
        ServiceId = "service-2" // The target service-identifier.
    });
}

```

Conversing with *Chat Completion Agent*

Conversing with your *Chat Completion Agent* is based on a *Chat History* instance, no different from interacting with a [Chat Completion AI service](#).

```

// Define agent
ChatCompletionAgent agent = ...;

// Create a ChatHistory object to maintain the conversation state.
ChatHistory chat = [];

// Add a user message to the conversation
chat.Add(new ChatMessageContent(AuthorRole.User, "<user input>"));

// Generate the agent response(s)
await foreach (ChatMessageContent response in agent.InvokeAsync(chat))
{
    // Process agent response(s)...
}

```

How-To:

For an end-to-end example for a *Chat Completion Agent*, see:

- [How-To: Chat Completion Agent](#)

Exploring the *Semantic Kernel Open AI Assistant Agent* (Experimental)

Article • 10/09/2024

⚠ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Detailed API documentation related to this discussion is available at:

- [OpenAIAssistantAgent](#)
- [OpenAIAssistantDefinition](#)
- [OpenAIClientProvider](#)

What is an Assistant?

The *OpenAI Assistant API* is a specialized interface designed for more advanced and interactive AI capabilities, enabling developers to create personalized and multi-step task-oriented agents. Unlike the Chat Completion API, which focuses on simple conversational exchanges, the Assistant API allows for dynamic, goal-driven interactions with additional features like code-interpreter and file-search.

- [Open AI Assistant Guide ↗](#)
- [Open AI Assistant API ↗](#)
- [Assistant API in Azure](#)

Creating an *Open AI Assistant Agent*

Creating an *Open AI Assistant* requires invoking a remote service, which is handled asynchronously. To manage this, the *Open AI Assistant Agent* is instantiated through a static factory method, ensuring the process occurs in a non-blocking manner. This method abstracts the complexity of the asynchronous call, returning a promise or future once the assistant is fully initialized and ready for use.

C#

```
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.CreateAsync(
        OpenAIClientProvider.ForAzureOpenAI(/*<...service configuration>*/),
```

```
new OpenAIAssistantDefinition("<model name>")
{
    Name = "<agent name>",
    Instructions = "<agent instructions>",
},
new Kernel());
```

Retrieving an *Open AI Assistant Agent*

Once created, the identifier of the assistant may be accessed via its identifier. This identifier may be used to create an *Open AI Assistant Agent* from an existing assistant definition.

For .NET, the agent identifier is exposed as a `string` via the property defined by any agent.

C#

```
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.RetrieveAsync(
        OpenAIClientProvider.ForAzureOpenAI(/*<...service configuration>*/),
        "<your agent id>",
        new Kernel());
```

Using an *Open AI Assistant Agent*

As with all aspects of the *Assistant API*, conversations are stored remotely. Each conversation is referred to as a *thread* and identified by a unique `string` identifier. Interactions with your *OpenAI Assistant Agent* are tied to this specific thread identifier which must be specified when calling the agent/

C#

```
// Define agent
OpenAIAssistantAgent agent = ...;

// Create a thread for the agent conversation.
string threadId = await agent.CreateThreadAsync();

// Add a user message to the conversation
chat.Add(threadId, new ChatMessageContent(AuthorRole.User, "<user input>"));

// Generate the agent response(s)
await foreach (ChatMessageContent response in agent.InvokeAsync(threadId))
{
    // Process agent response(s)...
}
```

```
// Delete the thread when it is no longer needed  
await agent.DeleteThreadAsync(threadId);
```

Deleting an *Open AI Assistant Agent*

Since the assistant's definition is stored remotely, it supports the capability to self-delete. This enables the agent to be removed from the system when it is no longer needed.

Note: Attempting to use an agent instance after being deleted results in an exception.

For .NET, the agent identifier is exposed as a `string` via the `Agent.Id` property defined by any agent.

C#

```
// Perform the deletion  
await agent.DeleteAsync();  
  
// Inspect whether an agent has been deleted  
bool isDeleted = agent.IsDeleted();
```

How-To

For an end-to-end example for a *Open AI Assistant Agent*, see:

- [How-To: Open AI Assistant Agent Code Interpreter](#)
- [How-To: Open AI Assistant Agent File Search](#)

[Agent Collaboration inAgent Chat](#)

Exploring Agent Collaboration in *Agent Chat* (Experimental)

Article • 10/09/2024

⚠ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Detailed API documentation related to this discussion is available at:

- [AgentChat](#)
- [AgentGroupChat](#)
- [Microsoft.SemanticKernel.Agents.Chat](#)

What is *Agent Chat*?

Agent Chat provides a framework that enables interaction between multiple agents, even if they are of different types. This makes it possible for a *Chat Completion Agent* and an *Open AI Assistant Agent* to work together within the same conversation. *Agent Chat* also defines entry points for initiating collaboration between agents, whether through multiple responses or a single agent response.

As an abstract class, *Agent Chat* can be subclassed to support custom scenarios.

One such subclass, *Agent Group Chat*, offers a concrete implementation of *Agent Chat*, using a strategy-based approach to manage conversation dynamics.

Creating an *Agent Group Chat*

To create an *Agent Group Chat*, you may either specify the participating agents or create an empty chat and subsequently add agent participants. Configuring the *Chat-Settings* and *Strategies* is also performed during *Agent Group Chat* initialization. These settings define how the conversation dynamics will function within the group.

Note: The default *Chat-Settings* result in a conversation that is limited to a single response. See [Agent Chat Behavior](#) for details on configuring *_Chat-Settings*.

Creating Agent Group Chat with Agents:

```
C#  
  
// Define agents  
ChatCompletionAgent agent1 = ...;  
OpenAIAssistantAgent agent2 = ...;  
  
// Create chat with participating agents.  
AgentGroupChat chat = new(agent1, agent2);
```

Adding Agents to a Agent Group Chat:

```
C#  
  
// Define agents  
ChatCompletionAgent agent1 = ...;  
OpenAIAssistantAgent agent2 = ...;  
  
// Create an empty chat.  
AgentGroupChat chat = new();  
  
// Add agents to an existing chat.  
chat.AddAgent(agent1);  
chat.AddAgent(agent2);
```

Using Agent Group Chat

Agent Chat supports two modes of operation: *Single-Turn* and *Multi-Turn*. In *single-turn*, a specific agent is designated to provide a response. In *multi-turn*, all agents in the conversation take turns responding until a termination criterion is met. In both modes, agents can collaborate by responding to one another to achieve a defined goal.

Providing Input

Adding an input message to an *Agent Chat* follows the same pattern as with a *Chat History* object.

```
C#  
  
AgentGroupChat chat = new();  
  
chat.AddChatMessage(new ChatMessageContent(AuthorRole.User, "<message content>"));
```

Single-Turn Agent Invocation

In a multi-turn invocation, the system must decide which agent responds next and when the conversation should end. In contrast, a single-turn invocation simply returns a response from the specified agent, allowing the caller to directly manage agent participation.

After an agent participates in the *Agent Chat* through a single-turn invocation, it is added to the set of *agents* eligible for multi-turn invocation.

```
C#
```

```
// Define an agent
ChatCompletionAgent agent = ...;

// Create an empty chat.
AgentGroupChat chat = new();

// Invoke an agent for its response
ChatMessageContent[] messages = await
chat.InvokeAsync(agent).ToArrayAsync();
```

Multi-Turn Agent Invocation

While agent collaboration requires that a system must be in place that not only determines which agent should respond during each turn but also assesses when the conversation has achieved its intended goal, initiating multi-turn collaboration remains straightforward.

Agent responses are returned asynchronously as they are generated, allowing the conversation to unfold in real-time.

Note: In following sections, [Agent Selection](#) and [Chat Termination](#), will delve into the *Execution Settings* in detail. The default *Execution Settings* employs sequential or round-robin selection and limits agent participation to a single turn.

.NET *Execution Settings* API: [AgentGroupChatSettings](#)

```
C#
```

```
// Define agents
ChatCompletionAgent agent1 = ...;
OpenAIAssistantAgent agent2 = ...;

// Create chat with participating agents.
```

```
AgentGroupChat chat =
    new(agent1, agent2)
{
    // Override default execution settings
    ExecutionSettings =
    {
        TerminationStrategy = { MaximumIterations = 10 }
    }
};

// Invoke agents
await foreach (ChatMessageContent response in chat.InvokeAsync())
{
    // Process agent response(s)...
}
```

Accessing Chat History

The *Agent Chat* conversation history is always accessible, even though messages are delivered through the invocation pattern. This ensures that past exchanges remain available throughout the conversation.

Note: The most recent message is provided first (descending order: newest to oldest).

C#

```
// Define and use a chat
AgentGroupChat chat = ...;

// Access history for a previously utilized AgentGroupChat
ChatMessageContent[] history = await
    chat.GetChatMessagesAsync().ToArrayAsync();
```

Since different agent types or configurations may maintain their own version of the conversation history, agent specific history is also available by specifying an agent. (For example: [Open AI Assistant](#) versus [Chat Completion Agent](#).)

C#

```
// Agents to participate in chat
ChatCompletionAgent agent1 = ...;
OpenAIAssistantAgent agent2 = ...;

// Define a group chat
AgentGroupChat chat = ...;
```

```
// Access history for a previously utilized AgentGroupChat
ChatMessageContent[] history1 = await
chat.GetChatMessagesAsync(agent1).ToArrayAsync();
ChatMessageContent[] history2 = await
chat.GetChatMessagesAsync(agent2).ToArrayAsync();
```

Defining Agent Group Chat Behavior

Collaboration among agents to solve complex tasks is a core agentic pattern. To use this pattern effectively, a system must be in place that not only determines which agent should respond during each turn but also assesses when the conversation has achieved its intended goal. This requires managing agent selection and establishing clear criteria for conversation termination, ensuring seamless cooperation between agents toward a solution. Both of these aspects are governed by the *Execution Settings* property.

The following sections, [Agent Selection](#) and [Chat Termination](#), will delve into these considerations in detail.

Agent Selection

In multi-turn invocation, agent selection is guided by a *Selection Strategy*. This strategy is defined by a base class that can be extended to implement custom behaviors tailored to specific needs. For convenience, two predefined concrete *Selection Strategies* are also available, offering ready-to-use approaches for handling agent selection during conversations.

If known, an initial agent may be specified to always take the first turn. A history reducer may also be employed to limit token usage when using a strategy based on a *Kernel Function*.

.NET Selection Strategy API:

- [SelectionStrategy](#)
- [SequentialSelectionStrategy](#)
- [KernelFunctionSelectionStrategy](#)
- [Microsoft.SemanticKernel.Agents.History](#)

C#

```
// Define the agent names for use in the function template
const string WriterName = "Writer";
const string ReviewerName = "Reviewer";

// Initialize a Kernel with a chat-completion service
```

```

Kernel kernel = ...;

// Create the agents
ChatCompletionAgent writerAgent =
    new()
{
    Name = WriterName,
    Instructions = "<writer instructions>",
    Kernel = kernel
};

ChatCompletionAgent reviewerAgent =
    new()
{
    Name = ReviewerName,
    Instructions = "<reviewer instructions>",
    Kernel = kernel
};

// Define a kernel function for the selection strategy
KernelFunction selectionFunction =
    AgentGroupChat.CreatePromptFunctionForStrategy(
        $$$"""
        Determine which participant takes the next turn in a conversation
        based on the the most recent participant.
        State only the name of the participant to take the next turn.
        No participant should take more than one turn in a row.

        Choose only from these participants:
        - {{ReviewerName}}
        - {{WriterName}}

        Always follow these rules when selecting the next participant:
        - After {{WriterName}}, it is {{ReviewerName}}'s turn.
        - After {{ReviewerName}}, it is {{WriterName}}'s turn.

        History:
        {{$history}}
        """,
        safeParameterNames: "history");

// Define the selection strategy
KernelFunctionSelectionStrategy selectionStrategy =
    new(selectionFunction, kernel)
{
    // Always start with the writer agent.
    InitialAgent = writerAgent,
    // Parse the function response.
    ResultParser = (result) => result.GetValue<string>() ?? WriterName,
    // The prompt variable name for the history argument.
    HistoryVariableName = "history",
    // Save tokens by not including the entire history in the prompt
    HistoryReducer = new ChatHistoryTruncationReducer(3),
};

```

```
// Create a chat using the defined selection strategy.
AgentGroupChat chat =
    new(writerAgent, reviewerAgent)
{
    ExecutionSettings = new() { SelectionStrategy = selectionStrategy }
};
```

Chat Termination

In *multi-turn* invocation, the *Termination Strategy* dictates when the final turn takes place. This strategy ensures the conversation ends at the appropriate point.

This strategy is defined by a base class that can be extended to implement custom behaviors tailored to specific needs. For convenience, several predefined concrete *Selection Strategies* are also available, offering ready-to-use approaches for defining termination criteria for an *Agent Chat* conversations.

.NET Selection Strategy API:

- [TerminationStrategy](#)
- [RegexTerminationStrategy](#)
- [KernelFunctionSelectionStrategy](#)
- [KernelFunctionTerminationStrategy](#)
- [AggregatorTerminationStrategy](#)
- [Microsoft.SemanticKernel.Agents.History](#)

C#

```
// Initialize a Kernel with a chat-completion service
Kernel kernel = ...;

// Create the agents
ChatCompletionAgent writerAgent =
    new()
{
    Name = "Writer",
    Instructions = "<writer instructions>",
    Kernel = kernel
};

ChatCompletionAgent reviewerAgent =
    new()
{
    Name = "Reviewer",
    Instructions = "<reviewer instructions>",
    Kernel = kernel
};
```

```

// Define a kernel function for the selection strategy
KernelFunction terminationFunction =
    AgentGroupChat.CreatePromptFunctionForStrategy(
        $$$"""
        Determine if the reviewer has approved. If so, respond with a
single word: yes

        History:
        {{\$history}}
        """,
        safeParameterNames: "history");

// Define the termination strategy
KernelFunctionTerminationStrategy terminationStrategy =
    new(selectionFunction, kernel)
{
    // Only the reviewer may give approval.
    Agents = [reviewerAgent],
    // Parse the function response.
    ResultParser = (result) =>
        result.GetValue<string>()?.Contains("yes",
StringComparison.OrdinalIgnoreCase) ?? false,
    // The prompt variable name for the history argument.
    HistoryVariableName = "history",
    // Save tokens by not including the entire history in the prompt
    HistoryReducer = new ChatHistoryTruncationReducer(1),
    // Limit total number of turns no matter what
    MaximumIterations = 10,
};

// Create a chat using the defined termination strategy.
AgentGroupChat chat =
    new(writerAgent, reviewerAgent)
{
    ExecutionSettings = new() { TerminationStrategy =
terminationStrategy }
};

```

Resetting Chat Completion State

Regardless of whether *Agent Group Chat* is invoked using the single-turn or multi-turn approach, the state of the *Agent Group Chat* is updated to indicate it is *completed* once the termination criteria is met. This ensures that the system recognizes when a conversation has fully concluded. To continue using an *Agent Group Chat* instance after it has reached the *Completed* state, this state must be reset to allow further interactions. Without resetting, additional interactions or agent responses will not be possible.

In the case of a multi-turn invocation that reaches the maximum turn limit, the system will cease agent invocation but will not mark the instance as *completed*. This allows for

the possibility of extending the conversation without needing to reset the *Completion* state.

```
C#  
  
// Define an use chat  
AgentGroupChat chat = ...;  
  
// Evaluate if completion is met and reset.  
if (chat.IsComplete)  
{  
    // Opt to take action on the chat result...  
  
    // Reset completion state to continue use  
    chat.IsComplete = false;  
}
```

Clear Full Conversation State

When done using an *Agent Chat* where an [Open AI Assistant](#) participated, it may be necessary to delete the remote *thread* associated with the *assistant*. *Agent Chat* supports resetting or clearing the entire conversation state, which includes deleting any remote *thread* definition. This ensures that no residual conversation data remains linked to the *assistant* once the chat concludes.

A full reset does not remove the *agents* that had joined the *Agent Chat* and leaves the *Agent Chat* in a state where it can be reused. This allows for the continuation of interactions with the same agents without needing to reinitialize them, making future conversations more efficient.

```
C#  
  
// Define an use chat  
AgentGroupChat chat = ...;  
  
// Clear the all conversation state  
await chat.ResetAsync();
```

How-To

For an end-to-end example for using *Agent Group Chat* for Agent collaboration, see:

- [How to Coordinate Agent Collaboration using *Agent Group Chat*](#)

Create an Agent from a Template

Create an Agent from a Semantic Kernel Template (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Prompt Templates in Semantic Kernel

An agent's role is primarily shaped by the instructions it receives, which dictate its behavior and actions. Similar to invoking a `Kernel prompt`, an agent's instructions can include templated parameters—both values and functions—that are dynamically substituted during execution. This enables flexible, context-aware responses, allowing the agent to adjust its output based on real-time input.

Additionally, an agent can be configured directly using a *Prompt Template Configuration*, providing developers with a structured and reusable way to define its behavior. This approach offers a powerful tool for standardizing and customizing agent instructions, ensuring consistency across various use cases while still maintaining dynamic adaptability.

Related API's:

- [PromptTemplateConfig](#)
- [KernelFunctionYaml.FromPromptYaml](#)
- [IPromptTemplateFactory](#)
- [KernelPromptTemplateFactory](#)
- [Handlebars](#)
- [Prompty](#)
- [Liquid](#)

Agent Instructions as a Template

Creating an agent with template parameters provides greater flexibility by allowing its instructions to be easily customized based on different scenarios or requirements. This

approach enables the agent's behavior to be tailored by substituting specific values or functions into the template, making it adaptable to a variety of tasks or contexts. By leveraging template parameters, developers can design more versatile agents that can be configured to meet diverse use cases without needing to modify the core logic.

Chat Completion Agent

C#

```
// Initialize a Kernel with a chat-completion service
Kernel kernel = ...;

ChatCompletionAgent agent =
    new()
{
    Kernel = kernel,
    Name = "StoryTeller",
    Instructions = "Tell a story about {{$topic}} that is {{$length}}
sentences long.",
    Arguments = new KernelArguments()
    {
        { "topic", "Dog" },
        { "length", "3" },
    }
};
```

Open AI Assistant Agent

Templated instructions are especially powerful when working with an [Open AI Assistant Agent](#). With this approach, a single assistant definition can be created and reused multiple times, each time with different parameter values tailored to specific tasks or contexts. This enables a more efficient setup, allowing the same assistant framework to handle a wide range of scenarios while maintaining consistency in its core behavior.

C#

```
// Retrieve an existing assistant definition by identifier
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.RetrieveAsync(
        this.GetClientProvider(),
        "<stored agent-identifier>",
        new Kernel(),
        new KernelArguments()
        {
            { "topic", "Dog" },
            { "length", "3" },
        });
});
```

Agent Definition from a *Prompt Template*

The same *Prompt Template Config* used to create a *Kernel Prompt Function* can also be leveraged to define an agent. This allows for a unified approach in managing both prompts and agents, promoting consistency and reuse across different components. By externalizing agent definitions from the codebase, this method simplifies the management of multiple agents, making them easier to update and maintain without requiring changes to the underlying logic. This separation also enhances flexibility, enabling developers to modify agent behavior or introduce new agents by simply updating the configuration, rather than adjusting the code itself.

YAML Template

YAML

```
name: GenerateStory
template: |
  Tell a story about {{$topic}} that is {{$length}} sentences long.
template_format: semantic-kernel
description: A function that generates a story about a topic.
input_variables:
  - name: topic
    description: The topic of the story.
    is_required: true
  - name: length
    description: The number of sentences in the story.
    is_required: true
```

Agent Initialization

C#

```
// Read YAML resource
string generateStoryYaml = File.ReadAllText("./GenerateStory.yaml");
// Convert to a prompt template config
PromptTemplateConfig templateConfig =
KernelFunctionYaml.ToPromptTemplateConfig(generateStoryYaml);

// Create agent with Instructions, Name and Description
// provided by the template config.
ChatCompletionAgent agent =
new(templateConfig)
{
  Kernel = this.CreateKernelWithChatCompletion(),
  // Provide default values for template parameters
  Arguments = new KernelArguments()
{
```

```
        { "topic", "Dog" },
        { "length", "3" },
    }
};
```

Overriding Template Values for Direct Invocation

When invoking an agent directly, without using *Agent Chat*, the agent's parameters can be overridden as needed. This allows for greater control and customization of the agent's behavior during specific tasks, enabling you to modify its instructions or settings on the fly to suit particular requirements.

C#

```
// Initialize a Kernel with a chat-completion service
Kernel kernel = ...;

ChatCompletionAgent agent =
    new()
{
    Kernel = kernel,
    Name = "StoryTeller",
    Instructions = "Tell a story about {{$topic}} that is {{$length}}
sentences long.",
    Arguments = new KernelArguments()
    {
        { "topic", "Dog" },
        { "length", "3" },
    }
};

// Create a ChatHistory object to maintain the conversation state.
ChatHistory chat = [];

KernelArguments overrideArguments =
    new()
{
    { "topic", "Cat" },
    { "length", "3" },
});

// Generate the agent response(s)
await foreach (ChatMessageContent response in agent.InvokeAsync(chat,
overrideArguments))
{
    // Process agent response(s)...
}
```

How-To

For an end-to-end example for creating an agent from a *pmompt-template*, see:

- [How-To: Chat Completion Agent](#)

[Configuring Agents with Plugins](#)

Configuring Agents with Semantic Kernel Plugins (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Functions and Plugins in Semantic Kernel

Function calling is a powerful tool that allows developers to add custom functionalities and expand the capabilities of AI applications. The *Semantic Kernel Plugin* architecture offers a flexible framework to support [Function Calling](#). For an *Agent*, integrating [Plugins](#) and [Function Calling](#) is built on this foundational *Semantic Kernel* feature.

Once configured, an agent will choose when and how to call an available function, as it would in any usage outside of the *Agent Framework*.

- [KernelFunctionFactory](#)
- [KernelFunction](#)
- [KernelPluginFactory](#)
- [KernelPlugin](#)
- [Kernel.Plugins](#)

Adding Plugins to an Agent

Any [Plugin](#) available to an *Agent* is managed within its respective *Kernel* instance. This setup enables each *Agent* to access distinct functionalities based on its specific role.

[Plugins](#) can be added to the *Kernel* either before or after the *Agent* is created. The process of initializing [Plugins](#) follows the same patterns used for any *Semantic Kernel* implementation, allowing for consistency and ease of use in managing AI capabilities.

Note: For a [Chat Completion Agent](#), the function calling mode must be explicitly enabled. [Open AI Assistant](#) agent is always based on automatic function calling.

```

// Factory method to produce an agent with a specific role.
// Could be incorporated into DI initialization.
ChatCompletionAgent CreateSpecificAgent(Kernel kernel, string credentials)
{
    // Clone kernel instance to allow for agent specific plug-in definition
    Kernel agentKernel = kernel.Clone();

    // Initialize plug-in from type
    agentKernel.CreatePluginFromType<StatelessPlugin>();

    // Initialize plug-in from object
    agentKernel.CreatePluginFromObject(new StatefulPlugin(credentials));

    // Create the agent
    return
        new ChatCompletionAgent()
    {
        Name = "<agent name>",
        Instructions = "<agent instructions>",
        Kernel = agentKernel,
        Arguments = new KernelArguments(
            new OpenAIPromptExecutionSettings()
            {
                FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
            }
        );
    };
}

```

Adding Functions to an Agent

A [Plugin](#) is the most common approach for configuring [Function Calling](#). However, individual functions can also be supplied independently including *prompt functions*.

C#

```

// Factory method to produce an agent with a specific role.
// Could be incorporated into DI initialization.
ChatCompletionAgent CreateSpecificAgent(Kernel kernel)
{
    // Clone kernel instance to allow for agent specific plug-in definition
    Kernel agentKernel = kernel.Clone();

    // Initialize plug-in from a static function
    agentKernel.CreateFunctionFromMethod(StatelessPlugin.AStaticMethod);

    // Initialize plug-in from a prompt
    agentKernel.CreateFunctionFromPrompt("<your prompt instructions>");

    // Create the agent
    return

```

```
new ChatCompletionAgent()
{
    Name = "<agent name>",
    Instructions = "<agent instructions>",
    Kernel = agentKernel,
    Arguments = new KernelArguments(
        new OpenAIPromptExecutionSettings()
    {
        FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
    })
};

}
```

Limitations for Agent Function Calling

When directly invoking a *Chat Completion Agent*, all *Function Choice Behaviors* are supported. However, when using an *Open AI Assistant* or *Agent Chat*, only *Automatic Function Calling* is currently available.

How-To

For an end-to-end example for using function calling, see:

- How-To: *Chat Completion Agent*

[How to Stream Agent Responses](#)

How to Stream Agent Responses. (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

What is a Streamed Response?

A streamed response delivers the message content in small, incremental chunks. This approach enhances the user experience by allowing them to view and engage with the message as it unfolds, rather than waiting for the entire response to load. Users can begin processing information immediately, improving the sense of responsiveness and interactivity. As a result, it minimizes delays and keeps users more engaged throughout the communication process.

Streaming References:

- [Open AI Streaming Guide ↗](#)
- [Open AI Chat Completion Streaming ↗](#)
- [Open AI Assistant Streaming ↗](#)
- [Azure OpenAI Service REST API](#)

Streaming in Semantic Kernel

[AI Services](#) that support streaming in Semantic Kernel use different content types compared to those used for fully-formed messages. These content types are specifically designed to handle the incremental nature of streaming data. The same content types are also utilized within the Agent Framework for similar purposes. This ensures consistency and efficiency across both systems when dealing with streaming information.

- [StreamingChatMessageContent](#)
- [StreamingTextContent](#)
- [StreamingFileReferenceContent](#)

- [StreamingAnnotationContent](#)

Streaming Agent Invocation

The *Agent Framework* supports *streamed* responses when using [Agent Chat](#) or when directly invoking a [Chat Completion Agent](#) or [Open AI Assistant Agent](#). In either mode, the framework delivers responses asynchronously as they are streamed. Alongside the streamed response, a consistent, non-streamed history is maintained to track the conversation. This ensures both real-time interaction and a reliable record of the conversation's flow.

Streamed response from *Chat Completion Agent*

When invoking a streamed response from a [Chat Completion Agent](#), the *Chat History* is updated after the full response is received. Although the response is streamed incrementally, the history records only the complete message. This ensures that the *Chat History* reflects fully formed responses for consistency.

```
// Define agent
ChatCompletionAgent agent = ...;

// Create a ChatHistory object to maintain the conversation state.
ChatHistory chat = [];

// Add a user message to the conversation
chat.Add(new ChatMessageContent(AuthorRole.User, "<user input>"));

// Generate the streamed agent response(s)
await foreach (StreamingChatMessageContent response in
agent.InvokeStreamingAsync(chat))
{
    // Process streamed response(s)...
}
```

Streamed response from *Open AI Assistant Agent*

When invoking a streamed response from an [Open AI Assistant Agent](#), an optional *Chat History* can be provided to capture the complete messages for further analysis if needed. Since the assistant maintains the conversation state as a remote thread, capturing these messages is not always necessary. The decision to store and analyze the full response depends on the specific requirements of the interaction.

```

// Define agent
OpenAIAssistantAgent agent = ...;

// Create a thread for the agent conversation.
string threadId = await agent.CreateThreadAsync();

// Add a user message to the conversation
chat.Add(threadId, new ChatMessageContent(AuthorRole.User, "<user input>"));

// Generate the streamed agent response(s)
await foreach (StreamingChatMessageContent response in
agent.InvokeStreamingAsync(threadId))
{
    // Process streamed response(s)...
}

// Delete the thread when it is no longer needed
await agent.DeleteThreadAsync(threadId);

```

Streaming with *Agent Chat*

When using *Agent Chat*, the full conversation history is always preserved and can be accessed directly through the *Agent Chat* instance. Therefore, the key difference between streamed and non-streamed invocations lies in the delivery method and the resulting content type. In both cases, users can still access the complete history, but streamed responses provide real-time updates as the conversation progresses. This allows for greater flexibility in handling interactions, depending on the application's needs.

```

// Define agents
ChatCompletionAgent agent1 = ...;
OpenAIAssistantAgent agent2 = ...;

// Create chat with participating agents.
AgentGroupChat chat =
    new(agent1, agent2)
{
    // Override default execution settings
    ExecutionSettings =
    {
        TerminationStrategy = { MaximumIterations = 10 }
    }
};

// Invoke agents
string lastAgent = string.Empty;
await foreach (StreamingChatMessageContent response in
chat.InvokeStreamingAsync())

```

```
{  
    if (!lastAgent.Equals(response.AuthorName, StringComparison.OrdinalIgnoreCase))  
    {  
        // Process begining of agent response  
        lastAgent = response.AuthorName;  
    }  
  
    // Process streamed content...  
}
```

How-To: Chat Completion Agent (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Overview

In this sample, we will explore configuring a plugin to access *Github API* and provide templated instructions to a *Chat Completion Agent* to answer questions about a *Github* repository. The approach will be broken down step-by-step to highlight the key parts of the coding process. As part of the task, the agent will provide document citations within the response.

Streaming will be used to deliver the agent's responses. This will provide real-time updates as the task progresses.

Getting Started

Before proceeding with feature coding, make sure your development environment is fully set up and configured.

Start by creating a *Console* project. Then, include the following package references to ensure all required dependencies are available.

To add package dependencies from the command-line use the `dotnet` command:

PowerShell

```
dotnet add package Azure.Identity  
dotnet add package Microsoft.Extensions.Configuration  
dotnet add package Microsoft.Extensions.Configuration.Binder  
dotnet add package Microsoft.Extensions.Configuration.UserSecrets  
dotnet add package Microsoft.Extensions.Configuration.EnvironmentVariables  
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI  
dotnet add package Microsoft.SemanticKernel.Actors.Core --prerelease
```

If managing *NuGet* packages in *Visual Studio*, ensure `Include prerelease` is checked.

The project file (`.csproj`) should contain the following `PackageReference` definitions:

XML

```
<ItemGroup>
  <PackageReference Include="Azure.Identity" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.UserSecrets" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="<stable>" />
  <PackageReference Include="Microsoft.SemanticKernel.Agents.Core" Version="<latest>" />
  <PackageReference
    Include="Microsoft.SemanticKernel.Connectors.AzureOpenAI" Version="<latest>" />
</ItemGroup>
```

The *Agent Framework* is experimental and requires warning suppression. This may be addressed in as a property in the project file (`.csproj`):

XML

```
<PropertyGroup>
  <NoWarn>$(NoWarn);CA2007;IDE1006;SKEXP0001;SKEXP0110;OPENAI001</NoWarn>
</PropertyGroup>
```

Additionally, copy the GitHub plug-in and models (`GitHubPlugin.cs` and `GitHubModels.cs`) from [Semantic Kernel LearnResources Project](#). Add these files in your project folder.

Configuration

This sample requires configuration setting in order to connect to remote services. You will need to define settings for either *Open AI* or *Azure Open AI* and also for *GitHub*.

Note: For information on GitHub *Personal Access Tokens*, see: [Managing your personal access tokens](#).

PowerShell

```
# Open AI
dotnet user-secrets set "OpenAISettings:ApiKey" "<api-key>"
dotnet user-secrets set "OpenAISettings:ChatModel" "gpt-4o"

# Azure Open AI
dotnet user-secrets set "AzureOpenAISettings:ApiKey" "<api-key>" # Not required if using token-credential
dotnet user-secrets set "AzureOpenAISettings:Endpoint" "<model-endpoint>"
dotnet user-secrets set "AzureOpenAISettings:ChatModelDeployment" "gpt-4o"

# GitHub
dotnet user-secrets set "GitHubSettings:BaseUrl" "https://api.github.com"
dotnet user-secrets set "GitHubSettings:Token" "<personal access token>"
```

The following class is used in all of the Agent examples. Be sure to include it in your project to ensure proper functionality. This class serves as a foundational component for the examples that follow.

```
c#
using System.Reflection;
using Microsoft.Extensions.Configuration;

namespace AgentsSample;

public class Settings
{
    private readonly IConfigurationRoot configRoot;

    private AzureOpenAISettings azureOpenAI;
    private OpenAISettings openAI;

    public AzureOpenAISettings AzureOpenAI => this.azureOpenAI ??
this.GetSettings<Settings.AzureOpenAISettings>();
    public OpenAISettings OpenAI => this.openAI ??
this.GetSettings<Settings.OpenAISettings>();

    public class OpenAISettings
    {
        public string ChatModel { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }

    public class AzureOpenAISettings
    {
        public string ChatModelDeployment { get; set; } = string.Empty;
```

```

        public string Endpoint { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }

    public TSettings GetSettings<TSettings>() =>

this.configRoot.GetRequiredSection(typeof(TSettings).Name).Get<TSettings>()
!;

    public Settings()
{
    this.configRoot =
        new ConfigurationBuilder()
            .AddEnvironmentVariables()
            .AddUserSecrets(Assembly.GetExecutingAssembly(), optional:
true)
            .Build();
}
}

```

Coding

The coding process for this sample involves:

1. [Setup](#) - Initializing settings and the plug-in.
2. [Agent Definition](#) - Create the *Chat Completion Agent* with templated instructions and plug-in.
3. [The Chat Loop](#) - Write the loop that drives user / agent interaction.

The full example code is provided in the [Final](#) section. Refer to that section for the complete implementation.

Setup

Prior to creating a *Chat Completion Agent*, the configuration settings, plugins, and *Kernel* must be initialized.

Initialize the `Settings` class referenced in the previous [Configuration](#) section.

C#

```
Settings settings = new();
```

Initialize the plug-in using its settings.

Here, a message is displaying to indicate progress.

```
C#
```

```
Console.WriteLine("Initialize plugins...");  
GitHubSettings githubSettings = settings.GetSettings<GitHubSettings>();  
GitHubPlugin githubPlugin = new(githubSettings);
```

Now initialize a `Kernel` instance with an `IChatCompletionService` and the `GitHubPlugin` previously created.

```
C#
```

```
Console.WriteLine("Creating kernel...");  
IKernelBuilder builder = Kernel.CreateBuilder();  
  
builder.AddAzureOpenAIChatCompletion(  
    settings.AzureOpenAI.ChatModelDeployment,  
    settings.AzureOpenAI.Endpoint,  
    new AzureCliCredential());  
  
builder.Plugins.AddFromObject(githubPlugin);  
  
Kernel kernel = builder.Build();
```

Agent Definition

Finally we are ready to instantiate a *Chat Completion Agent* with its *Instructions*, associated *Kernel*, and the default *Arguments* and *Execution Settings*. In this case, we desire to have the any plugin functions automatically executed.

```
C#
```

```
Console.WriteLine("Defining agent...");  
ChatCompletionAgent agent =  
    new()  
    {  
        Name = "SampleAssistantAgent",  
        Instructions =  
            """  
                You are an agent designed to query and retrieve information from  
                a single GitHub repository in a read-only manner.  
                You are also able to access the profile of the active user.  
  
                Use the current date and time to provide up-to-date details or  
                time-sensitive responses.  
  
                The repository you are querying is a public repository with the  
                following name: {{$repository}}  
  
                The current date and time is: {{$now}}.  
            """
```

```

    """,
    Kernel = kernel,
    Arguments =
        new KernelArguments(new AzureOpenAIPromptExecutionSettings() {
FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() })
    {
        { "repository", "microsoft/semantic-kernel" }
    }
};

Console.WriteLine("Ready!");

```

The *Chat* Loop

At last, we are able to coordinate the interaction between the user and the *Agent*. Start by creating a *Chat History* object to maintain the conversation state and creating an empty loop.

```
C#
ChatHistory history = [];
bool isComplete = false;
do
{
    // processing logic here
} while (!isComplete);
```

Now let's capture user input within the previous loop. In this case, empty input will be ignored and the term `EXIT` will signal that the conversation is completed. Valid input will be added to the *Chat History* as a *User* message.

```
C#
Console.WriteLine();
Console.Write("> ");
string input = Console.ReadLine();
if (string.IsNullOrWhiteSpace(input))
{
    continue;
}
if (input.Trim().Equals("EXIT", StringComparison.OrdinalIgnoreCase))
{
    isComplete = true;
    break;
}

history.Add(new ChatMessageContent(AuthorRole.User, input));
```

```
Console.WriteLine();
```

To generate a *Agent* response to user input, invoke the agent using *Arguments* to provide the final template parameter that specifies the current date and time.

The *Agent* response is then then displayed to the user.

C#

```
DateTime now = DateTime.Now;
KernelArguments arguments =
    new()
{
    { "now", $"{now.ToShortDateString()} {now.ToShortTimeString()}" }
};
await foreach (ChatMessageContent response in agent.InvokeAsync(history,
    arguments))
{
    Console.WriteLine($"{response.Content}");
}
```

Final

Bringing all the steps together, we have the final code for this example. The complete implementation is provided below.

Try using these suggested inputs:

1. What is my username?
2. Describe the repo.
3. Describe the newest issue created in the repo.
4. List the top 10 issues closed within the last week.
5. How were these issues labeled?
6. List the 5 most recently opened issues with the "Agents" label

C#

```
using System;
using System.Threading.Tasks;
using Azure.Identity;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Actors;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;
using Plugins;
```

```
namespace AgentsSample;

public static class Program
{
    public static async Task Main()
    {
        // Load configuration from environment variables or user secrets.
        Settings settings = new();

        Console.WriteLine("Initialize plugins...");
        GitHubSettings githubSettings = settings.GetSettings<GitHubSettings>();
        GitHubPlugin githubPlugin = new(githubSettings);

        Console.WriteLine("Creating kernel...");
        IKernelBuilder builder = Kernel.CreateBuilder();

        builder.AddAzureOpenAIChatCompletion(
            settings.AzureOpenAI.ChatModelDeployment,
            settings.AzureOpenAI.Endpoint,
            new AzureCliCredential());

        builder.Plugins.AddFromObject(githubPlugin);

        Kernel kernel = builder.Build();

        Console.WriteLine("Defining agent...");
        ChatCompletionAgent agent =
            new()
            {
                Name = "SampleAssistantAgent",
                Instructions =
                    """
                        You are an agent designed to query and retrieve
                        information from a single GitHub repository in a read-only manner.
                        You are also able to access the profile of the
                        active user.

                        Use the current date and time to provide up-to-date
                        details or time-sensitive responses.

                        The repository you are querying is a public
                        repository with the following name: {{$repository}}
                    """
            },
            Kernel = kernel,
            Arguments =
                new KernelArguments(new
                    AzureOpenAIPromptExecutionSettings() { FunctionChoiceBehavior =
                    FunctionChoiceBehavior.Auto() })
                {
                    { "repository", "microsoft/semantic-kernel" }
                }
            );
    }
}
```

```

Console.WriteLine("Ready!");

ChatHistory history = [];
bool isComplete = false;
do
{
    Console.WriteLine();
    Console.Write("> ");
    string input = Console.ReadLine();
    if (string.IsNullOrWhiteSpace(input))
    {
        continue;
    }
    if (input.Trim().Equals("EXIT",
StringComparison.OrdinalIgnoreCase))
    {
        isComplete = true;
        break;
    }

    history.Add(new ChatMessageContent(AuthorRole.User, input));

    Console.WriteLine();

    DateTime now = DateTime.Now;
    KernelArguments arguments =
        new()
        {
            { "now", $"{now.ToShortDateString()}"
{now.ToShortTimeString()}" }
        };
    await foreach (ChatMessageContent response in
agent.InvokeAsync(history, arguments))
    {
        // Display response.
        Console.WriteLine($"{response.Content}");
    }

} while (!isComplete);
}
}

```

[How-To:Open AI Assistant AgentCode Interpreter](#)

How-To: Open AI Assistant Agent Code Interpreter (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Overview

In this sample, we will explore how to use the *code-interpreter* tool of an [Open AI Assistant Agent](#) to complete data-analysis tasks. The approach will be broken down step-by-step to highlight the key parts of the coding process. As part of the task, the agent will generate both image and text responses. This will demonstrate the versatility of this tool in performing quantitative analysis.

Streaming will be used to deliver the agent's responses. This will provide real-time updates as the task progresses.

Getting Started

Before proceeding with feature coding, make sure your development environment is fully set up and configured.

Start by creating a *Console* project. Then, include the following package references to ensure all required dependencies are available.

To add package dependencies from the command-line use the `dotnet` command:

PowerShell

```
dotnet add package Azure.Identity  
dotnet add package Microsoft.Extensions.Configuration  
dotnet add package Microsoft.Extensions.Configuration.Binder  
dotnet add package Microsoft.Extensions.Configuration.UserSecrets  
dotnet add package Microsoft.Extensions.Configuration.EnvironmentVariables  
dotnet add package Microsoft.SemanticKernel  
dotnet add package Microsoft.SemanticKernel.Actors.Core --prerelease
```

If managing *NuGet* packages in *Visual Studio*, ensure `Include prerelease` is checked.

The project file (`.csproj`) should contain the following `PackageReference` definitions:

XML

```
<ItemGroup>
  <PackageReference Include="Azure.Identity" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.UserSecrets" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="<stable>" />
  <PackageReference Include="Microsoft.SemanticKernel" Version="<latest>" />
  <PackageReference Include="Microsoft.SemanticKernel.Agents.OpenAI" Version="<latest>" />
</ItemGroup>
```

The *Agent Framework* is experimental and requires warning suppression. This may be addressed in as a property in the project file (`.csproj`):

XML

```
<PropertyGroup>
  <NoWarn>$(NoWarn);CA2007;IDE1006;SKEXP0001;SKEXP0110;OPENAI001</NoWarn>
</PropertyGroup>
```

Additionally, copy the `PopulationByAdmin1.csv` and `PopulationByCountry.csv` data files from [Semantic Kernel LearnResources Project](#). Add these files in your project folder and configure to have them copied to the output directory:

XML

```
<ItemGroup>
  <None Include="PopulationByAdmin1.csv">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Include="PopulationByCountry.csv">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

```
</None>
</ItemGroup>
```

Configuration

This sample requires configuration setting in order to connect to remote services. You will need to define settings for either *Open AI* or *Azure Open AI*.

PowerShell

```
# Open AI
dotnet user-secrets set "OpenAISettings:ApiKey" "<api-key>"
dotnet user-secrets set "OpenAISettings:ChatModel" "gpt-4o"

# Azure Open AI
dotnet user-secrets set "AzureOpenAISettings:ApiKey" "<api-key>" # Not
required if using token-credential
dotnet user-secrets set "AzureOpenAISettings:Endpoint" "<model-endpoint>"
dotnet user-secrets set "AzureOpenAISettings:ChatModelDeployment" "gpt-4o"
```

The following class is used in all of the Agent examples. Be sure to include it in your project to ensure proper functionality. This class serves as a foundational component for the examples that follow.

```
c#
using System.Reflection;
using Microsoft.Extensions.Configuration;

namespace AgentsSample;

public class Settings
{
    private readonly IConfigurationRoot configRoot;

    private AzureOpenAISettings azureOpenAI;
    private OpenAISettings openAI;

    public AzureOpenAISettings AzureOpenAI => this.azureOpenAI ??=
this.GetSettings<Settings.AzureOpenAISettings>();
    public OpenAISettings OpenAI => this.openAI ??=
this.GetSettings<Settings.OpenAISettings>();

    public class OpenAISettings
    {
        public string ChatModel { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }
}
```

```

public class AzureOpenAISettings
{
    public string ChatModelDeployment { get; set; } = string.Empty;
    public string Endpoint { get; set; } = string.Empty;
    public string ApiKey { get; set; } = string.Empty;
}

public TSettings GetSettings<TSettings>() =>

this.configRoot.GetRequiredSection(typeof(TSettings).Name).Get<TSettings>()
!;

public Settings()
{
    this.configRoot =
        new ConfigurationBuilder()
            .AddEnvironmentVariables()
            .AddUserSecrets(Assembly.GetExecutingAssembly(), optional:
true)
            .Build();
}
}

```

Coding

The coding process for this sample involves:

1. [Setup](#) - Initializing settings and the plug-in.
2. [Agent Definition](#) - Create the *OpenAI_Assistant_Agent* with templated instructions and plug-in.
3. [The Chat Loop](#) - Write the loop that drives user / agent interaction.

The full example code is provided in the [Final](#) section. Refer to that section for the complete implementation.

Setup

Prior to creating an *Open AI Assistant Agent*, ensure the configuration settings are available and prepare the file resources.

Instantiate the `Settings` class referenced in the previous [Configuration](#) section. Use the settings to also create an `OpenAIClientProvider` that will be used for the [Agent Definition](#) as well as file-upload.

C#

```
Settings settings = new();

OpenAIClientProvider clientProvider =
    OpenAIClientProvider.ForAzureOpenAI(new AzureCliCredential(), new
Uri(settings.AzureOpenAI.Endpoint));
```

Use the `OpenAIClientProvider` to access a `FileClient` and upload the two data-files described in the previous [Configuration](#) section, preserving the *File Reference* for final clean-up.

C#

```
Console.WriteLine("Uploading files...");
FileClient fileClient = clientProvider.Client.GetFileClient();
OpenAIFileInfo fileDataCountryDetail = await
fileClient.UploadFileAsync("PopulationByAdmin1.csv",
FileUploadPurpose.Assistants);
OpenAIFileInfo fileDataCountryList = await
fileClient.UploadFileAsync("PopulationByCountry.csv",
FileUploadPurpose.Assistants);
```

Agent Definition

We are now ready to instantiate an *OpenAI Assistant Agent*. The agent is configured with its target model, *Instructions*, and the *Code Interpreter* tool enabled. Additionally, we explicitly associate the two data files with the *Code Interpreter* tool.

C#

```
Console.WriteLine("Defining agent...");
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.CreateAsync(
        clientProvider,
        new
OpenAIAssistantDefinition(settings.AzureOpenAI.ChatModelDeployment)
{
    Name = "SampleAssistantAgent",
    Instructions =
    """
        Analyze the available data to provide an answer to the
user's question.
        Always format response using markdown.
        Always include a numerical index that starts at 1 for any
lists or tables.
        Always sort lists in ascending order.
    """,
    EnableCodeInterpreter = true,
    CodeInterpreterFileIds = [fileDataCountryList.Id,
```

```
fileDataCountryDetail.Id],  
    },  
    new Kernel());
```

The *Chat Loop*

At last, we are able to coordinate the interaction between the user and the *Agent*. Start by creating an *Assistant Thread* to maintain the conversation state and creating an empty loop.

Let's also ensure the resources are removed at the end of execution to minimize unnecessary charges.

C#

```
Console.WriteLine("Creating thread...");  
string threadId = await agent.CreateThreadAsync();  
  
Console.WriteLine("Ready!");  
  
try  
{  
    bool isComplete = false;  
    List<string> fileIds = [];  
    do  
    {  
        } while (!isComplete);  
}  
finally  
{  
    Console.WriteLine();  
    Console.WriteLine("Cleaning-up...");  
    await Task.WhenAll(  
        [  
            agent.DeleteThreadAsync(threadId),  
            agent.DeleteAsync(),  
            fileClient.DeleteFileAsync(fileDataCountryList.Id),  
            fileClient.DeleteFileAsync(fileDataCountryDetail.Id),  
        ]);  
}
```

Now let's capture user input within the previous loop. In this case, empty input will be ignored and the term `EXIT` will signal that the conversation is completed. Valid input will be added to the *Assistant Thread* as a *User message*.

C#

```

Console.WriteLine();
Console.Write("> ");
string input = Console.ReadLine();
if (string.IsNullOrWhiteSpace(input))
{
    continue;
}
if (input.Trim().Equals("EXIT", StringComparison.OrdinalIgnoreCase))
{
    isComplete = true;
    break;
}

await agent.AddChatMessageAsync(threadId, new
ChatMessageContent(AuthorRole.User, input));

Console.WriteLine();

```

Before invoking the *Agent* response, let's add some helper methods to download any files that may be produced by the *Agent*.

Here we're place file content in the system defined temporary directory and then launching the system defined viewer application.

C#

```

private static async Task DownloadResponseImageAsync(FileClient client,
ICollection<string> fileIds)
{
    if (fileIds.Count > 0)
    {
        Console.WriteLine();
        foreach (string fileId in fileIds)
        {
            await DownloadFileContentAsync(client, fileId, launchViewer:
true);
        }
    }
}

private static async Task DownloadFileContentAsync(FileClient client, string
fileId, bool launchViewer = false)
{
    OpenAIFileInfo fileInfo = client.GetFile(fileId);
    if (fileInfo.Purpose == OpenAIFilePurpose.AssistantsOutput)
    {
        string filePath =
            Path.Combine(
                Path.GetTempPath(),
                Path.GetFileName(Path.ChangeExtension(fileInfo.Filename,
".png")));

```

```

        BinaryData content = await client.DownloadFileAsync(fileId);
        await using FileStream fileStream = new(filePath,
        FileMode.CreateNew);
        await content.ToStream().CopyToAsync(fileStream);
        Console.WriteLine($"File saved to: {filePath}.");

        if (launchViewer)
        {
            Process.Start(
                new ProcessStartInfo
                {
                    FileName = "cmd.exe",
                    Arguments = $"{"/C start {filePath}"}
                });
        }
    }
}

```

To generate an *Agent* response to user input, invoke the agent by specifying the *Assistant Thread*. In this example, we choose a streamed response and capture any generated *File References* for download and review at the end of the response cycle. It's important to note that generated code is identified by the presence of a *Metadata* key in the response message, distinguishing it from the conversational reply.

C#

```

bool isCode = false;
await foreach (StreamingChatMessageContent response in
agent.InvokeStreamingAsync(threadId))
{
    if (isCode !=
(response.Metadata?.ContainsKey(OpenAIAssistantAgent.CodeInterpreterMetadata
Key) ?? false))
    {
        Console.WriteLine();
        isCode = !isCode;
    }

    // Display response.
    Console.Write($"{response.Content}");

    // Capture file IDs for downloading.
    fileIds.AddRange(response.Items.OfType<StreamingFileReferenceContent>
().Select(item => item.FileId));
}

Console.WriteLine();

// Download any files referenced in the response.
await DownloadResponseImageAsync(fileClient, fileIds);
fileIds.Clear();

```

Final

Bringing all the steps together, we have the final code for this example. The complete implementation is provided below.

Try using these suggested inputs:

1. Compare the files to determine the number of countries do not have a state or province defined compared to the total count
2. Create a table for countries with state or province defined. Include the count of states or provinces and the total population
3. Provide a bar chart for countries whose names start with the same letter and sort the x axis by highest count to lowest (include all countries)

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Azure.Identity;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Agents.OpenAI;
using Microsoft.SemanticKernel.ChatCompletion;
using OpenAI.Files;

namespace AgentsSample;

public static class Program
{
    public static async Task Main()
    {
        // Load configuration from environment variables or user secrets.
        Settings settings = new();

        OpenAIClientProvider clientProvider =
            OpenAIClientProvider.ForAzureOpenAI(new AzureCliCredential(),
new Uri(settings.AzureOpenAI.Endpoint));

        Console.WriteLine("Uploading files...");
        FileClient fileClient = clientProvider.Client.GetFileClient();
        OpenAIFileInfo fileDataCountryDetail = await
fileClient.UploadFileAsync("PopulationByAdmin1.csv",
FileUploadPurpose.Assistants);
        OpenAIFileInfo fileDataCountryList = await
fileClient.UploadFileAsync("PopulationByCountry.csv",
FileUploadPurpose.Assistants);

        Console.WriteLine("Defining agent...");
```

```
        OpenAIAssistantAgent agent =
            await OpenAIAssistantAgent.CreateAsync(
                clientProvider,
                new
                    OpenAIAssistantDefinition(settings.AzureOpenAI.ChatModelDeployment)
                {
                    Name = "SampleAssistantAgent",
                    Instructions =
                        """
                            Analyze the available data to provide an answer to
                            the user's question.
                            Always format response using markdown.
                            Always include a numerical index that starts at 1
                            for any lists or tables.
                            Always sort lists in ascending order.
                            """,
                    EnableCodeInterpreter = true,
                    CodeInterpreterFileIds = [fileDataCountryList.Id,
fileDataCountryDetail.Id],
                },
                new Kernel());
           

Console.WriteLine("Creating thread...");
string threadId = await agent.CreateThreadAsync();

Console.WriteLine("Ready!");

try
{
    bool isComplete = false;
    List<string> fileIds = [];
    do
    {
        Console.WriteLine();
        Console.Write("> ");
        string input = Console.ReadLine();
        if (string.IsNullOrWhiteSpace(input))
        {
            continue;
        }
        if (input.Trim().Equals("EXIT",
 StringComparison.OrdinalIgnoreCase))
        {
            isComplete = true;
            break;
        }

        await agent.AddChatMessageAsync(threadId, new
ChatMessageContent(AuthorRole.User, input));

        Console.WriteLine();

        bool isCode = false;
        await foreach (StreamingChatMessageContent response in
agent.InvokeStreamingAsync(threadId))
```

```

        {
            if (isCode !=
(response.Metadata?.ContainsKey(OpenAIAssistantAgent.CodeInterpreterMetadata
Key) ?? false))
            {
                Console.WriteLine();
                isCode = !isCode;
            }

            // Display response.
            Console.Write(${response.Content});

            // Capture file IDs for downloading.

fileIds.AddRange(response.Items.OfType<StreamingFileReferenceContent>
().Select(item => item.FileId));
        }
        Console.WriteLine();

        // Download any files referenced in the response.
        await DownloadResponseImageAsync(fileClient, fileIds);
        fileIds.Clear();

    } while (!isComplete);
}
finally
{
    Console.WriteLine();
    Console.WriteLine("Cleaning-up...");
    await Task.WhenAll(
    [
        agent.DeleteThreadAsync(threadId),
        agent.DeleteAsync(),
        fileClient.DeleteFileAsync(fileDataCountryList.Id),
        fileClient.DeleteFileAsync(fileDataCountryDetail.Id),
    ]);
}
}

private static async Task DownloadResponseImageAsync(FileClient client,
ICollection<string> fileIds)
{
    if (fileIds.Count > 0)
    {
        Console.WriteLine();
        foreach (string fileId in fileIds)
        {
            await DownloadFileContentAsync(client, fileId, launchViewer:
true);
        }
    }
}

private static async Task DownloadFileContentAsync(FileClient client,
string fileId, bool launchViewer = false)

```

```
{  
    OpenAIFileInfo fileInfo = client.GetFile(fileId);  
    if (fileInfo.Purpose == OpenAIFilePurpose.AssistantsOutput)  
    {  
        string filePath =  
            Path.Combine(  
                Path.GetTempPath(),  
                Path.GetFileName(Path.ChangeExtension(fileInfo.Filename,  
".png")));  
  
        BinaryData content = await client.DownloadFileAsync(fileId);  
        await using FileStream fileStream = new(filePath,  
 FileMode.CreateNew);  
        await content.ToStream().CopyToAsync(fileStream);  
        Console.WriteLine($"File saved to: {filePath}");  
  
        if (launchViewer)  
        {  
            Process.Start(  
                new ProcessStartInfo  
                {  
                    FileName = "cmd.exe",  
                    Arguments = $"/C start {filePath}"  
                });  
        }  
    }  
}
```

How-To:Open AI Assistant AgentCode File Search

How-To: Open AI Assistant Agent File Search (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Overview

In this sample, we will explore how to use the *file-search* tool of an *Open AI Assistant Agent* to complete comprehension tasks. The approach will be step-by-step, ensuring clarity and precision throughout the process. As part of the task, the agent will provide document citations within the response.

Streaming will be used to deliver the agent's responses. This will provide real-time updates as the task progresses.

Getting Started

Before proceeding with feature coding, make sure your development environment is fully set up and configured.

To add package dependencies from the command-line use the `dotnet` command:

PowerShell

```
dotnet add package Azure.Identity  
dotnet add package Microsoft.Extensions.Configuration  
dotnet add package Microsoft.Extensions.Configuration.Binder  
dotnet add package Microsoft.Extensions.Configuration.UserSecrets  
dotnet add package Microsoft.Extensions.Configuration.EnvironmentVariables  
dotnet add package Microsoft.SemanticKernel  
dotnet add package Microsoft.SemanticKernel.Actors.Core --prerelease
```

If managing *NuGet* packages in *Visual Studio*, ensure `Include prerelease` is checked.

The project file (.csproj) should contain the following `PackageReference` definitions:

XML

```
<ItemGroup>
  <PackageReference Include="Azure.Identity" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration" Version="<stable>" />
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.UserSecrets" Version="<stable>" />
  <PackageReference
    Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="<stable>" />
  <PackageReference Include="Microsoft.SemanticKernel" Version="<latest>" />
  <PackageReference Include="Microsoft.SemanticKernel.Agents.OpenAI" Version="<latest>" />
</ItemGroup>
```

The *Agent Framework* is experimental and requires warning suppression. This may be addressed in as a property in the project file (.csproj):

XML

```
<PropertyGroup>
  <NoWarn>$(NoWarn);CA2007;IDE1006;SKEXP0001;SKEXP0110;OPENAI001</NoWarn>
</PropertyGroup>
```

Additionally, copy the `Grimms-The-King-of-the-Golden-Mountain.txt`, `Grimms-The-Water-of-Life.txt` and `Grimms-The-White-Snake.txt` public domain content from [Semantic Kernel LearnResources Project](#). Add these files in your project folder and configure to have them copied to the output directory:

XML

```
<ItemGroup>
  <None Include="Grimms-The-King-of-the-Golden-Mountain.txt">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Include="Grimms-The-Water-of-Life.txt">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  <None Include="Grimms-The-White-Snake.txt">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
```

```
</None>
</ItemGroup>
```

Configuration

This sample requires configuration setting in order to connect to remote services. You will need to define settings for either *Open AI* or *Azure Open AI*.

PowerShell

```
# Open AI
dotnet user-secrets set "OpenAISettings:ApiKey" "<api-key>"
dotnet user-secrets set "OpenAISettings:ChatModel" "gpt-4o"

# Azure Open AI
dotnet user-secrets set "AzureOpenAISettings:ApiKey" "<api-key>" # Not
required if using token-credential
dotnet user-secrets set "AzureOpenAISettings:Endpoint" "https://lightspeed-
team-shared-openai-eastus.openai.azure.com/"
dotnet user-secrets set "AzureOpenAISettings:ChatModelDeployment" "gpt-4o"
```

The following class is used in all of the Agent examples. Be sure to include it in your project to ensure proper functionality. This class serves as a foundational component for the examples that follow.

```
c#
using System.Reflection;
using Microsoft.Extensions.Configuration;

namespace AgentsSample;

public class Settings
{
    private readonly IConfigurationRoot configRoot;

    private AzureOpenAISettings azureOpenAI;
    private OpenAISettings openAI;

    public AzureOpenAISettings AzureOpenAI => this.azureOpenAI ??
this.GetSettings<Settings.AzureOpenAISettings>();
    public OpenAISettings OpenAI => this.openAI ??
this.GetSettings<Settings.OpenAISettings>();

    public class OpenAISettings
    {
        public string ChatModel { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }
}
```

```

public class AzureOpenAISettings
{
    public string ChatModelDeployment { get; set; } = string.Empty;
    public string Endpoint { get; set; } = string.Empty;
    public string ApiKey { get; set; } = string.Empty;
}

public TSettings GetSettings<TSettings>() =>

this.configRoot.GetRequiredSection(typeof(TSettings).Name).Get<TSettings>()
();

public Settings()
{
    this.configRoot =
        new ConfigurationBuilder()
            .AddEnvironmentVariables()
            .AddUserSecrets(Assembly.GetExecutingAssembly(), optional:
true)
            .Build();
}
}

```

Coding

The coding process for this sample involves:

1. [Setup](#) - Initializing settings and the plug-in.
2. [Agent Definition](#) - Create the *Chat_Completion_Agent* with templated instructions and plug-in.
3. [The Chat Loop](#) - Write the loop that drives user / agent interaction.

The full example code is provided in the [Final](#) section. Refer to that section for the complete implementation.

Setup

Prior to creating an *Open AI Assistant Agent*, ensure the configuration settings are available and prepare the file resources.

Instantiate the `Settings` class referenced in the previous [Configuration](#) section. Use the settings to also create an `OpenAIProvider` that will be used for the [Agent Definition](#) as well as file-upload and the creation of a `VectorStore`.

```
Settings settings = new();

OpenAIClientProvider clientProvider =
    OpenAIClientProvider.ForAzureOpenAI(
        new AzureCliCredential(),
        new Uri(settings.AzureOpenAI.Endpoint));
```

Now create an empty _Vector Store for use with the *File Search* tool:

Use the `OpenAIClientProvider` to access a `VectorStoreClient` and create a `VectorStore`.

C#

```
Console.WriteLine("Creating store...");
VectorStoreClient storeClient =
    clientProvider.Client.GetVectorStoreClient();
VectorStore store = await storeClient.CreateVectorStoreAsync();
```

Let's declare the the three content-files described in the previous [Configuration](#) section:

C#

```
private static readonly string[] _fileNames =
[
    "Grimms-The-King-of-the-Golden-Mountain.txt",
    "Grimms-The-Water-of-Life.txt",
    "Grimms-The-White-Snake.txt",
];
```

Now upload those files and add them to the *Vector Store* by using the previously created `FileClient` and `VectorStore` client to upload each file and add it to the *Vector Store*, preserving the resulting *File References*.

C#

```
Dictionary<string, OpenAIFileInfo> fileReferences = [];

Console.WriteLine("Uploading files...");
FileClient fileClient = clientProvider.Client.GetFileClient();
foreach (string fileName in _fileNames)
{
    OpenAIFileInfo fileInfo = await fileClient.UploadFileAsync(fileName,
FileUploadPurpose.Assistants);
    await storeClient.AddFileToVectorStoreAsync(store.Id, fileInfo.Id);
    fileReferences.Add(fileInfo.Id, fileInfo);
}
```

Agent Definition

We are now ready to instantiate an *OpenAI Assistant Agent*. The agent is configured with its target model, *Instructions*, and the *File Search* tool enabled. Additionally, we explicitly associate the *Vector Store* with the *File Search* tool.

We will utilize the `OpenAIProvider` again as part of creating the `OpenAIAssistantAgent`:

C#

```
Console.WriteLine("Defining agent...");
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.CreateAsync(
        clientProvider,
        new
OpenAIAssistantDefinition(settings.AzureOpenAI.ChatModelDeployment)
{
    Name = "SampleAssistantAgent",
    Instructions =
    """
        The document store contains the text of fictional stories.
        Always analyze the document store to provide an answer to
        the user's question.
        Never rely on your knowledge of stories not included in the
        document store.
        Always format response using markdown.
    """,
    EnableFileSearch = true,
    VectorStoreId = store.Id,
},
new Kernel());
```

The *Chat Loop*

At last, we are able to coordinate the interaction between the user and the *Agent*. Start by creating an *Assistant Thread* to maintain the conversation state and creating an empty loop.

Let's also ensure the resources are removed at the end of execution to minimize unnecessary charges.

C#

```
Console.WriteLine("Creating thread...");
string threadId = await agent.CreateThreadAsync();

Console.WriteLine("Ready!");
```

```

try
{
    bool isComplete = false;
    do
    {
        // Processing occurs here
    } while (!isComplete);
}
finally
{
    Console.WriteLine();
    Console.WriteLine("Cleaning-up...");
    await Task.WhenAll(
        [
            agent.DeleteThreadAsync(threadId),
            agent.DeleteAsync(),
            storeClient.DeleteVectorStoreAsync(store.Id),
            ..fileReferences.Select(fileReference =>
fileClient.DeleteFileAsync(fileReference.Key))
        ]);
}

```

Now let's capture user input within the previous loop. In this case, empty input will be ignored and the term `EXIT` will signal that the conversation is completed. Valid input will be added to the *Assistant Thread* as a *User message*.

C#

```

Console.WriteLine();
Console.Write("> ");
string input = Console.ReadLine();
if (string.IsNullOrWhiteSpace(input))
{
    continue;
}
if (input.Trim().Equals("EXIT", StringComparison.OrdinalIgnoreCase))
{
    isComplete = true;
    break;
}

await agent.AddChatMessageAsync(threadId, new
ChatMessageContent(AuthorRole.User, input));
Console.WriteLine();

```

Before invoking the *Agent* response, let's add a helper method to reformat the unicode annotation brackets to ANSI brackets.

C#

```
private static string ReplaceUnicodeBrackets(this string content) =>
    content?.Replace('【', '[').Replace('】', ']');
```

To generate an *Agent* response to user input, invoke the agent by specifying the *Assistant Thread*. In this example, we choose a streamed response and capture any associated *Citation Annotations* for display at the end of the response cycle. Note each streamed chunk is being reformatted using the previous helper method.

C#

```
List<StreamingAnnotationContent> footnotes = [];
await foreach (StreamingChatMessageContent chunk in
agent.InvokeStreamingAsync(threadId))
{
    // Capture annotations for footnotes
    footnotes.AddRange(chunk.Items.OfType<StreamingAnnotationContent>());

    // Render chunk with replacements for unicode brackets.
    Console.WriteLine(chunk.Content.ReplaceUnicodeBrackets());
}

Console.WriteLine();

// Render footnotes for captured annotations.
if (footnotes.Count > 0)
{
    Console.WriteLine();
    foreach (StreamingAnnotationContent footnote in footnotes)
    {
        Console.WriteLine($"#{footnote.Quote.ReplaceUnicodeBrackets()} - {fileReferences[footnote.FileId!].Filename} (Index: {footnote.StartIndex} - {footnote.EndIndex})");
    }
}
```

Final

Bringing all the steps together, we have the final code for this example. The complete implementation is provided below.

Try using these suggested inputs:

1. What is the paragraph count for each of the stories?
2. Create a table that identifies the protagonist and antagonist for each story.
3. What is the moral in The White Snake?

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Azure.Identity;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Agents.OpenAI;
using Microsoft.SemanticKernel.ChatCompletion;
using OpenAI.Files;
using OpenAI.VectorStores;

namespace AgentsSample;

public static class Program
{
    private static readonly string[] _fileNames =
    [
        "Grimms-The-King-of-the-Golden-Mountain.txt",
        "Grimms-The-Water-of-Life.txt",
        "Grimms-The-White-Snake.txt",
    ];

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    /// <returns>A <see cref="Task"/> representing the asynchronous
    operation.</returns>
    public static async Task Main()
    {
        // Load configuration from environment variables or user secrets.
        Settings settings = new();

        OpenAIClientProvider clientProvider =
            OpenAIClientProvider.ForAzureOpenAI(
                new AzureCliCredential(),
                new Uri(settings.AzureOpenAI.Endpoint));

        Console.WriteLine("Creating store...");
        VectorStoreClient storeClient =
clientProvider.Client.GetVectorStoreClient();
        VectorStore store = await storeClient.CreateVectorStoreAsync();

        // Retain file references.
        Dictionary<string, OpenAIFileInfo> fileReferences = [];

        Console.WriteLine("Uploading files...");
        FileClient fileClient = clientProvider.Client.GetFileClient();
        foreach (string fileName in _fileNames)
        {
            OpenAIFileInfo fileInfo = await
fileClient.UploadFileAsync(fileName, FileUploadPurpose.Assistants);
            await storeClient.AddFileToVectorStoreAsync(store.Id,
fileInfo.Id);
            fileReferences.Add(fileInfo.Id, fileInfo);
        }
    }
}
```

```

}

Console.WriteLine("Defining agent...");
OpenAIAssistantAgent agent =
    await OpenAIAssistantAgent.CreateAsync(
        clientProvider,
        new
OpenAIAssistantDefinition(settings.AzureOpenAI.ChatModelDeployment)
{
    Name = "SampleAssistantAgent",
    Instructions =
    """
        The document store contains the text of fictional
stories.

        Always analyze the document store to provide an
answer to the user's question.

        Never rely on your knowledge of stories not included
in the document store.

        Always format response using markdown.

        """,
    EnableFileSearch = true,
    VectorStoreId = store.Id,
},
new Kernel());

Console.WriteLine("Creating thread...");
string threadId = await agent.CreateThreadAsync();

Console.WriteLine("Ready!");

try
{
    bool isComplete = false;
    do
    {
        Console.WriteLine();
        Console.Write("> ");
        string input = Console.ReadLine();
        if (string.IsNullOrWhiteSpace(input))
        {
            continue;
        }
        if (input.Trim().Equals("EXIT",
 StringComparison.OrdinalIgnoreCase))
        {
            isComplete = true;
            break;
        }

        await agent.AddChatMessageAsync(threadId, new
ChatMessageContent(AuthorRole.User, input));
        Console.WriteLine();
    }

    List<StreamingAnnotationContent> footnotes = [];
}

```

```

        await foreach (StreamingChatMessageContent chunk in
agent.InvokeStreamingAsync(threadId))
{
    // Capture annotations for footnotes

footnotes.AddRange(chunk.Items.OfType<StreamingAnnotationContent>());

    // Render chunk with replacements for unicode brackets.
    Console.WriteLine(chunk.Content.ReplaceUnicodeBrackets());
}

Console.WriteLine();

    // Render footnotes for captured annotations.
    if (footnotes.Count > 0)
    {
        Console.WriteLine();
        foreach (StreamingAnnotationContent footnote in
footnotes)
        {
            Console.WriteLine($"#{footnote.Quote.ReplaceUnicodeBrackets()} - {fileReferences[footnote.FileId!].Filename} (Index: {footnote.StartIndex} - {footnote.EndIndex})");
        }
    }
} while (!isComplete);
}
finally
{
    Console.WriteLine();
    Console.WriteLine("Cleaning-up...");
    await Task.WhenAll(
    [
        agent.DeleteThreadAsync(threadId),
        agent.DeleteAsync(),
        storeClient.DeleteVectorStoreAsync(store.Id),
        ..fileReferences.Select(fileReference =>
fileClient.DeleteFileAsync(fileReference.Key))
    ]);
}
}

private static string ReplaceUnicodeBrackets(this string content) =>
content?.Replace('【', '[').Replace('】', ']');
}

```

How to Coordinate Agent Collaboration using *Agent Group Chat*

How-To: Coordinate Agent Collaboration using Agent Group Chat (Experimental)

Article • 10/09/2024

⚠️ Warning

The *Semantic Kernel Agent Framework* is experimental, still in development and is subject to change.

Overview

In this sample, we will explore how to use *Agent Group Chat* to coordinate collaboration of two different agents working to review and rewrite user provided content. Each agent is assigned a distinct role:

- **Reviewer:** Reviews and provides direction to *Writer*.
- **Writer:** Updates user content based on *Reviewer* input.

The approach will be broken down step-by-step to highlight the key parts of the coding process.

Getting Started

Before proceeding with feature coding, make sure your development environment is fully set up and configured.

Start by creating a *Console* project. Then, include the following package references to ensure all required dependencies are available.

To add package dependencies from the command-line use the `dotnet` command:

PowerShell

```
dotnet add package Azure.Identity  
dotnet add package Microsoft.Extensions.Configuration  
dotnet add package Microsoft.Extensions.Configuration.Binder  
dotnet add package Microsoft.Extensions.Configuration.UserSecrets  
dotnet add package Microsoft.Extensions.Configuration.EnvironmentVariables
```

```
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI  
dotnet add package Microsoft.SemanticKernel.Agents.Core --prerelease
```

If managing *NuGet* packages in *Visual Studio*, ensure `Include prerelease` is checked.

The project file (`.csproj`) should contain the following `PackageReference` definitions:

XML

```
<ItemGroup>  
  <PackageReference Include="Azure.Identity" Version="<stable>" />  
  <PackageReference Include="Microsoft.Extensions.Configuration" Version="  
<stable>" />  
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder"  
Version="<stable>" />  
  <PackageReference  
Include="Microsoft.Extensions.Configuration.UserSecrets" Version="<stable>"  
/>  
  <PackageReference  
Include="Microsoft.Extensions.Configuration.EnvironmentVariables" Version="  
<stable>" />  
  <PackageReference Include="Microsoft.SemanticKernel.Agents.Core"  
Version="<latest>" />  
  <PackageReference  
Include="Microsoft.SemanticKernel.Connectors.AzureOpenAI" Version="<latest>"  
/>  
</ItemGroup>
```

The *Agent Framework* is experimental and requires warning suppression. This may be addressed in as a property in the project file (`.csproj`):

XML

```
<PropertyGroup>  
  <NoWarn>$(NoWarn);CA2007;IDE1006;SKEXP0001;SKEXP0110;OPENAI001</NoWarn>  
</PropertyGroup>
```

Configuration

This sample requires configuration setting in order to connect to remote services. You will need to define settings for either *Open AI* or *Azure Open AI*.

PowerShell

```

# Open AI
dotnet user-secrets set "OpenAISettings:ApiKey" "<api-key>"
dotnet user-secrets set "OpenAISettings:ChatModel" "gpt-4o"

# Azure Open AI
dotnet user-secrets set "AzureOpenAISettings:ApiKey" "<api-key>" # Not
required if using token-credential
dotnet user-secrets set "AzureOpenAISettings:Endpoint" "<model-endpoint>"
dotnet user-secrets set "AzureOpenAISettings:ChatModelDeployment" "gpt-4o"

```

The following class is used in all of the Agent examples. Be sure to include it in your project to ensure proper functionality. This class serves as a foundational component for the examples that follow.

```

c#

using System.Reflection;
using Microsoft.Extensions.Configuration;

namespace AgentsSample;

public class Settings
{
    private readonly IConfigurationRoot configRoot;

    private AzureOpenAISettings azureOpenAI;
    private OpenAISettings openAI;

    public AzureOpenAISettings AzureOpenAI => this.azureOpenAI ??=
this.GetSettings<Settings.AzureOpenAISettings>();
    public OpenAISettings OpenAI => this.openAI ??=
this.GetSettings<Settings.OpenAISettings>();

    public class OpenAISettings
    {
        public string ChatModel { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }

    public class AzureOpenAISettings
    {
        public string ChatModelDeployment { get; set; } = string.Empty;
        public string Endpoint { get; set; } = string.Empty;
        public string ApiKey { get; set; } = string.Empty;
    }

    public TSettings GetSettings<TSettings>() =>

this.configRoot.GetRequiredSection(typeof(TSettings).Name).Get<TSettings>()
();

    public Settings()

```

```
{  
    this.configRoot =  
        new ConfigurationBuilder()  
            .AddEnvironmentVariables()  
            .AddUserSecrets(Assembly.GetExecutingAssembly(), optional:  
true)  
            .Build();  
}  
}
```

Coding

The coding process for this sample involves:

1. [Setup](#) - Initializing settings and the plug-in.
2. [Agent Definition](#) - Create the two *Chat Completion Agent* instances (*Reviewer* and *Writer*).
3. [Chat Definition](#) - Create the *Agent Group Chat* and associated strategies.
4. [The Chat Loop](#) - Write the loop that drives user / agent interaction.

The full example code is provided in the [Final](#) section. Refer to that section for the complete implementation.

Setup

Prior to creating any *Chat Completion Agent*, the configuration settings, plugins, and *Kernel* must be initialized.

Instantiate the the `Settings` class referenced in the previous [Configuration](#) section.

C#

```
Settings settings = new();
```

Now initialize a `Kernel` instance with an `IChatCompletionService`.

C#

```
IKernelBuilder builder = Kernel.CreateBuilder();  
  
builder.AddAzureOpenAIChatCompletion(  
    settings.AzureOpenAI.ChatModelDeployment,  
    settings.AzureOpenAI.Endpoint,  
    new AzureCliCredential());
```

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

Let's also create a second *Kernel* instance via *cloning* and add a plug-in that will allow the review to place updated content on the clip-board.

C#

```
Kernel toolKernel = kernel.Clone();
toolKernel.Plugins.AddFromType<ClipboardAccess>();
```

The *Clipboard* plugin may be defined as part of the sample.

C#

```
private sealed class ClipboardAccess
{
    [KernelFunction]
    [Description("Copies the provided content to the clipboard.")]
    public static void SetClipboard(string content)
    {
        if (string.IsNullOrWhiteSpace(content))
        {
            return;
        }

        using Process clipProcess = Process.Start(
            new ProcessStartInfo
            {
                FileName = "clip",
                RedirectStandardInput = true,
                UseShellExecute = false,
            });
        
        clipProcess.StandardInput.WriteLine(content);
        clipProcess.StandardInput.Close();
    }
}
```

Agent Definition

Let's declare the agent names as `const` so they might be referenced in *Agent Group Chat* strategies:

C#

```
const string ReviewerName = "Reviewer";
```

```
const string WriterName = "Writer";
```

Defining the *Reviewer* agent uses the pattern explored in [How-To: Chat Completion Agent](#).

Here the *Reviewer* is given the role of responding to user input, providing direction to the *Writer* agent, and verifying result of the *Writer* agent.

C#

```
ChatCompletionAgent agentReviewer =
    new()
{
    Name = ReviewerName,
    Instructions =
        """
            Your responsibility is to review and identify how to improve user
            provided content.

            If the user has providing input or direction for content already
            provided, specify how to address this input.

            Never directly perform the correction or provide example.

            Once the content has been updated in a subsequent response, you
            will review the content again until satisfactory.

            Always copy satisfactory content to the clipboard using
            available tools and inform user.

        RULES:
        - Only identify suggestions that are specific and actionable.
        - Verify previous suggestions have been addressed.
        - Never repeat previous suggestions.

        """,
    Kernel = toolKernel,
    Arguments =
        new KernelArguments(
            new AzureOpenAIPromptExecutionSettings()
            {
                FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
            }
        );
}
```

The *Writer* agent is similar, but doesn't require the specification of *Execution Settings* since it isn't configured with a plug-in.

Here the *Writer* is given a single-purpose task, follow direction and rewrite the content.

C#

```
ChatCompletionAgent agentWriter =
    new()
{
```

```

        Name = WriterName,
        Instructions =
        """
            Your sole responsibility is to rewrite content according to
            review suggestions.

            - Always apply all review direction.
            - Always revise the content in its entirety without explanation.
            - Never address the user.
        """,
        Kernel = kernel,
    };

```

Chat Definition

Defining the *Agent Group Chat* requires considering the strategies for selecting the *Agent* turn and determining when to exit the *Chat* loop. For both of these considerations, we will define a *Kernel Prompt Function*.

The first to reason over *Agent* selection:

Using `AgentGroupChat.CreatePromptFunctionForStrategy` provides a convenient mechanism to avoid *HTML encoding* the message parameter.

C#

```

KernelFunction selectionFunction =
    AgentGroupChat.CreatePromptFunctionForStrategy(
        $$"""
            Examine the provided RESPONSE and choose the next participant.
            State only the name of the chosen participant without explanation.
            Never choose the participant named in the RESPONSE.

            Choose only from these participants:
            - {{ReviewerName}}
            - {{WriterName}}

            Always follow these rules when choosing the next participant:
            - If RESPONSE is user input, it is {{ReviewerName}}'s turn.
            - If RESPONSE is by {{ReviewerName}}, it is {{WriterName}}'s
turn.
            - If RESPONSE is by {{WriterName}}, it is {{ReviewerName}}'s
turn.

            RESPONSE:
            {{$lastmessage}}
        """",
        safeParameterNames: "lastmessage");

```

The second will evaluate when to exit the *Chat* loop:

```
C#  
  
const string TerminationToken = "yes";  
  
KernelFunction terminationFunction =  
    AgentGroupChat.CreatePromptFunctionForStrategy(  
        $$$"  
            Examine the RESPONSE and determine whether the content has been  
            deemed satisfactory.  
            If content is satisfactory, respond with a single word without  
            explanation: {{TerminationToken}}.  
            If specific suggestions are being provided, it is not satisfactory.  
            If no correction is suggested, it is satisfactory.  
  
            RESPONSE:  
            {{$lastmessage}}  
        """,  
        safeParameterNames: "lastmessage");
```

Both of these *Strategies* will only require knowledge of the most recent *Chat* message. This will reduce token usage and help improve performance:

```
C#  
  
ChatHistoryTruncationReducer historyReducer = new(1);
```

Finally we are ready to bring everything together in our *Agent Group Chat* definition.

Creating `AgentGroupChat` involves:

1. Include both agents in the constructor.
2. Define a `KernelFunctionSelectionStrategy` using the previously defined `KernelFunction` and `Kernel` instance.
3. Define a `KernelFunctionTerminationStrategy` using the previously defined `KernelFunction` and `Kernel` instance.

Notice that each strategy is responsible for parsing the `KernelFunction` result.

```
C#  
  
AgentGroupChat chat =  
    new(agentReviewer, agentWriter)  
    {  
        ExecutionSettings = new AgentGroupChatSettings  
        {  
            SelectionStrategy =
```

```

        new KernelFunctionSelectionStrategy(selectionFunction,
kernel)
{
    // Always start with the editor agent.
    InitialAgent = agentReviewer,
    // Save tokens by only including the final response
    HistoryReducer = historyReducer,
    // The prompt variable name for the history argument.
    HistoryVariableName = "lastmessage",
    // Returns the entire result value as a string.
    ResultParser = (result) => result.GetValue<string>() ??
agentReviewer.Name
},
TerminationStrategy =
    new KernelFunctionTerminationStrategy(terminationFunction,
kernel)
{
    // Only evaluate for editor's response
    Agents = [agentReviewer],
    // Save tokens by only including the final response
    HistoryReducer = historyReducer,
    // The prompt variable name for the history argument.
    HistoryVariableName = "lastmessage",
    // Limit total number of turns
    MaximumIterations = 12,
    // Customer result parser to determine if the response
is "yes"
    ResultParser = (result) => result.GetValue<string>
()?.Contains(TerminationToken, StringComparison.OrdinalIgnoreCase) ?? false
}
};

Console.WriteLine("Ready!");

```

The *Chat* Loop

At last, we are able to coordinate the interaction between the user and the *Agent Group Chat*. Start by creating creating an empty loop.

Note: Unlike the other examples, no external history or *thread* is managed. *Agent Group Chat* manages the conversation history internally.

C#

```

bool isComplete = false;
do
{

```

```
} while (!isComplete);
```

Now let's capture user input within the previous loop. In this case:

- Empty input will be ignored
- The term `EXIT` will signal that the conversation is completed
- The term `RESET` will clear the *Agent Group Chat* history
- Any term starting with `@` will be treated as a file-path whose content will be provided as input
- Valid input will be added to the *Agent Group Chaty* as a *User* message.

C#

```
Console.WriteLine();
Console.Write("> ");
string input = Console.ReadLine();
if (string.IsNullOrWhiteSpace(input))
{
    continue;
}
input = input.Trim();
if (input.Equals("EXIT", StringComparison.OrdinalIgnoreCase))
{
    isComplete = true;
    break;
}

if (input.Equals("RESET", StringComparison.OrdinalIgnoreCase))
{
    await chat.ResetAsync();
    Console.WriteLine("[Conversation has been reset]");
    continue;
}

if (input.StartsWith("@", StringComparison.Ordinal) && input.Length > 1)
{
    string filePath = input.Substring(1);
    try
    {
        if (!File.Exists(filePath))
        {
            Console.WriteLine($"Unable to access file: {filePath}");
            continue;
        }
        input = File.ReadAllText(filePath);
    }
    catch (Exception)
    {
        Console.WriteLine($"Unable to access file: {filePath}");
        continue;
    }
}
```

```
        }

    chat.AddChatMessage(new ChatMessageContent(AuthorRole.User, input));
```

To initiate the *Agent* collaboration in response to user input and display the *Agent* responses, invoke the *Agent Group Chat*; however, first be sure to reset the *Completion* state from any prior invocation.

Note: Service failures are being caught and displayed to avoid crashing the conversation loop.

C#

```
chat.IsComplete = false;

try
{
    await foreach (ChatMessageContent response in chat.InvokeAsync())
    {
        Console.WriteLine();
        Console.WriteLine($"{response.AuthorName.ToUpperInvariant()}:
{Environment.NewLine}{response.Content}");
    }
}
catch (HttpOperationException exception)
{
    Console.WriteLine(exception.Message);
    if (exception.InnerException != null)
    {
        Console.WriteLine(exception.InnerException.Message);
        if (exception.InnerException.Data.Count > 0)
        {

Console.WriteLine(JsonSerializer.Serialize(exception.InnerException.Data,
new JsonSerializerOptions() { WriteIndented = true }));
        }
    }
}
```

Final

Bringing all the steps together, we have the final code for this example. The complete implementation is provided below.

Try using these suggested inputs:

1. Hi
2. {"message: "hello world"}
3. {"message": "hello world"}
4. Semantic Kernel (SK) is an open-source SDK that enables developers to build and orchestrate complex AI workflows that involve natural language processing (NLP) and machine learning models. It provides a flexible platform for integrating AI capabilities such as semantic search, text summarization, and dialogue systems into applications. With SK, you can easily combine different AI services and models, define their relationships, and orchestrate interactions between them.
5. make this two paragraphs
6. thank you
7. @\WomensSuffrage.txt
8. its good, but is it ready for my college professor?

C#

```
// Copyright (c) Microsoft. All rights reserved.

using System;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Text.Json;
using System.Threading.Tasks;
using Azure.Identity;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Actors;
using Microsoft.SemanticKernel.Actors.Chat;
using Microsoft.SemanticKernel.Actors.History;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;

namespace AgentsSample;

public static class Program
{
    public static async Task Main()
    {
        // Load configuration from environment variables or user secrets.
        Settings settings = new();

        Console.WriteLine("Creating kernel...");
        IKernelBuilder builder = Kernel.CreateBuilder();

        builder.AddAzureOpenAIChatCompletion(
            settings.AzureOpenAI.ChatModelDeployment,
            settings.AzureOpenAI.Endpoint,
            new AzureCliCredential());

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

```
Kernel toolKernel = kernel.Clone();
toolKernel.Plugins.AddFromType<ClipboardAccess>();

Console.WriteLine("Defining agents...");

const string ReviewerName = "Reviewer";
const string WriterName = "Writer";

ChatCompletionAgent agentReviewer =
    new()
{
    Name = ReviewerName,
    Instructions =
    """
        Your responsibility is to review and identify how to
        improve user provided content.
        If the user has providing input or direction for content
        already provided, specify how to address this input.
        Never directly perform the correction or provide
        example.
        Once the content has been updated in a subsequent
        response, you will review the content again until satisfactory.
        Always copy satisfactory content to the clipboard using
        available tools and inform user.

    RULES:
    - Only identify suggestions that are specific and
    actionable.
    - Verify previous suggestions have been addressed.
    - Never repeat previous suggestions.
    """,
    Kernel = toolKernel,
    Arguments = new KernelArguments(new
AzureOpenAIPromptExecutionSettings() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto() })
};

ChatCompletionAgent agentWriter =
    new()
{
    Name = WriterName,
    Instructions =
    """
        Your sole responsibility is to rewrite content according
        to review suggestions.

        - Always apply all review direction.
        - Always revise the content in its entirety without
        explanation.
        - Never address the user.
    """,
    Kernel = kernel,
};
```

```

KernelFunction selectionFunction =
    AgentGroupChat.CreatePromptFunctionForStrategy(
        $$$"""
        Examine the provided RESPONSE and choose the next
participant.
        State only the name of the chosen participant without
explanation.
        Never choose the participant named in the RESPONSE.

        Choose only from these participants:
        - {{{ReviewerName}}}
        - {{{WriterName}}}

        Always follow these rules when choosing the next
participant:
        - If RESPONSE is user input, it is {{{ReviewerName}}}’s
turn.
        - If RESPONSE is by {{{ReviewerName}}}, it is
{{{{WriterName}}}}’s turn.
        - If RESPONSE is by {{{WriterName}}}, it is
{{{{ReviewerName}}}}’s turn.

        RESPONSE:
        {{\$lastmessage}}
        """",
        safeParameterNames: "lastmessage");

    const string TerminationToken = "yes";

KernelFunction terminationFunction =
    AgentGroupChat.CreatePromptFunctionForStrategy(
        $$$"""
        Examine the RESPONSE and determine whether the content has
been deemed satisfactory.
        If content is satisfactory, respond with a single word
without explanation: {{{TerminationToken}}}.

        If specific suggestions are being provided, it is not
satisfactory.

        If no correction is suggested, it is satisfactory.

        RESPONSE:
        {{\$lastmessage}}
        """",
        safeParameterNames: "lastmessage");

    ChatHistoryTruncationReducer historyReducer = new(1);

    AgentGroupChat chat =
        new(agentReviewer, agentWriter)
    {
        ExecutionSettings = new AgentGroupChatSettings
        {
            SelectionStrategy =
                new

```

```

KernelFunctionSelectionStrategy(selectionFunction, kernel)
{
    // Always start with the editor agent.
    InitialAgent = agentReviewer,
    // Save tokens by only including the final
response
    HistoryReducer = historyReducer,
    // The prompt variable name for the history
argument.
    HistoryVariableName = "lastmessage",
    // Returns the entire result value as a string.
    ResultParser = (result) =>
result.GetValue<string>() ?? agentReviewer.Name
},
TerminationStrategy =
new
KernelFunctionTerminationStrategy(terminationFunction, kernel)
{
    // Only evaluate for editor's response
    Agents = [agentReviewer],
    // Save tokens by only including the final
response
    HistoryReducer = historyReducer,
    // The prompt variable name for the history
argument.
    HistoryVariableName = "lastmessage",
    // Limit total number of turns
    MaximumIterations = 12,
    // Customer result parser to determine if the
response is "yes"
    ResultParser = (result) =>
result.GetValue<string>()?.Contains(TerminationToken,
 StringComparison.OrdinalIgnoreCase) ?? false
}
};

Console.WriteLine("Ready!");

bool isComplete = false;
do
{
    Console.WriteLine();
    Console.Write("> ");
    string input = Console.ReadLine();
    if (string.IsNullOrWhiteSpace(input))
    {
        continue;
    }
    input = input.Trim();
    if (input.Equals("EXIT", StringComparison.OrdinalIgnoreCase))
    {
        isComplete = true;
        break;
    }
}

```

```

        if (input.Equals("RESET", StringComparison.OrdinalIgnoreCase))
        {
            await chat.ResetAsync();
            Console.WriteLine("[Conversation has been reset]");
            continue;
        }

        if (input.StartsWith("@", StringComparison.OrdinalIgnoreCase) &&
input.Length > 1)
        {
            string filePath = input.Substring(1);
            try
            {
                if (!File.Exists(filePath))
                {
                    Console.WriteLine($"Unable to access file:
{filePath}");
                    continue;
                }
                input = File.ReadAllText(filePath);
            }
            catch (Exception)
            {
                Console.WriteLine($"Unable to access file: {filePath}");
                continue;
            }
        }

        chat.AddChatMessage(new ChatMessageContent(AuthorRole.User,
input));

        chat.IsComplete = false;

        try
        {
            await foreach (ChatMessageContent response in
chat.InvokeAsync())
            {
                Console.WriteLine();
                Console.WriteLine($""
{response.AuthorName.ToUpperInvariant()}: {Environment.NewLine}
{response.Content}");
            }
        }
        catch (HttpOperationException exception)
        {
            Console.WriteLine(exception.Message);
            if (exception.InnerException != null)
            {
                Console.WriteLine(exception.InnerException.Message);
                if (exception.InnerException.Data.Count > 0)
                {

Console.WriteLine(JsonSerializer.Serialize(exception.InnerException.Data,

```

```
new JsonSerializerOptions() { WriteIndented = true }));  
        }  
    }  
}
```

```
    }  
} while (!isComplete);  
}

```
private sealed class ClipboardAccess
{
 [KernelFunction]
 [Description("Copies the provided content to the clipboard.")]
 public static void SetClipboard(string content)
 {
 if (string.IsNullOrWhiteSpace(content))
 {
 return;
 }

 using Process clipProcess = Process.Start(
 new ProcessStartInfo
 {
 FileName = "clip",
 RedirectStandardInput = true,
 UseShellExecute = false,
 });

 clipProcess.StandardInput.Write(content);
 clipProcess.StandardInput.Close();
 }
}
```


```

Overview of the Process Framework

Article • 09/28/2024

Welcome to the Process Framework within Microsoft's Semantic Kernel—a cutting-edge approach designed to optimize AI integration with your business processes. This framework empowers developers to efficiently create, manage, and deploy business processes while leveraging the powerful capabilities of AI, alongside your existing code and systems.

A Process is a structured sequence of activities or tasks that deliver a service or product, adding value in alignment with specific business goals for customers.

ⓘ Note

Process Framework package is currently experimental and is subject to change until it is moved to preview and GA.

Introduction to the Process Framework

The Process Framework provides a robust solution for automating complex workflows. Each step within the framework performs tasks by invoking user-defined Kernel Functions, utilizing an event-driven model to manage workflow execution.

By embedding AI into your business processes, you can significantly enhance productivity and decision-making capabilities. With the Process Framework, you benefit from seamless AI integration, facilitating smarter and more responsive workflows. This framework streamlines operations, fosters improved collaboration between business units, and boosts overall efficiency.

Key Features

- **Leverage Semantic Kernel:** Steps can utilize one or multiple Kernel Functions, enabling you to tap into all aspects of Semantic Kernel within your processes.
- **Reusability & Flexibility:** Steps and processes can be reused across different applications, promoting modularity and scalability.
- **Event-Driven Architecture:** Utilize events and metadata to trigger actions and transitions between process steps effectively.
- **Full Control and Auditability:** Maintain control of processes in a defined and repeatable manner, complete with audit capabilities through Open Telemetry.

Core Concepts

- **Process:** A collection of steps arranged to achieve a specific business goal for customers.
- **Step:** An activity within a process that has defined inputs and outputs, contributing to a larger goal.
- **Pattern:** The specific sequence type that dictates how steps are executed for the process to be fully completed.

Business Process Examples

Business processes are a part of our daily routines. Here are three examples you might have encountered this week:

1. **Account Opening:** This process includes multiple steps such as credit pulls and ratings, fraud detection, creating customer accounts in core systems, and sending welcome information to the customer, including their customer ID.
2. **Food Delivery:** Ordering food for delivery is a familiar process. From receiving the order via phone, website, or app, to preparing each food item, ensuring quality control, driver assignment, and final delivery, there are many steps in this process that can be streamlined.
3. **Support Ticket:** We have all submitted support tickets—whether for new services, IT support, or other needs. This process can involve multiple subprocesses based on business and customer requirements, ultimately aiming for satisfaction by addressing customer needs effectively.

Getting Started

Are you ready to harness the power of the Process Framework?

Begin your journey by exploring our [.NET samples](#) on GitHub. While Python support is on the horizon, the .NET examples provide an excellent starting point for understanding the framework's capabilities and applications.

ⓘ Note

Process Framework is available for .NET currently. The Process Framework for Python is in progress.

By diving into the Process Framework, developers can transform traditional workflows into intelligent, adaptive systems. Start building with the tools at your disposal and redefine what's possible with AI-driven business processes.

Core Components of the Process Framework

Article • 09/28/2024

The Process Framework is built upon a modular architecture that enables developers to construct sophisticated workflows through its core components. Understanding these components is essential for effectively leveraging the framework.

Process

A Process serves as the overarching container that orchestrates the execution of Steps. It defines the flow and routing of data between Steps, ensuring that process goals are achieved efficiently. Processes handle inputs and outputs, providing flexibility and scalability across various workflows.

Process Features

- **Stateful:** Supports querying information such as tracking status and percent completion, as well as the ability to pause and resume.
- **Reusable:** A Process can be invoked within other processes, promoting modularity and reusability.
- **Event Driven:** Employs event-based flow with listeners to route data to Steps and other Processes.
- **Scalable:** Utilizes well-established runtimes for global scalability and rollouts.
- **Cloud Event Integrated:** Incorporates industry-standard eventing for triggering a Process or Step.

Creating A Process

To create a new Process, add the Process Package to your project and define a name for your process.

Step

Steps are the fundamental building blocks within a Process. Each Step corresponds to a discrete unit of work and encapsulates one or more Kernel Functions. Steps can be created independently of their use in specific Processes, enhancing their reusability. They emit events based on the work performed, which can trigger subsequent Steps.

Step Features

- **Stateful:** Facilitates tracking information such as status and defined tags.
- **Reusable:** Steps can be employed across multiple Processes.
- **Dynamic:** Steps can be created dynamically by a Process as needed, depending on the required pattern.
- **Flexible:** Offers different types of Steps for developers by leveraging Kernel Functions, including Code-only, API calls, AI Agents, and Human-in-the-loop.
- **Auditable:** Telemetry is enabled across both Steps and Processes.

Defining a Step

To create a Step, define a public class to name the Step and add it to the `KernelStepBase`. Within your class, you can incorporate one or multiple Kernel Functions.

Register a Step into a Process

Once your class is created, you need to register it within your Process. For the first Step in the Process, add `isEntryPoint: true` so the Process knows where to start.

Step Events

Steps have several events available, including:

- **OnEvent:** Triggered when the class completes its execution.
- **OnFunctionResult:** Activated when the defined Kernel Function emits results, allowing output to be sent to one or many Steps.
- **SendOutputTo:** Defines the Step and Input for sending results to a subsequent Step.

Pattern

Patterns standardize common process flows, simplifying the implementation of frequently used operations. They promote a consistent approach to solving recurring problems across various implementations, enhancing both maintainability and readability.

Pattern Types

- **Fan In:** The input for the next Step is supported by multiple outputs from previous Steps.
- **Fan Out:** The output of previous Steps is directed into multiple Steps further down the Process.
- **Cycle:** Steps continue to loop until completion based on input and output.
- **Map Reduce:** Outputs from a Step are consolidated into a smaller amount and directed to the next Step's input.

Setting up a Pattern

Once your class is created for your Step and registered within the Process, you can define the events that should be sent downstream to other Steps or set conditions for Steps to be restarted based on the output from your Step.

Deployment of the Process Framework

Article • 09/28/2024

Deploying workflows built with the Process Framework can be done seamlessly across local development environments and cloud runtimes. This flexibility enables developers to choose the best approach tailored to their specific use cases.

Local Development

The Process Framework provides an in-process runtime that allows developers to run processes directly on their local machines or servers without requiring complex setups or additional infrastructure. This runtime supports both memory and file-based persistence, ideal for rapid development and debugging. You can quickly test processes with immediate feedback, accelerating the development cycle and enhancing efficiency.

Cloud Runtimes

For scenarios requiring scalability and distributed processing, the Process Framework supports cloud runtimes such as [Orleans](#) and [Dapr](#). These options empower developers to deploy processes in a distributed manner, facilitating high availability and load balancing across multiple instances. By leveraging these cloud runtimes, organizations can streamline their operations and manage substantial workloads with ease.

- **Orleans Runtime:** This framework provides a programming model for building distributed applications and is particularly well-suited for handling virtual actors in a resilient manner, complementing the Process Framework's event-driven architecture.
- **Dapr (Distributed Application Runtime):** Dapr simplifies microservices development by providing a foundational framework for building distributed systems. It supports state management, service invocation, and pub/sub messaging, making it easier to connect various components within a cloud environment.

Using either runtime, developers can scale applications according to demand, ensuring that processes run smoothly and efficiently, regardless of workload.

With the flexibility to choose between local testing environments and robust cloud platforms, the Process Framework is designed to meet diverse deployment needs. This

enables developers to concentrate on building innovative AI-powered processes without the burden of infrastructure complexities.

 **Note**

Orleans will be supported first with the .NET Process Framework, followed by Dapr in the upcoming release of the Python version of the Process Framework.

Best Practices for the Process Framework

Article • 09/28/2024

Utilizing the Process Framework effectively can significantly enhance your workflow automation. Here are some best practices to help you optimize your implementation and avoid common pitfalls.

File and Folder Layout Structure

Organizing your project files in a logical and maintainable structure is crucial for collaboration and scalability. A recommended file layout may include:

- **Processes/**: A directory for all defined processes.
- **Steps/**: A dedicated directory for reusable Steps.
- **Functions/**: A folder containing your Kernel Function definitions.

An organized structure not only simplifies navigation within the project but also enhances code reusability and facilitates collaboration among team members.

Common Pitfalls

To ensure smooth implementation and operation of the Process Framework, be mindful of these common pitfalls to avoid:

- **Overcomplicating Steps**: Keep Steps focused on a single responsibility. Avoid creating complex Steps that perform multiple tasks, as this can complicate debugging and maintenance.
- **Ignoring Event Handling**: Events are vital for smooth communication between Steps. Ensure that you handle all potential events and errors within the process to prevent unexpected behavior or crashes.
- **Performance and Quality**: As processes scale, it's crucial to continuously monitor performance. Leverage telemetry from your Steps to gain insights into how Processes are functioning.

By following these best practices, you can maximize the effectiveness of the Process Framework, enabling more robust and manageable workflows. Keeping organization, simplicity, and performance in mind will lead to a smoother development experience and higher-quality applications.

Support for Semantic Kernel

Article • 10/10/2024

 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](#)
- [Transparency Documentation](#)

Next step

[Run the samples](#)

Contributing to Semantic Kernel

Article • 09/24/2024

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](#).

Running your own Hackathon

Article • 09/24/2024

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!

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](#)

Semantic Kernel - .Net V1 Migration Guide

Article • 09/24/2024

ⓘ 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.AzureAISearch</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
SKEexception	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": ""
}
]
}
```

OpenAI Connector Migration Guide

Article • 09/24/2024

Coming as part of the new **1.18** version of Semantic Kernel we migrated our `OpenAI` and `AzureOpenAI` services to use the new `OpenAI SDK v2.0` and `Azure OpenAI SDK v2.0` SDKs.

As those changes were major breaking changes when implementing ours we looked forward to break as minimal as possible the dev experience.

This guide prepares you for the migration that you may need to do to use our new OpenAI Connector is a complete rewrite of the existing OpenAI Connector and is designed to be more efficient, reliable, and scalable. This manual will guide you through the migration process and help you understand the changes that have been made to the OpenAI Connector.

Those changes are needed for anyone using `OpenAI` or `AzureOpenAI` connectors with Semantic Kernel version `1.18.0-rc` or above.

1. Package Setup when using Azure only services

If you are working with Azure services you will need to change the package from `Microsoft.SemanticKernel.Connectors.OpenAI` to `Microsoft.SemanticKernel.Connectors.AzureOpenAI`. This is necessary as we created two distinct connectors for each.

ⓘ Important

The `Microsoft.SemanticKernel.Connectors.AzureOpenAI` package depends on the `Microsoft.SemanticKernel.Connectors.OpenAI` package so there's no need to add both to your project when using OpenAI related types.

diff

Before

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

After

+ `using Microsoft.SemanticKernel.Connectors.AzureOpenAI;`

1.1 AzureOpenAIclient

When using Azure with OpenAI, before where you were using `OpenAIclient` you will need to update your code to use the new `AzureOpenAIclient` type.

1.2 Services

All services below now belong to the `Microsoft.SemanticKernel.Connectors.AzureOpenAI` namespace.

- `AzureOpenAIAudioToTextService`
- `AzureOpenAIChatCompletionService`
- `AzureOpenAITextEmbeddingGenerationService`
- `AzureOpenAITextToAudioService`
- `AzureOpenAITextToImageService`

2. Text Generation Deprecation

The latest `OpenAI` SDK does not support text generation modality, when migrating to their underlying SDK we had to drop support as well and remove `TextGeneration` specific services.

If you were using OpenAI's `gpt-3.5-turbo-instruct` legacy model with any of the `OpenAITextGenerationService` or `AzureOpenAITextGenerationService` you will need to update your code to target a chat completion model instead, using `OpenAIChatCompletionService` or `AzureOpenAIChatCompletionService` instead.

ⓘ Note

OpenAI and AzureOpenAI `ChatCompletion` services also implement the `ITextGenerationService` interface and that may not require any changes to your code if you were targeting the `ITextGenerationService` interface.

tags: `AddOpenAITextGeneration`, `AddAzureOpenAITextGeneration`

3. ChatCompletion Multiple Choices Deprecated

The latest `OpenAI` SDK does not support multiple choices, when migrating to their underlying SDK we had to drop the support and remove `ResultsPerPrompt` also from the `OpenAIPromptExecutionSettings`.

tags: `results_per_prompt`

4. OpenAI File Service Deprecation

The `OpenAIFileService` was deprecated in the latest version of the OpenAI Connector. We strongly recommend to update your code to use the new `OpenAIClient.GetFileClient()` for file management operations.

5. OpenAI ChatCompletion custom endpoint

The `OpenAIChatCompletionService` **experimental** constructor for custom endpoints will not attempt to auto-correct the endpoint and use it as is.

We have the two only specific cases where we attempted to auto-correct the endpoint.

1. If you provided `chat/completions` path before. Now those need to be removed as they are added automatically to the end of your original endpoint by `OpenAI` `SDK`.

diff

```
- http://any-host-and-port/v1/chat/completions
+ http://any-host-and-port/v1
```

2. If you provided a custom endpoint without any path. We won't be adding the `v1/` as the first path. Now the `v1` path needs to be provided as part of your endpoint.

diff

```
- http://any-host-and-port/
+ http://any-host-and-port/v1
```

6. SemanticKernel MetaPackage

To be retrocompatible with the new OpenAI and AzureOpenAI Connectors, our `Microsoft.SemanticKernel` meta package changed its dependency to use the new `Microsoft.SemanticKernel.Connectors.AzureOpenAI` package that depends on the

`Microsoft.SemanticKernel.Connectors.OpenAI` package. This way if you are using the metapackage, no change is needed to get access to `Azure` related types.

7. Chat Message Content Changes

7.1 OpenAIChatMessageContent

- The `Tools` property type has changed from `IReadOnlyList<ChatCompletionsToolCall>` to `IReadOnlyList<ChatToolCall>`.
- Inner content type has changed from `ChatCompletionsFunctionToolCall` to `ChatToolCall`.
- Metadata type `FunctionToolCalls` has changed from `IEnumerable<ChatCompletionsFunctionToolCall>` to `IEnumerable<ChatToolCall>`.

7.2 OpenAIStreamingChatMessageContent

- The `FinishReason` property type has changed from `CompletionsFinishReason` to `FinishReason`.
- The `ToolCallUpdate` property has been renamed to `ToolCallUpdates` and its type has changed from `StreamingToolCallUpdate?` to `IReadOnlyList<StreamingToolCallUpdate>?`.
- The `AuthorName` property is not initialized because it's not provided by the underlying library anymore.

8. Metrics for AzureOpenAI Connector

The meter `s_meter = new("Microsoft.SemanticKernel.Connectors.OpenAI");` and the relevant counters still have old names that contain "openai" in them, such as:

- `semantic_kernel.connectors.openai.tokens.prompt`
- `semantic_kernel.connectors.openai.tokens.completion`
- `semantic_kernel.connectors.openai.tokens.total`

9. Using Azure with your data (Data Sources)

With the new `AzureOpenAIclient`, you can now specify your datasource thru the options and that requires a small change in your code to the new type.

Before

```
C#  
  
var promptExecutionSettings = new OpenAIPromptExecutionSettings  
{  
    AzureChatExtensionsOptions = new AzureChatExtensionsOptions  
    {  
        Extensions = [ new AzureSearchChatExtensionConfiguration  
        {  
            SearchEndpoint = new  
Uri(TestConfiguration.AzureAISeach.Endpoint),  
            Authentication = new  
OnYourDataApiKeyAuthenticationOptions(TestConfiguration.AzureAISeach.ApiKey  
),  
            IndexName = TestConfiguration.AzureAISeach.IndexName  
        }]  
    };  
};
```

After

```
C#  
  
var promptExecutionSettings = new AzureOpenAIPromptExecutionSettings  
{  
    AzureChatDataSource = new AzureSearchChatDataSource  
    {  
        Endpoint = new Uri(TestConfiguration.AzureAISeach.Endpoint),  
        Authentication =  
DataSourceAuthentication.FromApiKey(TestConfiguration.AzureAISeach.ApiKey),  
        IndexName = TestConfiguration.AzureAISeach.IndexName  
    }  
};
```

Tags: `WithData`, `AzureOpenAIChatCompletionWithDataConfig`,
`AzureOpenAIChatCompletionWithDataService`

10. Breaking glass scenarios

Breaking glass scenarios are scenarios where you may need to update your code to use the new OpenAI Connector. Below are some of the breaking changes that you may need to be aware of.

10.1 KernelContent Metadata

Some of the keys in the content metadata dictionary have changed and removed.

- Changed: `Created` -> `CreatedAt`
- Changed: `LogProbabilityInfo` -> `ContentTokenLogProbabilities`
- Changed: `PromptFilterResults` -> `ContentFilterResultForPrompt`
- Changed: `ContentFilterResultsForPrompt` -> `ContentFilterResultForResponse`
- Removed: `FinishDetails`
- Removed: `Index`
- Removed: `Enhancements`

10.2 Prompt Filter Results

The `PromptFilterResults` metadata type has changed from `IReadOnlyList<ContentFilterResultsForPrompt>` to `ContentFilterResultForPrompt`.

10.3 Content Filter Results

The `ContentFilterResultsForPrompt` type has changed from `ContentFilterResultsForChoice` to `ContentFilterResultForResponse`.

10.4 Finish Reason

The `FinishReason` metadata string value has changed from `stop` to `Stop`

10.5 Tool Calls

The `ToolCalls` metadata string value has changed from `tool_calls` to `ToolCalls`

10.6 LogProbs / Log Probability Info

The `LogProbabilityInfo` type has changed from `ChatChoiceLogProbabilityInfo` to `IReadOnlyList<ChatTokenLogProbabilityInfo>`.

10.7 Token Usage

The Token usage naming convention from `OpenAI` changed from `Completion`, `Prompt` tokens to `Output` and `Input` respectively. You will need to update your code to use the new naming.

The type also changed from `CompletionsUsage` to `ChatTokenUsage`.

Example of Token Usage Metadata Changes ↗

diff

Before

```
- var usage = FunctionResult.Metadata?["Usage"] as CompletionsUsage;  
- var completionTokens = usage?.CompletionTokens;  
- var promptTokens = usage?.PromptTokens;
```

After

```
+ var usage = FunctionResult.Metadata?["Usage"] as ChatTokenUsage;  
+ var promptTokens = usage?.InputTokens;  
+ var completionTokens = completionTokens: usage?.OutputTokens;
```

10.8 OpenAIclient

The `OpenAIclient` type previously was a Azure specific namespace type but now it is an `OpenAI` SDK namespace type, you will need to update your code to use the new `OpenAIclient` type.

When using Azure, you will need to update your code to use the new `AzureOpenAIclient` type.

10.9 OpenAIclientOptions

The `OpenAIclientOptions` type previously was a Azure specific namespace type but now it is an `OpenAI` SDK namespace type, you will need to update your code to use the new `AzureOpenAIclientOptions` type if you are using the new `AzureOpenAIclient` with any of the specific options for the Azure client.

10.10 Pipeline Configuration

The new `OpenAI` SDK uses a different pipeline configuration, and has a dependency on `System.ClientModel` package. You will need to update your code to use the new `HttpClientPipelineTransport` transport configuration where before you were using `HttpClientTransport` from `Azure.Core.Pipeline`.

Example of Pipeline Configuration ↗

diff

```
var clientOptions = new OpenAIclientOptions  
{  
    Before: From Azure.Core.Pipeline
```

```
-    Transport = new HttpClientTransport(httpClient),
-    RetryPolicy = new RetryPolicy(maxRetries: 0), // Disable Azure SDK
retry policy if and only if a custom HttpClient is provided.
-    Retry = { NetworkTimeout = Timeout.InfiniteTimeSpan } // Disable Azure
SDK default timeout
```

After: From OpenAI SDK -> System.ClientModel

```
+    Transport = new HttpClientPipelineTransport(httpClient),
+    RetryPolicy = new ClientRetryPolicy(maxRetries: 0); // Disable retry
policy if and only if a custom HttpClient is provided.
+    NetworkTimeout = Timeout.InfiniteTimeSpan; // Disable default timeout
};
```

Function Calling Migration Guide

Article • 09/24/2024

Semantic Kernel is gradually transitioning from the current function calling capabilities, represented by the `ToolCallBehavior` class, to the new enhanced capabilities, represented by the `FunctionChoiceBehavior` class. The new capability is service-agnostic and is not tied to any specific AI service, unlike the current model. Therefore, it resides in Semantic Kernel abstractions and will be used by all AI connectors working with function-calling capable AI models.

This guide is intended to help you to migrate your code to the new function calling capabilities.

Migrate `ToolCallBehavior.AutoInvokeKernelFunctions` behavior

The `ToolCallBehavior.AutoInvokeKernelFunctions` behavior is equivalent to the `FunctionChoiceBehavior.Auto` behavior in the new model.

C#

```
// Before
var executionSettings = new OpenAIPromptExecutionSettings { ToolCallBehavior =
    ToolCallBehavior.AutoInvokeKernelFunctions };

// After
var executionSettings = new OpenAIPromptExecutionSettings {
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() };
```

Migrate `ToolCallBehavior.EnableKernelFunctions` behavior

The `ToolCallBehavior.EnableKernelFunctions` behavior is equivalent to the `FunctionChoiceBehavior.Auto` behavior with disabled auto invocation.

C#

```
// Before
var executionSettings = new OpenAIPromptExecutionSettings { ToolCallBehavior
= ToolCallBehavior.EnableKernelFunctions };

// After
var executionSettings = new OpenAIPromptExecutionSettings {
FunctionChoiceBehavior = FunctionChoiceBehavior.Auto(autoInvoke: false) };
```

Migrate ToolCallBehavior.EnableFunctions behavior

The `ToolCallBehavior.EnableFunctions` behavior is equivalent to the `FunctionChoiceBehavior.Auto` behavior that configured with list of functions with disabled auto invocation.

C#

```
var function = kernel.CreateFunctionFromMethod(() => DayOfWeek.Friday,
"GetDayOfWeek", "Returns the current day of the week.");

// Before
var executionSettings = new OpenAIPromptExecutionSettings() {
ToolCallBehavior = ToolCallBehavior.EnableFunctions(functions:
[function.Metadata.ToOpenAIFunction()]) };

// After
var executionSettings = new OpenAIPromptExecutionSettings {
FunctionChoiceBehavior = FunctionChoiceBehavior.Auto(functions: [function],
autoInvoke: false) };
```

Migrate ToolCallBehavior.RequireFunction behavior

The `ToolCallBehavior.RequireFunction` behavior is equivalent to the `FunctionChoiceBehavior.Required` behavior that configured with list of functions with disabled auto invocation.

C#

```
var function = kernel.CreateFunctionFromMethod(() => DayOfWeek.Friday,
"GetDayOfWeek", "Returns the current day of the week.");

// Before
var executionSettings = new OpenAIPromptExecutionSettings() {
```

```
ToolCallBehavior = ToolCallBehavior.RequireFunction(functions:  
[function.Metadata.ToOpenAIFunction()]) };  
  
// After  
var executionSettings = new OpenAIPromptExecutionSettings {  
FunctionChoiceBehavior = FunctionChoiceBehavior.Required(functions:  
[function], autoInvoke: false) };
```

Replace the usage of connector-specific function call classes

Function calling functionality in Semantic Kernel allows developers to access a list of functions chosen by the AI model in two ways:

- Using connector-specific function call classes like `ChatToolCall` or `ChatCompletionsFunctionToolCall`, available via the `ToolCalls` property of the OpenAI-specific `OpenAIChatMessageContent` item in chat history.
- Using connector-agnostic function call classes like `FunctionCallContent`, available via the `Items` property of the connector-agnostic `ChatMessageContent` item in chat history.

Both ways are supported at the moment by the current and new models. However, we strongly recommend using the connector-agnostic approach to access function calls, as it is more flexible and allows your code to work with any AI connector that supports the new function-calling model. Moreover, considering that the current model will be deprecated soon, now is a good time to migrate your code to the new model to avoid breaking changes in the future.

So, if you use [Manual Function Invocation](#) with the connector-specific function call classes like in this code snippet:

C#

```
using System.Text.Json;  
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.ChatCompletion;  
using Microsoft.SemanticKernel.Connectors.OpenAI;  
using OpenAI.Chat;  
  
var chatHistory = new ChatHistory();  
  
var settings = new OpenAIPromptExecutionSettings() { ToolCallBehavior =  
ToolCallBehavior.EnableKernelFunctions };  
  
var result = await
```

```

chatCompletionService.GetChatMessageContentAsync(chatHistory, settings,
kernel);

// Current way of accessing function calls using connector specific classes.
var toolCalls =
((OpenAIChatMessageContent)result).ToolCalls.OfType<ChatToolCall>
().ToList();

while (toolCalls.Count > 0)
{
    // Adding function call from AI model to chat history
    chatHistory.Add(result);

    // Iterating over the requested function calls and invoking them
    foreach (var toolCall in toolCalls)
    {
        string content = kernel.Plugins.TryGetFunctionAndArguments(toolCall,
out KernelFunction? function, out KernelArguments? arguments) ?
            JsonSerializer.Serialize(await function.InvokeAsync(kernel,
arguments)).GetValue<object>() :
            "Unable to find function. Please try again!";

        // Adding the result of the function call to the chat history
        chatHistory.Add(new ChatMessageContent(
            AuthorRole.Tool,
            content,
            metadata: new Dictionary<string, object?>(1) { {
                OpenAIChatMessageContent.ToolIdProperty, toolCall.Id } }}));
    }

    // Sending the functions invocation results back to the AI model to get
    // the final response
    result = await
chatCompletionService.GetChatMessageContentAsync(chatHistory, settings,
kernel);
    toolCalls =
((OpenAIChatMessageContent)result).ToolCalls.OfType<ChatToolCall>
().ToList();
}

```

You can refactor it to use the connector-agnostic classes:

C#

```

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;

var chatHistory = new ChatHistory();

var settings = new PromptExecutionSettings() { FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto(autoInvoke: false) };

var messageContent = await

```

```
chatCompletionService.GetChatMessageContentAsync(chatHistory, settings,
kernel);

// New way of accessing function calls using connector agnostic function
// calling model classes.
var functionCalls =
FunctionCallContent.GetFunctionCalls(messageContent).ToArray();

while (functionCalls.Length != 0)
{
    // Adding function call from AI model to chat history
    chatHistory.Add(messageContent);

    // Iterating over the requested function calls and invoking them
    foreach (var functionCall in functionCalls)
    {
        var result = await functionCall.InvokeAsync(kernel);

        chatHistory.Add(result.ToChatMessage());
    }

    // Sending the functions invocation results to the AI model to get the
    final response
    messageContent = await
chatCompletionService.GetChatMessageContentAsync(chatHistory, settings,
kernel);
    functionCalls =
FunctionCallContent.GetFunctionCalls(messageContent).ToArray();
}
```

The code snippets above demonstrate how to migrate your code that uses the OpenAI AI connector. A similar migration process can be applied to the Gemini and Mistral AI connectors when they are updated to support the new function calling model.

Next steps

Now after you have migrated your code to the new function calling model, you can proceed to learn how to configure various aspects of the model that might better correspond to your specific scenarios by referring to the [function choice behaviors article](#)

[Function Choice Behaviors](#)

Stepwise Planner Migration Guide

Article • 10/16/2024

This migration guide shows how to migrate from `FunctionCallingStepwisePlanner` to a new recommended approach for planning capability - [Auto Function Calling](#). The new approach produces the results more reliably and uses fewer tokens compared to `FunctionCallingStepwisePlanner`.

Plan generation

Following code shows how to generate a new plan with Auto Function Calling by using `FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()`. After sending a request to AI model, the plan will be located in `chatHistory` object where a message with `Assistant` role will contain a list of functions (steps) to call.

Old approach:

```
C#  
  
Kernel kernel = Kernel  
    .CreateBuilder()  
    .AddOpenAIChatCompletion("gpt-4",  
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))  
    .Build();  
  
FunctionCallingStepwisePlanner planner = new();  
  
FunctionCallingStepwisePlannerResult result = await  
planner.ExecuteAsync(kernel, "Check current UTC time and return current  
weather in Boston city.");  
  
ChatHistory generatedPlan = result.ChatHistory;
```

New approach:

```
C#  
  
Kernel kernel = Kernel  
    .CreateBuilder()  
    .AddOpenAIChatCompletion("gpt-4",  
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))  
    .Build();  
  
IChatCompletionService chatCompletionService =  
kernel.GetRequiredService<IChatCompletionService>();
```

```

ChatHistory chatHistory = [];
chatHistory.AddUserMessage("Check current UTC time and return current
weather in Boston city.");

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

await chatCompletionService.GetChatMessageContentAsync(chatHistory,
executionSettings, kernel);

ChatHistory generatedPlan = chatHistory;

```

Execution of the new plan

Following code shows how to execute a new plan with Auto Function Calling by using `FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()`. This approach is useful when only result is needed without plan steps. In this case, `Kernel` object can be used to pass a goal to `InvokePromptAsync` method. The result of plan execution will be located in `FunctionResult` object.

Old approach:

```

C#

Kernel kernel = Kernel
    .CreateBuilder()
    .AddOpenAIChatCompletion("gpt-4",
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))
    .Build();

FunctionCallingStepwisePlanner planner = new();

FunctionCallingStepwisePlannerResult result = await
planner.ExecuteAsync(kernel, "Check current UTC time and return current
weather in Boston city.");

string planResult = result.FinalAnswer;

```

New approach:

```

C#

Kernel kernel = Kernel
    .CreateBuilder()
    .AddOpenAIChatCompletion("gpt-4",
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))
    .Build();

```

```

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

FunctionResult result = await kernel.InvokePromptAsync("Check current UTC
time and return current weather in Boston city.", new(executionSettings));

string planResult = result.ToString();

```

Execution of the existing plan

Following code shows how to execute an existing plan with Auto Function Calling by using `FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()`. This approach is useful when `ChatHistory` is already present (e.g. stored in cache) and it should be re-executed again and final result should be provided by AI model.

Old approach:

```

C#

Kernel kernel = Kernel
    .CreateBuilder()
    .AddOpenAIChatCompletion("gpt-4",
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))
    .Build();

FunctionCallingStepwisePlanner planner = new();
ChatHistory existingPlan = GetExistingPlan(); // plan can be stored in
database or cache for reusability.

FunctionCallingStepwisePlannerResult result = await
planner.ExecuteAsync(kernel, "Check current UTC time and return current
weather in Boston city.", existingPlan);

string planResult = result.FinalAnswer;

```

New approach:

```

C#

Kernel kernel = Kernel
    .CreateBuilder()
    .AddOpenAIChatCompletion("gpt-4",
Environment.GetEnvironmentVariable("OpenAI__ApiKey"))
    .Build();

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

ChatHistory existingPlan = GetExistingPlan(); // plan can be stored in

```

database or cache for reusability.

```
OpenAIPromptExecutionSettings executionSettings = new() {  
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() };  
  
ChatMessageContent result = await  
chatCompletionService.GetChatMessageContentAsync(existingPlan,  
executionSettings, kernel);  
  
string planResult = result.Content;
```

The code snippets above demonstrate how to migrate your code that uses Stepwise Planner to use Auto Function Calling. Learn more about [Function Calling with chat completion](#).

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.
AzureAllInferenceKernelBuilderExtensions	Provides extension methods for IKernelBuilder to configure Azure AI Inference connectors.
AzureAllInferenceServiceCollectionExtensions	Provides extension methods for IServiceCollection to configure Azure AI Inference connectors.
AzureAISearchKernelBuilderExtensions	Extension methods to register Azure AI Search IVectorStore instances on the IKernelBuilder .
AzureAISearchServiceCollectionExtensions	Extension methods to register Azure AI Search IVectorStore instances on an IServiceCollection .
AzureCosmosDBMongoDBKernelBuilderExtensions	Extension methods to register Azure CosmosDB MongoDB IVectorStore instances on the IKernelBuilder .
AzureCosmosDBMongoDBServiceCollectionExtensions	Extension methods to register Azure CosmosDB MongoDB IVectorStore instances on an IServiceCollection .
AzureCosmosDBNoSQLKernelBuilderExtensions	Extension methods to register Azure CosmosDB NoSQL IVectorStore instances on the IKernelBuilder .

AzureCosmos DBNoSQLServiceCollection Extensions	Extension methods to register Azure CosmosDB NoSQL IVectorStore instances on an IServiceCollection .
AzureOpenAIKernelBuilder Extensions	Provides extension methods for IKernelBuilder to configure Azure OpenAI connectors.
AzureOpenAIService CollectionExtensions	Provides extension methods for IServiceCollection to configure Azure OpenAI connectors.
BinaryContent	Provides access to binary content.
CancelKernelEventArgs	Provides an EventArgs for cancelable operations related to Kernel -based operations.
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 AI model.
FunctionCallContentBuilder	A builder class for creating FunctionCallContent objects from incremental function call updates represented by StreamingFunctionCallUpdateContent .
FunctionChoiceBehavior	Represents the base class for different function choice behaviors. These behaviors define the way functions are chosen by AI model and various aspects of their invocation by AI connectors.
FunctionChoiceBehavior Configuration	Represents function choice behavior configuration produced by a FunctionChoiceBehavior .
FunctionChoiceBehavior ConfigurationContext	The context is to be provided by the choice behavior consumer – AI connector in order to obtain the choice behavior configuration.
FunctionChoiceBehavior Options	Represents the options for a function choice behavior. At the moment this is empty but it is being included for future use.
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.

FunctionEventArgs	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.
GoogleAIKernelBuilder Extensions	Extensions for adding GoogleAI generation services to the application.
GoogleAIMemoryBuilder Extensions	Provides extension methods for the MemoryBuilder class to configure GoogleAI connector.
GoogleAIServiceCollection Extensions	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.
HuggingFaceKernelBuilder Extensions	Provides extension methods for the IKernelBuilder class to configure Hugging Face connectors.
HuggingFaceService CollectionExtensions	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> .
KernelBuilderExtensions	Extension methods to register Data services on the IKernelBuilder .
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.
KernelFunctionAttribute	Specifies that a method on a class imported as a plugin should be included as a KernelFunction in the resulting KernelPlugin .
KernelFunctionCanceledException	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.
KernelFunctionFromMethodOptions	Optional options that can be provided when creating a KernelFunction from a method.
KernelFunctionMarkdown	Factory methods for creating instances.
KernelFunctionMetadata	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.
KernelParameterMetadata	Provides read-only metadata for a KernelFunction parameter.
KernelPlugin	Represents a plugin that may be registered with a Kernel .
KernelPluginCollection	Provides a collection of KernelPlugins .
KernelPluginExtensions	Provides extension methods for working with KernelPlugins and collections of them.
KernelPluginFactory	Provides static factory methods for creating commonly-used plugin implementations.
KernelPromptTemplateFactory	Provides an implementation of IPromptTemplateFactory for the 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.
OpenAIChatHistoryExtensions	Chat history extensions.
OpenAIKernelBuilderExtensions	Sponsor extensions class for IKernelBuilder .
OpenAIServiceCollectionExtensions	Sponsor extensions class for IServiceCollection .
OutputVariable	Represents an output variable returned from a prompt function.
PineconeKernelBuilderExtensions	Extension methods to register Pinecone IVectorStore instances on the IKernelBuilder .
PineconeServiceCollectionExtensions	Extension methods to register Pinecone IVectorStore instances on an IServiceCollection .
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.
QdrantKernelBuilder Extensions	Extension methods to register Qdrant IVectorStore instances on the IKernelBuilder .
QdrantServiceCollection Extensions	Extension methods to register Qdrant IVectorStore instances on an IServiceCollection .
RedisKernelBuilderExtensions	Extension methods to register Redis IVectorStore instances on the IKernelBuilder .
RedisServiceCollection Extensions	Extension methods to register Redis IVectorStore instances on an IServiceCollection .
RestApiOperationResponse	The REST API operation response.
RestApiOperationResponse Converter	Converts a object of RestApiOperationResponse type to string type.
ServiceCollectionExtensions	Extension methods to register Data services on an IServiceCollection .
StreamingChatMessage Content	Abstraction of chat message content chunks when using streaming from IChatCompletionService interface.
StreamingFileReference Content	Content type to support file references.
StreamingFunctionCall UpdateContent	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.
VertexAIKernelBuilder Extensions	Extensions for adding VertexAI generation services to the application.
VertexAIMemoryBuilder Extensions	Provides extension methods for the MemoryBuilder class to configure VertexAI connector.
VertexAIServiceCollection Extensions	Extensions for adding VertexAI generation services to the application.

WeaviateKernelBuilderExtensions	Extension methods to register Weaviate IVectorStore instances on the IKernelBuilder .
WeaviateServiceCollectionExtensions	Extension methods to register Weaviate IVectorStore instances on an IServiceCollection

Structs

[] [Expand table](#)

FunctionChoice	Represents an AI model's decision-making strategy for calling functions, offering predefined choices: Auto, Required, and None. Auto allows the model to decide if and which functions to call, Required enforces calling one or more functions, and None prevents any function calls, generating only a user-facing message.
----------------	---

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 .
IKernelBuilderPlugins	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.
IPromptTemplateFactory	Represents a factory for prompt templates for one or more prompt template formats.
IReadOnlyKernelPluginCollection	Provides a read-only collection of KernelPlugins .

Kernel Class

Reference

The Kernel of Semantic Kernel.

This is the main entry point for Semantic Kernel. It provides the ability to run functions and manage filters, plugins, 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, KernelPlugin] | list[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, Callable[[FILTER_CONTEXT_TYPE,
Callable[[FILTER_CONTEXT_TYPE], None]], None]]] = None,
prompt_rendering_filters: list[tuple[int, Callable[[FILTER_CONTEXT_TYPE,
Callable[[FILTER_CONTEXT_TYPE], None]], None]]] = None,
auto_function_invocation_filters: list[tuple[int,
Callable[[FILTER_CONTEXT_TYPE, Callable[[FILTER_CONTEXT_TYPE], None]], None]], None]]] = None)
```

Parameters

[+] Expand table

Name	Description
plugins	The plugins to be used by the kernel, will be rewritten to a dict with plugin name as key Default value: None
services	The services to be used by the kernel, will be rewritten to a dict with service_id as key

Name	Description
	Default value: None
ai_service_selector	The AI service selector to be used by the kernel, default is based on order of execution settings. Default value: None
**kwargs Required*	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.

Keyword-Only Parameters

[\[\] Expand table](#)

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

Methods

[\[\] Expand table](#)

add_embedding_to_object	Gather all fields to embed, batch the embedding generation and store.
invoke	Execute a function and return the FunctionResult.
invoke_function_call	Processes the provided FunctionCallContent and updates the chat history.
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>

add_embedding_to_object

Gather all fields to embed, batch the embedding generation and store.

Python

```
async add_embedding_to_object(inputs: TDataModel | Sequence[TDataModel],
field_to_embed: str, field_to_store: str, execution_settings: dict[str,
PromptExecutionSettings], container_mode: bool = False, cast_function:
Callable[[list[float]], Any] | None = None, **kwargs: Any)
```

Parameters

[\[\]](#) Expand table

Name	Description
inputs Required*	
field_to_embed Required*	
field_to_store Required*	
execution_settings Required*	
container_mode	Default value: False
cast_function	Default value: None

invoke

Execute a function and return the FunctionResult.

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	<xref:semantic_kernel.kernel.KernelFunction> The function or functions to execute, this value has precedence when supplying both this and using function_name and plugin_name, if this is none, function_name and plugin_name are used and cannot be None. Default value: None
arguments	<xref:semantic_kernel.kernel.KernelArguments> The arguments to pass to the function(s), optional Default value: None
function_name	<xref:<xref:semantic_kernel.kernel.str None>> The name of the function to execute Default value: None
plugin_name	<xref:<xref:semantic_kernel.kernel.str None>> The name of the plugin to execute Default value: None
metadata	dict[str,<xref: Any>] The metadata to pass to the function(s) Default value: {}
kwargs Required*	dict[str,<xref: Any>] arguments that can be used instead of supplying KernelArguments

Exceptions

[+] Expand table

Type	Description
KernelInvokeException	If an error occurs during function invocation

invoke_function_call

Processes the provided FunctionCallContent and updates the chat history.

Python

```
async invoke_function_call(function_call: FunctionCallContent,
chat_history: ChatHistory, arguments: KernelArguments | None = None,
function_call_count: int | None = None, request_index: int | None = None,
function_behavior: FunctionChoiceBehavior = None) ->
AutoFunctionInvocationContext | None
```

Parameters

[\[\] Expand table](#)

Name	Description
function_call Required*	
chat_history Required*	
arguments	Default value: None
function_call_count	Default value: None
request_index	Default value: None
function_behavior	Default value: None

invoke_prompt

Invoke a function from the provided prompt.

Python

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

Parameters

[\[\] Expand table](#)

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

Returns

[Expand table](#)

Type	Description
<code>FunctionResult list[FunctionResult] None</code>	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(prompt: str, function_name: str | None = None,
plugin_name: str | None = None, arguments: KernelArguments | None = None,
template_format: Literal['semantic-kernel', 'handlebars', 'jinja2'] =
'semantic-kernel', return_function_results: bool | None = False,
**kwargs: Any) -> AsyncIterable[list[StreamingContentMixin] |
FunctionResult | list[FunctionResult]]
```

Parameters

[\[\] Expand table](#)

Name	Description
prompt Required*	str The prompt to use
function_name	str The name of the function, optional Default value: None
plugin_name	str The name of the plugin, optional Default value: None
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
return_function_results	bool If True, the function results are yielded as a list[FunctionResult] Default value: False
kwargs Required*	dict[str,<xref: Any>] arguments that can be used instead of supplying KernelArguments

Returns

[\[\] Expand table](#)

Type	Description
AsyncIterable[StreamingContentMixin]	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.

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[FunctionResult], Any]
```

Parameters

[] Expand table

Name	Description
function	<xref:semantic_kernel.kernel.KernelFunction> The function to execute, this value has precedence when supplying both this and using function_name and plugin_name, if this is none, function_name and plugin_name are used and cannot be None. Default value: None
arguments	<xref:<xref:semantic_kernel.kernel.KernelArguments None>> The arguments to pass to the function(s), optional Default value: None
function_name	<xref:<xref:semantic_kernel.kernel.str None>> The name of the function to execute Default value: None
plugin_name	<xref:<xref:semantic_kernel.kernel.str None>> The name of the plugin to execute Default value: None
metadata	dict[str,<xref: Any>] The metadata to pass to the function(s) Default value: {}
return_function_results	bool If True, the function results are yielded as a list[FunctionResult] Default value: False
content Required*	content (<xref:in addition to the streaming>)
yielded. Required*	yielded. (<xref:otherwise only the streaming content is>)
kwargs	dict[str,<xref: Any>]

Name	Description
Required*	arguments that can be used instead of supplying KernelArguments

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`] objects.

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)}
```

function_invocation_filters

Filters applied during function invocation, from KernelFilterExtension.

Python

```
function_invocation_filters: list[tuple[int,  
Callable[[FILTER_CONTEXT_TYPE, Callable[[FILTER_CONTEXT_TYPE], None]],  
None]]]
```

prompt_rendering_filters

Filters applied during prompt rendering, from KernelFilterExtension.

Python

```
prompt_rendering_filters: list[tuple[int, Callable[[FILTER_CONTEXT_TYPE,  
Callable[[FILTER_CONTEXT_TYPE], None]], None]]]
```

auto_function_invocation_filters

Filters applied during auto function invocation, from KernelFilterExtension.

Python

```
auto_function_invocation_filters: list[tuple[int,  
Callable[[FILTER_CONTEXT_TYPE, Callable[[FILTER_CONTEXT_TYPE], None]],  
None]]]
```

plugins

A dict with the plugins registered with the Kernel, from KernelFunctionExtension.

Python

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

services

A dict with the services registered with the Kernel, from KernelServicesExtension.

Python

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

ai_service_selector

The AI service selector to be used by the kernel, from KernelServicesExtension.

Python

```
ai_service_selector: AIServiceSelector
```

retry_mechanism

The retry mechanism to be used by the kernel, from KernelReliabilityExtension.

Python

```
retry_mechanism: RetryMechanismBase
```