AI apps for .NET developers

Learn to build AI apps with .NET. Browse sample code, tutorials, quickstarts, conceptual articles, and more.

Get started

OVERVIEW

Develop .NET apps with AI features

Microsoft.Extensions.AI libraries

GET STARTED

Connect to and prompt an AI model

Build an Azure AI chat app

Generate images using Azure AI

Essential concepts

CONCEPT

How generative AI and LLMs work

Understand tokens

Preserve semantic meaning with embeddings

Semantic search with vector databases

Prompt engineering

Evaluation libraries

Common tasks

HOW-TO GUIDE

Authenticate App Service to Azure OpenAI

Authenticate App Service to a vector database

Use Redis with the Semantic Kernel SDK

Use custom and local AI models with the Semantic Kernel SDK

Work with content filtering

Tutorials

TUTORIAL

Scale Azure OpenAI with Azure Container Apps

.NET enterprise chat sample using RAG

Implement RAG using vector search

Evaluate a model's response

Training

TUTORIAL

Fundamentals of Azure OpenAI Service

Generate conversations Azure OpenAI completions

.NET enterprise chat sample using RAG

Develop AI agents using Azure OpenAI

API reference

REFERENCE

ChatClientBuilder

IChatClient

IEmbeddingGenerator

Develop .NET apps with AI features

Article • 05/02/2025

With .NET, you can use artificial intelligence (AI) to automate and accomplish complex tasks in your applications using the tools, platforms, and services that are familiar to you.

Why choose .NET to build AI apps?

Millions of developers use .NET to create applications that run on the web, on mobile and desktop devices, or in the cloud. By using .NET to integrate AI into your applications, you can take advantage of all that .NET has to offer:

- A unified story for building web UIs, APIs, and applications.
- Supported on Windows, macOS, and Linux.
- Is open-source and community-focused.
- Runs on top of the most popular web servers and cloud platforms.
- Provides powerful tooling to edit, debug, test, and deploy.

What can you build with AI and .NET?

The opportunities with AI are near endless. Here are a few examples of solutions you can build using AI in your .NET applications:

- Language processing: Create virtual agents or chatbots to talk with your data and generate content and images.
- Computer vision: Identify objects in an object or video.
- Audio generation: Use synthesized voices to interact with customers.
- Classification: Label the severity of a customer-reported issue.
- Task automation: Automatically perform the next step in a workflow as tasks are completed.

Recommended learning path

We recommend the following sequence of tutorials and articles for an introduction to developing applications with AI and .NET:

Expand table

Scenario	Tutorial
Create a chat application	Build an Azure AI chat app with .NET

Scenario	Tutorial
Summarize text	Summarize text using Azure AI chat app with .NET
Chat with your data	Get insight about your data from an .NET Azure AI chat app
Call .NET functions with AI	Extend Azure AI using tools and execute a local function with .NET
Generate images	Generate images using Azure AI with .NET
Train your own model	ML.NET tutorial

Browse the table of contents to learn more about the core concepts, starting with How generative AI and LLMs work.

Next steps

- Quickstart: Build an Azure AI chat app with .NET
- Video series: Machine Learning and AI with .NET

Connect to and prompt an AI model

Article • 05/18/2025

In this quickstart, you learn how to create a .NET console chat app to connect to and prompt an OpenAI or Azure OpenAI model. The app uses the Microsoft.Extensions.AI library so you can write code using AI abstractions rather than a specific SDK. AI abstractions enable you to change the underlying AI model with minimal code changes.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8.0 SDK.
- An API key from OpenAI so you can run this sample.

> **Note**
>
> You can also use **Semantic Kernel** to accomplish the tasks in this article. Semantic Kernel is a lightweight, open-source SDK that lets you build AI agents and integrate the latest AI models into your .NET apps.

Create the app

Complete the following steps to create a .NET console app to connect to an AI model.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

   .NET CLI

   ```
   dotnet new console -o ExtensionsAI
   ```

2. Change directory into the app folder:

   .NET CLI

   ```
   cd ExtensionsAI
   ```

3. Install the required packages:

   Bash

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

4. Open the app in Visual Studio Code or your editor of choice.

Configure the app

1. Navigate to the root of your .NET project from a terminal or command prompt.

2. Run the following commands to configure your OpenAI API key as a secret for the sample app:

```Bash
dotnet user-secrets init
dotnet user-secrets set OpenAIKey <your-OpenAI-key>
dotnet user-secrets set ModelName <your-OpenAI-model-name>
```

Add the app code

The app uses the Microsoft.Extensions.AI package to send and receive requests to the AI model.

1. Copy the benefits.md file to your project directory. Configure the project to copy this file to the output directory. If you're using Visual Studio, right-click on the file in Solution Explorer, select **Properties**, and then set **Copy to Output Directory** to **Copy if newer**.

2. In the `Program.cs` file, add the following code to connect and authenticate to the AI model.

```C#
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;

IConfigurationRoot config = new ConfigurationBuilder()
    .AddUserSecrets<Program>()
    .Build();
string? model = config["ModelName"];
string? key = config["OpenAIKey"];
```

```
IChatClient client =
    new OpenAIClient(key).GetChatClient(model).AsIChatClient();
```

3. Add code to read the `benefits.md` file content and then create a prompt for the model. The prompt instructs the model to summarize the file's text content in 20 words or less.

```C#
string text = File.ReadAllText("benefits.md");
string prompt = $"""
    Summarize the the following text in 20 words or less:
    {text}
    """;
```

4. Call the `GetResponseAsync` method to send the prompt to the model to generate a response.

```C#
// Submit the prompt and print out the response.
ChatResponse response = await client.GetResponseAsync(
    prompt,
    new ChatOptions { MaxOutputTokens = 400 });
Console.WriteLine(response);
```

5. Run the app:

```.NET CLI
dotnet run
```

   The app prints out the completion response from the AI model. Customize the text content of the `benefits.md` file or the length of the summary to see the differences in the responses.

Next steps

- Quickstart - Build an AI chat app with .NET
- Generate text and conversations with .NET and Azure OpenAI Completions

.NET + AI ecosystem tools and SDKs

Article • 05/29/2025

The .NET ecosystem provides many powerful tools, libraries, and services to develop AI applications. .NET supports both cloud and local AI model connections, many different SDKs for various AI and vector database services, and other tools to help you build intelligent apps of varying scope and complexity.

> **Important**
>
> Not all of the SDKs and services presented in this article are maintained by Microsoft. When considering an SDK, make sure to evaluate its quality, licensing, support, and compatibility to ensure they meet your requirements.

Microsoft.Extensions.AI libraries

Microsoft.Extensions.AI is a set of core .NET libraries that provide a unified layer of C# abstractions for interacting with AI services, such as small and large language models (SLMs and LLMs), embeddings, and middleware. These APIs were created in collaboration with developers across the .NET ecosystem, including Semantic Kernel. The low-level APIs, such as IChatClient and IEmbeddingGenerator<TInput,TEmbedding>, were extracted from Semantic Kernel and moved into the Microsoft.Extensions.AI namespace.

`Microsoft.Extensions.AI` provides abstractions that can be implemented by various services, all adhering to the same core concepts. This library is not intended to provide APIs tailored to any specific provider's services. The goal of `Microsoft.Extensions.AI` is to act as a unifying layer within the .NET ecosystem, enabling developers to choose their preferred frameworks and libraries while ensuring seamless integration and collaboration across the ecosystem.

Semantic Kernel for .NET

If you just want to use the low-level services, such as IChatClient and IEmbeddingGenerator<TInput,TEmbedding>, you can reference the `Microsoft.Extensions.AI.Abstractions` package directly from your app. However, if you want to use higher-level, more opinionated approaches to AI, then you should use Semantic Kernel.

Semantic Kernel, which has a dependency on the `Microsoft.Extensions.AI.Abstractions` package, is an open-source library that enables AI integration and orchestration capabilities in your .NET apps. Its connectors provides concrete implementations of IChatClient and

IEmbeddingGenerator<TInput,TEmbedding> for different services, including OpenAI, Amazon Bedrock, and Google Gemini.

The Semantic Kernel SDK is generally the recommended AI orchestration tool for .NET apps that use one or more AI services in combination with other APIs or web services, data stores, and custom code. Semantic Kernel benefits enterprise developers in the following ways:

- Streamlines integration of AI capabilities into existing applications to enable a cohesive solution for enterprise products.
- Minimizes the learning curve of working with different AI models or services by providing abstractions that reduce complexity.
- Improves reliability by reducing the unpredictable behavior of prompts and responses from AI models. You can fine-tune prompts and plan tasks to create a controlled and predictable user experience.

For more information, see the Semantic Kernel documentation.

.NET SDKs for building AI apps

Many different SDKs are available to build .NET apps with AI capabilities depending on the target platform or AI model. OpenAI models offer powerful generative AI capabilities, while other Azure AI Services provide intelligent solutions for a variety of specific scenarios.

.NET SDKs for OpenAI models

Expand table

NuGet package	Supported models	Maintainer or vendor	Documentation
Microsoft.SemanticKernel	OpenAI models Azure OpenAI supported models	Semantic Kernel (Microsoft)	Semantic Kernel documentation
Azure OpenAI SDK	Azure OpenAI supported models	Azure SDK for .NET (Microsoft)	Azure OpenAI services documentation
OpenAI SDK	OpenAI supported models	OpenAI SDK for .NET (OpenAI)	OpenAI services documentation

.NET SDKs for Azure AI Services

Azure offers many other AI services to build specific application capabilities and workflows. Most of these services provide a .NET SDK to integrate their functionality into custom apps.

Some of the most commonly used services are shown in the following table. For a complete list of available services and learning resources, see the Azure AI Services documentation.

Expand table

Service	Description
Azure AI Search	Bring AI-powered cloud search to your mobile and web apps.
Azure AI Content Safety	Detect unwanted or offensive content.
Azure AI Document Intelligence	Turn documents into intelligent data-driven solutions.
Azure AI Language	Build apps with industry-leading natural language understanding capabilities.
Azure AI Speech	Speech to text, text to speech, translation, and speaker recognition.
Azure AI Translator	AI-powered translation technology with support for more than 100 languages and dialects.
Azure AI Vision	Analyze content in images and videos.

Develop with local AI models

.NET apps can also connect to local AI models for many different development scenarios. Semantic Kernel is the recommended tool to connect to local models using .NET. Semantic Kernel can connect to many different models hosted across a variety of platforms and abstracts away lower-level implementation details.

For example, you can use Ollama to connect to local AI models with .NET, including several small language models (SLMs) developed by Microsoft:

Expand table

Model	Description
phi3 models	A family of powerful SLMs with groundbreaking performance at low cost and low latency.
orca models	Research models in tasks such as reasoning over user-provided data, reading comprehension, math problem solving, and text summarization.

> **Note**
>
> The preceding SLMs can also be hosted on other services, such as Azure.

Connect to vector databases and services

AI applications often use data vector databases and services to improve relevancy and provide customized functionality. Many of these services provide a native SDK for .NET, while others offer a REST service you can connect to through custom code. Semantic Kernel provides an extensible component model that enables you to use different vector stores without needing to learn each SDK.

Semantic Kernel provides connectors for the following vector databases and services:

Expand table

Vector service	Semantic Kernel connector	.NET SDK
Azure AI Search	Microsoft.SemanticKernel.Connectors.AzureAISearch	Azure.Search.Documents
Azure Cosmos DB for NoSQL	Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL	Microsoft.Azure.Cosmos
Azure Cosmos DB for MongoDB	Microsoft.SemanticKernel.Connectors.AzureCosmosDBMongoDB	MongoDb.Driver
Azure PostgreSQL Server	Microsoft.SemanticKernel.Connectors.Postgres	Npgsql
Azure SQL Database	Microsoft.SemanticKernel.Connectors.SqlServer	Microsoft.Data.SqlClient
Chroma	Microsoft.SemanticKernel.Connectors.Chroma	NA
DuckDB	Microsoft.SemanticKernel.Connectors.DuckDB	DuckDB.NET.Data.Full
Milvus	Microsoft.SemanticKernel.Connectors.Milvus	Milvus.Client
MongoDB Atlas Vector Search	Microsoft.SemanticKernel.Connectors.MongoDB	MongoDb.Driver
Pinecone	Microsoft.SemanticKernel.Connectors.Pinecone	REST API
Postgres	Microsoft.SemanticKernel.Connectors.Postgres	Npgsql
Qdrant	Microsoft.SemanticKernel.Connectors.Qdrant	Qdrant.Client

Vector service	Semantic Kernel connector	.NET SDK
Redis	Microsoft.SemanticKernel.Connectors.Redis	StackExchange.Redis
Weaviate	Microsoft.SemanticKernel.Connectors.Weaviate	REST API

To discover .NET SDK and API support, visit the documentation for each respective service.

Next steps

- What is Semantic Kernel?
- Quickstart - Summarize text using Azure AI chat app with .NET

Microsoft.Extensions.AI libraries

06/23/2025

.NET developers need to integrate and interact with a growing variety of artificial intelligence (AI) services in their apps. The `Microsoft.Extensions.AI` libraries provide a unified approach for representing generative AI components, and enable seamless integration and interoperability with various AI services. This article introduces the libraries and provides in-depth usage examples to help you get started.

The packages

The Microsoft.Extensions.AI.Abstractions package provides the core exchange types, including IChatClient and IEmbeddingGenerator<TInput,TEmbedding>. Any .NET library that provides an LLM client can implement the `IChatClient` interface to enable seamless integration with consuming code.

The Microsoft.Extensions.AI package has an implicit dependency on the `Microsoft.Extensions.AI.Abstractions` package. This package enables you to easily integrate components such as automatic function tool invocation, telemetry, and caching into your applications using familiar dependency injection and middleware patterns. For example, it provides the UseOpenTelemetry(ChatClientBuilder, ILoggerFactory, String, Action<OpenTelemetryChatClient>) extension method, which adds OpenTelemetry support to the chat client pipeline.

Which package to reference

Libraries that provide implementations of the abstractions typically reference only `Microsoft.Extensions.AI.Abstractions`.

To also have access to higher-level utilities for working with generative AI components, reference the `Microsoft.Extensions.AI` package instead (which itself references `Microsoft.Extensions.AI.Abstractions`). Most consuming applications and services should reference the `Microsoft.Extensions.AI` package along with one or more libraries that provide concrete implementations of the abstractions.

Install the packages

For information about how to install NuGet packages, see dotnet package add or Manage package dependencies in .NET applications.

API usage examples

The following subsections show specific IChatClient usage examples:

- Request a chat response
- Request a streaming chat response
- Tool calling
- Cache responses
- Use telemetry
- Provide options
- Pipelines of functionality
- Custom IChatClient middleware
- Dependency injection
- Stateless vs. stateful clients

The following sections show specific IEmbeddingGenerator usage examples:

- Create embeddings
- Pipelines of functionality

The `IChatClient` interface

The IChatClient interface defines a client abstraction responsible for interacting with AI services that provide chat capabilities. It includes methods for sending and receiving messages with multi-modal content (such as text, images, and audio), either as a complete set or streamed incrementally. Additionally, it allows for retrieving strongly typed services provided by the client or its underlying services.

.NET libraries that provide clients for language models and services can provide an implementation of the `IChatClient` interface. Any consumers of the interface are then able to interoperate seamlessly with these models and services via the abstractions. You can see a simple implementation at Sample implementations of IChatClient and IEmbeddingGenerator.

Request a chat response

With an instance of IChatClient, you can call the IChatClient.GetResponseAsync method to send a request and get a response. The request is composed of one or more messages, each of which is composed of one or more pieces of content. Accelerator methods exist to simplify common cases, such as constructing a request for a single piece of text content.

```
C#
```

```
using Microsoft.Extensions.AI;
using OllamaSharp;

IChatClient client = new OllamaApiClient(
    new Uri("http://localhost:11434/"), "phi3:mini");

Console.WriteLine(await client.GetResponseAsync("What is AI?"));
```

The core `IChatClient.GetResponseAsync` method accepts a list of messages. This list represents the history of all messages that are part of the conversation.

C#

```
Console.WriteLine(await client.GetResponseAsync(
[
    new(ChatRole.System, "You are a helpful AI assistant"),
    new(ChatRole.User, "What is AI?"),
]));
```

The ChatResponse that's returned from `GetResponseAsync` exposes a list of ChatMessage instances that represent one or more messages generated as part of the operation. In common cases, there is only one response message, but in some situations, there can be multiple messages. The message list is ordered, such that the last message in the list represents the final message to the request. To provide all of those response messages back to the service in a subsequent request, you can add the messages from the response back into the messages list.

C#

```
List<ChatMessage> history = [];
while (true)
{
    Console.Write("Q: ");
    history.Add(new(ChatRole.User, Console.ReadLine()));

    ChatResponse response = await client.GetResponseAsync(history);
    Console.WriteLine(response);

    history.AddMessages(response);
}
```

Request a streaming chat response

The inputs to IChatClient.GetStreamingResponseAsync are identical to those of `GetResponseAsync`. However, rather than returning the complete response as part of a

ChatResponse object, the method returns an IAsyncEnumerable<T> where `T` is ChatResponseUpdate, providing a stream of updates that collectively form the single response.

C#

```
await foreach (ChatResponseUpdate update in client.GetStreamingResponseAsync("What
is AI?"))
{
    Console.Write(update);
}
```

> **Tip**
>
> Streaming APIs are nearly synonymous with AI user experiences. C# enables compelling scenarios with its `IAsyncEnumerable<T>` support, allowing for a natural and efficient way to stream data.

As with `GetResponseAsync`, you can add the updates from IChatClient.GetStreamingResponseAsync back into the messages list. Because the updates are individual pieces of a response, you can use helpers like ToChatResponse(IEnumerable<ChatResponseUpdate>) to compose one or more updates back into a single ChatResponse instance.

Helpers like AddMessages compose a ChatResponse and then extract the composed messages from the response and add them to a list.

C#

```
List<ChatMessage> chatHistory = [];
while (true)
{
    Console.Write("Q: ");
    chatHistory.Add(new(ChatRole.User, Console.ReadLine()));

    List<ChatResponseUpdate> updates = [];
    await foreach (ChatResponseUpdate update in
        client.GetStreamingResponseAsync(history))
    {
        Console.Write(update);
        updates.Add(update);
    }
    Console.WriteLine();

    chatHistory.AddMessages(updates);
}
```

Tool calling

Some models and services support *tool calling*. To gather additional information, you can configure the ChatOptions with information about tools (usually .NET methods) that the model can request the client to invoke. Instead of sending a final response, the model requests a function invocation with specific arguments. The client then invokes the function and sends the results back to the model with the conversation history. The `Microsoft.Extensions.AI.Abstractions` library includes abstractions for various message content types, including function call requests and results. While `IChatClient` consumers can interact with this content directly, `Microsoft.Extensions.AI` provides helpers that can enable automatically invoking the tools in response to corresponding requests. The `Microsoft.Extensions.AI.Abstractions` and `Microsoft.Extensions.AI` libraries provide the following types:

- AIFunction: Represents a function that can be described to an AI model and invoked.
- AIFunctionFactory: Provides factory methods for creating `AIFunction` instances that represent .NET methods.
- FunctionInvokingChatClient: Wraps an `IChatClient` as another `IChatClient` that adds automatic function-invocation capabilities.

The following example demonstrates a random function invocation (this example depends on the OllamaSharp NuGet package):

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;

string GetCurrentWeather() => Random.Shared.NextDouble() > 0.5 ? "It's sunny" :
"It's raining";

IChatClient client = new OllamaApiClient(new Uri("http://localhost:11434"),
"llama3.1");

client = ChatClientBuilderChatClientExtensions
    .AsBuilder(client)
    .UseFunctionInvocation()
    .Build();

ChatOptions options = new() { Tools =
[AIFunctionFactory.Create(GetCurrentWeather)] };

var response = client.GetStreamingResponseAsync("Should I wear a rain coat?",
options);
await foreach (var update in response)
{
```

```
    Console.Write(update);
}
```

The preceding code:

- Defines a function named `GetCurrentWeather` that returns a random weather forecast.
- Instantiates a ChatClientBuilder with an `OllamaSharp.OllamaApiClient` and configures it to use function invocation.
- Calls `GetStreamingResponseAsync` on the client, passing a prompt and a list of tools that includes a function created with Create.
- Iterates over the response, printing each update to the console.

Cache responses

If you're familiar with Caching in .NET, it's good to know that Microsoft.Extensions.AI provides other such delegating `IChatClient` implementations. The DistributedCachingChatClient is an `IChatClient` that layers caching around another arbitrary `IChatClient` instance. When a novel chat history is submitted to the `DistributedCachingChatClient`, it forwards it to the underlying client and then caches the response before sending it back to the consumer. The next time the same history is submitted, such that a cached response can be found in the cache, the `DistributedCachingChatClient` returns the cached response rather than forwarding the request along the pipeline.

C#

```
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Options;
using OllamaSharp;

var sampleChatClient = new OllamaApiClient(new Uri("http://localhost:11434"),
"llama3.1");

IChatClient client = new ChatClientBuilder(sampleChatClient)
    .UseDistributedCache(new MemoryDistributedCache(
        Options.Create(new MemoryDistributedCacheOptions())))
    .Build();

string[] prompts = ["What is AI?", "What is .NET?", "What is AI?"];

foreach (var prompt in prompts)
{
    await foreach (var update in client.GetStreamingResponseAsync(prompt))
    {
        Console.Write(update);
    }
```

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

This example depends on the Microsoft.Extensions.Caching.Memory NuGet package. For more information, see Caching in .NET.

Use telemetry

Another example of a delegating chat client is the OpenTelemetryChatClient. This implementation adheres to the OpenTelemetry Semantic Conventions for Generative AI systems. Similar to other `IChatClient` delegators, it layers metrics and spans around other arbitrary `IChatClient` implementations.

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;
using OpenTelemetry.Trace;

// Configure OpenTelemetry exporter.
string sourceName = Guid.NewGuid().ToString();
TracerProvider tracerProvider = OpenTelemetry.Sdk.CreateTracerProviderBuilder()
    .AddSource(sourceName)
    .AddConsoleExporter()
    .Build();

IChatClient ollamaClient = new OllamaApiClient(
    new Uri("http://localhost:11434/"), "phi3:mini");

IChatClient client = new ChatClientBuilder(ollamaClient)
    .UseOpenTelemetry(
        sourceName: sourceName,
        configure: c => c.EnableSensitiveData = true)
    .Build();

Console.WriteLine((await client.GetResponseAsync("What is AI?")).Text);
```

(The preceding example depends on the OpenTelemetry.Exporter.Console NuGet package.)

Alternatively, the LoggingChatClient and corresponding UseLogging(ChatClientBuilder, ILoggerFactory, Action<LoggingChatClient>) method provide a simple way to write log entries to an ILogger for every request and response.

Provide options

Every call to GetResponseAsync or GetStreamingResponseAsync can optionally supply a ChatOptions instance containing additional parameters for the operation. The most common parameters among AI models and services show up as strongly typed properties on the type, such as ChatOptions.Temperature. Other parameters can be supplied by name in a weakly typed manner, via the ChatOptions.AdditionalProperties dictionary, or via an options instance that the underlying provider understands, via the ChatOptions.RawRepresentationFactory property.

You can also specify options when building an `IChatClient` with the fluent ChatClientBuilder API by chaining a call to the ConfigureOptions(ChatClientBuilder, Action<ChatOptions>) extension method. This delegating client wraps another client and invokes the supplied delegate to populate a `ChatOptions` instance for every call. For example, to ensure that the ChatOptions.ModelId property defaults to a particular model name, you can use code like the following:

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;

IChatClient client = new OllamaApiClient(new Uri("http://localhost:11434"));

client = ChatClientBuilderChatClientExtensions.AsBuilder(client)
    .ConfigureOptions(options => options.ModelId ??= "phi3")
    .Build();

// Will request "phi3".
Console.WriteLine(await client.GetResponseAsync("What is AI?"));
// Will request "llama3.1".
Console.WriteLine(await client.GetResponseAsync("What is AI?", new() { ModelId =
"llama3.1" }));
```

Functionality pipelines

`IChatClient` instances can be layered to create a pipeline of components that each add additional functionality. These components can come from `Microsoft.Extensions.AI`, other NuGet packages, or custom implementations. This approach allows you to augment the behavior of the `IChatClient` in various ways to meet your specific needs. Consider the following code snippet that layers a distributed cache, function invocation, and OpenTelemetry tracing around a sample chat client:

C#

```
// Explore changing the order of the intermediate "Use" calls.
IChatClient client = new ChatClientBuilder(new OllamaApiClient(new
```

```
Uri("http://localhost:11434"), "llama3.1"))
    .UseDistributedCache(new MemoryDistributedCache(Options.Create(new
MemoryDistributedCacheOptions())))
    .UseFunctionInvocation()
    .UseOpenTelemetry(sourceName: sourceName, configure: c =>
c.EnableSensitiveData = true)
    .Build();
```

Custom `IChatClient` middleware

To add additional functionality, you can implement `IChatClient` directly or use the DelegatingChatClient class. This class serves as a base for creating chat clients that delegate operations to another `IChatClient` instance. It simplifies chaining multiple clients, allowing calls to pass through to an underlying client.

The `DelegatingChatClient` class provides default implementations for methods like `GetResponseAsync`, `GetStreamingResponseAsync`, and `Dispose`, which forward calls to the inner client. A derived class can then override only the methods it needs to augment the behavior, while delegating other calls to the base implementation. This approach is useful for creating flexible and modular chat clients that are easy to extend and compose.

The following is an example class derived from `DelegatingChatClient` that uses the System.Threading.RateLimiting library to provide rate-limiting functionality.

C#

```
using Microsoft.Extensions.AI;
using System.Runtime.CompilerServices;
using System.Threading.RateLimiting;

public sealed class RateLimitingChatClient(
    IChatClient innerClient, RateLimiter rateLimiter)
        : DelegatingChatClient(innerClient)
{
    public override async Task<ChatResponse> GetResponseAsync(
        IEnumerable<ChatMessage> messages,
        ChatOptions? options = null,
        CancellationToken cancellationToken = default)
    {
        using var lease = await rateLimiter.AcquireAsync(permitCount: 1,
cancellationToken)
            .ConfigureAwait(false);
        if (!lease.IsAcquired)
            throw new InvalidOperationException("Unable to acquire lease.");

        return await base.GetResponseAsync(messages, options, cancellationToken)
            .ConfigureAwait(false);
    }
```

```
    public override async IAsyncEnumerable<ChatResponseUpdate>
GetStreamingResponseAsync(
        IEnumerable<ChatMessage> messages,
        ChatOptions? options = null,
        [EnumeratorCancellation] CancellationToken cancellationToken = default)
    {
        using var lease = await rateLimiter.AcquireAsync(permitCount: 1,
cancellationToken)
            .ConfigureAwait(false);
        if (!lease.IsAcquired)
            throw new InvalidOperationException("Unable to acquire lease.");

        await foreach (var update in base.GetStreamingResponseAsync(messages,
options, cancellationToken)
            .ConfigureAwait(false))
        {
            yield return update;
        }
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
            rateLimiter.Dispose();

        base.Dispose(disposing);
    }
}
```

As with other `IChatClient` implementations, the `RateLimitingChatClient` can be composed:

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;
using System.Threading.RateLimiting;

var client = new RateLimitingChatClient(
    new OllamaApiClient(new Uri("http://localhost:11434"), "llama3.1"),
    new ConcurrencyLimiter(new() { PermitLimit = 1, QueueLimit = int.MaxValue }));

Console.WriteLine(await client.GetResponseAsync("What color is the sky?"));
```

To simplify the composition of such components with others, component authors should create a `Use*` extension method for registering the component into a pipeline. For example, consider the following `UseRatingLimiting` extension method:

C#

```
using Microsoft.Extensions.AI;
using System.Threading.RateLimiting;

public static class RateLimitingChatClientExtensions
{
    public static ChatClientBuilder UseRateLimiting(
        this ChatClientBuilder builder,
        RateLimiter rateLimiter) =>
        builder.Use(innerClient =>
            new RateLimitingChatClient(innerClient, rateLimiter)
        );
}
```

Such extensions can also query for relevant services from the DI container; the IServiceProvider used by the pipeline is passed in as an optional parameter:

C#

```
using Microsoft.Extensions.AI;
using Microsoft.Extensions.DependencyInjection;
using System.Threading.RateLimiting;

public static class RateLimitingChatClientExtensions
{
    public static ChatClientBuilder UseRateLimiting(
        this ChatClientBuilder builder,
        RateLimiter? rateLimiter = null) =>
        builder.Use((innerClient, services) =>
            new RateLimitingChatClient(
                innerClient,
                services.GetRequiredService<RateLimiter>())
        );
}
```

Now it's easy for the consumer to use this in their pipeline, for example:

C#

```
HostApplicationBuilder builder = Host.CreateApplicationBuilder(args);

IChatClient client = new OllamaApiClient(
    new Uri("http://localhost:11434/"),
    "phi3:mini");

builder.Services.AddChatClient(services =>
        client
        .AsBuilder()
        .UseDistributedCache()
        .UseRateLimiting()
```

```
        .UseOpenTelemetry()
        .Build(services));
```

The previous extension methods demonstrate using a `Use` method on ChatClientBuilder. `ChatClientBuilder` also provides Use overloads that make it easier to write such delegating handlers. For example, in the earlier `RateLimitingChatClient` example, the overrides of `GetResponseAsync` and `GetStreamingResponseAsync` only need to do work before and after delegating to the next client in the pipeline. To achieve the same thing without writing a custom class, you can use an overload of `Use` that accepts a delegate that's used for both `GetResponseAsync` and `GetStreamingResponseAsync`, reducing the boilerplate required:

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;
using System.Threading.RateLimiting;

RateLimiter rateLimiter = new ConcurrencyLimiter(new()
{
    PermitLimit = 1,
    QueueLimit = int.MaxValue
});

IChatClient client = new OllamaApiClient(new Uri("http://localhost:11434"),
"llama3.1");

client = ChatClientBuilderChatClientExtensions
    .AsBuilder(client)
    .UseDistributedCache()
    .Use(async (messages, options, nextAsync, cancellationToken) =>
    {
        using var lease = await rateLimiter.AcquireAsync(permitCount: 1,
cancellationToken).ConfigureAwait(false);
        if (!lease.IsAcquired)
            throw new InvalidOperationException("Unable to acquire lease.");

        await nextAsync(messages, options, cancellationToken);
    })
    .UseOpenTelemetry()
    .Build();
```

For scenarios where you need a different implementation for `GetResponseAsync` and `GetStreamingResponseAsync` in order to handle their unique return types, you can use the Use(Func<IEnumerable<ChatMessage>,ChatOptions,IChatClient,CancellationToken, Task<ChatResponse>>, Func<IEnumerable<ChatMessage>,ChatOptions, IChatClient,CancellationToken,IAsyncEnumerable<ChatResponseUpdate>>) overload that accepts a delegate for each.

Dependency injection

IChatClient implementations are often provided to an application via dependency injection (DI). In this example, an IDistributedCache is added into the DI container, as is an `IChatClient`. The registration for the `IChatClient` uses a builder that creates a pipeline containing a caching client (which then uses an `IDistributedCache` retrieved from DI) and the sample client. The injected `IChatClient` can be retrieved and used elsewhere in the app.

C#

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

// App setup.
var builder = Host.CreateApplicationBuilder();
builder.Services.AddDistributedMemoryCache();
builder.Services.AddChatClient(new OllamaApiClient(new
Uri("http://localhost:11434"), "llama3.1"))
    .UseDistributedCache();
var host = builder.Build();

// Elsewhere in the app.
var chatClient = host.Services.GetRequiredService<IChatClient>();
Console.WriteLine(await chatClient.GetResponseAsync("What is AI?"));
```

What instance and configuration is injected can differ based on the current needs of the application, and multiple pipelines can be injected with different keys.

Stateless vs. stateful clients

Stateless services require all relevant conversation history to be sent back on every request. In contrast, *stateful* services keep track of the history and require only additional messages to be sent with a request. The IChatClient interface is designed to handle both stateless and stateful AI services.

When working with a stateless service, callers maintain a list of all messages. They add in all received response messages and provide the list back on subsequent interactions.

C#

```
List<ChatMessage> history = [];
while (true)
{
    Console.Write("Q: ");
    history.Add(new(ChatRole.User, Console.ReadLine()));
```

```
    var response = await client.GetResponseAsync(history);
    Console.WriteLine(response);

    history.AddMessages(response);
}
```

For stateful services, you might already know the identifier used for the relevant conversation. You can put that identifier into ChatOptions.ConversationId. Usage then follows the same pattern, except there's no need to maintain a history manually.

C#

```
ChatOptions statefulOptions = new() { ConversationId = "my-conversation-id" };
while (true)
{
    Console.Write("Q: ");
    ChatMessage message = new(ChatRole.User, Console.ReadLine());

    Console.WriteLine(await client.GetResponseAsync(message, statefulOptions));
}
```

Some services might support automatically creating a conversation ID for a request that doesn't have one, or creating a new conversation ID that represents the current state of the conversation after incorporating the last round of messages. In such cases, you can transfer the ChatResponse.ConversationId over to the `ChatOptions.ConversationId` for subsequent requests. For example:

C#

```
ChatOptions options = new();
while (true)
{
    Console.Write("Q: ");
    ChatMessage message = new(ChatRole.User, Console.ReadLine());

    ChatResponse response = await client.GetResponseAsync(message, options);
    Console.WriteLine(response);

    options.ConversationId = response.ConversationId;
}
```

If you don't know ahead of time whether the service is stateless or stateful, you can check the response ConversationId and act based on its value. If it's set, then that value is propagated to the options and the history is cleared so as to not resend the same history again. If the response `ConversationId` isn't set, then the response message is added to the history so that it's sent back to the service on the next turn.

C#

```
List<ChatMessage> chatHistory = [];
ChatOptions chatOptions = new();
while (true)
{
    Console.Write("Q: ");
    chatHistory.Add(new(ChatRole.User, Console.ReadLine()));

    ChatResponse response = await client.GetResponseAsync(chatHistory);
    Console.WriteLine(response);

    chatOptions.ConversationId = response.ConversationId;
    if (response.ConversationId is not null)
    {
        chatHistory.Clear();
    }
    else
    {
        chatHistory.AddMessages(response);
    }
}
```

The `IEmbeddingGenerator` interface

The IEmbeddingGenerator<TInput,TEmbedding> interface represents a generic generator of embeddings. For the generic type parameters, `TInput` is the type of input values being embedded, and `TEmbedding` is the type of generated embedding, which inherits from the Embedding class.

The `Embedding` class serves as a base class for embeddings generated by an `IEmbeddingGenerator`. It's designed to store and manage the metadata and data associated with embeddings. Derived types, like Embedding<T>, provide the concrete embedding vector data. For example, an `Embedding<float>` exposes a `ReadOnlyMemory<float> Vector { get; }` property for access to its embedding data.

The `IEmbeddingGenerator` interface defines a method to asynchronously generate embeddings for a collection of input values, with optional configuration and cancellation support. It also provides metadata describing the generator and allows for the retrieval of strongly typed services that can be provided by the generator or its underlying services.

Most users don't need to implement the `IEmbeddingGenerator` interface. However, if you're a library author, you can see a simple implementation at Sample implementations of IChatClient and IEmbeddingGenerator.

Create embeddings

The primary operation performed with an IEmbeddingGenerator<TInput,TEmbedding> is embedding generation, which is accomplished with its GenerateAsync method.

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;

IEmbeddingGenerator<string, Embedding<float>> generator =
    new OllamaApiClient(new Uri("http://localhost:11434/"), "phi3:mini");

foreach (Embedding<float> embedding in
    await generator.GenerateAsync(["What is AI?", "What is .NET?"]))
{
    Console.WriteLine(string.Join(", ", embedding.Vector.ToArray()));
}
```

Accelerator extension methods also exist to simplify common cases, such as generating an embedding vector from a single input.

C#

```
ReadOnlyMemory<float> vector = await generator.GenerateVectorAsync("What is AI?");
```

Pipelines of functionality

As with `IChatClient`, `IEmbeddingGenerator` implementations can be layered. `Microsoft.Extensions.AI` provides a delegating implementation for `IEmbeddingGenerator` for caching and telemetry.

C#

```
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Options;
using OllamaSharp;
using OpenTelemetry.Trace;

// Configure OpenTelemetry exporter
string sourceName = Guid.NewGuid().ToString();
TracerProvider tracerProvider = OpenTelemetry.Sdk.CreateTracerProviderBuilder()
    .AddSource(sourceName)
    .AddConsoleExporter()
    .Build();
```

```
// Explore changing the order of the intermediate "Use" calls to see
// what impact that has on what gets cached and traced.
IEmbeddingGenerator<string, Embedding<float>> generator = new
EmbeddingGeneratorBuilder<string, Embedding<float>>(
        new OllamaApiClient(new Uri("http://localhost:11434/"), "phi3:mini"))
    .UseDistributedCache(
        new MemoryDistributedCache(
            Options.Create(new MemoryDistributedCacheOptions())))
    .UseOpenTelemetry(sourceName: sourceName)
    .Build();

GeneratedEmbeddings<Embedding<float>> embeddings = await generator.GenerateAsync(
[
    "What is AI?",
    "What is .NET?",
    "What is AI?"
]);

foreach (Embedding<float> embedding in embeddings)
{
    Console.WriteLine(string.Join(", ", embedding.Vector.ToArray()));
}
```

The `IEmbeddingGenerator` enables building custom middleware that extends the functionality of an `IEmbeddingGenerator`. The DelegatingEmbeddingGenerator<TInput,TEmbedding> class is an implementation of the `IEmbeddingGenerator<TInput, TEmbedding>` interface that serves as a base class for creating embedding generators that delegate their operations to another `IEmbeddingGenerator<TInput, TEmbedding>` instance. It allows for chaining multiple generators in any order, passing calls through to an underlying generator. The class provides default implementations for methods such as GenerateAsync and `Dispose`, which forward the calls to the inner generator instance, enabling flexible and modular embedding generation.

The following is an example implementation of such a delegating embedding generator that rate-limits embedding generation requests:

C#

```
using Microsoft.Extensions.AI;
using System.Threading.RateLimiting;

public class RateLimitingEmbeddingGenerator(
    IEmbeddingGenerator<string, Embedding<float>> innerGenerator, RateLimiter
rateLimiter)
        : DelegatingEmbeddingGenerator<string, Embedding<float>>(innerGenerator)
{
    public override async Task<GeneratedEmbeddings<Embedding<float>>>
GenerateAsync(
        IEnumerable<string> values,
        EmbeddingGenerationOptions? options = null,
        CancellationToken cancellationToken = default)
```

```
    {
        using var lease = await rateLimiter.AcquireAsync(permitCount: 1,
cancellationToken)
            .ConfigureAwait(false);

        if (!lease.IsAcquired)
        {
            throw new InvalidOperationException("Unable to acquire lease.");
        }

        return await base.GenerateAsync(values, options, cancellationToken);
    }

    protected override void Dispose(bool disposing)
    {
        if (disposing)
        {
            rateLimiter.Dispose();
        }

        base.Dispose(disposing);
    }
}
```

This can then be layered around an arbitrary `IEmbeddingGenerator<string, Embedding<float>>` to rate limit all embedding generation operations.

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;
using System.Threading.RateLimiting;

IEmbeddingGenerator<string, Embedding<float>> generator =
    new RateLimitingEmbeddingGenerator(
        new OllamaApiClient(new Uri("http://localhost:11434/"), "phi3:mini"),
        new ConcurrencyLimiter(new()
        {
            PermitLimit = 1,
            QueueLimit = int.MaxValue
        }));

foreach (Embedding<float> embedding in
    await generator.GenerateAsync(["What is AI?", "What is .NET?"]))
{
    Console.WriteLine(string.Join(", ", embedding.Vector.ToArray()));
}
```

In this way, the `RateLimitingEmbeddingGenerator` can be composed with other `IEmbeddingGenerator<string, Embedding<float>>` instances to provide rate-limiting functionality.

Build with Microsoft.Extensions.AI

You can start building with `Microsoft.Extensions.AI` in the following ways:

- **Library developers**: If you own libraries that provide clients for AI services, consider implementing the interfaces in your libraries. This allows users to easily integrate your NuGet package via the abstractions. For example implementations, see Sample implementations of IChatClient and IEmbeddingGenerator.
- **Service consumers**: If you're developing libraries that consume AI services, use the abstractions instead of hardcoding to a specific AI service. This approach gives your consumers the flexibility to choose their preferred provider.
- **Application developers**: Use the abstractions to simplify integration into your apps. This enables portability across models and services, facilitates testing and mocking, leverages middleware provided by the ecosystem, and maintains a consistent API throughout your app, even if you use different services in different parts of your application.
- **Ecosystem contributors**: If you're interested in contributing to the ecosystem, consider writing custom middleware components.

For more samples, see the dotnet/ai-samples GitHub repository. For an end-to-end sample, see eShopSupport.

See also

- Request a response with structured output
- Build an AI chat app with .NET
- Dependency injection in .NET
- Caching in .NET
- Rate limit an HTTP handler in .NET

Semantic Kernel overview for .NET

Article • 04/09/2025

In this article, you explore Semantic Kernel core concepts and capabilities. Semantic Kernel is a powerful and recommended choice for working with AI in .NET applications. In the sections ahead, you learn:

- How to add semantic kernel to your project
- Semantic Kernel core concepts

This article serves as an introductory overview of Semantic Kernel specifically in the context of .NET. For more comprehensive information and training about Semantic Kernel, see the following resources:

- Semantic Kernel documentation
- Semantic Kernel training

Add Semantic Kernel to a .NET project

The Semantic Kernel SDK is available as a NuGet package for .NET and integrates with standard app configurations.

Install the Microsoft.SemanticKernel package using the following command:

.NET CLI

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

Or, in .NET 10+:

.NET CLI

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

> **Note**
>
> Although `Microsoft.SemanticKernel` provides core features of Semantic Kernel, additional capabilities require you to install additional packages. For example, the Microsoft.SemanticKernel.Plugins.Memory package provides to access memory related features. For more information, see the Semantic Kernel documentation.

Create and configure a `Kernel` instance using the `KernelBuilder` class to access and work with Semantic Kernel. The `Kernel` holds services, data, and connections to orchestrate integrations between your code and AI models.

Configure the `Kernel` in a .NET console app:

C#

```
var builder = Kernel.CreateBuilder();

// Add builder configuration and services

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

Configure the Kernel in an ASP.NET Core app:

C#

```
var builder = WebApplication.CreateBuilder();
builder.Services.AddKernel();

// Add builder configuration and services

var app = builder.Build();
```

Understand Semantic Kernel

Semantic Kernel is an open-source SDK that integrates and orchestrates AI models and services like OpenAI, Azure OpenAI, and Hugging Face with conventional programming languages like C#, Python, and Java.

The Semantic Kernel SDK benefits enterprise developers in the following ways:

- Streamlines integration of AI capabilities into existing applications to enable a cohesive solution for enterprise products.
- Minimizes the learning curve of working with different AI models or services by providing abstractions that reduce complexity.
- Improves reliability by reducing the unpredictable behavior of prompts and responses from AI models. You can fine-tune prompts and plan tasks to create a controlled and predictable user experience.

Semantic Kernel is built around several core concepts:

- **Connections**: Interface with external AI services and data sources.
- **Plugins**: Encapsulate functions that applications can use.

- **Planner**: Orchestrates execution plans and strategies based on user behavior.
- **Memory**: Abstracts and simplifies context management for AI apps.

These building blocks are explored in more detail in the following sections.

Connections

The Semantic Kernel SDK includes a set of connectors that enable developers to integrate LLMs and other services into their existing applications. These connectors serve as the bridge between the application code and the AI models or services. Semantic Kernel handles many common connection concerns and challenges for you so you can focus on building your own workflows and features.

The following code snippet creates a `Kernel` and adds a connection to an Azure OpenAI model:

C#

```
using Microsoft.SemanticKernel;

// Create kernel
var builder = Kernel.CreateBuilder();

// Add a chat completion service:
builder.Services.AddAzureOpenAIChatCompletion(
    "your-resource-name",
    "your-endpoint",
    "your-resource-key",
    "deployment-model");
var kernel = builder.Build();
```

Plugins

Semantic Kernel plugins encapsulate standard language functions for applications and AI models to consume. You can create your own plugins or rely on plugins provided by the SDK. These plugins streamline tasks where AI models are advantageous and efficiently combine them with more traditional C# methods. Plugin functions are generally categorized into two types: *semantic functions* and *native functions*.

Semantic functions

Semantic functions are essentially AI prompts defined in your code that Semantic Kernel can customize and call as needed. You can templatize these prompts to use variables, custom prompt and completion formatting, and more.

The following code snippet defines and registers a semantic function:

C#

```
var userInput = Console.ReadLine();

// Define semantic function inline.
string skPrompt = @"Summarize the provided unstructured text in a sentence that is
easy to understand.
                    Text to summarize: {{$userInput}}";

// Register the function
kernel.CreateSemanticFunction(
    promptTemplate: skPrompt,
    functionName: "SummarizeText",
    pluginName: "SemanticFunctions"
);
```

Native functions

Native functions are C# methods that Semantic Kernel can call directly to manipulate or retrieve data. They perform operations that are better suited for traditional code instructions instead of LLM prompts.

The following code snippet defines and registers a native function:

C#

```
// Define native function
public class NativeFunctions {

    [SKFunction, Description("Retrieve content from local file")]
    public async Task<string> RetrieveLocalFile(string fileName, int maxSize =
5000)
    {
        string content = await File.ReadAllTextAsync(fileName);
        if (content.Length <= maxSize) return content;
        return content.Substring(0, maxSize);
    }
}

//Import native function
string plugInName = "NativeFunction";
string functionName = "RetrieveLocalFile";

var nativeFunctions = new NativeFunctions();
kernel.ImportFunctions(nativeFunctions, plugInName);
```

Planner

The planner is a core component of Semantic Kernel that provides AI orchestration to manage seamless integration between AI models and plugins. This layer devises execution strategies from user requests and dynamically orchestrates Plugins to perform complex tasks with AI-assisted planning.

Consider the following pseudo-code snippet:

C#

```
// Native function definition and kernel configuration code omitted for brevity

// Configure and create the plan
string planDefinition = "Read content from a local file and summarize the
content.";
SequentialPlanner sequentialPlanner = new SequentialPlanner(kernel);

string assetsFolder = @"../../assets";
string fileName = Path.Combine(assetsFolder,"docs","06_SemanticKernel",
"aci_documentation.txt");

ContextVariables contextVariables = new ContextVariables();
contextVariables.Add("fileName", fileName);

var customPlan = await sequentialPlanner.CreatePlanAsync(planDefinition);

// Execute the plan
KernelResult kernelResult = await kernel.RunAsync(contextVariables, customPlan);
Console.WriteLine($"Summarization: {kernelResult.GetValue<string>()}");
```

The preceding code creates an executable, sequential plan to read content from a local file and summarize the content. The plan sets up instructions to read the file using a native function and then analyze it using an AI model.

Memory

Semantic Kernel's Vector stores provide abstractions over embedding models, vector databases, and other data to simplify context management for AI applications. Vector stores are agnostic to the underlying LLM or Vector database, offering a uniform developer experience. You can configure memory features to store data in a variety of sources or service, including Azure AI Search and Azure Cache for Redis.

Consider the following code snippet:

C#

```
var facts = new Dictionary<string,string>();
facts.Add(
    "Azure Machine Learning; https://learn.microsoft.com/en-us/azure/machine-
learning/",
    @"Azure Machine Learning is a cloud service for accelerating and
    managing the machine learning project lifecycle. Machine learning
professionals,
    data scientists, and engineers can use it in their day-to-day workflows"
);

facts.Add(
    "Azure SQL Service; https://learn.microsoft.com/en-us/azure/azure-sql/",
    @"Azure SQL is a family of managed, secure, and intelligent products
    that use the SQL Server database engine in the Azure cloud."
);

string memoryCollectionName = "SummarizedAzureDocs";

foreach (var fact in facts) {
    await memoryBuilder.SaveReferenceAsync(
        collection: memoryCollectionName,
        description: fact.Key.Split(";")[0].Trim(),
        text: fact.Value,
        externalId: fact.Key.Split(";")[1].Trim(),
        externalSourceName: "Azure Documentation"
    );
}
```

The preceding code loads a set of facts into memory so that the data is available to use when interacting with AI models and orchestrating tasks.

Quickstart - Summarize text with OpenAI | Quickstart - Chat with your data

Get started with .NET AI and the Model Context Protocol

Article • 05/05/2025

The Model Context Protocol (MCP) is an open protocol designed to standardize integrations between AI apps and external tools and data sources. By using MCP, developers can enhance the capabilities of AI models, enabling them to produce more accurate, relevant, and context-aware responses.

For example, using MCP, you can connect your LLM to resources such as:

- Document databases or storage services.
- Web APIs that expose business data or logic.
- Tools that manage files or performing local tasks on a user's device.

Many Microsoft products already support MCP, including:

- Copilot Studio
- Visual Studio Code GitHub Copilot agent mode
- Semantic Kernel.

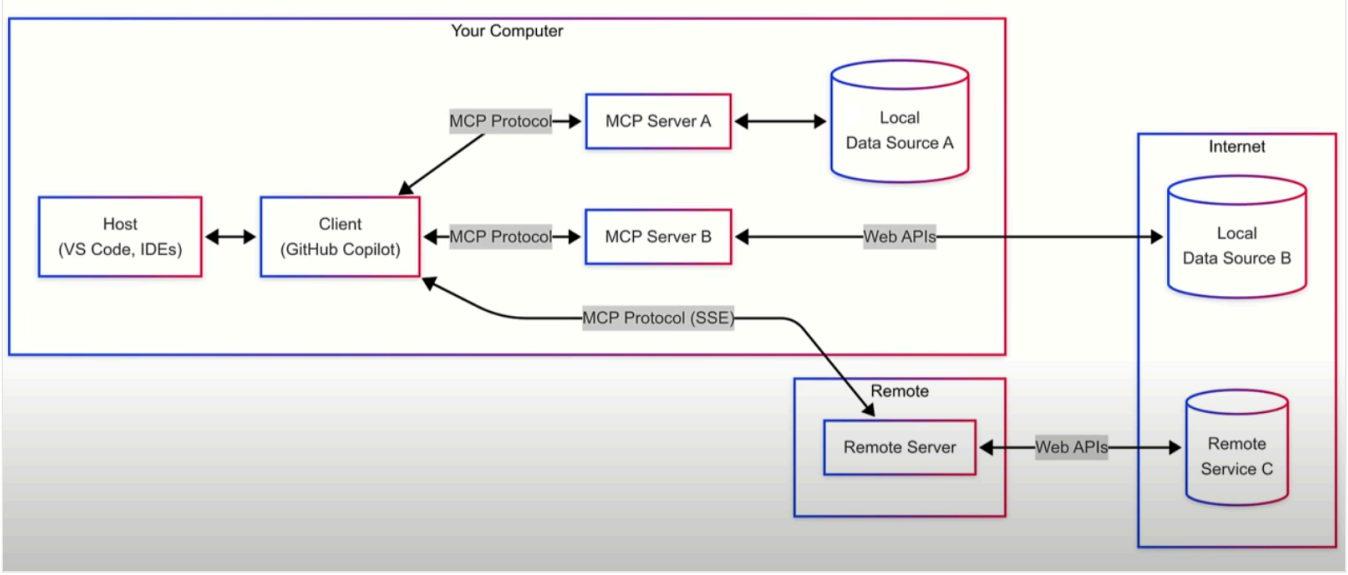
You can use the MCP C# SDK to quickly create your own MCP integrations and switch between different AI models without significant code changes.

MCP client-server architecture

MCP uses a client-server architecture that enables an AI-powered app (the host) to connect to multiple MCP servers through MCP clients:

- **MCP Hosts**: AI tools, code editors, or other software that enhance their AI models using contextual resources through MCP. For example, GitHub Copilot in Visual Studio Code can act as an MCP host and use MCP clients and servers to expand its capabilities.
- **MCP Clients**: Clients used by the host application to connect to MCP servers to retrieve contextual data.
- **MCP Servers**: Services that expose capabilities to clients through MCP. For example, an MCP server might provide an abstraction over a REST API or local data source to provide business data to the AI model.

The following diagram illustrates this architecture:



MCP client and server can exchange a set of standard messages:

Expand table

Message	Description
`InitializeRequest`	This request is sent by the client to the server when it first connects, asking it to begin initialization.
`ListToolsRequest`	Sent by the client to request a list of tools the server has.
`CallToolRequest`	Used by the client to invoke a tool provided by the server.
`ListResourcesRequest`	Sent by the client to request a list of available server resources.
`ReadResourceRequest`	Sent by the client to the server to read a specific resource URI.
`ListPromptsRequest`	Sent by the client to request a list of available prompts and prompt templates from the server.
`GetPromptRequest`	Used by the client to get a prompt provided by the server.
`PingRequest`	A ping, issued by either the server or the client, to check that the other party is still alive.
`CreateMessageRequest`	A request by the server to sample an LLM via the client. The client has full discretion over which model to select. The client should also inform the user before beginning sampling, to allow them to inspect the request (human in the loop) and decide whether to approve it.
`SetLevelRequest`	A request by the client to the server, to enable or adjust logging.

Develop with the MCP C# SDK

As a .NET developer, you can use MCP by creating MCP clients and servers to enhance your apps with custom integrations. MCP reduces the complexity involved in connecting an AI model to various tools, services, and data sources.

The official MCP C# SDK is available through NuGet and enables you to build MCP clients and servers for .NET apps and libraries. The SDK is maintained through collaboration between Microsoft, Anthropic, and the MCP open protocol organization.

To get started, add the MCP C# SDK to your project:

.NET CLI

```
dotnet add package ModelContextProtocol --prerelease
```

Instead of building unique connectors for each integration point, you can often leverage or reference prebuilt integrations from various providers such as GitHub and Docker:

- Available MPC clients
- Available MCP servers

Integration with Microsoft.Extensions.AI

The MCP C# SDK depends on the Microsoft.Extensions.AI libraries to handle various AI interactions and tasks. These extension libraries provides core types and abstractions for working with AI services, so developers can focus on coding against conceptual AI capabilities rather than specific platforms or provider implementations.

View the MCP C# SDK dependencies on the NuGet package page.

More .NET MCP development resources

Various tools, services, and learning resources are available in the .NET and Azure ecosystems to help you build MCP clients and servers or integrate with existing MCP servers.

Get started with the following development tools:

- Semantic Kernel allows you to add plugins for MCP servers. Semantic Kernel supports both local MCP servers through standard I/O and remote servers that connect through SSE over HTTPS.
- Azure Functions remote MCP servers combine MCP standards with the flexible architecture of Azure Functions. Visit the Remote MCP functions sample repository for code examples.

- Azure MCP Server implements the MCP specification to seamlessly connect AI agents with key Azure services like Azure Storage, Cosmos DB, and more.

Learn more about .NET and MCP using these resources:

- Microsoft partners with Anthropic to create official C# SDK for Model Context Protocol
- Build a Model Context Protocol (MCP) server in C#
- MCP C# SDK README

Related content

- Overview of the .NET + AI ecosystem
- Microsoft.Extensions.AI
- Semantic Kernel overview for .NET

Build an AI chat app with .NET

Article • 05/17/2025

In this quickstart, you learn how to create a conversational .NET console chat app using an OpenAI or Azure OpenAI model. The app uses the Microsoft.Extensions.AI library so you can write code using AI abstractions rather than a specific SDK. AI abstractions enable you to change the underlying AI model with minimal code changes.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8.0 SDK.
- An API key from OpenAI so you can run this sample.

> **Note**
>
> You can also use **Semantic Kernel** to accomplish the tasks in this article. Semantic Kernel is a lightweight, open-source SDK that lets you build AI agents and integrate the latest AI models into your .NET apps.

Create the app

Complete the following steps to create a .NET console app to connect to an AI model.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

   .NET CLI

   ```
   dotnet new console -o ChatAppAI
   ```

2. Change directory into the app folder:

   .NET CLI

   ```
   cd ChatAppAI
   ```

3. Install the required packages:

   Bash

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

4. Open the app in Visual Studio Code (or your editor of choice).

```Bash
code .
```

Configure the app

1. Navigate to the root of your .NET project from a terminal or command prompt.
2. Run the following commands to configure your OpenAI API key as a secret for the sample app:

```Bash
dotnet user-secrets init
dotnet user-secrets set OpenAIKey <your-OpenAI-key>
dotnet user-secrets set ModelName <your-OpenAI-model-name>
```

Add the app code

This app uses the Microsoft.Extensions.AI package to send and receive requests to the AI model. The app provides users with information about hiking trails.

1. In the `Program.cs` file, add the following code to connect and authenticate to the AI model.

```C#
var config = new ConfigurationBuilder().AddUserSecrets<Program>().Build();
string model = config["ModelName"];
string key = config["OpenAIKey"];

// Create the IChatClient
IChatClient chatClient =
    new OpenAIClient(key).GetChatClient(model).AsIChatClient();
```

2. Create a system prompt to provide the AI model with initial role context and instructions about hiking recommendations:

```
C#
```

```
// Start the conversation with context for the AI model
List<ChatMessage> chatHistory =
    [
        new ChatMessage(ChatRole.System, """
            You are a friendly hiking enthusiast who helps people discover
fun hikes in their area.
            You introduce yourself when first saying hello.
            When helping people out, you always ask them for this information
            to inform the hiking recommendation you provide:

            1. The location where they would like to hike
            2. What hiking intensity they are looking for

            You will then provide three suggestions for nearby hikes that
vary in length
            after you get that information. You will also share an
interesting fact about
            the local nature on the hikes when making a recommendation. At
the end of your
            response, ask if there is anything else you can help with.
        """)
    ];
```

3. Create a conversational loop that accepts an input prompt from the user, sends the prompt to the model, and prints the response completion:

```
C#
```

```
// Loop to get user input and stream AI response
while (true)
{
    // Get user prompt and add to chat history
    Console.WriteLine("Your prompt:");
    string? userPrompt = Console.ReadLine();
    chatHistory.Add(new ChatMessage(ChatRole.User, userPrompt));

    // Stream the AI response and add to chat history
    Console.WriteLine("AI Response:");
    string response = "";
    await foreach (ChatResponseUpdate item in
        chatClient.GetStreamingResponseAsync(chatHistory))
    {
        Console.Write(item.Text);
        response += item.Text;
    }
    chatHistory.Add(new ChatMessage(ChatRole.Assistant, response));
```

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

4. Use the `dotnet run` command to run the app:

   .NET CLI

   ```
   dotnet run
   ```

   The app prints out the completion response from the AI model. Send additional follow up prompts and ask other questions to experiment with the AI chat functionality.

Next steps

- Quickstart - Chat with a local AI model
- Generate images using AI with .NET

Request a response with structured output

Article • 05/17/2025

In this quickstart, you create a chat app that requests a response with *structured output*. A structured output response is a chat response that's of a type you specify instead of just plain text. The chat app you create in this quickstart analyzes sentiment of various product reviews, categorizing each review according to the values of a custom enumeration.

Prerequisites

- .NET 8 or a later version
- Visual Studio Code (optional)

Configure the AI service

To provision an Azure OpenAI service and model using the Azure portal, complete the steps in the Create and deploy an Azure OpenAI Service resource article. In the "Deploy a model" step, select the `gpt-4o` model.

Create the chat app

Complete the following steps to create a console app that connects to the `gpt-4o` AI model.

1. In a terminal window, navigate to the directory where you want to create your app, and create a new console app with the `dotnet new` command:

   .NET CLI

   ```
   dotnet new console -o SOChat
   ```

2. Navigate to the `SOChat` directory, and add the necessary packages to your app:

   .NET CLI

   ```
   dotnet add package Azure.AI.OpenAI
   dotnet add package Azure.Identity
   dotnet add package Microsoft.Extensions.AI
   dotnet add package Microsoft.Extensions.AI.OpenAI --prerelease
   dotnet add package Microsoft.Extensions.Configuration
   dotnet add package Microsoft.Extensions.Configuration.UserSecrets
   ```

3. Run the following commands to add app secrets for your Azure OpenAI endpoint, model name, and tenant ID:

```Bash
dotnet user-secrets init
dotnet user-secrets set AZURE_OPENAI_ENDPOINT <your-Azure-OpenAI-endpoint>
dotnet user-secrets set AZURE_OPENAI_GPT_NAME gpt-4o
dotnet user-secrets set AZURE_TENANT_ID <your-tenant-ID>
```

> **Note**
>
> Depending on your environment, the tenant ID might not be needed. In that case, remove it from the code that instantiates the DefaultAzureCredential.

4. Open the new app in your editor of choice.

Add the code

1. Define the enumeration that describes the different sentiments.

```C#
public enum Sentiment
{
    Positive,
    Negative,
    Neutral
}
```

2. Create the IChatClient that will communicate with the model.

```C#
IConfigurationRoot config = new ConfigurationBuilder()
    .AddUserSecrets<Program>()
    .Build();

string endpoint = config["AZURE_OPENAI_ENDPOINT"];
string model = config["AZURE_OPENAI_GPT_NAME"];
string tenantId = config["AZURE_TENANT_ID"];

// Get a chat client for the Azure OpenAI endpoint.
AzureOpenAIClient azureClient =
    new(
        new Uri(endpoint),
```

```
        new DefaultAzureCredential(new DefaultAzureCredentialOptions() {
TenantId = tenantId }));
IChatClient chatClient = azureClient
    .GetChatClient(deploymentName: model)
    .AsIChatClient();
```

> **Note**
>
> DefaultAzureCredential searches for authentication credentials from your environment or local tooling. You'll need to assign the `Azure AI Developer` role to the account you used to sign in to Visual Studio or the Azure CLI. For more information, see Authenticate to Azure AI services with .NET.

3. Send a request to the model with a single product review, and then print the analyzed sentiment to the console. You declare the requested structured output type by passing it as the type argument to the ChatClientStructuredOutputExtensions.GetResponseAsync<T>(IChatClient, String, ChatOptions, Nullable<Boolean>, CancellationToken) extension method.

```
C#

string review = "I'm happy with the product!";
var response = await chatClient.GetResponseAsync<Sentiment>($"What's the
sentiment of this review? {review}");
Console.WriteLine($"Sentiment: {response.Result}");
```

   This code produces output similar to:

```
Output

Sentiment: Positive
```

4. Instead of just analyzing a single review, you can analyze a collection of reviews.

```
C#

string[] inputs = [
    "Best purchase ever!",
    "Returned it immediately.",
    "Hello",
    "It works as advertised.",
    "The packaging was damaged but otherwise okay."
];

foreach (var i in inputs)
```

```
{
    var response2 = await chatClient.GetResponseAsync<Sentiment>($"What's the
sentiment of this review? {i}");
    Console.WriteLine($"Review: {i} | Sentiment: {response2.Result}");
}
```

This code produces output similar to:

Output

```
Review: Best purchase ever! | Sentiment: Positive
Review: Returned it immediately. | Sentiment: Negative
Review: Hello | Sentiment: Neutral
Review: It works as advertised. | Sentiment: Neutral
Review: The packaging was damaged but otherwise okay. | Sentiment: Neutral
```

5. And instead of requesting just the analyzed enumeration value, you can request the text response along with the analyzed value.

   Define a record type to contain the text response and analyzed sentiment:

   C#

   ```
   record SentimentRecord(string ResponseText, Sentiment ReviewSentiment);
   ```

   Send the request using the record type as the type argument to `GetResponseAsync<T>`:

   C#

   ```
   var review3 = "This product worked okay.";
   var response3 = await chatClient.GetResponseAsync<SentimentRecord>($"What's
   the sentiment of this review? {review3}");

   Console.WriteLine($"Response text: {response3.Result.ResponseText}");
   Console.WriteLine($"Sentiment: {response3.Result.ReviewSentiment}");
   ```

   This code produces output similar to:

   Output

   ```
   Response text: Certainly, I have analyzed the sentiment of the review you
   provided.
   Sentiment: Neutral
   ```

Clean up resources

If you no longer need them, delete the Azure OpenAI resource and GPT-4 model deployment.

1. In the Azure Portal, navigate to the Azure OpenAI resource.
2. Select the Azure OpenAI resource, and then select **Delete**.

See also

- Structured outputs (Azure OpenAI Service)
- Using JSON schema for structured output in .NET for OpenAI models
- Introducing Structured Outputs in the API (OpenAI)

Build a .NET AI vector search app

06/06/2025

In this quickstart, you create a .NET console app to perform semantic search on a *vector store* to find relevant results for the user's query. You learn how to generate embeddings for user prompts and use those embeddings to query the vector data store.

Vector stores, or vector databases, are essential for tasks like semantic search, retrieval augmented generation (RAG), and other scenarios that require grounding generative AI responses. While relational databases and document databases are optimized for structured and semi-structured data, vector databases are built to efficiently store, index, and manage data represented as embedding vectors. As a result, the indexing and search algorithms used by vector databases are optimized to efficiently retrieve data that can be used downstream in your applications.

About the libraries

The app uses the Microsoft.Extensions.AI and Microsoft.Extensions.VectorData libraries so you can write code using AI abstractions rather than a specific SDK. AI abstractions help create loosely coupled code that allows you to change the underlying AI model with minimal app changes.

Microsoft.Extensions.VectorData.Abstractions is a .NET library developed in collaboration with Semantic Kernel and the broader .NET ecosystem to provide a unified layer of abstractions for interacting with vector stores. The abstractions in `Microsoft.Extensions.VectorData.Abstractions` provide library authors and developers with the following functionality:

- Perform create-read-update-delete (CRUD) operations on vector stores.
- Use vector and text search on vector stores.

> **Note**
>
> The Microsoft.Extensions.VectorData.Abstractions library is currently in preview.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8.0 SDK.
- An API key from OpenAI so you can run this sample.

Create the app

Complete the following steps to create a .NET console app that can:

- Create and populate a vector store by generating embeddings for a data set.
- Generate an embedding for the user prompt.
- Query the vector store using the user prompt embedding.
- Display the relevant results from the vector search.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

   .NET CLI

   ```
   dotnet new console -o VectorDataAI
   ```

2. Change directory into the app folder:

   .NET CLI

   ```
   cd VectorDataAI
   ```

3. Install the required packages:

   Bash

   ```
   dotnet add package Microsoft.Extensions.AI.OpenAI --prerelease
   dotnet add package Microsoft.Extensions.VectorData.Abstractions --prerelease
   dotnet add package Microsoft.SemanticKernel.Connectors.InMemory --prerelease
   dotnet add package Microsoft.Extensions.Configuration
   dotnet add package Microsoft.Extensions.Configuration.UserSecrets
   dotnet add package System.Linq.AsyncEnumerable
   ```

   The following list describes each package in the `VectorDataAI` app:

   - Microsoft.Extensions.AI.OpenAI provides AI abstractions for OpenAI-compatible models or endpoints. This library also includes the official OpenAI library for the OpenAI service API as a dependency.
   - Microsoft.Extensions.VectorData.Abstractions enables Create-Read-Update-Delete (CRUD) and search operations on vector stores.
   - Microsoft.SemanticKernel.Connectors.InMemory provides an in-memory vector store class to hold queryable vector data records.
   - Microsoft.Extensions.Configuration provides an implementation of key-value pair—based configuration.

- Microsoft.Extensions.Configuration.UserSecrets is a user secrets configuration provider implementation for `Microsoft.Extensions.Configuration`.

4. Open the app in Visual Studio Code (or your editor of choice).

```Bash
code .
```

Configure the app

1. Navigate to the root of your .NET project from a terminal or command prompt.
2. Run the following commands to configure your OpenAI API key as a secret for the sample app:

```Bash
dotnet user-secrets init
dotnet user-secrets set OpenAIKey <your-OpenAI-key>
dotnet user-secrets set ModelName <your-OpenAI-model-name>
```

> **Note**
>
> For the model name, you need to specify a text embedding model such as `text-embedding-3-small` or `text-embedding-3-large` to generate embeddings for vector search in the sections that follow. For more information about embedding models, see Embeddings.

Add the app code

1. Add a new class named `CloudService` to your project with the following properties:

```C#
using Microsoft.Extensions.VectorData;

namespace VectorDataAI;

internal class CloudService
{
    [VectorStoreKey]
    public int Key { get; set; }
```

```
    [VectorStoreData]
    public string Name { get; set; }

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

    [VectorStoreVector(
        Dimensions: 384,
        DistanceFunction = DistanceFunction.CosineSimilarity)]
    public ReadOnlyMemory<float> Vector { get; set; }
}
```

The Microsoft.Extensions.VectorData attributes, such as VectorStoreKeyAttribute, influence how each property is handled when used in a vector store. The `Vector` property stores a generated embedding that represents the semantic meaning of the `Description` value for vector searches.

2. In the `Program.cs` file, add the following code to create a data set that describes a collection of cloud services:

C#

```
List<CloudService> cloudServices =
[
    new() {
            Key = 0,
            Name = "Azure App Service",
            Description = "Host .NET, Java, Node.js, and Python web
applications and APIs in a fully managed Azure service. You only need to
deploy your code to Azure. Azure takes care of all the infrastructure
management like high availability, load balancing, and autoscaling."
    },
    new() {
            Key = 1,
            Name = "Azure Service Bus",
            Description = "A fully managed enterprise message broker
supporting both point to point and publish-subscribe integrations. It's ideal
for building decoupled applications, queue-based load leveling, or
facilitating communication between microservices."
    },
    new() {
            Key = 2,
            Name = "Azure Blob Storage",
            Description = "Azure Blob Storage allows your applications to
store and retrieve files in the cloud. Azure Storage is highly scalable to
store massive amounts of data and data is stored redundantly to ensure high
availability."
    },
    new() {
            Key = 3,
```

```
            Name = "Microsoft Entra ID",
            Description = "Manage user identities and control access to your
apps, data, and resources."
    },
    new() {
            Key = 4,
            Name = "Azure Key Vault",
            Description = "Store and access application secrets like
connection strings and API keys in an encrypted vault with restricted access
to make sure your secrets and your application aren't compromised."
    },
    new() {
            Key = 5,
            Name = "Azure AI Search",
            Description = "Information retrieval at scale for traditional and
conversational search applications, with security and options for AI
enrichment and vectorization."
    }
];
```

3. Create and configure an `IEmbeddingGenerator` implementation to send requests to an embedding AI model:

C#

```
// Load the configuration values.
IConfigurationRoot config = new
ConfigurationBuilder().AddUserSecrets<Program>().Build();
string model = config["ModelName"];
string key = config["OpenAIKey"];

// Create the embedding generator.
IEmbeddingGenerator<string, Embedding<float>> generator =
    new OpenAIClient(new ApiKeyCredential(key))
      .GetEmbeddingClient(model: model)
      .AsIEmbeddingGenerator();
```

4. Create and populate a vector store with the cloud service data. Use the `IEmbeddingGenerator` implementation to create and assign an embedding vector for each record in the cloud service data:

C#

```
// Create and populate the vector store.
var vectorStore = new InMemoryVectorStore();
VectorStoreCollection<int, CloudService> cloudServicesStore =
    vectorStore.GetCollection<int, CloudService>("cloudServices");
await cloudServicesStore.EnsureCollectionExistsAsync();

foreach (CloudService service in cloudServices)
```

```
{
    service.Vector = await
generator.GenerateVectorAsync(service.Description);
    await cloudServicesStore.UpsertAsync(service);
}
```

The embeddings are numerical representations of the semantic meaning for each data record, which makes them compatible with vector search features.

5. Create an embedding for a search query and use it to perform a vector search on the vector store:

C#

```
// Convert a search query to a vector
// and search the vector store.
string query = "Which Azure service should I use to store my Word
documents?";
ReadOnlyMemory<float> queryEmbedding = await
generator.GenerateVectorAsync(query);

IAsyncEnumerable<VectorSearchResult<CloudService>> results =
    cloudServicesStore.SearchAsync(queryEmbedding, top: 1);

await foreach (VectorSearchResult<CloudService> result in results)
{
    Console.WriteLine($"Name: {result.Record.Name}");
    Console.WriteLine($"Description: {result.Record.Description}");
    Console.WriteLine($"Vector match score: {result.Score}");
}
```

6. Use the `dotnet run` command to run the app:

.NET CLI

```
dotnet run
```

The app prints out the top result of the vector search, which is the cloud service that's most relevant to the original query. You can modify the query to try different search scenarios.

Next steps

- Quickstart - Chat with a local AI model
- Generate images using AI with .NET

Invoke .NET functions using an AI model

Article • 05/18/2025

In this quickstart, you create a .NET console AI chat app to connect to an AI model with local function calling enabled. The app uses the Microsoft.Extensions.AI library so you can write code using AI abstractions rather than a specific SDK. AI abstractions enable you to change the underlying AI model with minimal code changes.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8.0 SDK.
- An API key from OpenAI so you can run this sample.

> **Note**
>
> You can also use **Semantic Kernel** to accomplish the tasks in this article. Semantic Kernel is a lightweight, open-source SDK that lets you build AI agents and integrate the latest AI models into your .NET apps.

Create the app

Complete the following steps to create a .NET console app to connect to an AI model.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

   .NET CLI

   ```
   dotnet new console -o FunctionCallingAI
   ```

2. Change directory into the app folder:

   .NET CLI

   ```
   cd FunctionCallingAI
   ```

3. Install the required packages:

   Bash

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

4. Open the app in Visual Studio Code or your editor of choice

```Bash
code .
```

Configure the app

1. Navigate to the root of your .NET project from a terminal or command prompt.
2. Run the following commands to configure your OpenAI API key as a secret for the sample app:

```Bash
dotnet user-secrets init
dotnet user-secrets set OpenAIKey <your-OpenAI-key>
dotnet user-secrets set ModelName <your-OpenAI-model-name>
```

Add the app code

The app uses the Microsoft.Extensions.AI package to send and receive requests to the AI model.

1. In the **Program.cs** file, add the following code to connect and authenticate to the AI model. The `ChatClient` is also configured to use function invocation, which allows the AI model to call .NET functions in your code.

```C#
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using OpenAI;

IConfigurationRoot config = new
ConfigurationBuilder().AddUserSecrets<Program>().Build();
string? model = config["ModelName"];
string? key = config["OpenAIKey"];
```

```
IChatClient client =
    new ChatClientBuilder(new OpenAIClient(key).GetChatClient(model ?? "gpt-
4o").AsIChatClient())
    .UseFunctionInvocation()
    .Build();
```

2. Create a new `ChatOptions` object that contains an inline function the AI model can call to get the current weather. The function declaration includes a delegate to run logic, and name and description parameters to describe the purpose of the function to the AI model.

C#

```
// Add a new plugin with a local .NET function
// that should be available to the AI model.
var chatOptions = new ChatOptions
{
    Tools = [AIFunctionFactory.Create((string location, string unit) =>
    {
        // Here you would call a weather API
        // to get the weather for the location.
        return "Periods of rain or drizzle, 15 C";
    },
    "get_current_weather",
    "Get the current weather in a given location")]
};
```

3. Add a system prompt to the `chatHistory` to provide context and instructions to the model. Send a user prompt with a question that requires the AI model to call the registered function to properly answer the question.

C#

```
// System prompt to provide context.
List<ChatMessage> chatHistory = [new(ChatRole.System, """
    You are a hiking enthusiast who helps people discover fun hikes in their
area. You are upbeat and friendly.
    """)];

// Weather conversation relevant to the registered function.
chatHistory.Add(new ChatMessage(ChatRole.User,
    "I live in Montreal and I'm looking for a moderate intensity hike. What's
the current weather like?"));
Console.WriteLine($"{chatHistory.Last().Role} >>> {chatHistory.Last()}");

ChatResponse response = await client.GetResponseAsync(chatHistory,
chatOptions);
Console.WriteLine($"Assistant >>> {response.Text}");
```

4. Use the `dotnet run` command to run the app:

   .NET CLI

   ```
   dotnet run
   ```

   The app prints the completion response from the AI model, which includes data provided by the .NET function. The AI model understood that the registered function was available and called it automatically to generate a proper response.

Next steps

- Quickstart - Build an AI chat app with .NET
- Generate text and conversations with .NET and Azure OpenAI Completions

Generate images using AI with .NET

Article • 05/18/2025

In this quickstart, you learn how to create a .NET console app to generate images using an OpenAI or Azure OpenAI DALLe AI model, which are specifically designed to generate images based on text prompts.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8.0 SDK.
- An API key from OpenAI so you can run this sample.

> **Note**
>
> You can also use Semantic Kernel to accomplish the tasks in this article. Semantic Kernel is a lightweight, open-source SDK that lets you build AI agents and integrate the latest AI models into your .NET apps.

Create the app

Complete the following steps to create a .NET console app to connect to an AI model.

1. In an empty directory on your computer, use the `dotnet new` command to create a new console app:

   .NET CLI

   ```
   dotnet new console -o ImagesAI
   ```

2. Change directory into the app folder:

   .NET CLI

   ```
   cd ImagesAI
   ```

3. Install the required packages:

   Bash

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

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

4. Open the app in Visual Studio Code or your editor of choice.

```Bash
code .
```

Configure the app

1. Navigate to the root of your .NET project from a terminal or command prompt.
2. Run the following commands to configure your OpenAI API key as a secret for the sample app:

```Bash
dotnet user-secrets init
dotnet user-secrets set OpenAIKey <your-OpenAI-key>
dotnet user-secrets set ModelName <your-OpenAI-model-name>
```

Add the app code

1. In the `Program.cs` file, add the following code to connect and authenticate to the AI model.

```C#
// Licensed to the .NET Foundation under one or more agreements.
// The .NET Foundation licenses this file to you under the MIT license.
// See the LICENSE file in the project root for more information.
using Microsoft.Extensions.Configuration;
using OpenAI.Images;
// Retrieve the local secrets that were set from the command line, using:
// dotnet user-secrets init
// dotnet user-secrets set OpenAIKey <your-openai-key>
var config = new ConfigurationBuilder().AddUserSecrets<Program>().Build();
string key = config["OpenAIKey"];
string modelName = config["ModelName"];

// Create the OpenAI ImageClient
ImageClient client = new(modelName, key);

// Generate the image
GeneratedImage generatedImage = await client.GenerateImageAsync("""
    A postal card with a happy hiker waving and a beautiful mountain in the
```

```
background.
    There is a trail visible in the foreground.
    The postal card has text in red saying: 'You are invited for a hike!'
    """,
    new ImageGenerationOptions
    {
        Size = GeneratedImageSize.W1024xH1024
    });

Console.WriteLine($"The generated image is ready
at:\n{generatedImage.ImageUri}");
```

The preceding code:

- Reads essential configuration values from the project user secrets to connect to the AI model.
- Creates an `OpenAI.Images.ImageClient` to connect to the AI model.
- Sends a prompt to the model that describes the desired image.
- Prints the URL of the generated image to the console output.

2. Run the app:

.NET CLI

```
dotnet run
```

Navigate to the image URL in the console output to view the generated image. Customize the text content of the prompt to create new images or modify the original.

Next steps

- Quickstart - Build an AI chat app with .NET
- Generate text and conversations with .NET and Azure OpenAI Completions

Chat with a local AI model using .NET

05/29/2025

In this quickstart, you learn how to create a conversational .NET console chat app using an OpenAI or Azure OpenAI model. The app uses the Microsoft.Extensions.AI library so you can write code using AI abstractions rather than a specific SDK. AI abstractions enable you to change the underlying AI model with minimal code changes.

Prerequisites

- Install .NET 8.0 or higher
- Install Ollama locally on your device
- Visual Studio Code (optional)

Run the local AI model

Complete the following steps to configure and run a local AI model on your device. Many different AI models are available to run locally and are trained for different tasks, such as generating code, analyzing images, generative chat, or creating embeddings. For this quickstart, you'll use the general purpose `phi3:mini` model, which is a small but capable generative AI created by Microsoft.

1. Open a terminal window and verify that Ollama is available on your device:

   ```
   Bash
   ollama
   ```

   If Ollama is available, it displays a list of available commands.

2. Start Ollama:

   ```
   Bash
   ollama serve
   ```

   If Ollama is running, it displays a list of available commands.

3. Pull the `phi3:mini` model from the Ollama registry and wait for it to download:

   ```
   Bash
   ```

```
ollama pull phi3:mini
```

4. After the download completes, run the model:

```Bash
ollama run phi3:mini
```

   Ollama starts the `phi3:mini` model and provides a prompt for you to interact with it.

Create the .NET app

Complete the following steps to create a .NET console app that connects to your local `phi3:mini` AI model.

1. In a terminal window, navigate to an empty directory on your device and create a new app with the `dotnet new` command:

```.NET CLI
dotnet new console -o LocalAI
```

2. Add the OllamaSharp package to your app:

```.NET CLI
dotnet add package OllamaSharp
```

3. Open the new app in your editor of choice, such as Visual Studio Code.

```.NET CLI
code .
```

Connect to and chat with the AI model

The Semantic Kernel SDK provides many services and features to connect to AI models and manage interactions. In the steps ahead, you'll create a simple app that connects to the local AI and stores conversation history to improve the chat experience.

1. Open the *Program.cs* file and replace the contents of the file with the following code:

C#

```
using Microsoft.Extensions.AI;
using OllamaSharp;

IChatClient chatClient =
    new OllamaApiClient(new Uri("http://localhost:11434/"), "phi3:mini");

// Start the conversation with context for the AI model
List<ChatMessage> chatHistory = new();

while (true)
{
    // Get user prompt and add to chat history
    Console.WriteLine("Your prompt:");
    var userPrompt = Console.ReadLine();
    chatHistory.Add(new ChatMessage(ChatRole.User, userPrompt));

    // Stream the AI response and add to chat history
    Console.WriteLine("AI Response:");
    var response = "";
    await foreach (ChatResponseUpdate item in
        chatClient.GetStreamingResponseAsync(chatHistory))
    {
        Console.Write(item.Text);
        response += item.Text;
    }
    chatHistory.Add(new ChatMessage(ChatRole.Assistant, response));
    Console.WriteLine();
}
```

The preceding code accomplishes the following:

- Creates an `OllamaChatClient` that implements the `IChatClient` interface.
  - This interface provides a loosely coupled abstraction you can use to chat with AI Models.
  - You can later change the underlying chat client implementation to another model, such as Azure OpenAI, without changing any other code.
- Creates a `ChatHistory` object to store the messages between the user and the AI model.
- Retrieves a prompt from the user and stores it in the `ChatHistory`.
- Sends the chat data to the AI model to generate a response.

> **Note**
>
> Ollama runs on port 11434 by default, which is why the AI model endpoint is set to `http://localhost:11434`.

2. Run the app and enter a prompt into the console to receive a response from the AI, such as the following:

Output

```
Your prompt:
Tell me three facts about .NET.

AI response:
1. **Cross-Platform Development:** One of the significant strengths of .NET,
particularly its newer iterations (.NET Core and .NET 5+), is cross-platform
support.
It allows developers to build applications that run on Windows, Linux, macOS,
and various other operating systems seamlessly, enhancing flexibility and
reducing barriers for a wider range of users.

2. **Rich Ecosystem and Library Support:** .NET has a rich ecosystem,
comprising an extensive collection of libraries (such as those provided by
the
official NuGet Package Manager), tools, and services. This allows developers
to work on web applications (.NET for desktop apps and ASP.NET Core
for modern web applications), mobile applications (.NET MAUI),
IoT solutions, AI/ML projects, and much more with a vast array of prebuilt
components available at their disposal.

3. **Type Safety:** .NET operates under the Common Language Infrastructure
(CLI)
model and employs managed code for executing applications. This approach
inherently
offers strong type safety checks which help in preventing many runtime errors
that
are common in languages like C/C++. It also enables features such as garbage
collection,
thus relieving developers from manual memory management. These
characteristics enhance
the reliability of .NET-developed software and improve productivity by
catching
issues early during development.
```

3. The response from the AI is accurate, but also verbose. The stored chat history enables the AI to modify its response. Instruct the AI to shorten the list it provided:

Output

```
Your prompt:
Shorten the length of each item in the previous response.

AI Response:
 **Cross-platform Capabilities:** .NET allows building for various operating
systems
through platforms like .NET Core, promoting accessibility (Windows, Linux,
macOS).
```

```
**Extensive Ecosystem:** Offers a vast library selection via NuGet and tools
for web
(.NET Framework), mobile development (.NET MAUI), IoT, AI, providing rich
capabilities to developers.

**Type Safety & Reliability:** .NET's CLI model enforces strong typing and
automatic
garbage collection, mitigating runtime errors, thus enhancing application
stability.
```

The updated response from the AI is much shorter the second time. Due to the available chat history, the AI was able to assess the previous result and provide shorter summaries.

Next steps

- Generate text and conversations with .NET and Azure OpenAI Completions

Create a minimal AI assistant using .NET

Article • 02/28/2025

In this quickstart, you'll learn how to create a minimal AI assistant using the OpenAI or Azure OpenAI SDK libraries. AI assistants provide agentic functionality to help users complete tasks using AI tools and models. In the sections ahead, you'll learn the following:

- Core components and concepts of AI assistants
- How to create an assistant using the Azure OpenAI SDK
- How to enhance and customize the capabilities of an assistant

Prerequisites

- Install .NET 8.0 or higher
- Visual Studio Code (optional)
- Visual Studio (optional)
- An access key for an OpenAI model

Core components of AI assistants

AI assistants are based around conversational threads with a user. The user sends prompts to the assistant on a conversation thread, which direct the assistant to complete tasks using the tools it has available. Assistants can process and analyze data, make decisions, and interact with users or other systems to achieve specific goals. Most assistants include the following components:

Expand table

Component	Description
Assistant	The core AI client and logic that uses Azure OpenAI models, manages conversation threads, and utilizes configured tools.
Thread	A conversation session between an assistant and a user. Threads store messages and automatically handle truncation to fit content into a model's context.
Message	A message created by an assistant or a user. Messages can include text, images, and other files. Messages are stored as a list on the thread.
Run	Activation of an assistant to begin running based on the contents of the thread. The assistant uses its configuration and the thread's messages to perform tasks by

Component	Description
	calling models and tools. As part of a run, the assistant appends messages to the thread.
Run steps	A detailed list of steps the assistant took as part of a run. An assistant can call tools or create messages during its run. Examining run steps allows you to understand how the assistant is getting to its final results.

Assistants can also be configured to use multiple tools in parallel to complete tasks, including the following:

- **Code interpreter tool**: Writes and runs code in a sandboxed execution environment.
- **Function calling**: Runs local custom functions you define in your code.
- **File search capabilities**: Augments the assistant with knowledge from outside its model.

By understanding these core components and how they interact, you can build and customize powerful AI assistants to meet your specific needs.

Create the .NET app

Complete the following steps to create a .NET console app and add the package needed to work with assistants:

1. In a terminal window, navigate to an empty directory on your device and create a new app with the `dotnet new` command:

   .NET CLI

   ```
   dotnet new console -o AIAssistant
   ```

2. Add the OpenAI package to your app:

   .NET CLI

   ```
   dotnet package add OpenAI --prerelease
   ```

3. Open the new app in your editor of choice, such as Visual Studio Code.

   .NET CLI

   ```
   code .
   ```

Create the AI assistant client

1. Open the *Program.cs* file and replace the contents of the file with the following code to create the required clients:

```C#
using OpenAI;
using OpenAI.Assistants;
using OpenAI.Files;
using Azure.AI.OpenAI;
using Azure.Identity;

// Create the OpenAI client
OpenAIClient openAIClient = new("your-apy-key");

// For Azure OpenAI, use the following client instead:
AzureOpenAIClient azureAIClient = new(
        new Uri("your-azure-openai-endpoint"),
        new DefaultAzureCredential());

#pragma warning disable OPENAI001
AssistantClient assistantClient = openAIClient.GetAssistantClient();
OpenAIFileClient fileClient = openAIClient.GetOpenAIFileClient();
```

2. Create an in-memory sample document and upload it to the `OpenAIFileClient`:

```C#
// Create an in-memory document to upload to the file client
using Stream document = BinaryData.FromBytes("""
    {
        "description": "This document contains the sale history data
for Contoso products.",
        "sales": [
            {
                "month": "January",
                "by_product": {
                    "113043": 15,
                    "113045": 12,
                    "113049": 2
                }
            },
            {
                "month": "February",
                "by_product": {
                    "113045": 22
                }
            },
            {
                "month": "March",
```

```
                "by_product": {
                    "113045": 16,
                    "113055": 5
                }
            }
        ]
    }
    """u8.ToArray()).ToStream();

// Upload the document to the file client
OpenAIFile salesFile = fileClient.UploadFile(
    document,
    "monthly_sales.json",
    FileUploadPurpose.Assistants);
```

3. Enable file search and code interpreter tooling capabilities via the `AssistantCreationOptions`:

C#

```
// Configure the assistant options
AssistantCreationOptions assistantOptions = new()
{
    Name = "Example: Contoso sales RAG",
    Instructions =
        "You are an assistant that looks up sales data and helps
visualize the information based"
        + " on user queries. When asked to generate a graph, chart, or
other visualization, use"
        + " the code interpreter tool to do so.",
    Tools =
    {
        new FileSearchToolDefinition(), // Enable the assistant to
search and access files
        new CodeInterpreterToolDefinition(), // Enable the assistant to
run code for data analysis
    },
    ToolResources = new()
    {
        FileSearch = new()
        {
            NewVectorStores =
            {
                new VectorStoreCreationHelper([salesFile.Id]),
            }
        }
    },
};
```

4. Create the `Assistant` and a thread to manage interactions between the user and the assistant:

C#

```
// Create the assistant
Assistant assistant = assistantClient.CreateAssistant("gpt-4o",
assistantOptions);

// Configure and create the conversation thread
ThreadCreationOptions threadOptions = new()
{
    InitialMessages = { "How well did product 113045 sell in February?
Graph its trend over time." }
};

ThreadRun threadRun = assistantClient.CreateThreadAndRun(assistant.Id,
threadOptions);

// Sent the prompt and monitor progress until the thread run is
complete
do
{
    Thread.Sleep(TimeSpan.FromSeconds(1));
    threadRun = assistantClient.GetRun(threadRun.ThreadId,
threadRun.Id);
}
while (!threadRun.Status.IsTerminal);

// Get the messages from the thread run
var messages = assistantClient.GetMessagesAsync(
    threadRun.ThreadId,
    new MessageCollectionOptions()
    {
        Order = MessageCollectionOrder.Ascending
    });
```

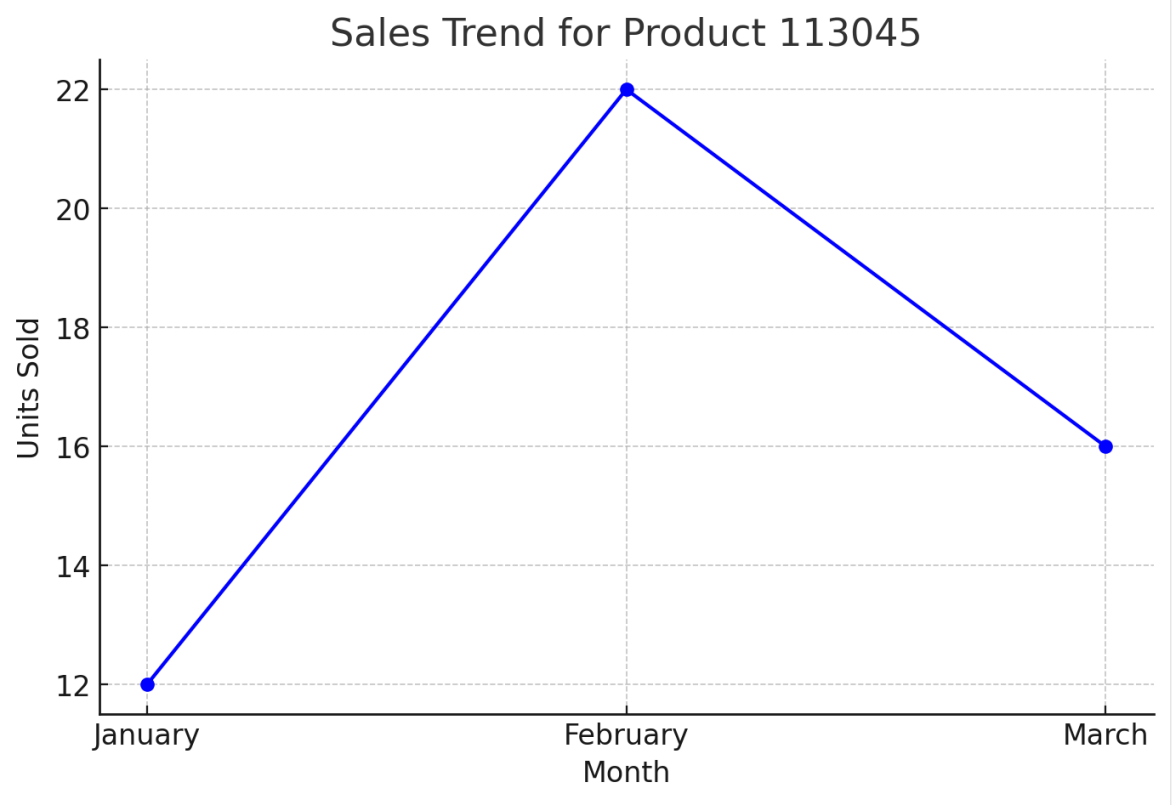
5. Print the messages and save the generated image from the conversation with the assistant:

C#

```
await foreach (ThreadMessage message in messages)
{
    // Print out the messages from the assistant
    Console.Write($"[{message.Role.ToString().ToUpper()}]: ");
    foreach (MessageContent contentItem in message.Content)
    {
        if (!string.IsNullOrEmpty(contentItem.Text))
        {
            Console.WriteLine($"{contentItem.Text}");

            if (contentItem.TextAnnotations.Count > 0)
            {
                Console.WriteLine();
```

```
            }

            // Include annotations, if any
            foreach (TextAnnotation annotation in
contentItem.TextAnnotations)
            {
                if (!string.IsNullOrEmpty(annotation.InputFileId))
                {
                    Console.WriteLine($"* File citation, file ID:
{annotation.InputFileId}");
                }
                if (!string.IsNullOrEmpty(annotation.OutputFileId))
                {
                    Console.WriteLine($"* File output, new file ID:
{annotation.OutputFileId}");
                }
            }
        }
        // Save the generated image file
        if (!string.IsNullOrEmpty(contentItem.ImageFileId))
        {
            OpenAIFile imageInfo =
fileClient.GetFile(contentItem.ImageFileId);
            BinaryData imageBytes =
fileClient.DownloadFile(contentItem.ImageFileId);
            using FileStream stream = File.OpenWrite($"
{imageInfo.Filename}.png");
            imageBytes.ToStream().CopyTo(stream);

            Console.WriteLine($"<image: {imageInfo.Filename}.png>");
        }
    }
    Console.WriteLine();
}
```

Locate and open the saved image in the app *bin* directory, which should resemble the following:



Next steps

- Generate text and conversations with .NET and Azure OpenAI Completions

Create a .NET AI app to chat with custom data using the AI app template extensions

08/01/2025

In this quickstart, you learn how to create a .NET AI app to chat with custom data using the .NET AI app template. The template is designed to streamline the getting started experience for building AI apps with .NET by handling common setup tasks and configurations for you.

Prerequisites

- .NET 9.0 SDK
- One of the following IDEs (optional):
  - Visual Studio 2022
  - Visual Studio Code with C# Dev Kit

Install the .NET AI app template

The **AI Chat Web App** template is available as a template package through NuGet. Use the dotnet new install command to install the package:

.NET CLI

```
dotnet new install Microsoft.Extensions.AI.Templates
```

Create the .NET AI app

After you install the AI app templates, you can use them to create starter apps through Visual Studio UI, Visual Studio Code, or the .NET CLI.

Visual Studio

1. Inside Visual Studio, navigate to **File** > **New** > **Project**.
2. On the **Create a new project** screen, search for **AI Chat Web App**. Select the matching result and then choose **Next**.
3. On the **Configure your new project** screen, enter the desired name and location for your project and then choose **Next**.
4. On the **Additional information** screen:

   - For the **Framework** option, select **.NET 9.0**.

- For the **AI service provider** option, select **GitHub Models**.
- For the **Vector store** option, select **Local on-disc (for prototyping)**.

5. Select **Create** to complete the process.

Explore the sample app

The sample app you created is a Blazor Interactive Server web app preconfigured with common AI and data services. The app handles the following concerns for you:

- Includes essential `Microsoft.Extensions.AI` packages and other dependencies in the `csproj` file to help you get started working with AI.
- Creates various AI services and registers them for dependency injection in the `Program.cs` file:
  - An `IChatClient` service to chat back and forth with the generative AI model
  - An `IEmbeddingGenerator` service that's used to generate embeddings, which are essential for vector search functionality
  - A `JsonVectorStore` to act as an in-memory vector store
- Registers a SQLite database context service to handle ingesting documents. The app is preconfigured to ingest whatever documents you add to the `Data` folder of the project, including the provided example files.
- Provides a complete chat UI using Blazor components. The UI handles rich formatting for the AI responses and provides features such as citations for response data.

Configure access to GitHub Models

To authenticate to GitHub models from your code, you'll need to create a GitHub personal access token :

1. Navigate to the **Personal access tokens** page of your GitHub account settings under **Developer Settings**.
2. Select **Generate new token**.
3. Enter a name for the token, and under **Permissions**, set **Models** to **Access: Read-only**.
4. Select **Generate token** at the bottom of the page.
5. Copy the token for use in the steps ahead.

Configure the app

The **AI Chat Web App** app is almost ready to go as soon as it's created. However, you need to configure the app to use the personal access token you set up for GitHub Models. By default,

the app template searches for this value in the project's local .NET user secrets. You can manage user secrets using either the Visual Studio UI or the .NET CLI.

Visual Studio

1. In Visual Studio, right-click on your project in the Solution Explorer and select **Manage User Secrets**. This opens a `secrets.json` file where you can store your API keys without them being tracked in source control.
2. Add the following key and value:

JSON

```
{
    "GitHubModels:Token": "<your-personal-access-token>"
}
```

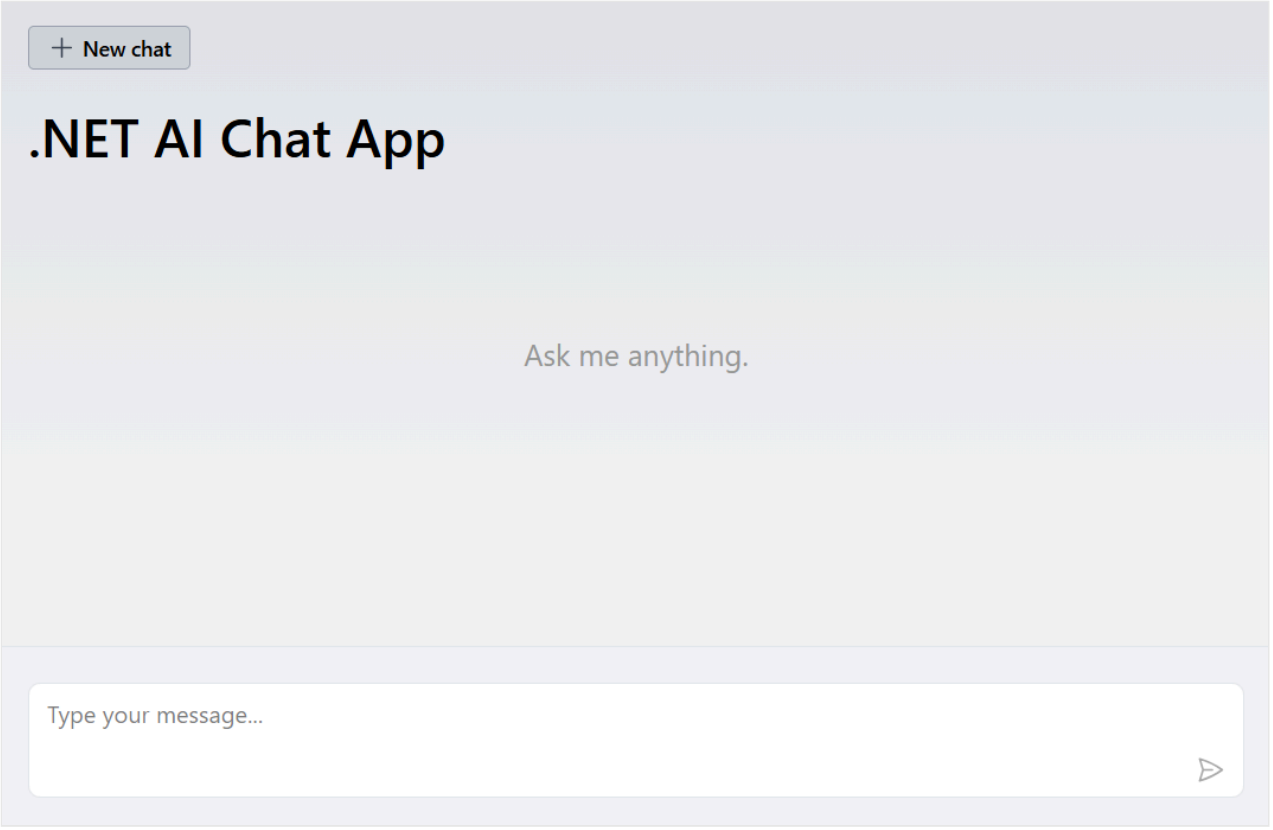
By default, the app template uses the `gpt-4o-mini` and `text-embedding-3-small` models. To try other models, update the name parameters in `Program.cs`:

C#

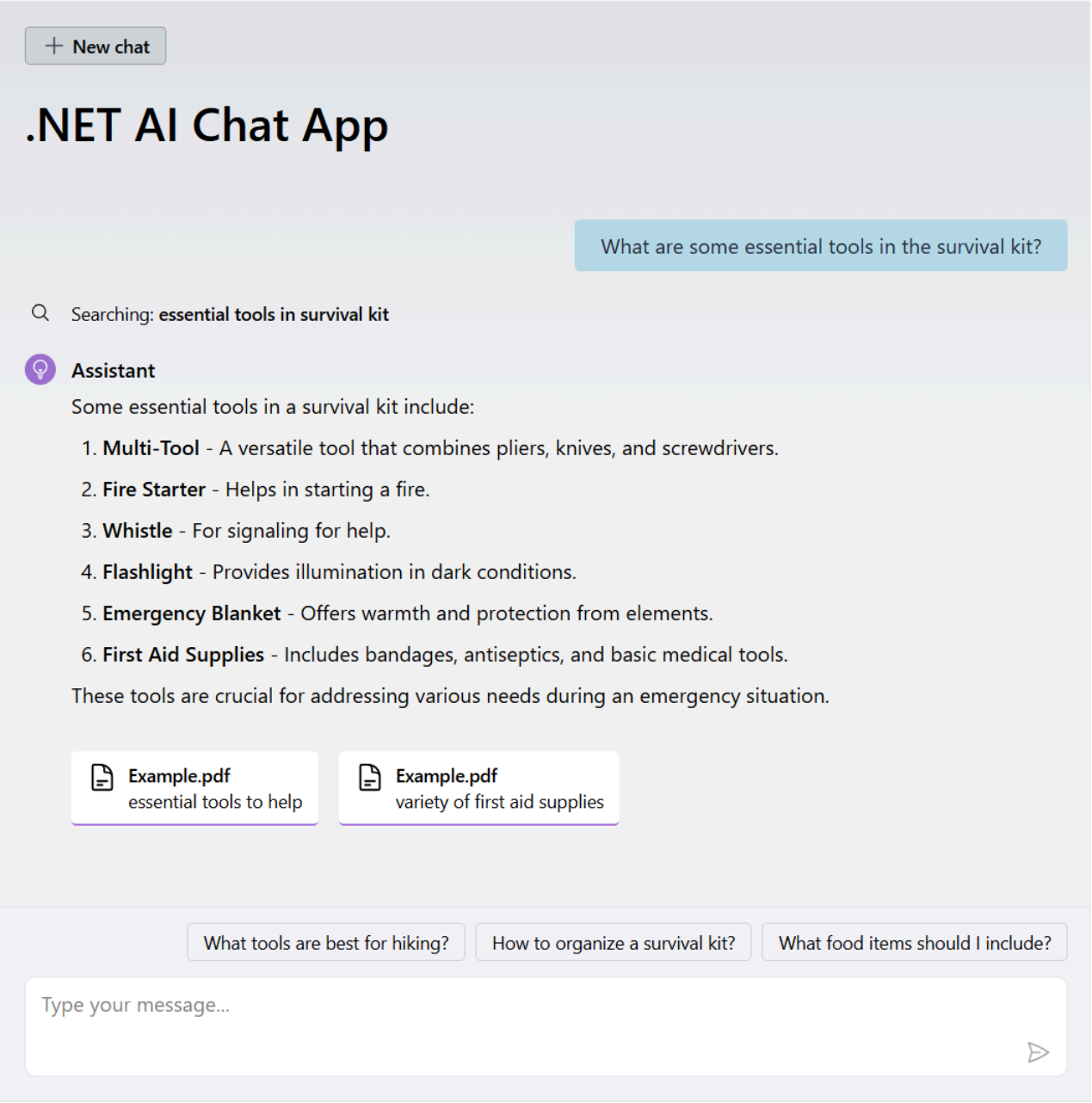
```
var chatClient = ghModelsClient.AsChatClient("gpt-4o-mini");
var embeddingGenerator = ghModelsClient.AsEmbeddingGenerator("text-embedding-3-small");
```

Run and test the app

1. Select the run button at the top of Visual Studio to launch the app. After a moment, you should see the following UI load in the browser:



2. Enter a prompt into the input box such as "*What are some essential tools in the survival kit?*" to ask your AI model a question about the ingested data from the example files.



The app responds with an answer to the question and provides citations of where it found the data. You can click on one of the citations to be directed to the relevant section of the example files.

Next steps

- Generate text and conversations with .NET and Azure OpenAI Completions

Create a minimal MCP server using C# and publish to NuGet

07/15/2025

In this quickstart, you create a minimal Model Context Protocol (MCP) server using the C# SDK for MCP, connect to it using GitHub Copilot, and publish it to NuGet. MCP servers are services that expose capabilities to clients through the Model Context Protocol (MCP).

> **Note**
>
> The `Microsoft.Extensions.AI.Templates` experience is currently in preview. The template uses the ModelContextProtocol library and the MCP registry server.json schema, which are both in preview.

Prerequisites

- .NET 10.0 SDK (preview 6 or higher)
- Visual Studio Code
- GitHub Copilot extension for Visual Studio Code
- NuGet.org account

Create the project

1. In a terminal window, install the MCP Server template (version 9.7.0-preview.2.25356.2 or newer):

   ```
   Bash

   dotnet new install Microsoft.Extensions.AI.Templates
   ```

2. Create a new MCP server app with the `dotnet new mcpserver` command:

   ```
   Bash

   dotnet new mcpserver -n SampleMcpServer
   ```

3. Navigate to the `SampleMcpServer` directory:

   ```
   Bash
   ```

```
cd SampleMcpServer
```

4. Build the project:

```Bash
dotnet build
```

5. Update the `<PackageId>` in the `.csproj` file to be unique on NuGet.org, for example `<NuGet.org username>.SampleMcpServer`.

Configure the MCP server in Visual Studio Code

Configure GitHub Copilot for Visual Studio Code to use your custom MCP server:

1. If you haven't already, open your project folder in Visual Studio Code.
2. Create a `.vscode` folder at the root of your project.
3. Add an `mcp.json` file in the `.vscode` folder with the following content:

```JSON
{
  "servers": {
    "SampleMcpServer": {
      "type": "stdio",
      "command": "dotnet",
      "args": [
        "run",
        "--project",
        "<RELATIVE PATH TO PROJECT DIRECTORY>"
      ]
    }
  }
}
```

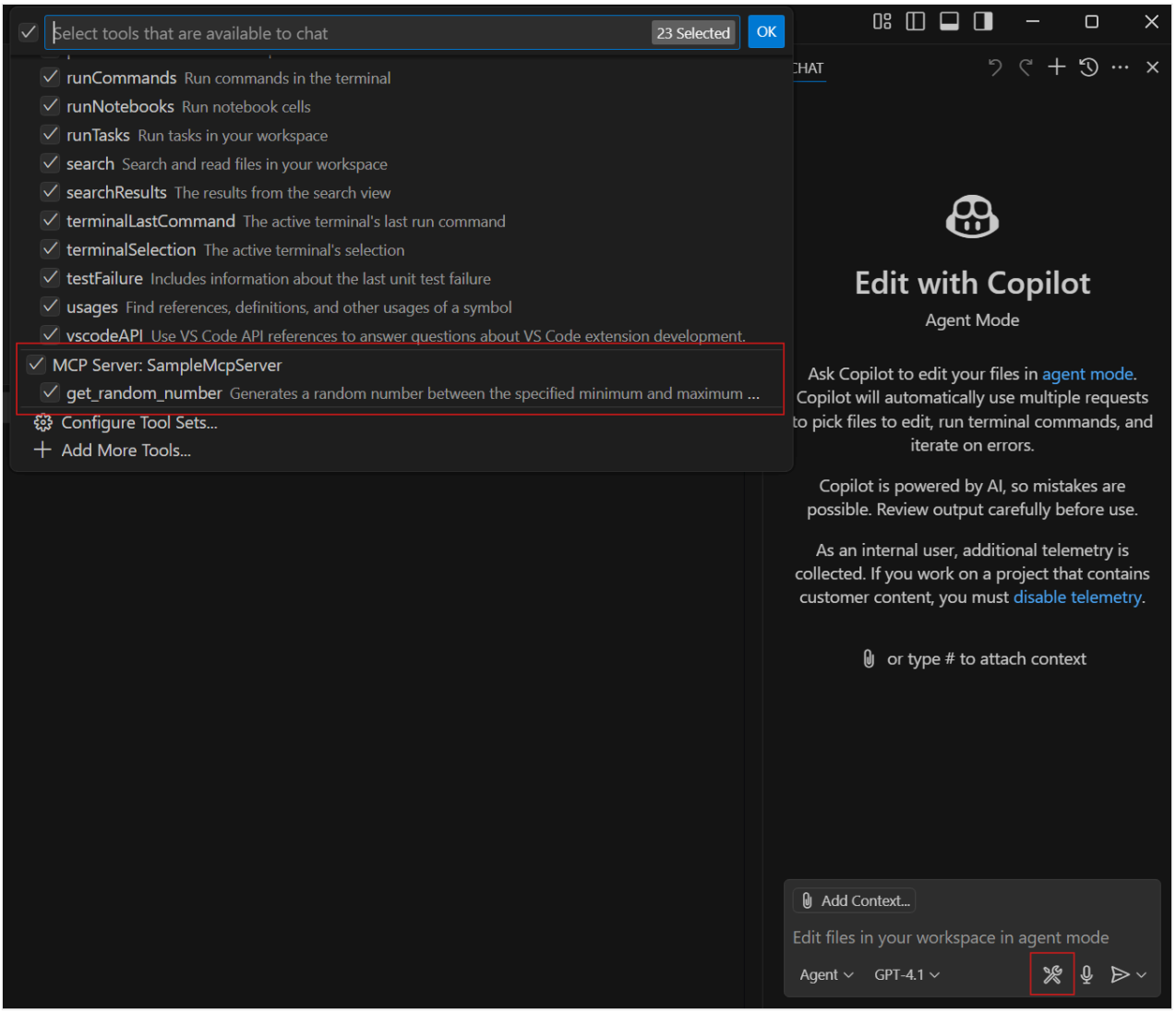
4. Save the file.

Test the MCP server

The MCP server template includes a tool called `get_random_number` you can use for testing and as a starting point for development.

1. Open GitHub Copilot in Visual Studio Code and switch to chat mode.

2. Select the **Select tools** icon to verify your **SampleMcpServer** is available with the sample tool listed.



3. Enter a prompt to run the **get_random_number** tool:

```
Console
```

```
Give me a random number between 1 and 100.
```

4. GitHub Copilot requests permission to run the **get_random_number** tool for your prompt. Select **Continue** or use the arrow to select a more specific behavior:

   - **Current session** always runs the operation in the current GitHub Copilot Agent Mode session.
   - **Current workspace** always runs the command for the current Visual Studio Code workspace.
   - **Always allow** sets the operation to always run for any GitHub Copilot Agent Mode session or any Visual Studio Code workspace.

5. Verify that the server responds with a random number:

Output

```
Your random number is 42.
```

Add inputs and configuration options

In this example, you enhance the MCP server to use a configuration value set in an environment variable. This could be configuration needed for the functioning of your MCP server, such as an API key, an endpoint to connect to, or a local directory path.

1. Add another tool method after the `GetRandomNumber` method in `Tools/RandomNumberTools.cs`. Update the tool code to use an environment variable.

C#

```
[McpServerTool]
[Description("Describes random weather in the provided city.")]
public string GetCityWeather(
    [Description("Name of the city to return weather for")] string city)
{
    // Read the environment variable during tool execution.
    // Alternatively, this could be read during startup and passed via
IOptions dependency injection
    var weather = Environment.GetEnvironmentVariable("WEATHER_CHOICES");
    if (string.IsNullOrWhiteSpace(weather))
    {
        weather = "balmy,rainy,stormy";
    }

    var weatherChoices = weather.Split(",");
    var selectedWeatherIndex =  Random.Shared.Next(0, weatherChoices.Length);

    return $"The weather in {city} is
{weatherChoices[selectedWeatherIndex]}.";
}
```

2. Update the `.vscode/mcp.json` to set the `WEATHER_CHOICES` environment variable for testing.

JSON

```
{
   "servers": {
     "SampleMcpServer": {
       "type": "stdio",
```

```
        "command": "dotnet",
        "args": [
          "run",
          "--project",
          "<RELATIVE PATH TO PROJECT DIRECTORY>"
        ],
        "env": {
           "WEATHER_CHOICES": "sunny,humid,freezing"
        }
      }
    }
  }
```

3. Try another prompt with Copilot in VS Code, such as:

```
Console

What is the weather in Redmond, Washington?
```

   VS Code should return a random weather description.

4. Update the `.mcp/server.json` to declare your environment variable input. The `server.json` file schema is defined by the MCP Registry project and is used by NuGet.org to generate VS Code MCP configuration.

   - Use the `environment_variables` property to declare environment variables used by your app that will be set by the client using the MCP server (for example, VS Code).
   - Use the `package_arguments` property to define CLI arguments that will be passed to your app. For more examples, see the MCP Registry project.

```
JSON

{
  "$schema": "https://modelcontextprotocol.io/schemas/draft/2025-07-
09/server.json",
  "description": "<your description here>",
  "name": "io.github.<your GitHub username here>/<your repo name>",
  "packages": [
    {
      "registry_name": "nuget",
      "name": "<your package ID here>",
      "version": "<your package version here>",
      "package_arguments": [],
      "environment_variables": [
        {
          "name": "WEATHER_CHOICES",
          "value": "{weather_choices}",
          "variables": {
```

```
            "weather_choices": {
              "description": "Comma separated list of weather descriptions to
randomly select.",
              "is_required": true,
              "is_secret": false
            }
          }
        }
      ]
    }
  ],
  "repository": {
    "url": "https://github.com/<your GitHub username here>/<your repo name>",
    "source": "github"
  },
  "version_detail": {
    "version": "<your package version here>"
  }
}
```

The only information used by NuGet.org in the `server.json` is the first `packages` array item with the `registry_name` value matching `nuget`. The other top-level properties aside from the `packages` property are currently unused and are intended for the upcoming central MCP Registry. You can leave the placeholder values until the MCP Registry is live and ready to accept MCP server entries.

You can test your MCP server again before moving forward.

Pack and publish to NuGet

1. Pack the project:

   Bash

   ```
   dotnet pack -c Release
   ```

2. Publish the package to NuGet:

   Bash

   ```
   dotnet nuget push bin/Release/*.nupkg --api-key <your-api-key> --source
   https://api.nuget.org/v3/index.json
   ```

   If you want to test the publishing flow before publishing to NuGet.org, you can register an account on the NuGet Gallery integration environment: https://int.nugettest.org. The `push` command would be modified to:

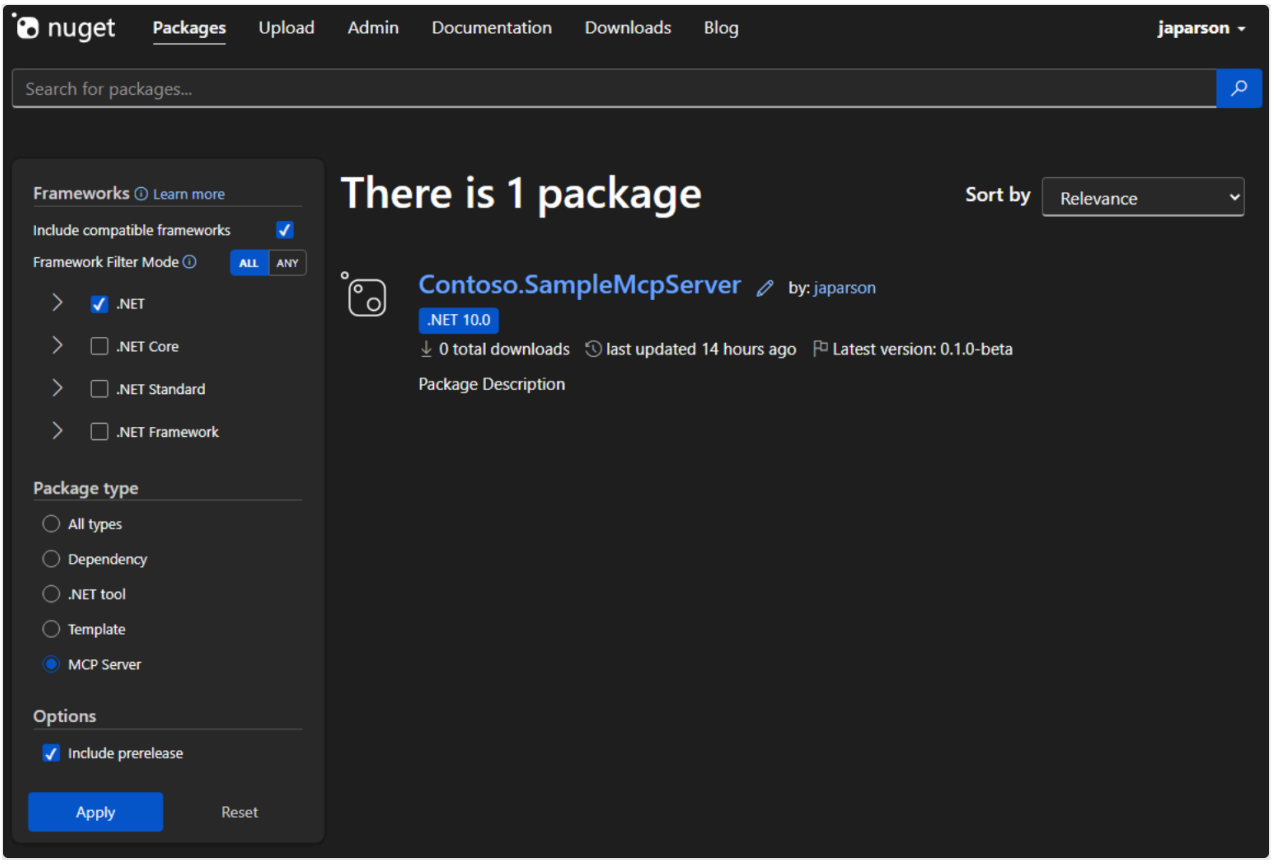
Bash

```
dotnet nuget push bin/Release/*.nupkg --api-key <your-api-key> --source
https://apiint.nugettest.org/v3/index.json
```

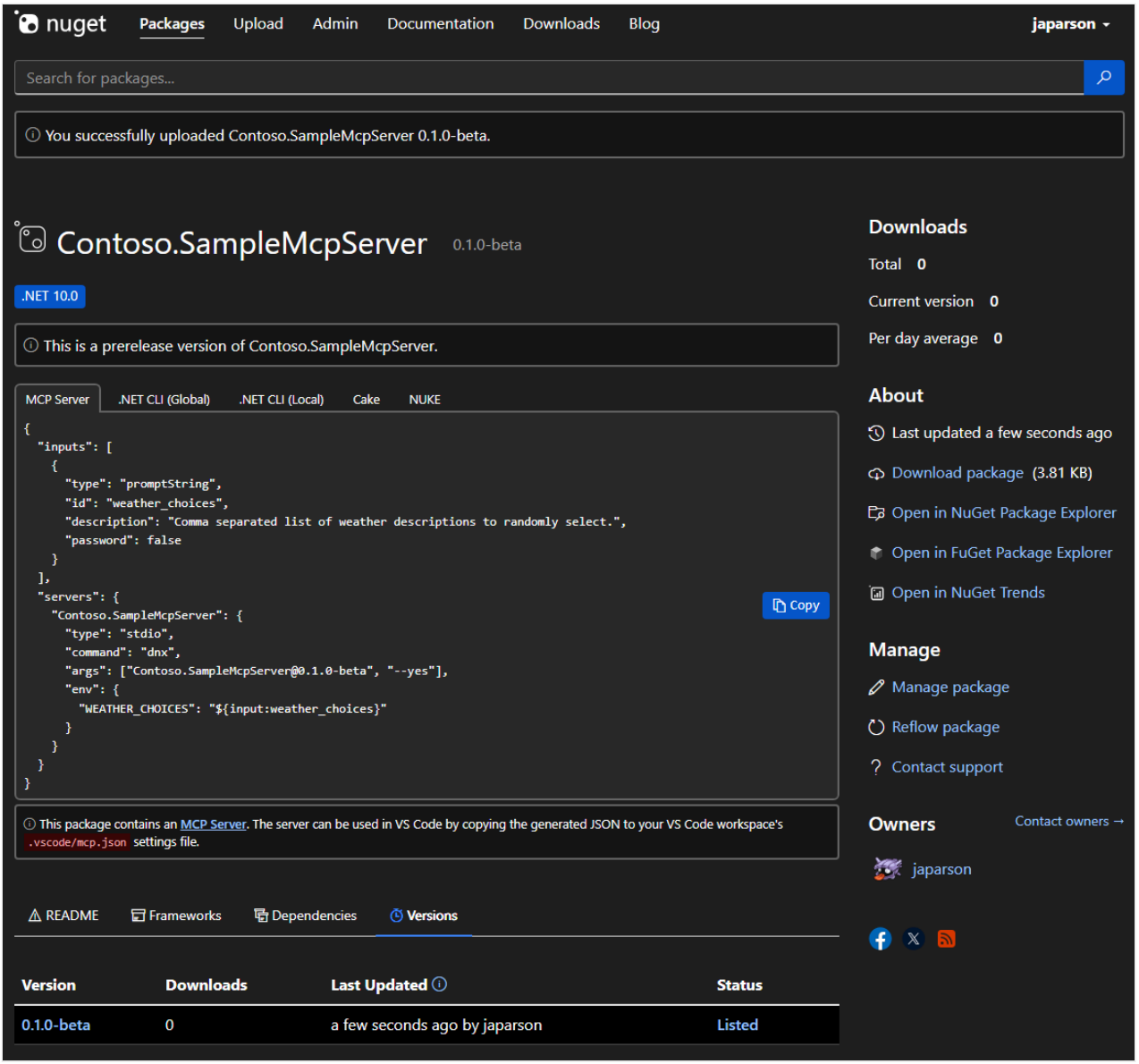
For more information, see Publish a package.

Discover MCP servers on NuGet.org

1. Search for your MCP server package on NuGet.org (or int.nugettest.org if you published to the integration environment) and select it from the list.



2. View the package details and copy the JSON from the "MCP Server" tab.



3. In your `mcp.json` file in the `.vscode` folder, add the copied JSON, which looks like this:

JSON

```
{
  "inputs": [
    {
      "type": "promptString",
      "id": "weather_choices",
      "description": "Comma separated list of weather descriptions to
randomly select.",
      "password": false
    }
  ],
  "servers": {
    "Contoso.SampleMcpServer": {
      "type": "stdio",
      "command": "dnx",
      "args": ["Contoso.SampleMcpServer@0.0.1-beta", "--yes"],
      "env": {
        "WEATHER_CHOICES": "${input:weather_choices}"
      }
```

```
    }
  }
}
```

If you published to the NuGet Gallery integration environment, you need to add `"--add-source", "https://apiint.nugettest.org/v3/index.json"` at the end of the `"args"` array.

4. Save the file.
5. In GitHub Copilot, select the **Select tools** icon to verify your **SampleMcpServer** is available with the tools listed.
6. Enter a prompt to run the new **get_city_weather** tool:

```
Console

What is the weather in Redmond?
```

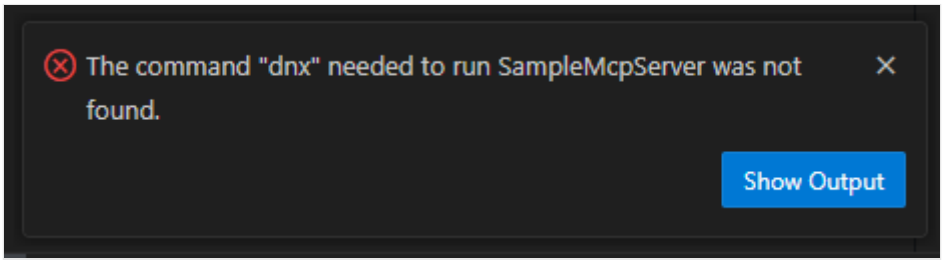
7. If you added inputs to your MCP server (for example, `WEATHER_CHOICES`), you will be prompted to provide values.
8. Verify that the server responds with the random weather:

```
Output

The weather in Redmond is balmy.
```

Common issues

The command "dnx" needed to run SampleMcpServer was not found.

If VS Code shows this error when starting the MCP server, you need to install a compatible version of the .NET SDK.



The `dnx` command is shipped as part of the .NET SDK, starting with version 10 preview 6. Install the .NET 10 SDK to resolve this issue.

GitHub Copilot does not use your tool (an answer is provided without invoking your tool).

Generally speaking, an AI agent like GitHub Copilot is informed that it has some tools available by the client application, such as VS Code. Some tools, such as the sample random number tool, might not be leveraged by the AI agent because it has similar functionality built in.

If your tool is not being used, check the following:

1. Verify that your tool appears in the list of tools that VS Code has enabled. See the screenshot in Test the MCP server for how to check this.
2. Explicitly reference the name of the tool in your prompt. In VS Code, you can reference your tool by name. For example, `Using #get_random_weather, what is the weather in Redmond?`.
3. Verify your MCP server is able to start. You can check this by clicking the "Start" button visible above your MCP server configuration in the VS Code user or workspace settings.

Related content

- Get started with .NET AI and the Model Context Protocol
- Model Context Protocol .NET samples
- Build a minimal MCP client
- Publish a package
- Find and evaluate NuGet packages for your project
- What's new in .NET 10

Create a minimal MCP client using .NET

Article • 05/30/2025

In this quickstart, you build a minimal Model Context Protocol (MCP) client using the C# SDK for MCP. You also learn how to configure the client to connect to an MCP server, such as the one created in the Build a minimal MCP server quickstart.

Prerequisites

- .NET 8.0 SDK or higher
- Visual Studio Code

> **Note**
>
> The MCP client you build in the sections ahead connects to the sample MCP server from the **Build a minimal MCP server** quickstart. You can also use your own MCP server if you provide your own connection configuration.

Create the .NET host app

Complete the following steps to create a .NET console app. The app acts as a host for an MCP client that connects to an MCP server.

Create the project

1. In a terminal window, navigate to the directory where you want to create your app, and create a new console app with the `dotnet new` command:

   Console

   ```
   dotnet new console -n MCPHostApp
   ```

2. Navigate into the newly created project folder:

   Console

   ```
   cd MCPHostApp
   ```

3. Run the following commands to add the necessary NuGet packages:

Console

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

4. Open the project folder in your editor of choice, such as Visual Studio Code:

Console

```
code .
```

Add the app code

Replace the contents of `Program.cs` with the following code:

C#

```
using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Extensions.AI;
using ModelContextProtocol.Client;
using ModelContextProtocol.Protocol.Transport;

// Create an IChatClient using Azure OpenAI.
IChatClient client =
    new ChatClientBuilder(
        new AzureOpenAIClient(new Uri("<your-azure-openai-endpoint>"),
        new DefaultAzureCredential())
        .GetChatClient("gpt-4o").AsIChatClient())
    .UseFunctionInvocation()
    .Build();

// Create the MCP client
// Configure it to start and connect to your MCP server.
var mcpClient = await McpClientFactory.CreateAsync(
    new StdioClientTransport(new()
    {
        Command = "dotnet run",
        Arguments = ["--project", "<path-to-your-mcp-server-project>"],
        Name = "Minimal MCP Server",
    }));

// List all available tools from the MCP server.
Console.WriteLine("Available tools:");
var tools = await mcpClient.ListToolsAsync();
foreach (var tool in tools)
{
```

```
    Console.WriteLine($"{tool}");
}
Console.WriteLine();

// Conversational loop that can utilize the tools via prompts.
List<ChatMessage> messages = [];
while (true)
{
    Console.Write("Prompt: ");
    messages.Add(new(ChatRole.User, Console.ReadLine()));

    List<ChatResponseUpdate> updates = [];
    await foreach (var update in client
        .GetStreamingResponseAsync(messages, new() { Tools = [.. tools] }))
    {
        Console.Write(update);
        updates.Add(update);
    }
    Console.WriteLine();

    messages.AddMessages(updates);
}
```

The preceding code accomplishes the following tasks:

- Initializes an `IChatClient` abstraction using the Microsoft.Extensions.AI libraries.
- Creates an MCP client and configures it to connect to your MCP server.
- Retrieves and displays a list of available tools from the MCP server, which is a standard MCP function.
- Implements a conversational loop that processes user prompts and utilizes the tools for responses.

Run and test the app

Complete the following steps to test your .NET host app:

1. In a terminal window open to the root of your project, run the following command to start the app:

   ```
   Console
   ```

   ```
   dotnet run
   ```

2. Once the app is running, enter a prompt to run the **ReverseEcho** tool:

   ```
   Console
   ```

```
Reverse the following: "Hello, minimal MCP server!"
```

3. Verify that the server responds with the echoed message:

Output

```
!revres PCM laminim ,olleH
```

Related content

Get started with .NET AI and the Model Context Protocol

How generative AI and LLMs work

05/29/2025

Generative AI is a type of artificial intelligence capable of creating original content, such as natural language, images, audio, and code. The output of a generative AI is based on the inputs provided by the user. One common way for users to interact with generative AI is through chat applications that use natural language as their input. ChatGPT, developed by OpenAI, is a popular example of this. Generative AI applications that use natural language as an input are powered by large language models (LLMs) to perform natural language processing (NLP).

How generative AI works

All generative AI is built on top of models. These models are trained with large sets of data in the form of content, such as natural language, images, audio, and code. Generative AI models use the patterns identified in the training data to produce new, statistically similar content.

The input provided by the user is used by the AI model to build an output. The input is first parsed into a form of data that the model can understand. The model then uses that data to identify matching patterns from its training that it combines to build the final output. Generative AI models are designed to produce unique content, so they won't generate the same output for identical inputs.

Generative AI applications that support natural language as an input or output utilize LLMs to do so. The LLM is used to perform NLP, which classifies the input text and determines its sentiment. That classification and sentiment analysis is used by the generative AI model to identify patterns and build the output. If the output is text, the LLM alone can be used to generate it. If the output is audio or images, additional models are used to provide the data and patterns for generating outputs in that format.

Common uses of generative AI

Generative AI applications support a variety of potential use cases and potential outputs, which are explored in the following sections.

Natural language generation

Some generative AI applications can produce natural language outputs, such as:

- Professionally worded summaries for a LinkedIn profile

- Rich descriptions of items in an online storefront
- Ideas for healthy meals

For example, given the user input *Give me three ideas for a healthy breakfast including peppers*, a generative AI application might produce the following output:

OK, here are three healthy breakfasts that include peppers:

1. *Egg and Pepper Scramble: Dice up some bell peppers and sauté them in a pan until they're slightly softened. Then, add in some beaten eggs and scramble everything together. Serve with whole-grain toast for a nutritious breakfast.*

2. *Pepper and Avocado Toast: Toast a slice of whole-grain bread and top it with sliced bell peppers, mashed avocado, and a sprinkle of salt and pepper. This is a great way to get some healthy fats from the avocado along with the vitamin C from the peppers.*

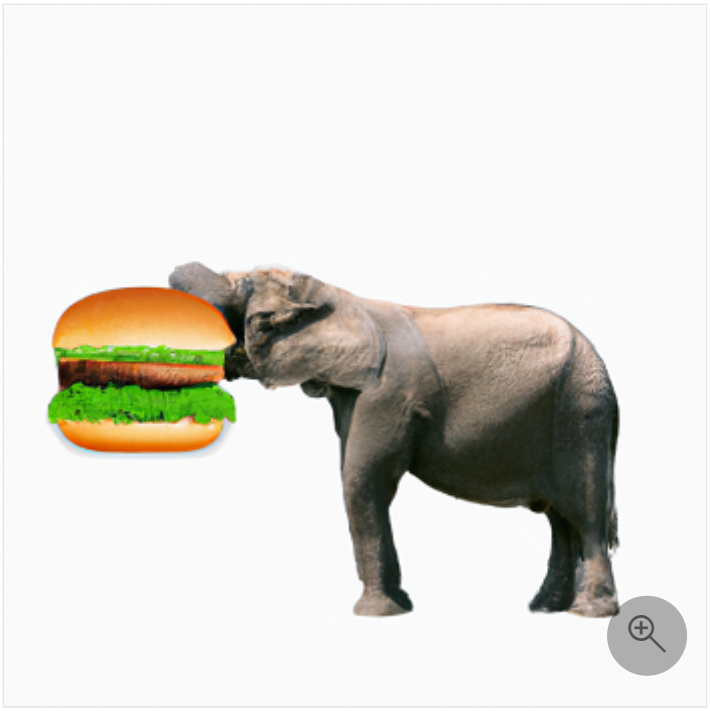
3. *Breakfast Burrito with Peppers: Sauté some diced bell peppers, onions, and scrambled eggs together in a pan. Then, spoon the mixture onto a whole-grain tortilla and top with some salsa or hot sauce. Roll it up and enjoy a satisfying and nutritious breakfast.*

Image generation

Some generative AI applications produce image outputs from natural language inputs. Depending on the application and its models, users might be able to specify:

- The people or things to appear the image
- The artistic style to create the image in
- References for generating similar images

Image generation can create virtual avatars for online accounts, design logos for a business, or provide artistic inspiration for creators. For example, a user may input the request, *Create an image of an elephant eating a burger*. A generative AI application might produce the following output:



Audio generation

Some generative AI applications produce audio outputs from natural language inputs. Depending on the application and its models, users might be able to:

- Synthesize natural sounding voices from input text
- Create music in a specific style or featuring certain instruments
- Modify input audio files based on a set criteria provided in natural language

Audio generation can provide spoken responses from digital voice assistants, add backing instruments to songs for music production, or reproduce a user's original voice from reference recordings.

Code generation

Some generative AI applications produce code from natural language inputs. Depending on the application and its models, users might be able to generate code that:

- Is in the language of their choice
- Accepts a specific input and output structure
- Uses specific technologies based on their reference documentation

Code generation can provide quick implementations of common operations, such as sort or filter data, speed up translating code from one language to another, or answer questions about a technology based on specified reference documentation. For example, given the input "Show me how to code a game of tic-tac-toe with C#", a generative AI application might produce the following output:

Here's a simple example of how you can code a game of tic-tac-toe in C#:

C#

```
using System;

namespace TicTacToe
{
    class Program
    {
        static char[] board = { '1', '2', '3', '4', '5', '6', '7', '8', '9' };
        static int player = 1; // Player 1 will start
        static int choice; // The number the player chooses
        static int flag = 0; // Set the flag value to 0

        // Check if the position is already marked or not
        private static int CheckPosition(char mark)
        {
            for (int i = 0; i < 9; i++)
            {
                if (board[i] == mark)
                {
                    return 1;
                }
            }
            return 0;
        }

        // The rest of the generated code has been omitted for brevity
        // ...
    }
}
```

This code creates a simple console-based tic-tac-toe game in C#. It uses a single-dimensional array to represent the board and checks for a win or draw after each move.

How LLMs work

When training an LLM, the training text is first broken down into tokens. Each token identifies a unique text value. A token can be a distinct word, a partial word, or a combination of words and punctuation. Each token is assigned an ID, which enables the text to be represented as a sequence of token IDs.

After the text has been broken down into tokens, a contextual vector, known as an embedding, is assigned to each token. These embedding vectors are multi-valued numeric data where each element of a token's vector represents a semantic attribute of the token. The elements of a token's vector are determined based on how commonly tokens are used together or in similar contexts.

The goal is to predict the next token in the sequence based on the preceding tokens. A weight is assigned to each token in the existing sequence that represents its relative influence on the next token. A calculation is then performed that uses the preceding tokens' weights and embeddings to predict the next vector value. The model then selects the most probable token to continue the sequence based on the predicted vector.

This process continues iteratively for each token in the sequence, with the output sequence being used regressively as the input for the next iteration. The output is built one token at a time. This strategy is analogous to how auto-complete works, where suggestions are based on what's been typed so far and updated with each new input.

During training, the complete sequence of tokens is known, but all tokens that come after the one currently being considered are ignored. The predicted value for the next token's vector is compared to the actual value and the loss is calculated. The weights are then incrementally adjusted to reduce the loss and improve the model.

Related content

- Understand Tokens
- Prompt engineering
- Large language models

Understand tokens

05/29/2025

Tokens are words, character sets, or combinations of words and punctuation that are generated by large language models (LLMs) when they decompose text. Tokenization is the first step in training. The LLM analyzes the semantic relationships between tokens, such as how commonly they're used together or whether they're used in similar contexts. After training, the LLM uses those patterns and relationships to generate a sequence of output tokens based on the input sequence.

Turn text into tokens

The set of unique tokens that an LLM is trained on is known as its *vocabulary*.

For example, consider the following sentence:

> `I heard a dog bark loudly at a cat`

This text could be tokenized as:

- `I`
- `heard`
- `a`
- `dog`
- `bark`
- `loudly`
- `at`
- `a`
- `cat`

By having a sufficiently large set of training text, tokenization can compile a vocabulary of many thousands of tokens.

Common tokenization methods

The specific tokenization method varies by LLM. Common tokenization methods include:

- **Word** tokenization (text is split into individual words based on a delimiter)
- **Character** tokenization (text is split into individual characters)
- **Subword** tokenization (text is split into partial words or character sets)

For example, the GPT models, developed by OpenAI, use a type of subword tokenization that's known as *Byte-Pair Encoding* (BPE). OpenAI provides a tool to visualize how text will be tokenized.

There are benefits and disadvantages to each tokenization method:

Expand table

Token size	Pros	Cons
Smaller tokens (character or subword tokenization)	- Enables the model to handle a wider range of inputs, such as unknown words, typos, or complex syntax. - Might allow the vocabulary size to be reduced, requiring fewer memory resources.	- A given text is broken into more tokens, requiring additional computational resources while processing. - Given a fixed token limit, the maximum size of the model's input and output is smaller.
Larger tokens (word tokenization)	- A given text is broken into fewer tokens, requiring fewer computational resources while processing. - Given the same token limit, the maximum size of the model's input and output is larger.	- Might cause an increased vocabulary size, requiring more memory resources. - Can limit the models ability to handle unknown words, typos, or complex syntax.

How LLMs use tokens

After the LLM completes tokenization, it assigns an ID to each unique token.

Consider our example sentence:

> `I heard a dog bark loudly at a cat`

After the model uses a word tokenization method, it could assign token IDs as follows:

- `I` (1)
- `heard` (2)
- `a` (3)
- `dog` (4)
- `bark` (5)
- `loudly` (6)
- `at` (7)
- `a` (the "a" token is already assigned an ID of 3)
- `cat` (8)

By assigning IDs, text can be represented as a sequence of token IDs. The example sentence would be represented as [1, 2, 3, 4, 5, 6, 7, 3, 8]. The sentence "`I heard a cat`" would be represented as [1, 2, 3, 8].

As training continues, the model adds any new tokens in the training text to its vocabulary and assigns it an ID. For example:

- `meow` (9)
- `run` (10)

The semantic relationships between the tokens can be analyzed by using these token ID sequences. Multi-valued numeric vectors, known as embeddings, are used to represent these relationships. An embedding is assigned to each token based on how commonly it's used together with, or in similar contexts to, the other tokens.

After it's trained, a model can calculate an embedding for text that contains multiple tokens. The model tokenizes the text, then calculates an overall embeddings value based on the learned embeddings of the individual tokens. This technique can be used for semantic document searches or adding vector stores to an AI.

During output generation, the model predicts a vector value for the next token in the sequence. The model then selects the next token from its vocabulary based on this vector value. In practice, the model calculates multiple vectors by using various elements of the previous tokens' embeddings. The model then evaluates all potential tokens from these vectors and selects the most probable one to continue the sequence.

Output generation is an iterative operation. The model appends the predicted token to the sequence so far and uses that as the input for the next iteration, building the final output one token at a time.

Token limits

LLMs have limitations regarding the maximum number of tokens that can be used as input or generated as output. This limitation often causes the input and output tokens to be combined into a maximum context window. Taken together, a model's token limit and tokenization method determine the maximum length of text that can be provided as input or generated as output.

For example, consider a model that has a maximum context window of 100 tokens. The model processes the example sentences as input text:

> `I heard a dog bark loudly at a cat`

By using a word-based tokenization method, the input is nine tokens. This leaves 91 **word** tokens available for the output.

By using a character-based tokenization method, the input is 34 tokens (including spaces). This leaves only 66 **character** tokens available for the output.

Token-based pricing and rate limiting

Generative AI services often use token-based pricing. The cost of each request depends on the number of input and output tokens. The pricing might differ between input and output. For example, see Azure OpenAI Service pricing.

Generative AI services might also be limited regarding the maximum number of tokens per minute (TPM). These rate limits can vary depending on the service region and LLM. For more information about specific regions, see Azure OpenAI Service quotas and limits.

Related content

- How generative AI and LLMs work
- Understand embeddings
- Work with vector databases

Embeddings in .NET

05/29/2025

Embeddings are the way LLMs capture semantic meaning. They are numeric representations of non-numeric data that an LLM can use to determine relationships between concepts. You can use embeddings to help an AI model understand the meaning of inputs so that it can perform comparisons and transformations, such as summarizing text or creating images from text descriptions. LLMs can use embeddings immediately, and you can store embeddings in vector databases to provide semantic memory for LLMs as-needed.

Use cases for embeddings

This section lists the main use cases for embeddings.

Use your own data to improve completion relevance

Use your own databases to generate embeddings for your data and integrate it with an LLM to make it available for completions. This use of embeddings is an important component of retrieval-augmented generation.

Increase the amount of text you can fit in a prompt

Use embeddings to increase the amount of context you can fit in a prompt without increasing the number of tokens required.

For example, suppose you want to include 500 pages of text in a prompt. The number of tokens for that much raw text will exceed the input token limit, making it impossible to directly include in a prompt. You can use embeddings to summarize and break down large amounts of that text into pieces that are small enough to fit in one input, and then assess the similarity of each piece to the entire raw text. Then you can choose a piece that best preserves the semantic meaning of the raw text and use it in your prompt without hitting the token limit.

Perform text classification, summarization, or translation

Use embeddings to help a model understand the meaning and context of text, and then classify, summarize, or translate that text. For example, you can use embeddings to help models classify texts as positive or negative, spam or not spam, or news or opinion.

Generate and transcribe audio

Use audio embeddings to process audio files or inputs in your app.

For example, Azure AI Speech supports a range of audio embeddings, including speech to text and text to speech. You can process audio in real-time or in batches.

Turn text into images or images into text

Semantic image processing requires image embeddings, which most LLMs can't generate. Use an image-embedding model such as ViT to create vector embeddings for images. Then you can use those embeddings with an image generation model to create or modify images using text or vice versa. For example, you can use the DALL·E model to generate images such as logos, faces, animals, and landscapes.

Generate or document code

Use embeddings to help a model create code from text or vice versa, by converting different code or text expressions into a common representation. For example, you can use embeddings to help a model generate or document code in C# or Python.

Choose an embedding model

You generate embeddings for your raw data by using an AI embedding model, which can encode non-numeric data into a vector (a long array of numbers). The model can also decode an embedding into non-numeric data that has the same or similar meaning as the original, raw data. There are many embedding models available for you to use, with OpenAI's `text-embedding-ada-002` model being one of the common models that's used. For more examples, see the list of Embedding models available on Azure OpenAI.

Store and process embeddings in a vector database

After you generate embeddings, you'll need a way to store them so you can later retrieve them with calls to an LLM. Vector databases are designed to store and process vectors, so they're a natural home for embeddings. Different vector databases offer different processing capabilities, so you should choose one based on your raw data and your goals. For information about your options, see available vector database solutions.

Using embeddings in your LLM solution

When building LLM-based applications, you can use Semantic Kernel to integrate embedding models and vector stores, so you can quickly pull in text data, and generate and store

embeddings. This lets you use a vector database solution to store and retrieve semantic memories.

Related content

- How GenAI and LLMs work
- Retrieval-augmented generation
- Training: Develop AI agents with Azure OpenAI and Semantic Kernel

Vector databases for .NET + AI

05/29/2025

Vector databases are designed to store and manage vector embeddings. Embeddings are numeric representations of non-numeric data that preserve semantic meaning. Words, documents, images, audio, and other types of data can all be vectorized. You can use embeddings to help an AI model understand the meaning of inputs so that it can perform comparisons and transformations, such as summarizing text, finding contextually related data, or creating images from text descriptions.

For example, you can use a vector database to:

- Identify similar images, documents, and songs based on their contents, themes, sentiments, and styles.
- Identify similar products based on their characteristics, features, and user groups.
- Recommend content, products, or services based on user preferences.
- Identify the best potential options from a large pool of choices to meet complex requirements.
- Identify data anomalies or fraudulent activities that are dissimilar from predominant or normal patterns.

Understand vector search

Vector databases provide vector search capabilities to find similar items based on their data characteristics rather than by exact matches on a property field. Vector search works by analyzing the vector representations of your data that you created using an AI embedding model such the Azure OpenAI embedding models. The search process measures the distance between the data vectors and your query vector. The data vectors that are closest to your query vector are the ones that are found to be most similar semantically.

Some services such as Azure Cosmos DB for MongoDB vCore provide native vector search capabilities for your data. Other databases can be enhanced with vector search by indexing the stored data using a service such as Azure AI Search, which can scan and index your data to provide vector search capabilities.

Vector search workflows with .NET and OpenAI

Vector databases and their search features are especially useful in RAG pattern workflows with Azure OpenAI. This pattern allows you to augment or enhance your AI model with additional semantically rich knowledge of your data. A common AI workflow using vector databases might include the following steps:

1. Create embeddings for your data using an OpenAI embedding model.
2. Store and index the embeddings in a vector database or search service.
3. Convert user prompts from your application to embeddings.
4. Run a vector search across your data, comparing the user prompt embedding to the embeddings your database.
5. Use a language model such as GPT-35 or GPT-4 to assembly a user friendly completion from the vector search results.

Visit the Implement Azure OpenAI with RAG using vector search in a .NET app tutorial for a hands-on example of this flow.

Other benefits of the RAG pattern include:

- Generate contextually relevant and accurate responses to user prompts from AI models.
- Overcome LLM tokens limits - the heavy lifting is done through the database vector search.
- Reduce the costs from frequent fine-tuning on updated data.

Available vector database solutions

AI applications often use data vector databases and services to improve relevancy and provide customized functionality. Many of these services provide a native SDK for .NET, while others offer a REST service you can connect to through custom code. Semantic Kernel provides an extensible component model that enables you to use different vector stores without needing to learn each SDK.

Semantic Kernel provides connectors for the following vector databases and services:

Expand table

Vector service	Semantic Kernel connector	.NET SDK
Azure AI Search	Microsoft.SemanticKernel.Connectors.AzureAISearch	Azure.Search.Documents
Azure Cosmos DB for NoSQL	Microsoft.SemanticKernel.Connectors.AzureCosmosDBNoSQL	Microsoft.Azure.Cosmos
Azure Cosmos DB for MongoDB	Microsoft.SemanticKernel.Connectors.AzureCosmosDBMongoDB	MongoDb.Driver

Vector service	Semantic Kernel connector	.NET SDK
Azure PostgreSQL Server	Microsoft.SemanticKernel.Connectors.Postgres	Npgsql
Azure SQL Database	Microsoft.SemanticKernel.Connectors.SqlServer	Microsoft.Data.SqlClient
Chroma	Microsoft.SemanticKernel.Connectors.Chroma	NA
DuckDB	Microsoft.SemanticKernel.Connectors.DuckDB	DuckDB.NET.Data.Full
Milvus	Microsoft.SemanticKernel.Connectors.Milvus	Milvus.Client
MongoDB Atlas Vector Search	Microsoft.SemanticKernel.Connectors.MongoDB	MongoDb.Driver
Pinecone	Microsoft.SemanticKernel.Connectors.Pinecone	REST API
Postgres	Microsoft.SemanticKernel.Connectors.Postgres	Npgsql
Qdrant	Microsoft.SemanticKernel.Connectors.Qdrant	Qdrant.Client
Redis	Microsoft.SemanticKernel.Connectors.Redis	StackExchange.Redis
Weaviate	Microsoft.SemanticKernel.Connectors.Weaviate	REST API

To discover .NET SDK and API support, visit the documentation for each respective service.

Related content

- Implement Azure OpenAI with RAG using vector search in a .NET app
- More Semantic Kernel .NET connectors

Prompt engineering in .NET

Article • 04/09/2025

In this article, you explore essential prompt engineering concepts. Many AI models are prompt-based, meaning they respond to user input text (a *prompt*) with a response generated by predictive algorithms (a *completion*). Newer models also often support completions in chat form, with messages based on roles (system, user, assistant) and chat history to preserve conversations.

Work with prompts

Consider this text generation example where *prompt* is the user input and *completion* is the model output:

Prompt: **"The president who served the shortest term was "**

Completion: "*Pedro Lascurain.*"

The completion appears correct, but what if your app is supposed to help U.S. history students? Pedro Lascurain's 45-minute term is the shortest term for any president, but he served as the president of Mexico. The U.S. history students are probably looking for "*William Henry Harrison*". Clearly, the app could be more helpful to its intended users if you gave it some context.

Prompt engineering adds context to the prompt by providing *instructions*, *examples*, and *cues* to help the model produce better completions.

Models that support text generation often don't require any specific format, but you should organize your prompts so it's clear what's an instruction and what's an example. Models that support chat-based apps use three roles to organize completions: a system role that controls the chat, a user role to represent user input, and an assistant role for responding to users. Divide your prompts into messages for each role:

- *System messages* give the model instructions about the assistant. A prompt can have only one system message, and it must be the first message.
- *User messages* include prompts from the user and show examples, historical prompts, or contain instructions for the assistant. An example chat completion must have at least one user message.
- *Assistant messages* show example or historical completions, and must contain a response to the preceding user message. Assistant messages aren't required, but if you include one it must be paired with a user message to form an example.

Use instructions to improve the completion

An instruction is text that tells the model how to respond. An instruction can be a directive or an imperative:

- *Directives* tell the model how to behave, but aren't simple commands—think character setup for an improv actor: **"You're helping students learn about U.S. history, so talk about the U.S. unless they specifically ask about other countries."**
- *Imperatives* are unambiguous commands for the model to follow. **"Translate to Tagalog:"**

Directives are more open-ended and flexible than imperatives:

- You can combine several directives in one instruction.
- Instructions usually work better when you use them with examples. However, because imperatives are unambiguous commands, models don't need examples to understand them (though you might use an example to show the model how to format responses). Because a directive doesn't tell the model exactly what to do, each example can help the model work better.
- It's usually better to break down a difficult instruction into a series of steps, which you can do with a sequence of directives. You should also tell the model to output the result of each step, so that you can easily make granular adjustments. Although you can break down the instruction into steps yourself, it's easier to just tell the model to do it, and to output the result of each step. This approach is called chain of thought prompting.

Primary and supporting content add context

You can provide content to add more context to instructions.

Primary content is text that you want the model to process with an instruction. Whatever action the instruction entails, the model will perform it on the primary content to produce a completion.

Supporting content is text that you refer to in an instruction, but which isn't the target of the instruction. The model uses the supporting content to complete the instruction, which means that supporting content also appears in completions, typically as some kind of structure (such as in headings or column labels).

Use labels with your instructional content to help the model figure out how to use it with the instruction. Don't worry too much about precision—labels don't have to match instructions exactly because the model will handle things like word form and capitalization.

Suppose you use the instruction **"Summarize US Presidential accomplishments"** to produce a list. The model might organize and order it in any number of ways. But what if you want the list

to group the accomplishments by a specific set of categories? Use supporting content to add that information to the instruction.

Adjust your instruction so the model groups by category, and append supporting content that specifies those categories:

C#

```
prompt = """
Instructions: Summarize US Presidential accomplishments, grouped by category.
Categories: Domestic Policy, US Economy, Foreign Affairs, Space Exploration.
Accomplishments: 'George Washington
- First president of the United States.
- First president to have been a military veteran.
- First president to be elected to a second term in office.
- Received votes from every presidential elector in an election.
- Filled the entire body of the United States federal judges; including the
Supreme Court.
- First president to be declared an honorary citizen of a foreign country, and an
honorary citizen of France.
John Adams ...' ///Text truncated
""";
```

Use examples to guide the model

An example is text that shows the model how to respond by providing sample user input and model output. The model uses examples to infer what to include in completions. Examples can come either before or after the instructions in an engineered prompt, but the two shouldn't be interspersed.

An example starts with a prompt and can optionally include a completion. A completion in an example doesn't have to include the verbatim response—it might just contain a formatted word, the first bullet in an unordered list, or something similar to indicate how each completion should start.

Examples are classified as zero-shot learning or few-shot learning based on whether they contain verbatim completions.

- **Zero-shot learning** examples include a prompt with no verbatim completion. This approach tests a model's responses without giving it example data output. Zero-shot prompts can have completions that include cues, such as indicating the model should output an ordered list by including **"1."** as the completion.
- **Few-shot learning** examples include several pairs of prompts with verbatim completions. Few-shot learning can change the model's behavior by adding to its existing knowledge.

Understand cues

A cue is text that conveys the desired structure or format of output. Like an instruction, a cue isn't processed by the model as if it were user input. Like an example, a cue shows the model what you want instead of telling it what to do. You can add as many cues as you want, so you can iterate to get the result you want. Cues are used with an instruction or an example and should be at the end of the prompt.

Suppose you use an instruction to tell the model to produce a list of presidential accomplishments by category, along with supporting content that tells the model what categories to use. You decide that you want the model to produce a nested list with all caps for categories, with each president's accomplishments in each category listed on one line that begins with their name, with presidents listed chronologically. After your instruction and supporting content, you could add three cues to show the model how to structure and format the list:

C#

```
prompt = """
Instructions: Summarize US Presidential accomplishments, grouped by category.
Categories: Domestic Policy, US Economy, Foreign Affairs, Space Exploration.
Accomplishments: George Washington
First president of the United States.
First president to have been a military veteran.
First president to be elected to a second term in office.
First president to receive votes from every presidential elector in an election.
First president to fill the entire body of the United States federal judges;
including the Supreme Court.
First president to be declared an honorary citizen of a foreign country, and an
honorary citizen of France.
John Adams ...  /// Text truncated

DOMESTIC POLICY
- George Washington:
- John Adams:
""";
```

- **DOMESTIC POLICY** shows the model that you want it to start each group with the category in all caps.
- **- George Washington:** shows the model to start each section with George Washington's accomplishments listed on one line.
- **- John Adams:** shows the model that it should list remaining presidents in chronological order.

Example prompt using .NET

.NET provides various tools to prompt and chat with different AI models. Use Semantic Kernel to connect to a wide variety of AI models and services, as well as other SDKs such as the official OpenAI .NET library. Semantic Kernel includes tools to create prompts with different roles and maintain chat history, as well as many other features.

Consider the following code example:

C#

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;

// Create a kernel with OpenAI chat completion
#pragma warning disable SKEXP0010
Kernel kernel = Kernel.CreateBuilder()
                    .AddOpenAIChatCompletion(
                        modelId: "phi3:mini",
                        endpoint: new Uri("http://localhost:11434"),
                        apiKey: "")
                    .Build();

var aiChatService = kernel.GetRequiredService<IChatCompletionService>();
var chatHistory = new ChatHistory();
chatHistory.Add(
    new ChatMessageContent(AuthorRole.System, "You are a helpful AI Assistant."));

while (true)
{
    // Get user prompt and add to chat history
    Console.WriteLine("Your prompt:");
    chatHistory.Add(new ChatMessageContent(AuthorRole.User, Console.ReadLine()));

    // Stream the AI response and add to chat history
    Console.WriteLine("AI Response:");
    var response = "";
    await foreach (var item in
        aiChatService.GetStreamingChatMessageContentsAsync(chatHistory))
    {
        Console.Write(item.Content);
        response += item.Content;
    }
    chatHistory.Add(new ChatMessageContent(AuthorRole.Assistant, response));
    Console.WriteLine();
}
```

The preceding code provides examples of the following concepts:

- Creates a chat history service to prompt the AI model for completions based on author roles.
- Configures the AI with an `AuthorRole.System` message.

- Accepts user input to allow for different types of prompts in the context of an `AuthorRole.User`.
- Asynchronously streams the completion from the AI to provide a dynamic chat experience.

Extend your prompt engineering techniques

You can also increase the power of your prompts with more advanced prompt engineering techniques that are covered in depth in their own articles.

- LLMs have token input limits that constrain the amount of text you can fit in a prompt. Use embeddings and vector database solutions to reduce the number of tokens you need to represent a given piece of text.
- LLMs aren't trained on your data unless you train them yourself, which can be costly and time-consuming. Use retrieval augmented generation (RAG) to make your data available to an LLM without training it.

Related content

- Prompt engineering techniques
- Configure prompts in Semantic Kernel

Chain of thought prompting

05/29/2025

GPT model performance and response quality benefits from *prompt engineering*, which is the practice of providing instructions and examples to a model to prime or refine its output. As they process instructions, models make more reasoning errors when they try to answer right away rather than taking time to work out an answer. You can help the model reason its way toward correct answers more reliably by asking for the model to include its chain of thought—that is, the steps it took to follow an instruction, along with the results of each step.

Chain of thought prompting is the practice of prompting a model to perform a task step-by-step and to present each step and its result in order in the output. This simplifies prompt engineering by offloading some execution planning to the model, and makes it easier to connect any problem to a specific step so you know where to focus further efforts.

It's generally simpler to just instruct the model to include its chain of thought, but you can use examples to show the model how to break down tasks. The following sections show both ways.

Use chain of thought prompting in instructions

To use an instruction for chain of thought prompting, include a directive that tells the model to perform the task step-by-step and to output the result of each step.

C#

```
prompt= """Instructions: Compare the pros and cons of EVs and petroleum-fueled
vehicles.
Break the task into steps, and output the result of each step as you perform
it.""";
```

Use chain of thought prompting in examples

You can use examples to indicate the steps for chain of thought prompting, which the model will interpret to mean it should also output step results. Steps can include formatting cues.

C#

```
prompt= """
        Instructions: Compare the pros and cons of EVs and petroleum-fueled
vehicles.

        Differences between EVs and petroleum-fueled vehicles:
        -
```

```
        Differences ordered according to overall impact, highest-impact first:
        1.

        Summary of vehicle type differences as pros and cons:
        Pros of EVs
        1.
        Pros of petroleum-fueled vehicles
        1.
        """;
```

Related content

- Prompt engineering techniques

Zero-shot and few-shot learning

05/29/2025

This article explains zero-shot learning and few-shot learning for prompt engineering in .NET, including their primary use cases.

GPT model performance benefits from *prompt engineering*, the practice of providing instructions and examples to a model to refine its output. Zero-shot learning and few-shot learning are techniques you can use when providing examples.

Zero-shot learning

Zero-shot learning is the practice of passing prompts that aren't paired with verbatim completions, although you can include completions that consist of cues. Zero-shot learning relies entirely on the model's existing knowledge to generate responses, which reduces the number of tokens created and can help you control costs. However, zero-shot learning doesn't add to the model's knowledge or context.

Here's an example zero-shot prompt that tells the model to evaluate user input to determine which of four possible intents the input represents, and then to preface the response with "**Intent:** ".

C#

```
prompt = $"""
Instructions: What is the intent of this request?
If you don't know the intent, don't guess; instead respond with "Unknown".
Choices: SendEmail, SendMessage, CompleteTask, CreateDocument, Unknown.
User Input: {request}
Intent:
""";
```

There are two primary use cases for zero-shot learning:

- **Work with fined-tuned LLMs** - Because it relies on the model's existing knowledge, zero-shot learning is not as resource-intensive as few-shot learning, and it works well with LLMs that have already been fined-tuned on instruction datasets. You might be able to rely solely on zero-shot learning and keep costs relatively low.
- **Establish performance baselines** - Zero-shot learning can help you simulate how your app would perform for actual users. This lets you evaluate various aspects of your model's current performance, such as accuracy or precision. In this case, you typically use zero-shot learning to establish a performance baseline and then experiment with few-shot learning to improve performance.

Few-shot learning

Few-shot learning is the practice of passing prompts paired with verbatim completions (few-shot prompts) to show your model how to respond. Compared to zero-shot learning, this means few-shot learning produces more tokens and causes the model to update its knowledge, which can make few-shot learning more resource-intensive. However, few-shot learning also helps the model produce more relevant responses.

C#

```
prompt = $"""
Instructions: What is the intent of this request?
If you don't know the intent, don't guess; instead respond with "Unknown".
Choices: SendEmail, SendMessage, CompleteTask, CreateDocument, Unknown.

User Input: Can you send a very quick approval to the marketing team?
Intent: SendMessage

User Input: Can you send the full update to the marketing team?
Intent: SendEmail

User Input: {request}
Intent:
""";
```

Few-shot learning has two primary use cases:

- **Tuning an LLM** - Because it can add to the model's knowledge, few-shot learning can improve a model's performance. It also causes the model to create more tokens than zero-shot learning does, which can eventually become prohibitively expensive or even infeasible. However, if your LLM isn't fined-tuned yet, you won't always get good performance with zero-shot prompts, and few-shot learning is warranted.
- **Fixing performance issues** - You can use few-shot learning as a follow-up to zero-shot learning. In this case, you use zero-shot learning to establish a performance baseline, and then experiment with few-shot learning based on the zero-shot prompts you used. This lets you add to the model's knowledge after seeing how it currently responds, so you can iterate and improve performance while minimizing the number of tokens you introduce.

Caveats

- Example-based learning doesn't work well for complex reasoning tasks. However, adding instructions can help address this.
- Few-shot learning requires creating lengthy prompts. Prompts with large number of tokens can increase computation and latency. This typically means increased costs.

There's also a limit to the length of the prompts.

- When you use several examples the model can learn false patterns, such as "Sentiments are twice as likely to be positive than negative."

Related content

- Prompt engineering techniques
- How GenAI and LLMs work

Retrieval-augmented generation (RAG) provides LLM knowledge

05/29/2025

This article describes how retrieval-augmented generation lets LLMs treat your data sources as knowledge without having to train.

LLMs have extensive knowledge bases through training. For most scenarios, you can select an LLM that is designed for your requirements, but those LLMs still require additional training to understand your specific data. Retrieval-augmented generation lets you make your data available to LLMs without training them on it first.

How RAG works

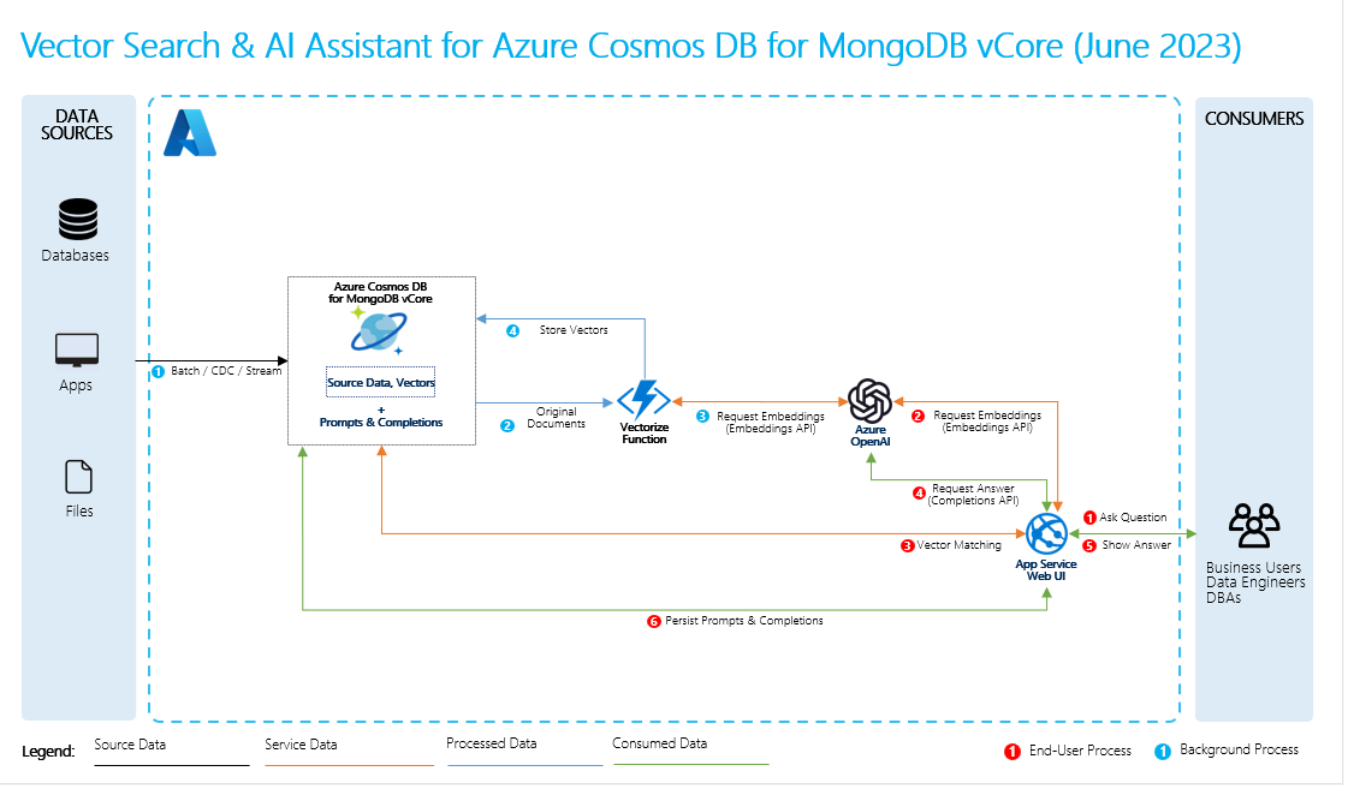
To perform retrieval-augmented generation, you create embeddings for your data along with common questions about it. You can do this on the fly or you can create and store the embeddings by using a vector database solution.

When a user asks a question, the LLM uses your embeddings to compare the user's question to your data and find the most relevant context. This context and the user's question then go to the LLM in a prompt, and the LLM provides a response based on your data.

Basic RAG process

To perform RAG, you must process each data source that you want to use for retrievals. The basic process is as follows:

1. Chunk large data into manageable pieces.
2. Convert the chunks into a searchable format.
3. Store the converted data in a location that allows efficient access. Additionally, it's important to store relevant metadata for citations or references when the LLM provides responses.
4. Feed your converted data to LLMs in prompts.



- **Source data**: This is where your data exists. It could be a file/folder on your machine, a file in cloud storage, an Azure Machine Learning data asset, a Git repository, or an SQL database.
- **Data chunking**: The data in your source needs to be converted to plain text. For example, word documents or PDFs need to be cracked open and converted to text. The text is then chunked into smaller pieces.
- **Converting the text to vectors**: These are embeddings. Vectors are numerical representations of concepts converted to number sequences, which make it easy for computers to understand the relationships between those concepts.
- **Links between source data and embeddings**: This information is stored as metadata on the chunks you created, which are then used to help the LLMs generate citations while generating responses.

Related content

- Prompt engineering

Understand OpenAI function calling

05/29/2025

Function calling is an OpenAI model feature that lets you describe functions and their arguments in prompts using JSON. Instead of invoking the function itself, the model returns a JSON output describing what functions should be called and the arguments to use.

Function calling simplifies how you connect external tools to your AI model. First, you specify each tool's functions to the model. Then the model decides which functions should be called, based on the prompt question. The model uses the function call results to build a more accurate and consistent response.

Potential use cases for function calling include:

- Answering questions by calling external APIs, for example, sending emails or getting the weather forecast.
- Answering questions with info from an internal datastore, for example, aggregating sales data to answer, "What are my best-selling products?".
- Creating structured data from text info, for example, building a user info object with details from the chat history.

Call functions with OpenAI

The general steps for calling functions with an OpenAI model are:

1. Send the user's question as a request with functions defined in the tools parameters.
2. The model decides which functions, if any, to call. The output contains a JSON object that lists the function calls and their arguments.

   > **Note**
   >
   > The model might hallucinate additional arguments.

3. Parse the output and call the requested functions with their specified arguments.
4. Send another request with the function results included as a new message.
5. The model responds with more function call requests or an answer to the user's question.

   - Continue invoking the requested function calls until the model responds with an answer.

You can force the model to request a specific function by setting the `tool_choice` parameter to the function's name. You can also force the model to respond with a message for the user by

setting the `tool_choice` parameter to `"none"`.

Call functions in parallel

Some models support parallel function calling, which enables the model to request multiple function calls in one output. The results of each function call are included together in one response back to the model. Parallel function calling reduces the number of API requests and time needed to generate an answer. Each function result is included as a new message in the conversation with a `tool_call_id` matching the `id` of the function call request.

Supported models

Not all OpenAI models are trained to support function calling. For a list of models that support function calling or parallel function calling, see OpenAI - Supported Models.

Function calling with the Semantic Kernel SDK

The Semantic Kernel SDK supports describing which functions are available to your AI using the KernelFunction decorator.

The Kernel builds the `tools` parameter of a request based on your decorators, orchestrates the requested function calls to your code, and returns results back to the model.

Token counts

Function descriptions are include in the system message of your request to a model. These function descriptions count against your model's token limit and are included in the cost of the request.

If your request exceeds the model's token limit, try the following modifications:

- Reduce the number of functions.
- Shorten the function and argument descriptions in your JSON.

Related content

- Understanding tokens
- Creating native functions for AI to call
- Prompt engineering

Get started with the 'Chat using your own data sample' for .NET

05/28/2025

This article shows you how to deploy and run the Chat with your own data sample for .NET. This sample implements a chat app using C#, Azure OpenAI Service, and Retrieval Augmented Generation (RAG) in Azure AI Search to get answers about employee benefits at a fictitious company. The employee benefits chat app is seeded with PDF files including an employee handbook, a benefits document and a list of company roles and expectations.

- Demo video

By following the instructions in this article, you will:

- Deploy a chat app to Azure.
- Get answers about employee benefits.
- Change settings to change behavior of responses.

Once you complete this procedure, you can start modifying the new project with your custom code.

This article is part of a collection of articles that show you how to build a chat app using Azure Open AI Service and Azure AI Search.

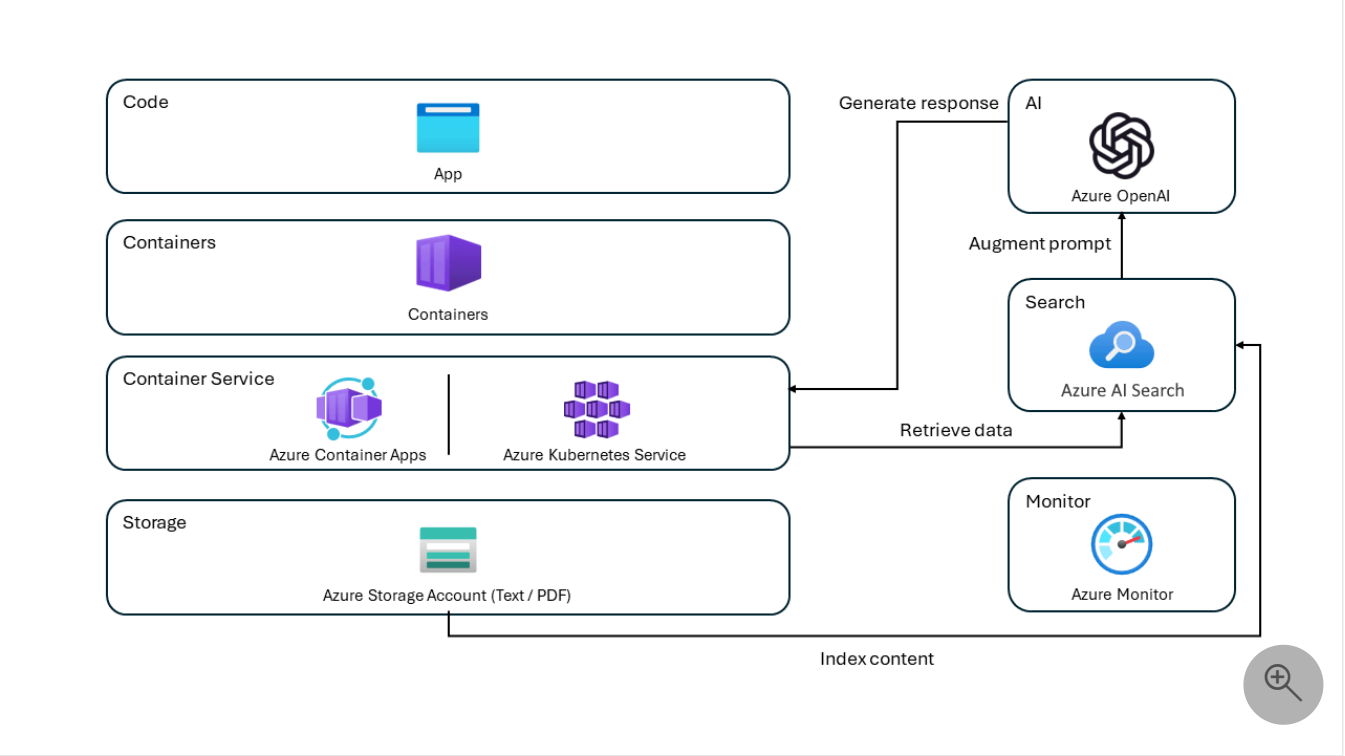
Other articles in the collection include:

- Python
- JavaScript
- Java

Architectural overview

In this sample application, a fictitious company called Contoso Electronics provides the chat app experience to its employees to ask questions about the benefits, internal policies, and job descriptions and roles.

The architecture of the chat app is shown in the following diagram:



- **User interface** - The application's chat interface is a Blazor WebAssembly application. This interface is what accepts user queries, routes request to the application backend, and displays generated responses.
- **Backend** - The application backend is an ASP.NET Core Minimal API. The backend hosts the Blazor static web application and is what orchestrates the interactions among the different services. Services used in this application include:
  - **Azure Cognitive Search** – Indexes documents from the data stored in an Azure Storage Account. This makes the documents searchable using vector search capabilities.
  - **Azure OpenAI Service** – Provides the Large Language Models (LLM) to generate responses. Semantic Kernel is used in conjunction with the Azure OpenAI Service to orchestrate the more complex AI workflows.

Cost

Most resources in this architecture use a basic or consumption pricing tier. Consumption pricing is based on usage, which means you only pay for what you use. To complete this article, there will be a charge, but it will be minimal. When you are done with the article, you can delete the resources to stop incurring charges.

For more information, see Azure Samples: Cost in the sample repo .

Prerequisites

A development container environment is available with all dependencies required to complete this article. You can run the development container in GitHub Codespaces (in a

browser) or locally using Visual Studio Code.

To follow along with this article, you need the following prerequisites:

Codespaces (recommended)

- An Azure subscription - Create one for free
- Azure account permissions - Your Azure account must have Microsoft.Authorization/roleAssignments/write permissions, such as User Access Administrator or Owner.
- GitHub account

Open development environment

Begin now with a development environment that has all the dependencies installed to complete this article.

GitHub Codespaces (recommended)

GitHub Codespaces runs a development container managed by GitHub with Visual Studio Code for the Web as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.

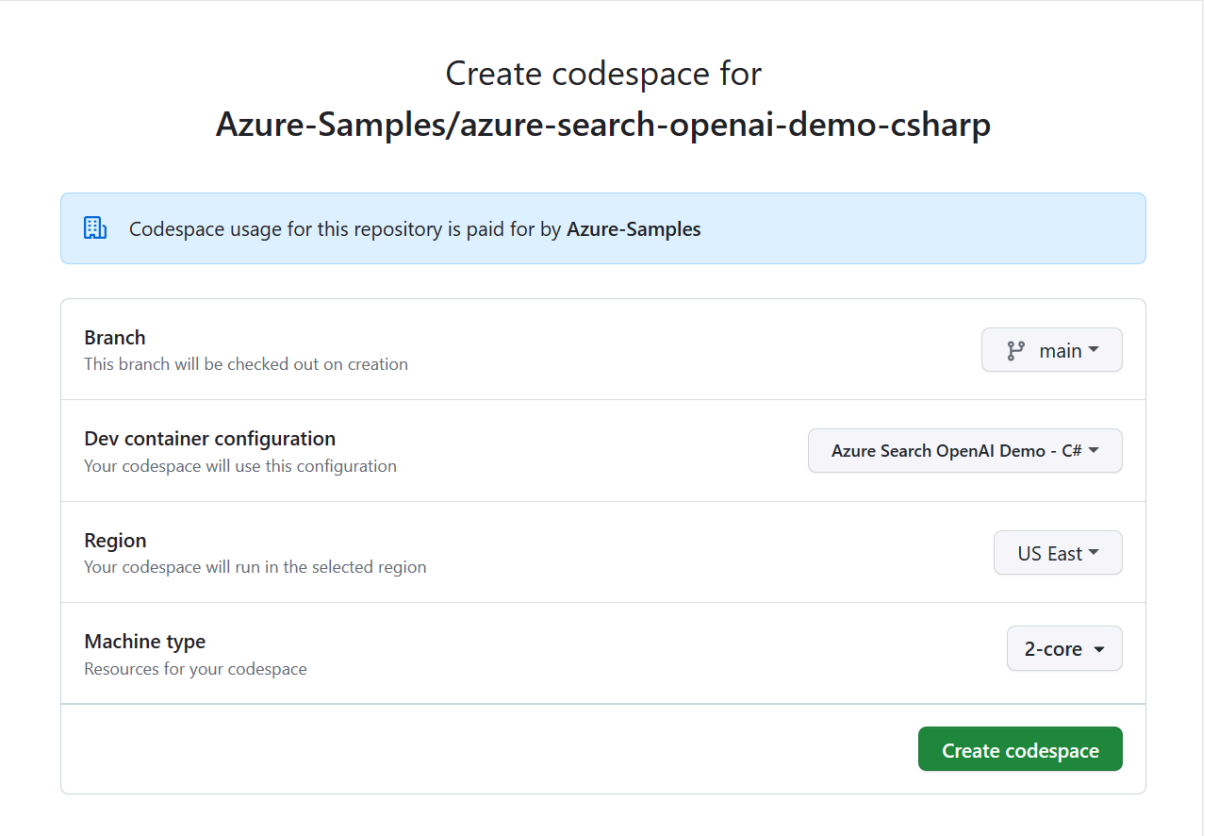
> **Important**
>
> All GitHub accounts can use Codespaces for up to 60 hours free each month with 2 core instances. For more information, see GitHub Codespaces monthly included storage and core hours.

1. Start the process to create a new GitHub codespace on the `main` branch of the Azure-Samples/azure-search-openai-demo-csharp GitHub repository.

2. To have both the development environment and the documentation available at the same time, right-click on the following **Open in GitHub Codespaces** button, and select *Open link in new windows*.

   Open in GitHub Codespaces

3. On the **Create codespace** page, review the codespace configuration settings and then select **Create new codespace**:



4. Wait for the codespace to start. This startup process can take a few minutes.

5. In the terminal at the bottom of the screen, sign in to Azure with the Azure Developer CLI.

```bash
azd auth login
```

6. Copy the code from the terminal and then paste it into a browser. Follow the instructions to authenticate with your Azure account.

7. The remaining tasks in this article take place in the context of this development container.

Deploy and run

The sample repository contains all the code and configuration files you need to deploy a chat app to Azure. The following steps walk you through the process of deploying the sample to Azure.

Deploy chat app to Azure

> **ⓘ Important**
>
> Azure resources created in this section incur immediate costs, primarily from the Azure AI Search resource. These resources may accrue costs even if you interrupt the command before it is fully executed.

1. Run the following Azure Developer CLI command to provision the Azure resources and deploy the source code:

   ```
   Bash

   azd up
   ```

2. When you're prompted to enter an environment name, keep it short and lowercase. For example, `myenv`. Its used as part of the resource group name.

3. When prompted, select a subscription to create the resources in.

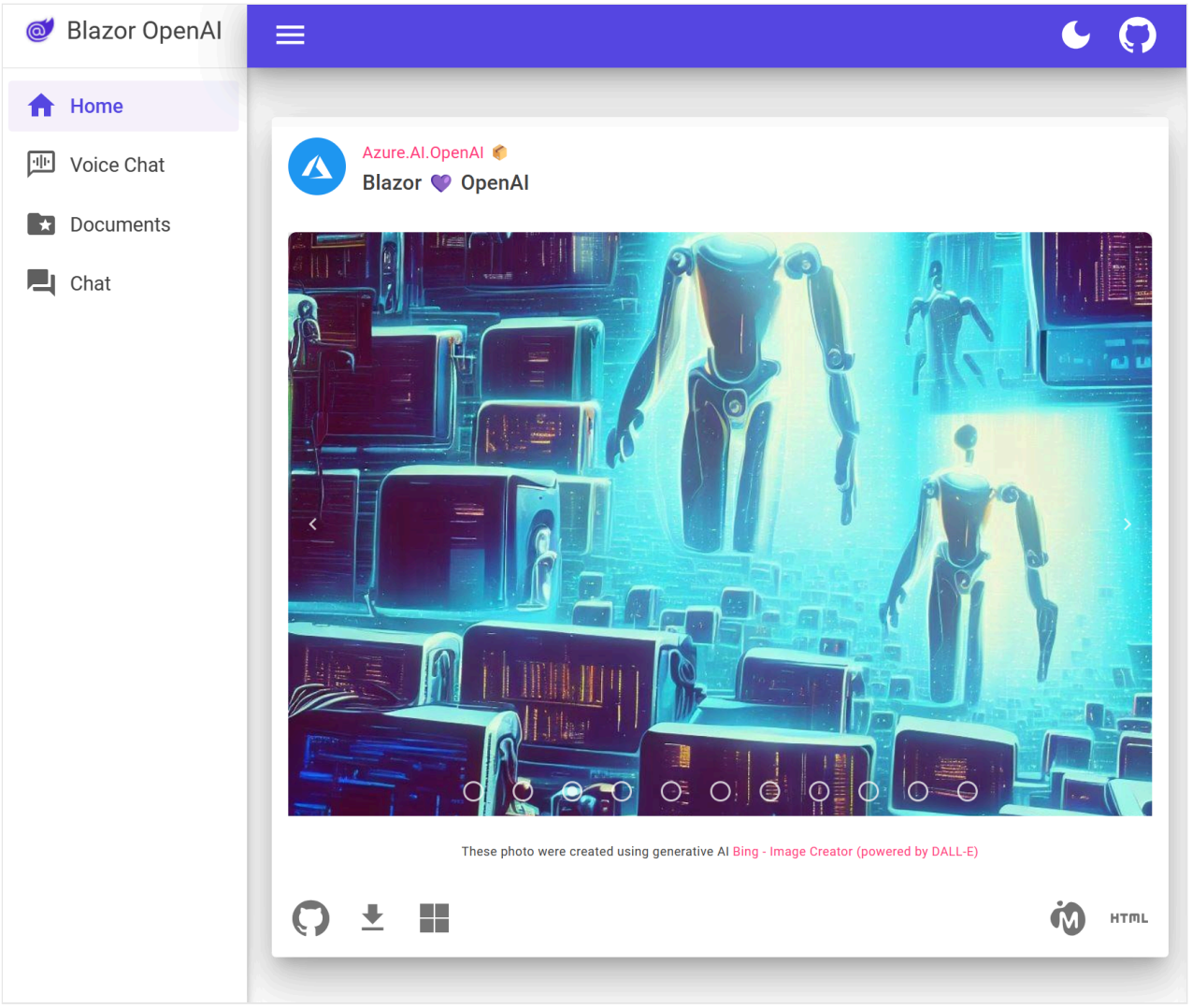
4. When you're prompted to select a location the first time, select a location near you. This location is used for most the resources including hosting.

5. If you're prompted for a location for the OpenAI model, select a location that is near you. If the same location is available as your first location, select that.

6. Wait until app is deployed. It may take up to 20 minutes for the deployment to complete.

7. After the application has been successfully deployed, you see a URL displayed in the terminal.

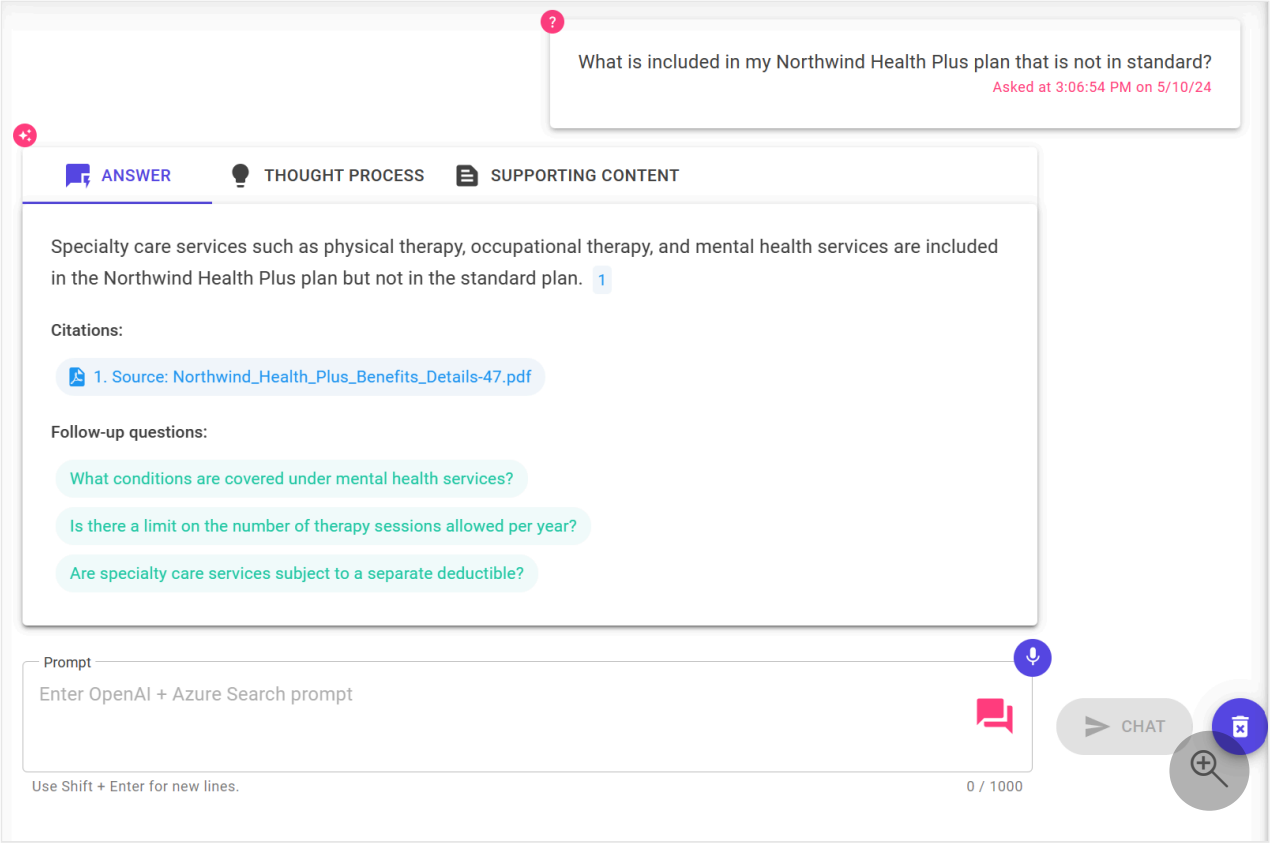
8. Select that URL labeled `Deploying service web` to open the chat application in a browser.



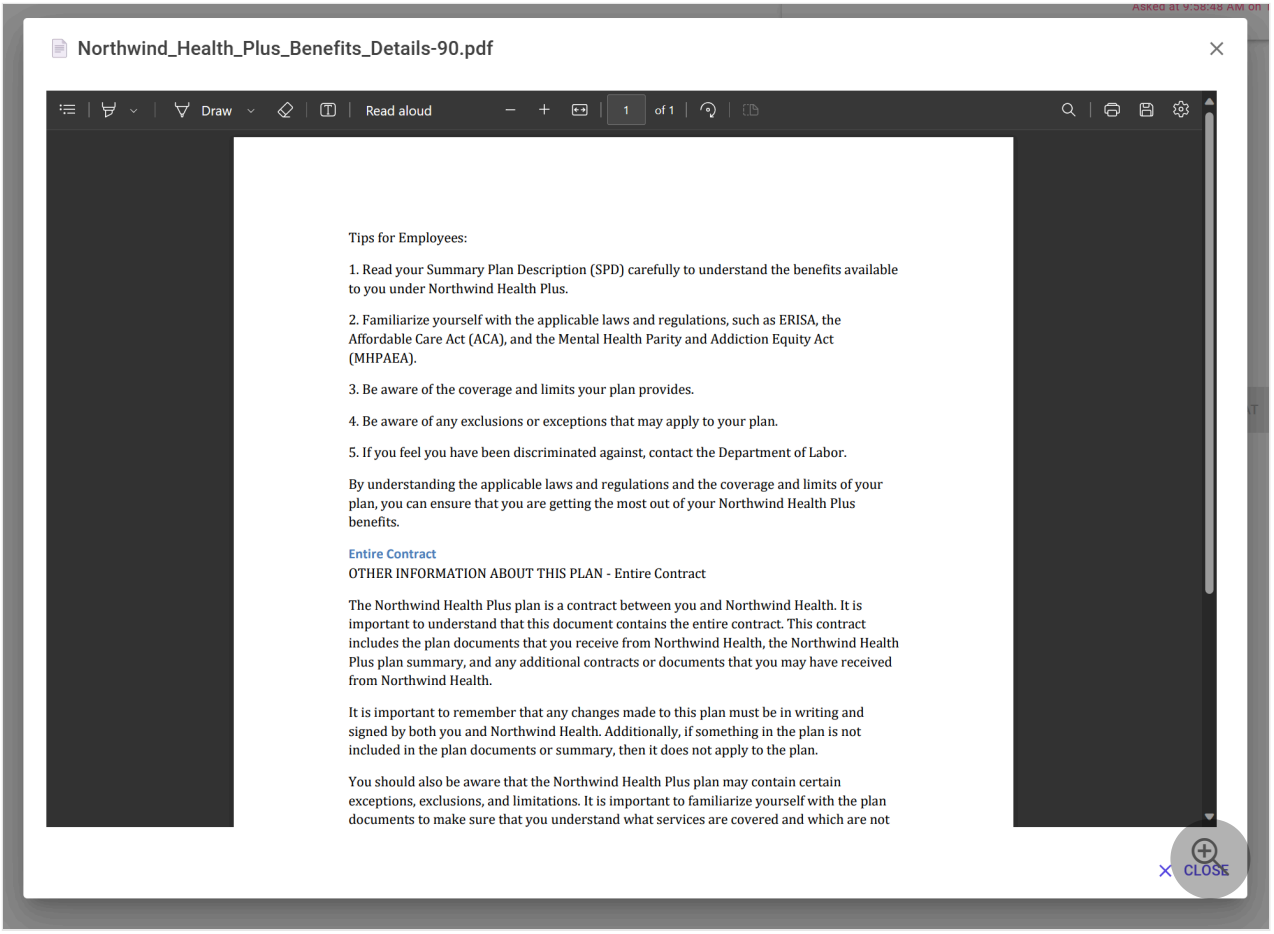
Use chat app to get answers from PDF files

The chat app is preloaded with employee benefits information from PDF files. You can use the chat app to ask questions about the benefits. The following steps walk you through the process of using the chat app.

1. In the browser, navigate to the **Chat** page using the left navigation.
2. Select or enter "What is included in my Northwind Health Plus plan that is not in standard?" in the chat text box. Your response is *similar* to the following image.



3. From the answer, select a citation. A pop-up window will open displaying the source of the information.



4. Navigate between the tabs at the top of the answer box to understand how the answer was generated.

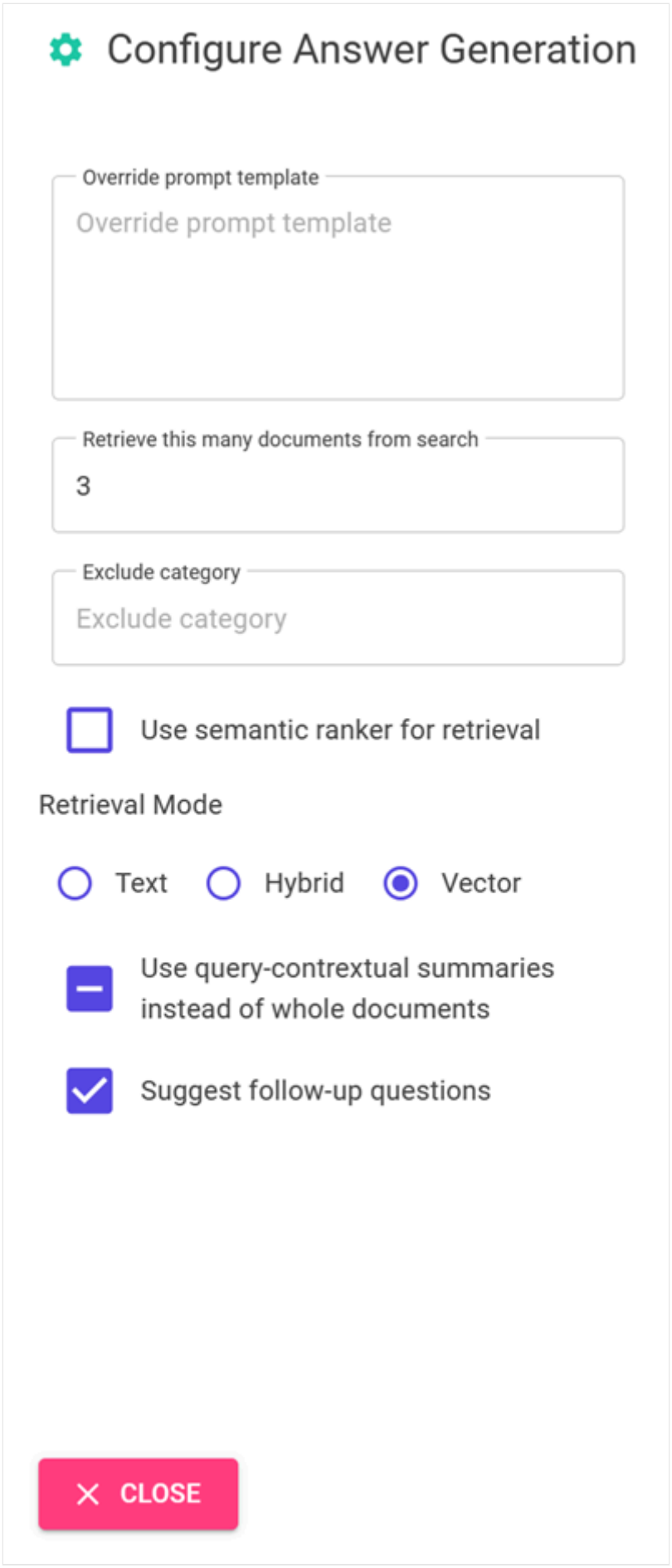
Expand table

Tab	Description
Thought process	This is a script of the interactions in chat. You can view the system prompt (`content`) and your user question (`content`).
Supporting content	This includes the information to answer your question and the source material. The number of source material citations is noted in the **Developer settings**. The default value is **3**.
Citation	This displays the source page that contains the citation.

5. When you're done, navigate back to the answer tab.

Use chat app settings to change behavior of responses

The intelligence of the chat is determined by the OpenAI model and the settings that are used to interact with the model.



Expand table

Setting	Description
Override prompt template	This is the prompt that is used to generate the answer.
Retrieve this many search results	This is the number of search results that are used to generate the answer. You can see these sources returned in the *Thought process* and *Supporting content* tabs of the citation.
Exclude category	This is the category of documents that are excluded from the search results.
Use semantic ranker for retrieval	This is a feature of Azure AI Search that uses machine learning to improve the relevance of search results.
Retrieval mode	**Vectors + Text** means that the search results are based on the text of the documents and the embeddings of the documents. **Vectors** means that the search results are based on the embeddings of the documents. **Text** means that the search results are based on the text of the documents.
Use query-contextual summaries instead of whole documents	When both `Use semantic ranker` and `Use query-contextual summaries` are checked, the LLM uses captions extracted from key passages, instead of all the passages, in the highest ranked documents.
Suggest follow-up questions	Have the chat app suggest follow-up questions based on the answer.

The following steps walk you through the process of changing the settings.

1. In the browser, select the gear icon in the upper right of the page.
2. If not selected, select the **Suggest follow-up questions** checkbox and ask the same question again.

   ```
   What is included in my Northwind Health Plus plan that is not in standard?
   ```

   The chat might return with follow-up question suggestions.
3. In the **Settings** tab, deselect **Use semantic ranker for retrieval**.
4. Ask the same question again.

   ```
   What is my deductible?
   ```
5. What is the difference in the answers?

The response that used the Semantic ranker provided a single answer. The response without semantic ranking returned a less direct answer.

Clean up resources

To finish, clean up the Azure and GitHub CodeSpaces resources you used.

Clean up Azure resources

The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Run the following Azure Developer CLI command to delete the Azure resources and remove the source code:

```bash
azd down --purge
```

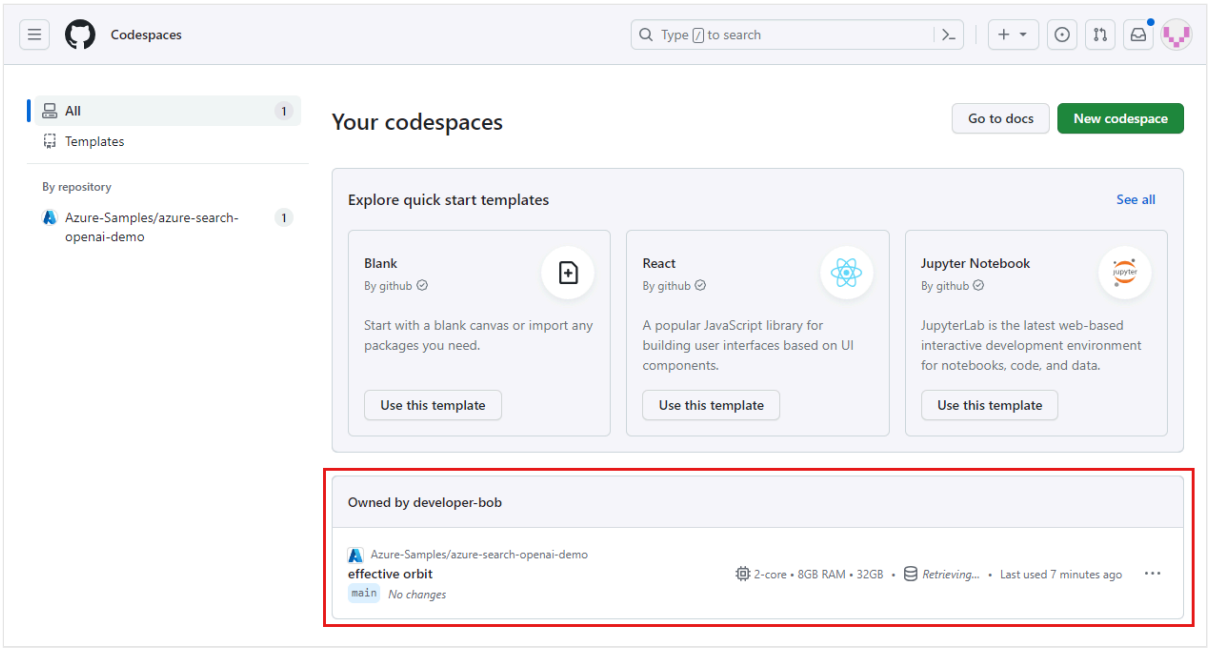
Clean up GitHub Codespaces

GitHub Codespaces

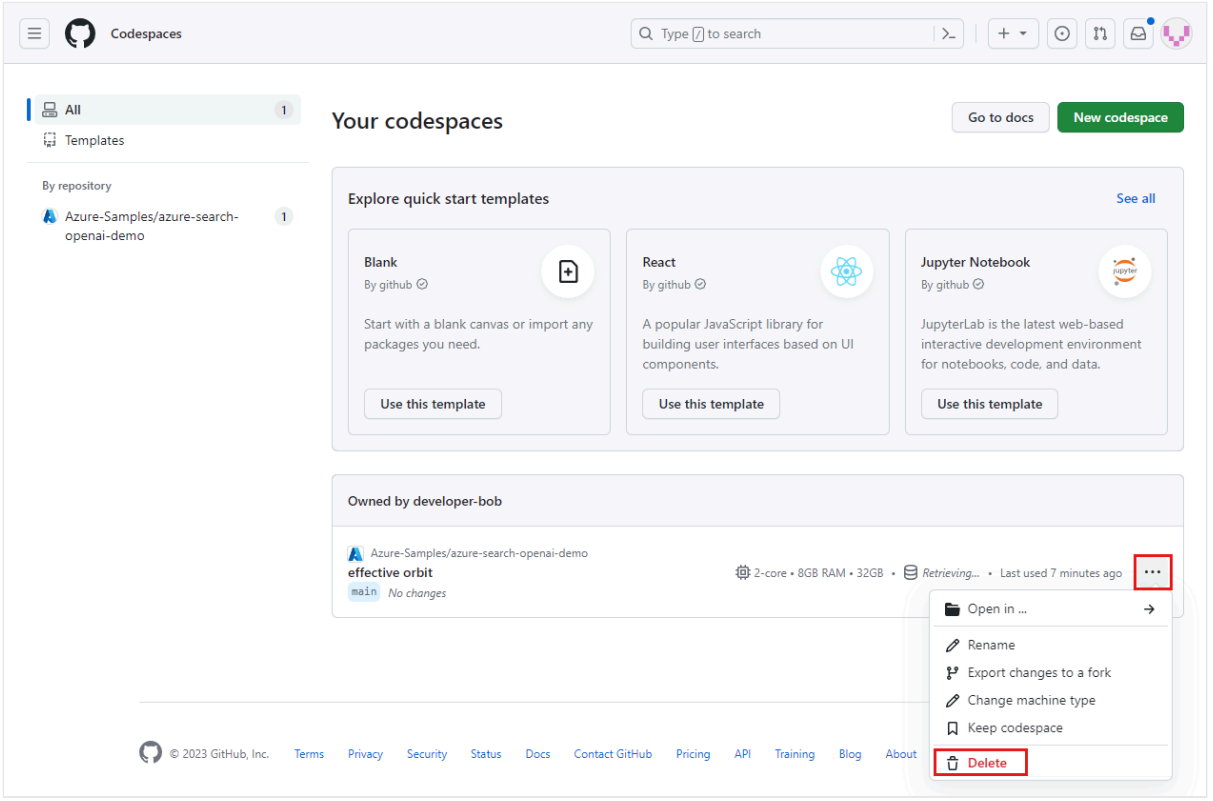
Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement you get for your account.

> **Important**
>
> For more information about your GitHub account's entitlements, see GitHub Codespaces monthly included storage and core hours.

1. Sign into the GitHub Codespaces dashboard (https://github.com/codespaces).
2. Locate your currently running codespaces sourced from the Azure-Samples/azure-search-openai-demo-csharp GitHub repository.



3. Open the context menu for the codespace and then select **Delete**.



Get help

This sample repository offers troubleshooting information.

If your issue isn't addressed, log your issue to the repository's Issues.

Next steps

- Get the source code for the sample used in this article
- Build a chat app with Azure OpenAI best practice solution architecture
- Access control in Generative AI Apps with Azure AI Search
- Build an Enterprise ready OpenAI solution with Azure API Management
- Outperforming vector search with hybrid retrieval and ranking capabilities

Implement Azure OpenAI with RAG using vector search in a .NET app

Article • 11/24/2024

This tutorial explores integration of the RAG pattern using Open AI models and vector search capabilities in a .NET app. The sample application performs vector searches on custom data stored in Azure Cosmos DB for MongoDB and further refines the responses using generative AI models, such as GPT-35 and GPT-4. In the sections that follow, you'll set up a sample application and explore key code examples that demonstrate these concepts.

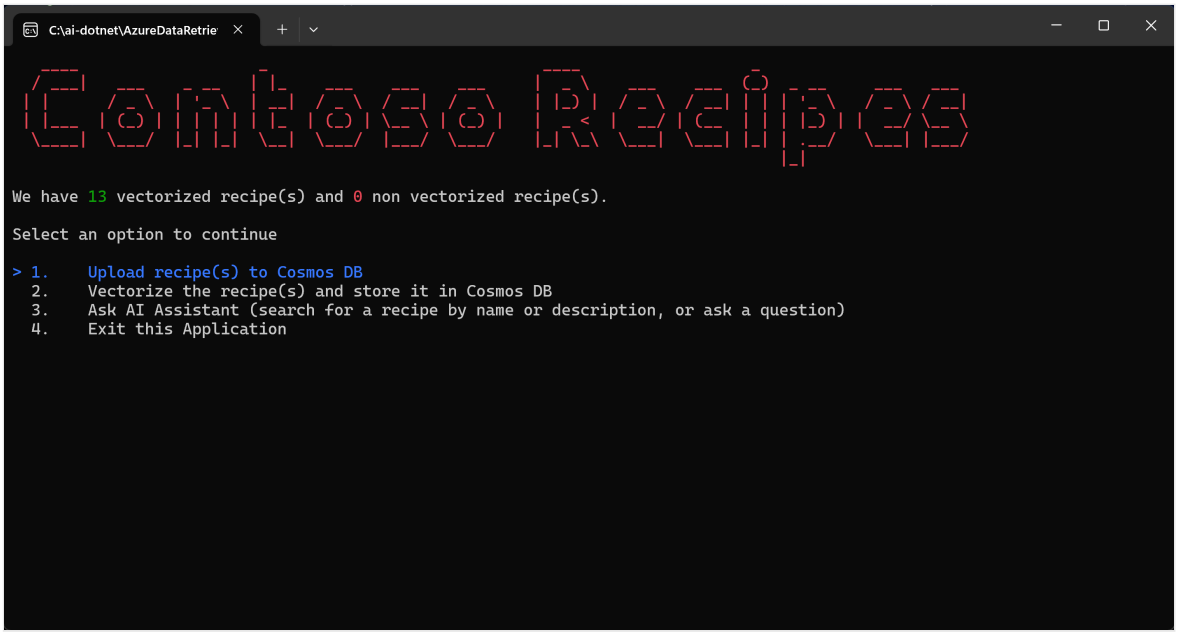
Prerequisites

- .NET 8.0 installed
- An Azure Account
- An Azure Cosmos DB for MongoDB vCore service
- An Azure Open AI service
  - Deploy `text-embedding-ada-002` model for embeddings
  - Deploy `gpt-35-turbo` model for chat completions

App overview

The Cosmos Recipe Guide app allows you to perform vector and AI driven searches against a set of recipe data. You can search directly for available recipes or prompt the app with ingredient names to find related recipes. The app and the sections ahead guide you through the following workflow to demonstrate this type of functionality:

1. Upload sample data to an Azure Cosmos DB for MongoDB database.
2. Create embeddings and a vector index for the uploaded sample data using the Azure OpenAI `text-embedding-ada-002` model.
3. Perform vector similarity search based on the user prompts.
4. Use the Azure OpenAI `gpt-35-turbo` completions model to compose more meaningful answers based on the search results data.



Get started

1. Clone the following GitHub repository:

   Bash

   ```
   git clone
   https://github.com/microsoft/AzureDataRetrievalAugmentedGenerationSampl
   es.git
   ```

2. In the *C#/CosmosDB-MongoDBvCore* folder, open the **CosmosRecipeGuide.sln** file.

3. In the *appsettings.json* file, replace the following config values with your Azure OpenAI and Azure CosmosDB for MongoDb values:

   JSON

   ```
   "OpenAIEndpoint": "https://<your-service-name>.openai.azure.com/",
   "OpenAIKey": "<your-api-key>",
   "OpenAIEmbeddingDeployment": "<your-ada-deployment-name>",
   "OpenAIcompletionsDeployment": "<your-gpt-deployment-name>",
   "MongoVcoreConnection": "<your-mongo-connection-string>"
   ```

4. Launch the app by pressing the **Start** button at the top of Visual Studio.

Explore the app

When you run the app for the first time, it connects to Azure Cosmos DB and reports that there are no recipes available yet. Follow the steps displayed by the app to begin

the core workflow.

1. Select **Upload recipe(s) to Cosmos DB** and press `Enter`. This command reads sample JSON files from the local project and uploads them to the Cosmos DB account.

   The code from the *Utility.cs* class parses the local JSON files.

   C#

   ```
   public static List<Recipe> ParseDocuments(string Folderpath)
   {
       List<Recipe> recipes = new List<Recipe>();

       Directory.GetFiles(Folderpath)
           .ToList()
           .ForEach(f =>
           {
               var jsonString= System.IO.File.ReadAllText(f);
               Recipe recipe = JsonConvert.DeserializeObject<Recipe>
   (jsonString);
               recipe.id = recipe.name.ToLower().Replace(" ", "");
               ret.Add(recipe);
           }
       );

       return recipes;
   }
   ```

   The `UpsertVectorAsync` method in the *VCoreMongoService.cs* file uploads the documents to Azure Cosmos DB for MongoDB.

   C#

   ```
   public async Task UpsertVectorAsync(Recipe recipe)
       {
           BsonDocument document = recipe.ToBsonDocument();

           if (!document.Contains("_id"))
           {
               Console.WriteLine("UpsertVectorAsync: Document does not
   contain _id.");
               throw new ArgumentException("UpsertVectorAsync: Document
   does not contain _id.");
           }

           string? _idValue = document["_id"].ToString();

           try
           {
               var filter = Builders<BsonDocument>.Filter.Eq("_id",
   ```

```
_idValue);
            var options = new ReplaceOptions { IsUpsert = true };
            await _recipeCollection.ReplaceOneAsync(filter, document,
options);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception: UpsertVectorAsync():
{ex.Message}");
            throw;
        }
    }
```

2. Select **Vectorize the recipe(s) and store them in Cosmos DB**.

   The JSON items uploaded to Cosmos DB do not contain embeddings and therefore are not optimized for RAG via vector search. An embedding is an information-dense, numerical representation of the semantic meaning of a piece of text. Vector searches are able to find items with contextually similar embeddings.

   The `GetEmbeddingsAsync` method in the *OpenAIService.cs* file creates an embedding for each item in the database.

   C#

```
public async Task<float[]?> GetEmbeddingsAsync(dynamic data)
{
    try
    {
        EmbeddingsOptions options = new EmbeddingsOptions(data)
        {
            Input = data
        };

        var response = await
_openAIClient.GetEmbeddingsAsync(openAIEmbeddingDeployment, options);

        Embeddings embeddings = response.Value;
        float[] embedding = embeddings.Data[0].Embedding.ToArray();

        return embedding;
    }
    catch (Exception ex)
    {
        Console.WriteLine($"GetEmbeddingsAsync Exception:
{ex.Message}");
        return null;
    }
}
```

The `CreateVectorIndexIfNotExists` in the *VCoreMongoService.cs* file creates a vector index, which enables you to perform vector similarity searches.

C#

```
public void CreateVectorIndexIfNotExists(string vectorIndexName)
{
    try
    {
        //Find if vector index exists in vectors collection
        using (IAsyncCursor<BsonDocument> indexCursor =
_recipeCollection.Indexes.List())
        {
            bool vectorIndexExists = indexCursor.ToList().Any(x =>
x["name"] == vectorIndexName);
            if (!vectorIndexExists)
            {
                BsonDocumentCommand<BsonDocument> command = new
BsonDocumentCommand<BsonDocument>(
                    BsonDocument.Parse(@"
                        { createIndexes: 'Recipe',
                            indexes: [{
                            name: 'vectorSearchIndex',
                            key: { embedding: 'cosmosSearch' },
                            cosmosSearchOptions: {
                                kind: 'vector-ivf',
                                numLists: 5,
                                similarity: 'COS',
                                dimensions: 1536 }
                            }]
                        }"));

                BsonDocument result = _database.RunCommand(command);
                if (result["ok"] != 1)
                {
                    Console.WriteLine("CreateIndex failed with
response: " + result.ToJson());
                }
            }
        }
    }
    catch (MongoException ex)
    {
        Console.WriteLine("MongoDbService InitializeVectorIndex: " +
ex.Message);
        throw;
    }
}
```

3. Select the **Ask AI Assistant (search for a recipe by name or description, or ask a question)** option in the application to run a user query.

The user query is converted to an embedding using the Open AI service and the embedding model. The embedding is then sent to Azure Cosmos DB for MongoDB and is used to perform a vector search. The `VectorSearchAsync` method in the *VCoreMongoService.cs* file performs a vector search to find vectors that are close to the supplied vector and returns a list of documents from Azure Cosmos DB for MongoDB vCore.

C#

```
public async Task<List<Recipe>> VectorSearchAsync(float[] queryVector)
    {
        List<string> retDocs = new List<string>();
        string resultDocuments = string.Empty;

        try
        {
            //Search Azure Cosmos DB for MongoDB vCore collection for
similar embeddings
            //Project the fields that are needed
            BsonDocument[] pipeline = new BsonDocument[]
            {
                BsonDocument.Parse(
                    @$"{{$search: {{
                            cosmosSearch:
                                {{ vector: [{string.Join(',',
queryVector)}],
                                   path: 'embedding',
                                   k: {_maxVectorSearchResults}}},
                                   returnStoredSource:true
                                }}
                            }}"),
                BsonDocument.Parse($"{{$project: {{embedding: 0}}}}"),
            };

            var bsonDocuments = await _recipeCollection
                .Aggregate<BsonDocument>(pipeline).ToListAsync();

            var recipes = bsonDocuments
                .ToList()
                .ConvertAll(bsonDocument =>
                    BsonSerializer.Deserialize<Recipe>(bsonDocument));
            return recipes;
        }
        catch (MongoException ex)
        {
            Console.WriteLine($"Exception: VectorSearchAsync():
{ex.Message}");
            throw;
        }
    }
```

The `GetChatCompletionAsync` method generates an improved chat completion response based on the user prompt and the related vector search results.

C#

```
public async Task<(string response, int promptTokens, int
responseTokens)> GetChatCompletionAsync(string userPrompt, string
documents)
{
    try
    {
        ChatMessage systemMessage = new ChatMessage(
            ChatRole.System, _systemPromptRecipeAssistant + documents);
        ChatMessage userMessage = new ChatMessage(
            ChatRole.User, userPrompt);

        ChatCompletionsOptions options = new()
        {
            Messages =
            {
                systemMessage,
                userMessage
            },
            MaxTokens = openAIMaxTokens,
            Temperature = 0.5f, //0.3f,
            NucleusSamplingFactor = 0.95f,
            FrequencyPenalty = 0,
            PresencePenalty = 0
        };

        Azure.Response<ChatCompletions> completionsResponse =
            await
openAIClient.GetChatCompletionsAsync(openAICompletionDeployment,
options);
        ChatCompletions completions = completionsResponse.Value;

        return (
            response: completions.Choices[0].Message.Content,
            promptTokens: completions.Usage.PromptTokens,
            responseTokens: completions.Usage.CompletionTokens
        );

    }
    catch (Exception ex)
    {
        string message = $"OpenAIService.GetChatCompletionAsync():
{ex.Message}";
        Console.WriteLine(message);
        throw;
    }
}
```

The app also uses prompt engineering to ensure Open AI service limits and formats the response for supplied recipes.

C#

```
//System prompts to send with user prompts to instruct the model for
chat session
private readonly string _systemPromptRecipeAssistant = @"
    You are an intelligent assistant for Contoso Recipes.
    You are designed to provide helpful answers to user questions about
    recipes, cooking instructions provided in JSON format below.

    Instructions:
    - Only answer questions related to the recipe provided below.
    - Don't reference any recipe not provided below.
    - If you're unsure of an answer, say ""I don't know"" and recommend
users search themselves.
    - Your response  should be complete.
    - List the Name of the Recipe at the start of your response
followed by step by step cooking instructions.
    - Assume the user is not an expert in cooking.
    - Format the content so that it can be printed to the Command Line
console.
    - In case there is more than one recipe you find, let the user pick
the most appropriate recipe.";
```

Scale Azure OpenAI for .NET chat using RAG with Azure Container Apps

05/29/2025

Learn how to add load balancing to your application to extend the chat app beyond the Azure OpenAI Service token and model quota limits. This approach uses Azure Container Apps to create three Azure OpenAI endpoints and a primary container to direct incoming traffic to one of the three endpoints.

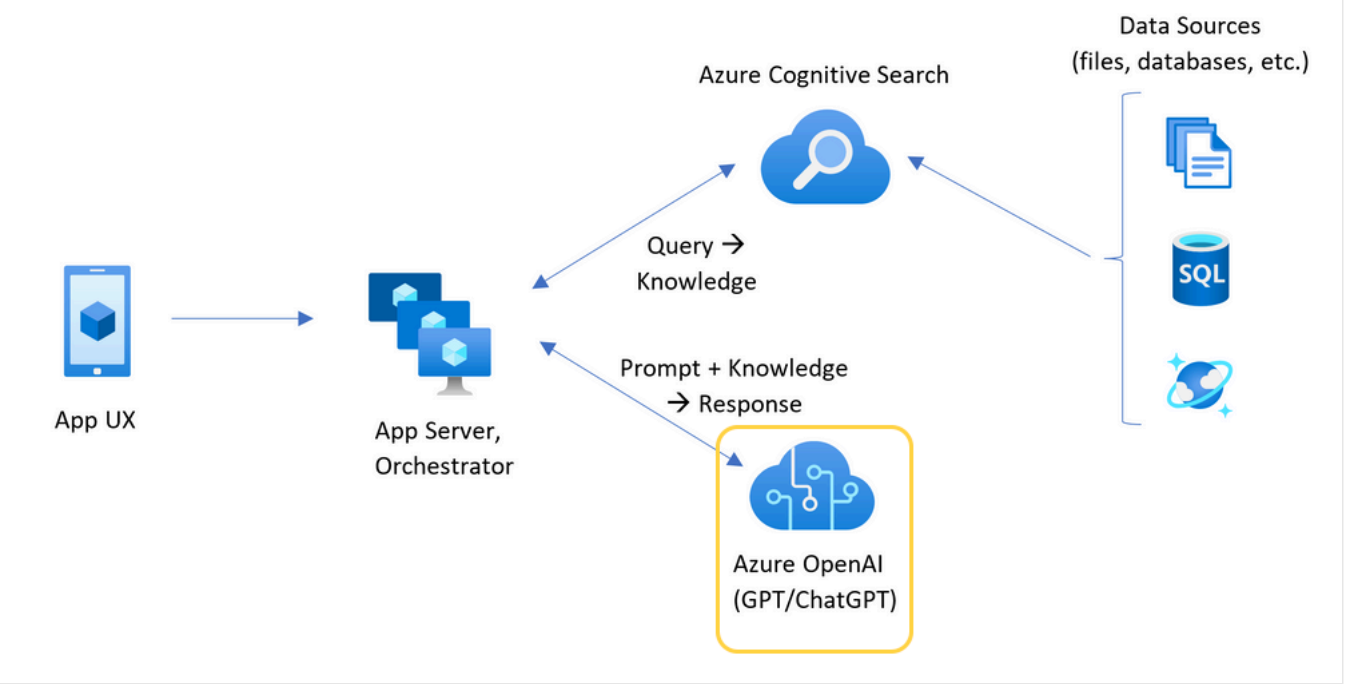
This article requires you to deploy two separate samples:

- Chat app
  - If you haven't deployed the chat app yet, wait until after the load balancer sample is deployed.
  - If you already deployed the chat app once, change the environment variable to support a custom endpoint for the load balancer and redeploy it again.
  - The chat app is available in these languages:
    - .NET
    - JavaScript
    - Python
- Load balancer app

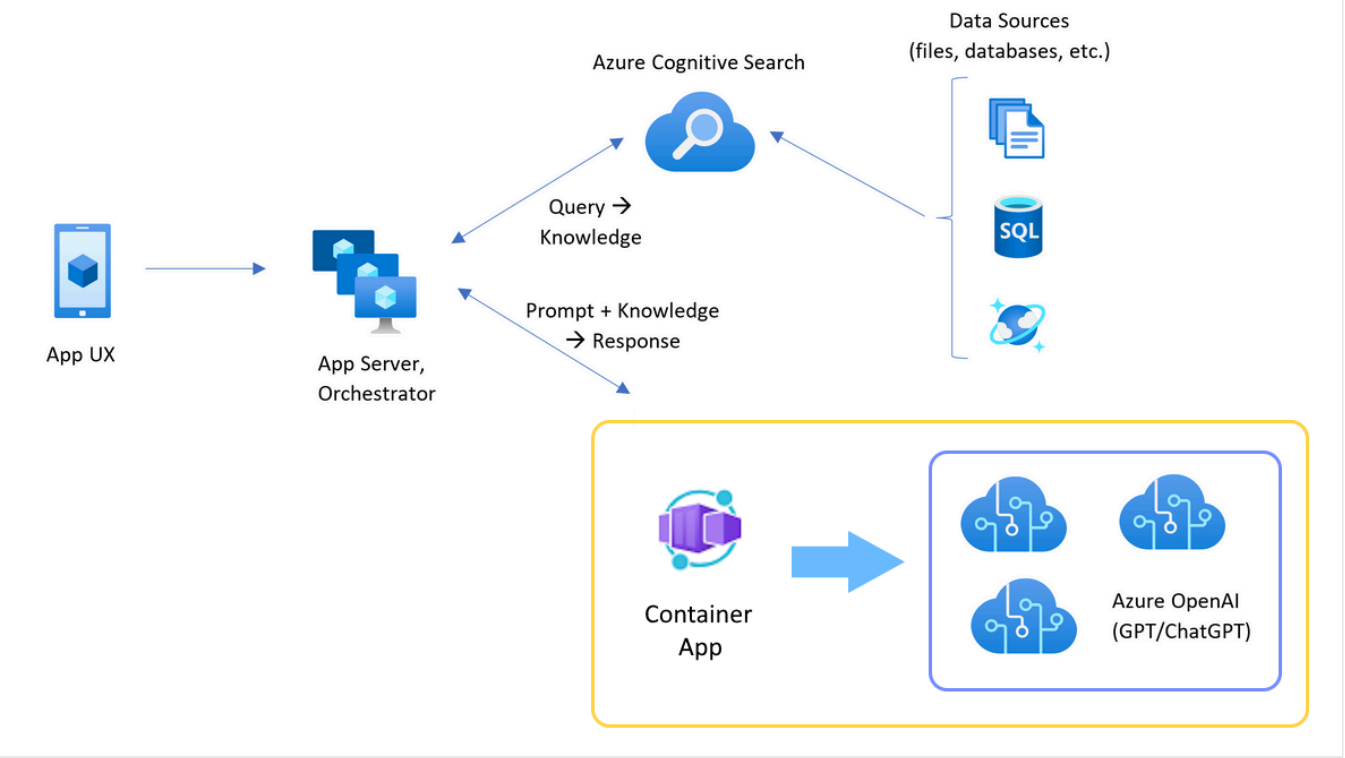
> **Note**
>
> This article uses one or more **AI app templates** as the basis for the examples and guidance in the article. AI app templates provide you with well-maintained reference implementations that are easy to deploy. They help to ensure a high-quality starting point for your AI apps.

Architecture for load balancing Azure OpenAI with Azure Container Apps

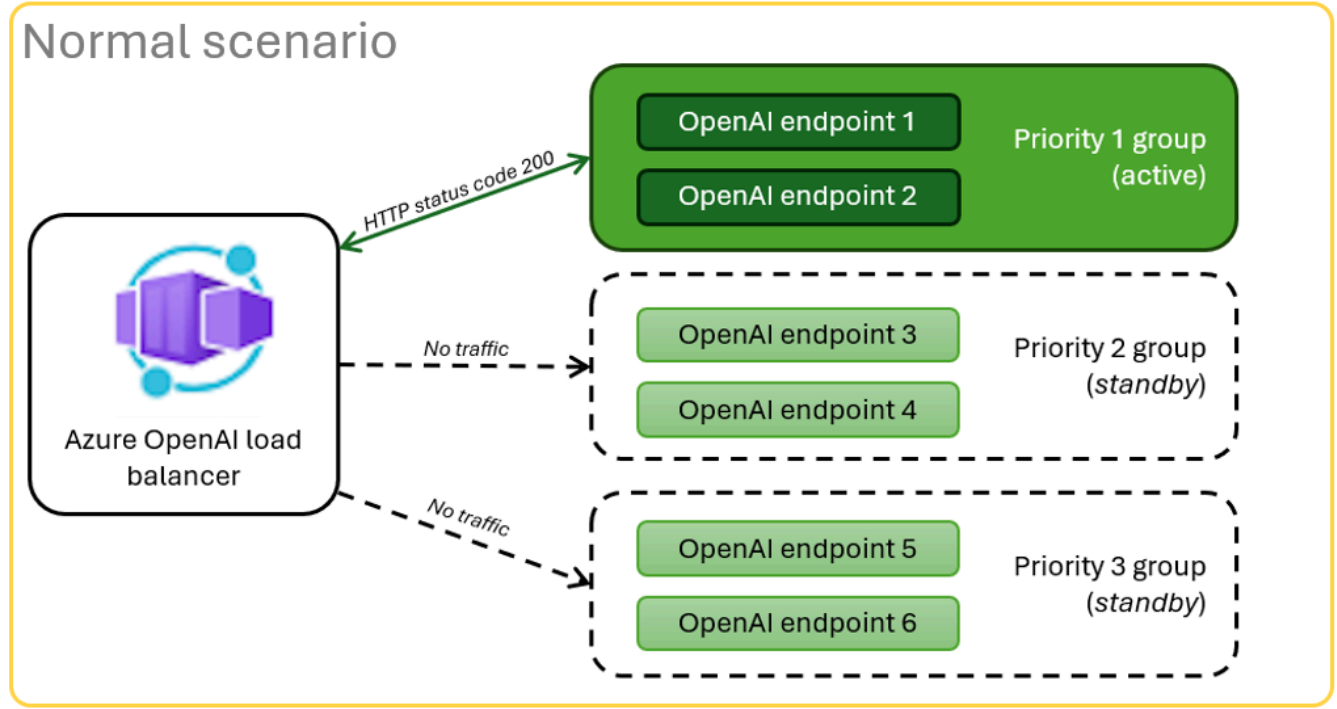
Because the Azure OpenAI resource has specific token and model quota limits, a chat app that uses a single Azure OpenAI resource is prone to have conversation failures because of those limits.



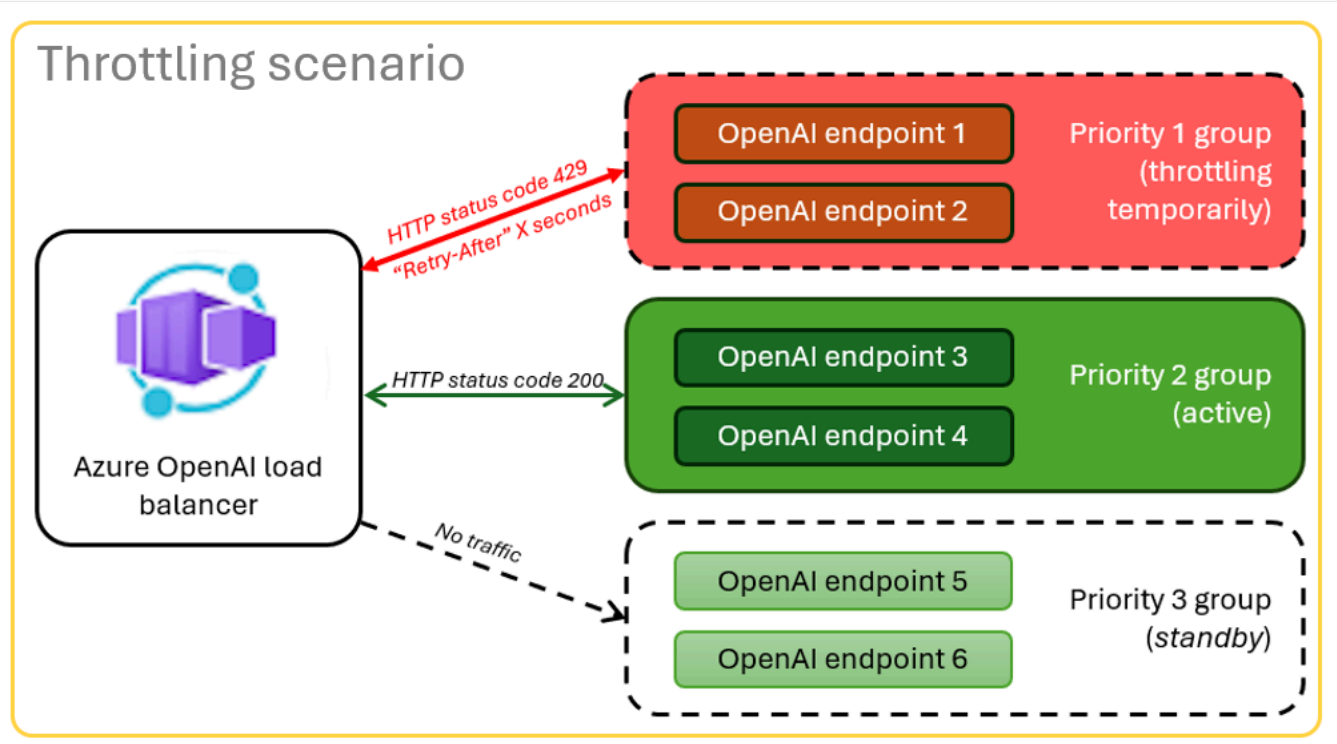
To use the chat app without hitting those limits, use a load-balanced solution with Container Apps. This solution seamlessly exposes a single endpoint from Container Apps to your chat app server.



The container app sits in front of a set of Azure OpenAI resources. The container app solves two scenarios: normal and throttled. During a *normal scenario* where token and model quota is available, the Azure OpenAI resource returns a 200 back through the container app and app server.



When a resource is in a *throttled scenario* because of quota limits, the container app can retry a different Azure OpenAI resource immediately to fulfill the original chat app request.



Prerequisites

- Azure subscription. Create one for free.

Dev containers are available for both samples, with all dependencies required to complete this article. You can run the dev containers in GitHub Codespaces (in a browser) or locally using

Visual Studio Code.

Codespaces (recommended)

- Only a GitHub account is required to use CodeSpaces

Open the Container Apps load balancer sample app

GitHub Codespaces (recommended)

GitHub Codespaces runs a development container managed by GitHub with Visual Studio Code for the Web as the user interface. For the most straightforward development environment, use GitHub Codespaces so that you have the correct developer tools and dependencies preinstalled to complete this article.

Open in GitHub Codespaces

Important

All GitHub accounts can use GitHub Codespaces for up to 60 hours free each month with two core instances. For more information, see GitHub Codespaces monthly included storage and core hours.

Deploy the Azure Container Apps load balancer

1. Sign in to the Azure Developer CLI to provide authentication to the provisioning and deployment steps:

   Bash

   ```
   azd auth login --use-device-code
   ```

2. Set an environment variable to use Azure CLI authentication to the post provision step:

   Bash

```
azd config set auth.useAzCliAuth "true"
```

3. Deploy the load balancer app:

```Bash
azd up
```

   Select a subscription and region for the deployment. They don't have to be the same subscription and region as the chat app.

4. Wait for the deployment to finish before you continue.

Get the deployment endpoint

1. Use the following command to display the deployed endpoint for the container app:

```Bash
azd env get-values
```

2. Copy the `CONTAINER_APP_URL` value. You use it in the next section.

Redeploy the chat app with the load balancer endpoint

These examples are completed on the chat app sample.

Initial deployment

1. Open the chat app sample's dev container by using one of the following choices.

Expand table

Language	GitHub Codespaces	Visual Studio Code
.NET	Open in GitHub Codespaces	Dev Containers Open
JavaScript	Open in GitHub Codespaces	Dev Containers Open

Language	GitHub Codespaces	Visual Studio Code
Python		

2. Sign in to the Azure Developer CLI (`AZD`):

   ```
   Bash

   azd auth login
   ```

   Finish the sign-in instructions.

3. Create an `AZD` environment with a name such as `chat-app`:

   ```
   Bash

   azd env new <name>
   ```

4. Add the following environment variable, which tells the chat app's backend to use a custom URL for the Azure OpenAI requests:

   ```
   Bash

   azd env set OPENAI_HOST azure_custom
   ```

5. Add the following environment variable. Substitute `<CONTAINER_APP_URL>` for the URL from the previous section. This action tells the chat app's backend what the value is of the custom URL for the Azure OpenAI request.

   ```
   Bash

   azd env set AZURE_OPENAI_CUSTOM_URL <CONTAINER_APP_URL>
   ```

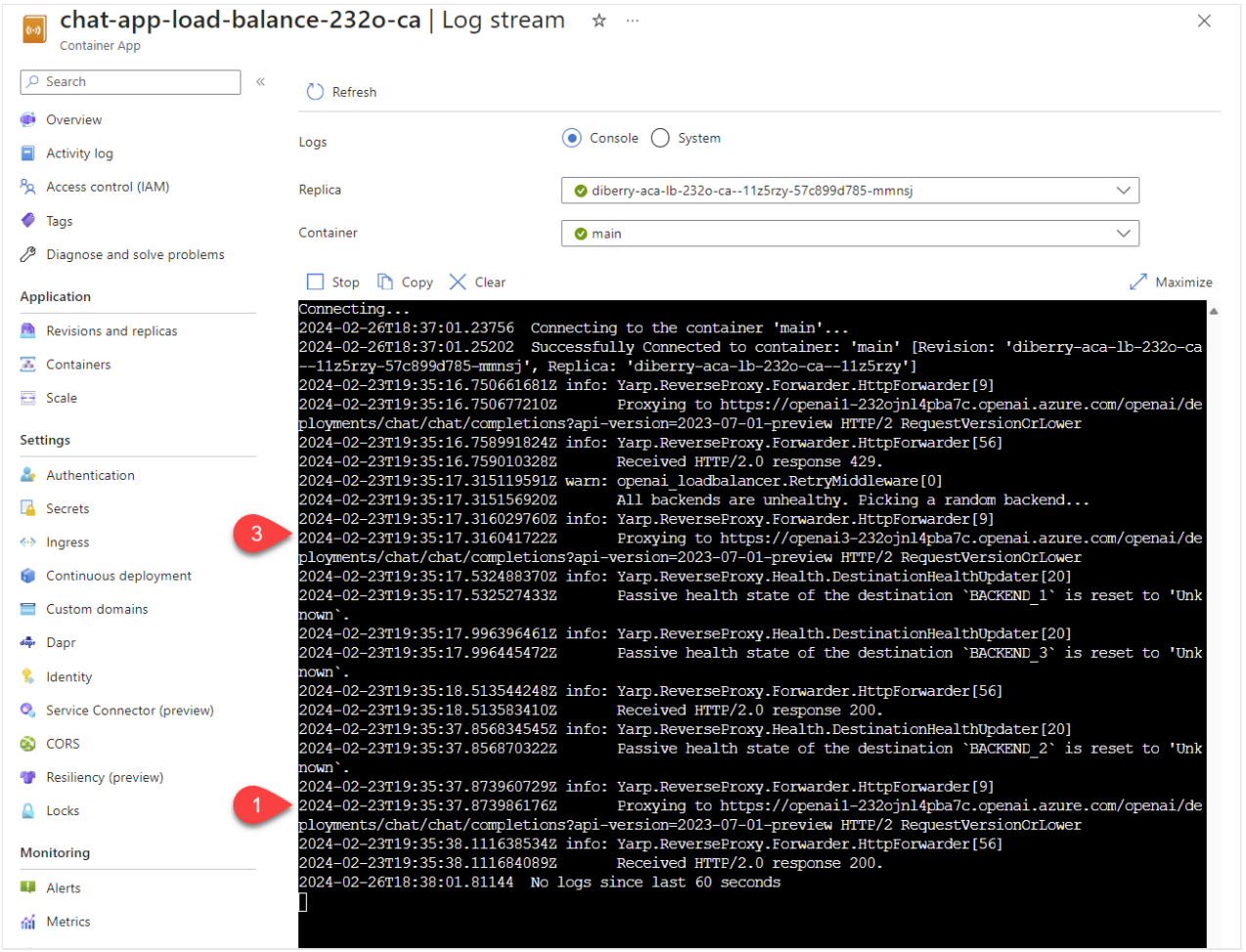
6. Deploy the chat app:

   ```
   Bash

   azd up
   ```

You can now use the chat app with the confidence that it's built to scale across many users without running out of quota.

Stream logs to see the load balancer results

1. In the Azure portal, search your resource group.
2. From the list of resources in the group, select the Azure Container Apps resource.
3. Select **Monitoring** > **Log stream** to view the log.
4. Use the chat app to generate traffic in the log.
5. Look for the logs, which reference the Azure OpenAI resources. Each of the three resources has its numeric identity in the log comment that begins with `Proxying to https://openai3`, where `3` indicates the third Azure OpenAI resource.



When the load balancer receives status that the request exceeds quota, the load balancer automatically rotates to another resource.

Configure the TPM quota

By default, each of the Azure OpenAI instances in the load balancer is deployed with a capacity of 30,000 tokens per minute (TPM). You can use the chat app with the confidence that it's built to scale across many users without running out of quota. Change this value when:

- You get deployment capacity errors: Lower the value.
- You need higher capacity: Raise the value.

1. Use the following command to change the value:

   ```bash
   azd env set OPENAI_CAPACITY 50
   ```

2. Redeploy the load balancer:

   ```bash
   azd up
   ```

Clean up resources

When you're finished with the chat app and the load balancer, clean up the resources. The Azure resources created in this article are billed to your Azure subscription. If you don't expect to need these resources in the future, delete them to avoid incurring more charges.

Clean up chat app resources

Return to the chat app article to clean up the resources:

- .NET
- JavaScript
- Python

Clean upload balancer resources

Run the following Azure Developer CLI command to delete the Azure resources and remove the source code:

```bash
azd down --purge --force
```

The switches provide:

- `purge`: Deleted resources are immediately purged so that you can reuse the Azure OpenAI Service tokens per minute.

- `force`: The deletion happens silently, without requiring user consent.

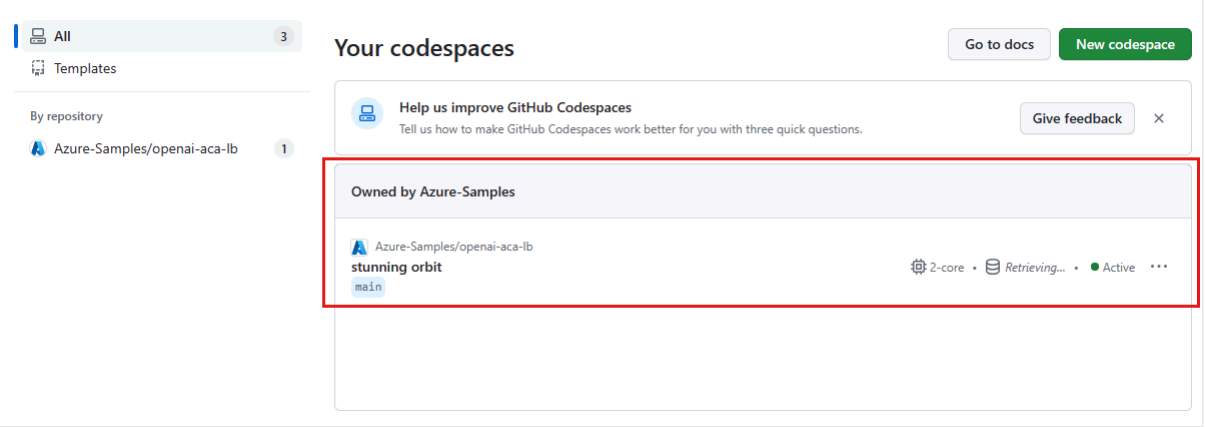
Clean up GitHub Codespaces and Visual Studio Code

GitHub Codespaces

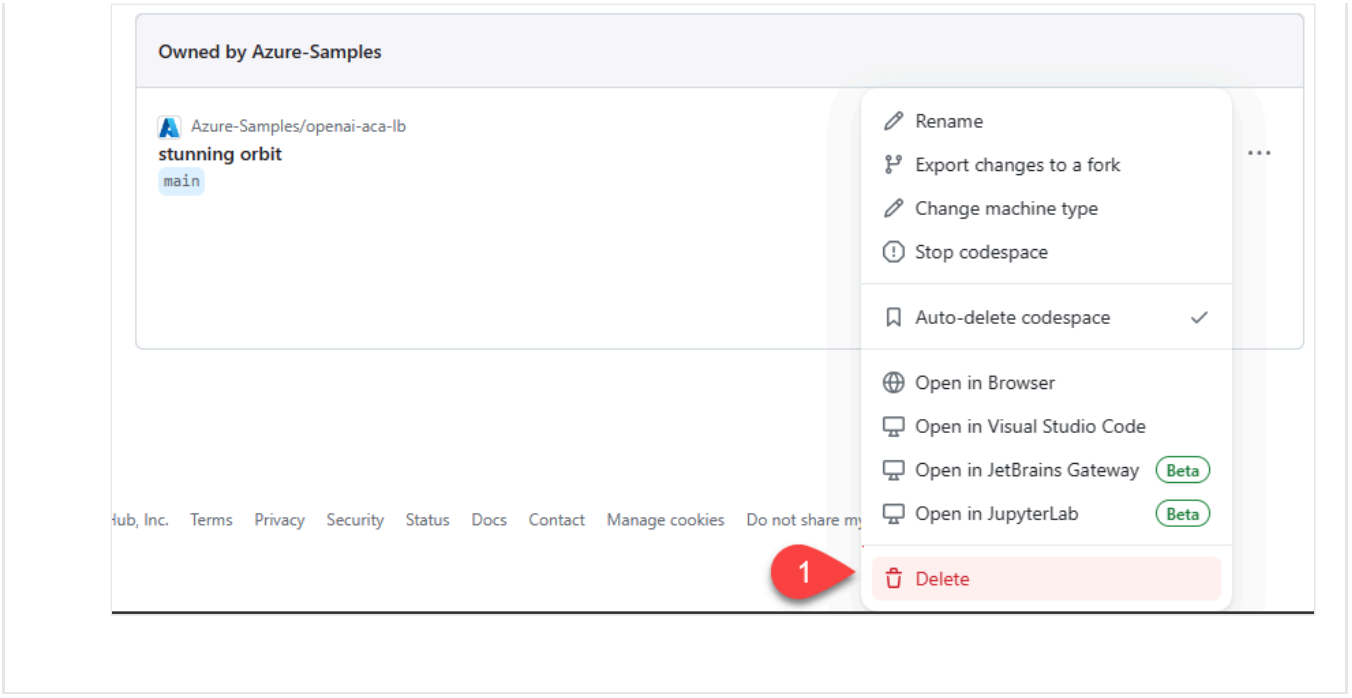
Deleting the GitHub Codespaces environment ensures that you can maximize the amount of free per-core hours entitlement that you get for your account.

> **ⓘ Important**
>
> For more information about your GitHub account's entitlements, see GitHub Codespaces monthly included storage and core hours.

1. Sign in to the GitHub Codespaces dashboard.
2. Locate your currently running codespaces that are sourced from the azure-samples/openai-aca-lb GitHub repository.



3. Open the context menu for the codespace, and then select **Delete**.



Get help

If you have trouble deploying the Azure API Management load balancer, add your issue to the repository's Issues webpage.

Sample code

Samples used in this article include:

- .NET chat app with RAG
- Load Balancer with Azure Container Apps

Next step

- Use Azure Load Testing to load test your chat app

Azure AI services authentication and authorization using .NET

Article • 04/09/2025

Application requests to Azure AI Services must be authenticated. In this article, you explore the options available to authenticate to Azure OpenAI and other AI services using .NET. These concepts apply to the Semantic Kernel SDK, as well as SDKs from specific services such as Azure OpenAI. Most AI services offer two primary ways to authenticate apps and users:

- **Key-based authentication** provides access to an Azure service using secret key values. These secret values are sometimes known as API keys or access keys depending on the service.
- **Microsoft Entra ID** provides a comprehensive identity and access management solution to ensure that the correct identities have the correct level of access to different Azure resources.

The sections ahead provide conceptual overviews for these two approaches, rather than detailed implementation steps. For more detailed information about connecting to Azure services, visit the following resources:

- Authenticate .NET apps to Azure services
- Identity fundamentals
- What is Azure RBAC?

> **Note**
>
> The examples in this article focus primarily on connections to Azure OpenAI, but the same concepts and implementation steps directly apply to many other Azure AI services as well.

Authentication using keys

Access keys allow apps and tools to authenticate to an Azure AI service, such as Azure OpenAI, using a secret key provided by the service. Retrieve the secret key using tools such as the Azure portal or Azure CLI and use it to configure your app code to connect to the AI service:

C#

```
builder.Services.AddAzureOpenAIChatCompletion(
    "deployment-model",
    "service-endpoint",
```

```
    "service-key"); // Secret key
var kernel = builder.Build();
```

Using keys is a straightforward option, but this approach should be used with caution. Keys aren't the recommended authentication option because they:

- Don't follow the principle of least privilege. They provide elevated permissions regardless of who uses them or for what task.
- Can accidentally be checked into source control or stored in unsafe locations.
- Can easily be shared with or sent to parties who shouldn't have access.
- Often require manual administration and rotation.

Instead, consider using Microsoft Entra ID for authentication, which is the recommended solution for most scenarios.

Authentication using Microsoft Entra ID

Microsoft Entra ID is a cloud-based identity and access management service that provides a vast set of features for different business and app scenarios. Microsoft Entra ID is the recommended solution to connect to Azure OpenAI and other AI services and provides the following benefits:

- Keyless authentication using identities.
- Role-based access control (RBAC) to assign identities the minimum required permissions.
- Can use the Azure.Identity client library to detect different credentials across environments without requiring code changes.
- Automatically handles administrative maintenance tasks such as rotating underlying keys.

The workflow to implement Microsoft Entra authentication in your app generally includes the following steps:

- Local development:
    1. Sign-in to Azure using a local dev tool such as the Azure CLI or Visual Studio.
    2. Configure your code to use the Azure.Identity client library and `DefaultAzureCredential` class.
    3. Assign Azure roles to the account you signed-in with to enable access to the AI service.
- Azure-hosted app:
    1. Deploy the app to Azure after configuring it to authenticate using the `Azure.Identity` client library.

2. Assign a managed identity to the Azure-hosted app.
3. Assign Azure roles to the managed identity to enable access to the AI service.

The key concepts of this workflow are explored in the following sections.

Authenticate to Azure locally

When developing apps locally that connect to Azure AI services, authenticate to Azure using a tool such as Visual Studio or the Azure CLI. Your local credentials can be discovered by the `Azure.Identity` client library and used to authenticate your app to Azure services, as described in the Configure the app code section.

For example, to authenticate to Azure locally using the Azure CLI, run the following command:

Azure CLI

```
az login
```

Configure the app code

Use the Azure.Identity client library from the Azure SDK to implement Microsoft Entra authentication in your code. The `Azure.Identity` libraries include the `DefaultAzureCredential` class, which automatically discovers available Azure credentials based on the current environment and tooling available. Visit the Azure SDK for .NET documentation for the full set of supported environment credentials and the order in which they are searched.

For example, configure Semantic Kernel to authenticate using `DefaultAzureCredential` using the following code:

C#

```
Kernel kernel = Kernel
    .CreateBuilder()
    .AddAzureOpenAITextGeneration(
        "your-model",
        "your-endpoint",
        new DefaultAzureCredential())
    .Build();
```

`DefaultAzureCredential` enables apps to be promoted from local development to production without code changes. For example, during development `DefaultAzureCredential` uses your local user credentials from Visual Studio or the Azure CLI to authenticate to the AI service.

When the app is deployed to Azure, `DefaultAzureCredential` uses the managed identity that is assigned to your app.

Assign roles to your identity

Azure role-based access control (Azure RBAC) is a system that provides fine-grained access management of Azure resources. Assign a role to the security principal used by `DefaultAzureCredential` to connect to an Azure AI service, whether that's an individual user, group, service principal, or managed identity. Azure roles are a collection of permissions that allow the identity to perform various tasks, such as generate completions or create and delete resources.

Assign roles such as **Cognitive Services OpenAI User** (role ID: `5e0bd9bd-7b93-4f28-af87-19fc36ad61bd`) to the relevant identity using tools such as the Azure CLI, Bicep, or the Azure Portal. For example, use the `az role assignment create` command to assign a role using the Azure CLI:

Azure CLI

```
az role assignment create \
        --role "5e0bd9bd-7b93-4f28-af87-19fc36ad61bd" \
        --assignee-object-id "$PRINCIPAL_ID" \
        --scope /subscriptions/"$SUBSCRIPTION_ID"/resourceGroups/"$RESOURCE_GROUP"
\
        --assignee-principal-type User
```

Learn more about Azure RBAC using the following resources:

- What is Azure RBAC?
- Grant a user access
- RBAC best practices

Assign a managed identity to your app

In most scenarios, Azure-hosted apps should use a managed identity to connect to other services such as Azure OpenAI. Managed identities provide a fully managed identity in Microsoft Entra ID for apps to use when connecting to resources that support Microsoft Entra authentication. `DefaultAzureCredential` discovers the identity associated with your app and uses it to authenticate to other Azure services.

There are two types of managed identities you can assign to your app:

- A **system-assigned identity** is tied to your application and is deleted if your app is deleted. An app can only have one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities.

Assign roles to a managed identity just like you would an individual user account, such as the **Cognitive Services OpenAI User** role. learn more about working with managed identities using the following resources:

- Managed identities overview
- Authenticate App Service to Azure OpenAI using Microsoft Entra ID
- How to use managed identities for App Service and Azure Functions

Authenticate to Azure OpenAI from an Azure hosted app using Microsoft Entra ID

05/29/2025

This article demonstrates how to use Microsoft Entra ID managed identities and the Microsoft.Extensions.AI library to authenticate an Azure hosted app to an Azure OpenAI resource.

A managed identity from Microsoft Entra ID allows your app to easily access other Microsoft Entra protected resources such as Azure OpenAI. The identity is managed by the Azure platform and doesn't require you to provision, manage, or rotate any secrets.

Prerequisites

- An Azure account that has an active subscription. Create an account for free.
- .NET SDK
- Create and deploy an Azure OpenAI Service resource
- Create and deploy a .NET application to App Service

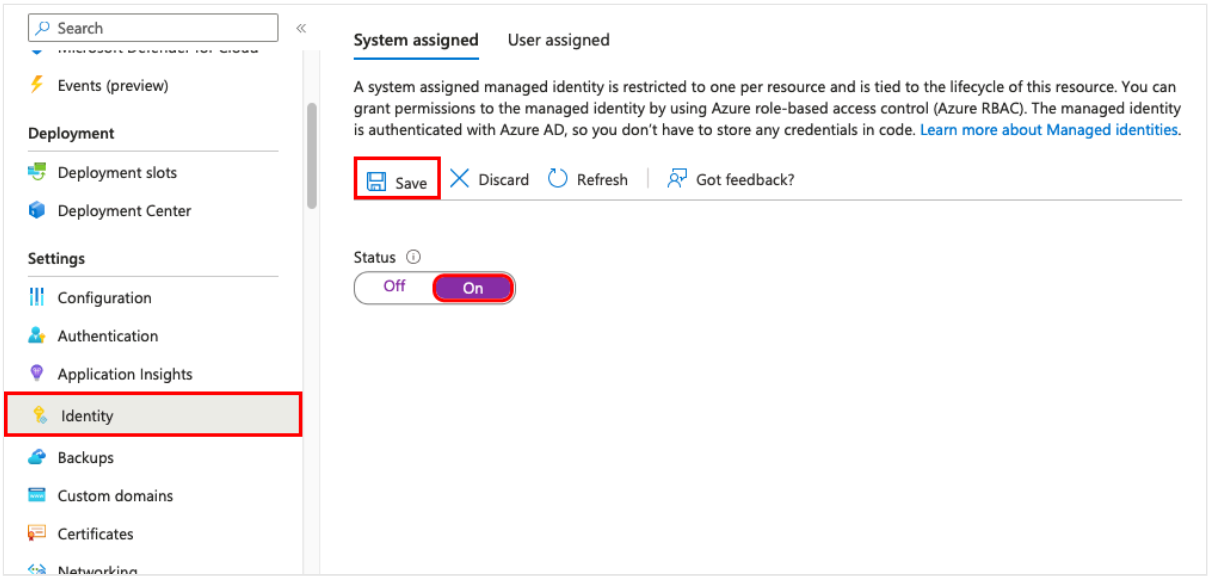
Add a managed identity to App Service

Managed identities provide an automatically managed identity in Microsoft Entra ID for applications to use when connecting to resources that support Microsoft Entra authentication. Applications can use managed identities to obtain Microsoft Entra tokens without having to manage any credentials. Your application can be assigned two types of identities:

- A **system-assigned identity** is tied to your application and is deleted if your app is deleted. An app can have only one system-assigned identity.
- A **user-assigned identity** is a standalone Azure resource that can be assigned to your app. An app can have multiple user-assigned identities.

System-assigned

1. Navigate to your app's page in the Azure portal, and then scroll down to the **Settings** group.
2. Select **Identity**.
3. On the **System assigned** tab, toggle *Status* to **On**, and then select **Save**.



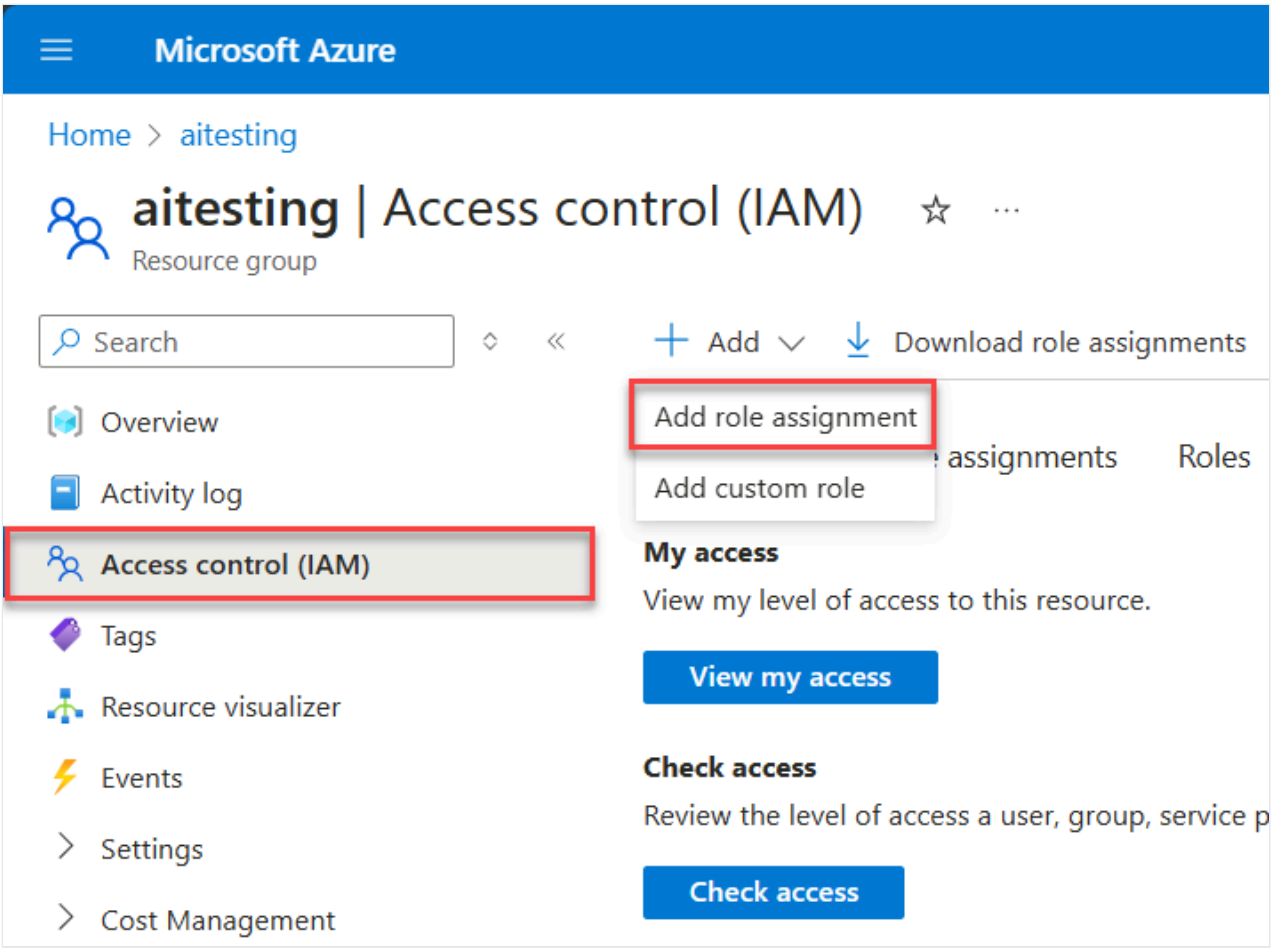
> **Note**
>
> The preceding screenshot demonstrates this process on an Azure App Service, but the steps are similar on other hosts such as Azure Container Apps.

Add an Azure OpenAI user role to the identity

1. In the Azure Portal, navigate to the scope that you want to grant **Azure OpenAI** access to. The scope can be a **Management group**, **Subscription**, **Resource group**, or a specific **Azure OpenAI** resource.

2. In the left navigation pane, select **Access control (IAM)**.

3. Select **Add**, then select **Add role assignment**.



4. On the **Role** tab, select the **Cognitive Services OpenAI User** role.

5. On the **Members** tab, select the managed identity.

6. On the **Review + assign** tab, select **Review + assign** to assign the role.

Implement identity authentication in your app code

1. Add the following NuGet packages to your app:

.NET CLI

```
dotnet add package Azure.Identity
dotnet add package Azure.AI.OpenAI
dotnet add package Microsoft.Extensions.Azure
dotnet add package Microsoft.Extensions.AI
dotnet add package Microsoft.Extensions.AI.OpenAI
```

The preceding packages each handle the following concerns for this scenario:

- Azure.Identity : Provides core functionality to work with Microsoft Entra ID
- Azure.AI.OpenAI : Enables your app to interface with the Azure OpenAI service

- Microsoft.Extensions.Azure: Provides helper extensions to register services for dependency injection
- Microsoft.Extensions.AI: Provides AI abstractions for common AI tasks
- Microsoft.Extensions.AI.OpenAI: Enables you to use OpenAI service types as AI abstractions provided by **Microsoft.Extensions.AI**

2. In the `Program.cs` file of your app, create a `DefaultAzureCredential` object to discover and configure available credentials:

```
C#

// For example, will discover Visual Studio or Azure CLI credentials
// in local environments and managed identity credentials in production
deployments
var credential = new DefaultAzureCredential(
    new DefaultAzureCredentialOptions
    {
        // If necessary, specify the tenant ID,
        // user-assigned identity client or resource ID, or other options
    }
);
```

3. Create an AI service and register it with the service collection:

```
C#

string endpoint = builder.Configuration["AZURE_OPENAI_ENDPOINT"];
string deployment = builder.Configuration["AZURE_OPENAI_GPT_NAME"];

builder.Services.AddChatClient(
    new AzureOpenAIClient(new Uri(endpoint), credential)
    .GetChatClient(deployment)
    .AsIChatClient());
```

4. Inject the registered service for use in your endpoints:

```
C#


app.MapGet("/test-prompt", async (IChatClient chatClient) =>
{
    return await chatClient.GetResponseAsync("Test prompt", new
ChatOptions());
})
.WithName("Test prompt");
```

> **Tip**

Learn more about ASP.NET Core dependency injection and how to register other AI services types in the Azure SDK for .NET **dependency injection** documentation.

Related content

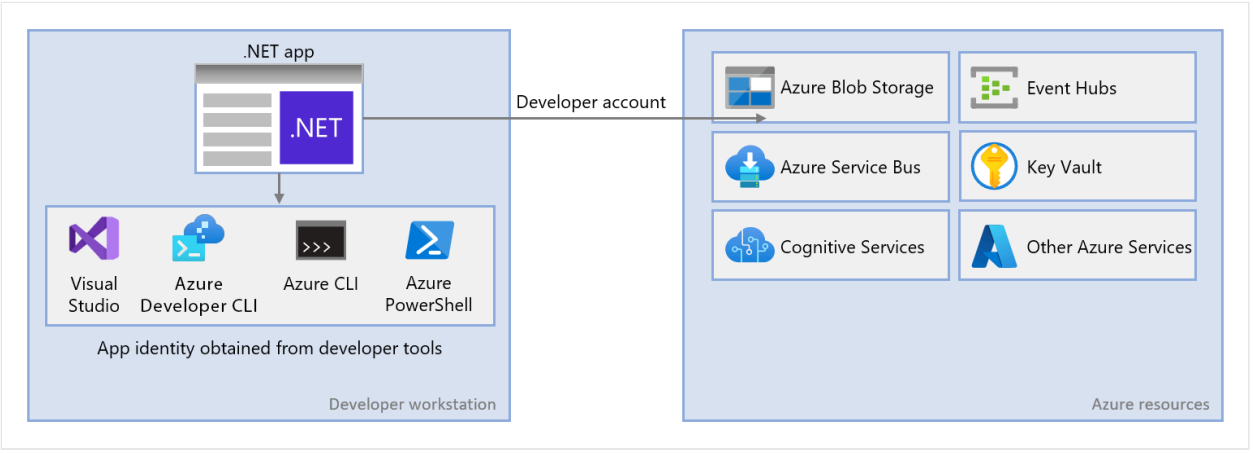
- How to use managed identities for App Service and Azure Functions
- Role-based access control for Azure OpenAI Service

Authenticate .NET apps to Azure services during local development using developer accounts

Article • 03/20/2025

During local development, applications need to authenticate to Azure to access various Azure services. Two common approaches for local authentication are to use a service principal or to use a developer account. This article explains how to use a developer account. In the sections ahead, you learn:

- How to use Microsoft Entra groups to efficiently manage permissions for multiple developer accounts
- How to assign roles to developer accounts to scope permissions
- How to sign-in to supported local development tools
- How to authenticate using a developer account from your app code



For an app to authenticate to Azure during local development using the developer's Azure credentials, the developer must be signed-in to Azure from one of the following developer tools:

- Azure CLI
- Azure Developer CLI
- Azure PowerShell
- Visual Studio

The Azure Identity library can detect that the developer is signed-in from one of these tools. The library can then obtain the Microsoft Entra access token via the tool to authenticate the app to Azure as the signed-in user.

This approach takes advantage of the developer's existing Azure accounts to streamline the authentication process. However, a developer's account likely has more permissions than required by the app, therefore exceeding the permissions the app runs with in production. As an alternative, you can create application service principals to use during local development, which can be scoped to have only the access needed by the app.

Create a Microsoft Entra group for local development

Create a Microsoft Entra group to encapsulate the roles (permissions) the app needs in local development rather than assigning the roles to individual service principal objects. This approach offers the following advantages:

- Every developer has the same roles assigned at the group level.
- If a new role is needed for the app, it only needs to be added to the group for the app.
- If a new developer joins the team, a new application service principal is created for the developer and added to the group, ensuring the developer has the right permissions to work on the app.

Azure portal

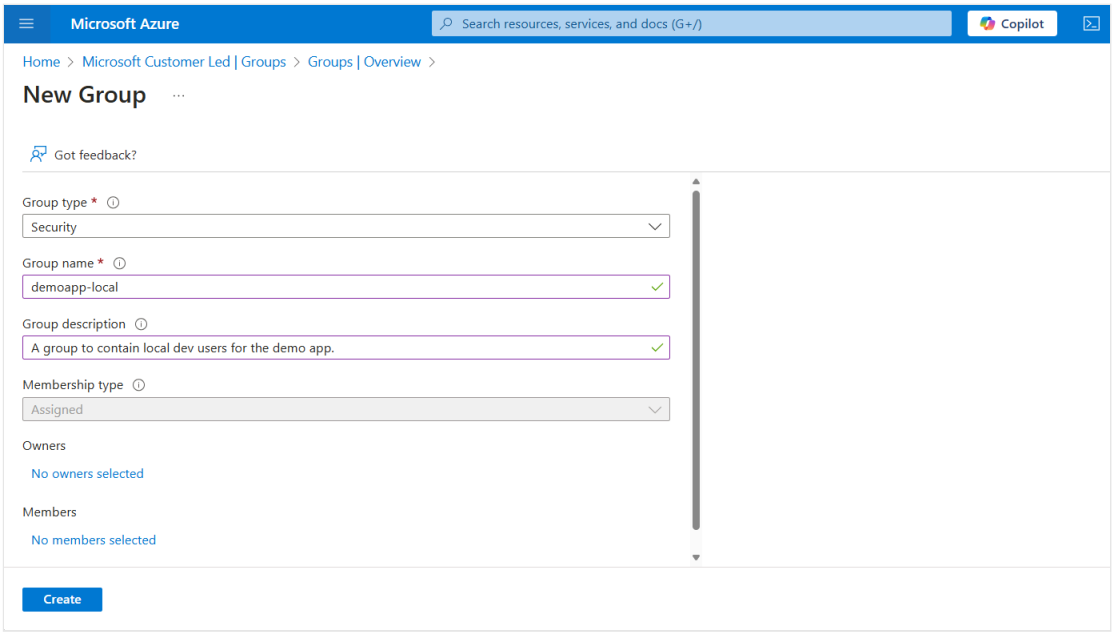
1. Navigate to the **Microsoft Entra ID** overview page in the Azure portal.

2. Select **All groups** from the left-hand menu.

3. On the **Groups** page, select **New group**.

4. On the **New group** page, fill out the following form fields:

    - **Group type**: Select **Security**.
    - **Group name**: Enter a name for the group that includes a reference to the app or environment name.
    - **Group description**: Enter a description that explains the purpose of the group.



5. Select the **No members selected** link under **Members** to add members to the group.

6. In the flyout panel that opens, search for the service principal you created earlier and select it from the filtered results. Choose the **Select** button at the bottom of the panel to confirm your selection.

7. Select **Create** at the bottom of the **New group** page to create the group and return to the **All groups** page. If you don't see the new group listed, wait a moment and refresh the page.

Assign roles to the group

Next, determine what roles (permissions) your app needs on what resources and assign those roles to the Microsoft Entra group you created. Groups can be assigned a role at the resource, resource group, or subscription scope. This example shows how to assign roles at the resource group scope, since most apps group all their Azure resources into a single resource group.

Azure portal

1. In the Azure portal, navigate to the **Overview** page of the resource group that contains your app.

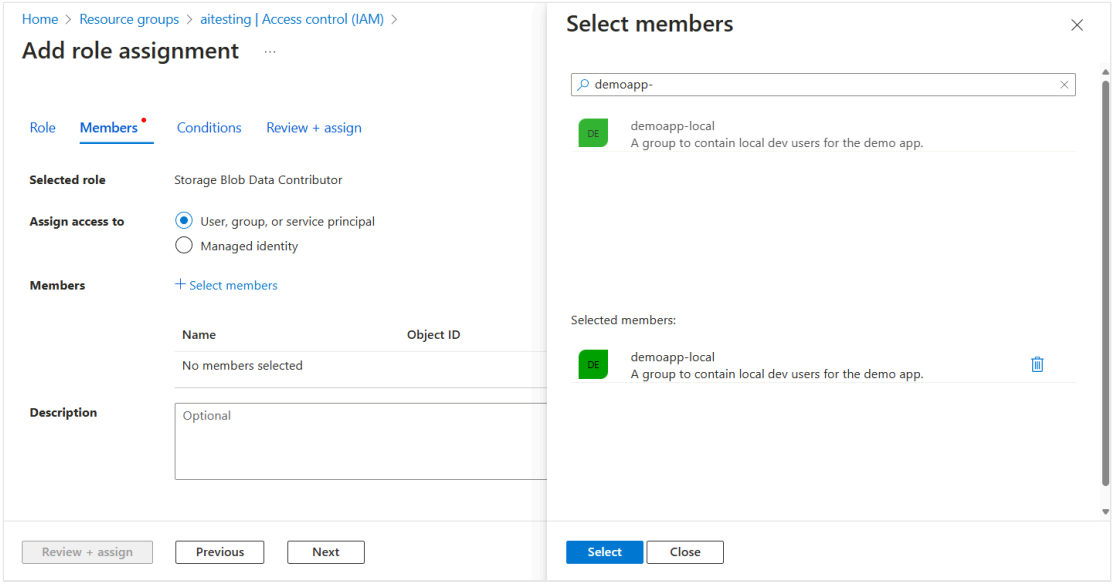
2. Select **Access control (IAM)** from the left navigation.

3. On the **Access control (IAM)** page, select **+ Add** and then choose **Add role assignment** from the drop-down menu. The **Add role assignment** page provides several tabs to configure and assign roles.

4. On the **Role** tab, use the search box to locate the role you want to assign. Select the role, and then choose **Next**.

5. On the **Members** tab:

    - For the **Assign access to** value, select **User, group, or service principal** .
    - For the **Members** value, choose **+ Select members** to open the **Select members** flyout panel.
    - Search for the Microsoft Entra group you created earlier and select it from the filtered results. Choose **Select** to select the group and close the flyout panel.
    - Select **Review + assign** at the bottom of the **Members** tab.



6. On the **Review + assign** tab, select **Review + assign** at the bottom of the page.

Sign-in to Azure using developer tooling

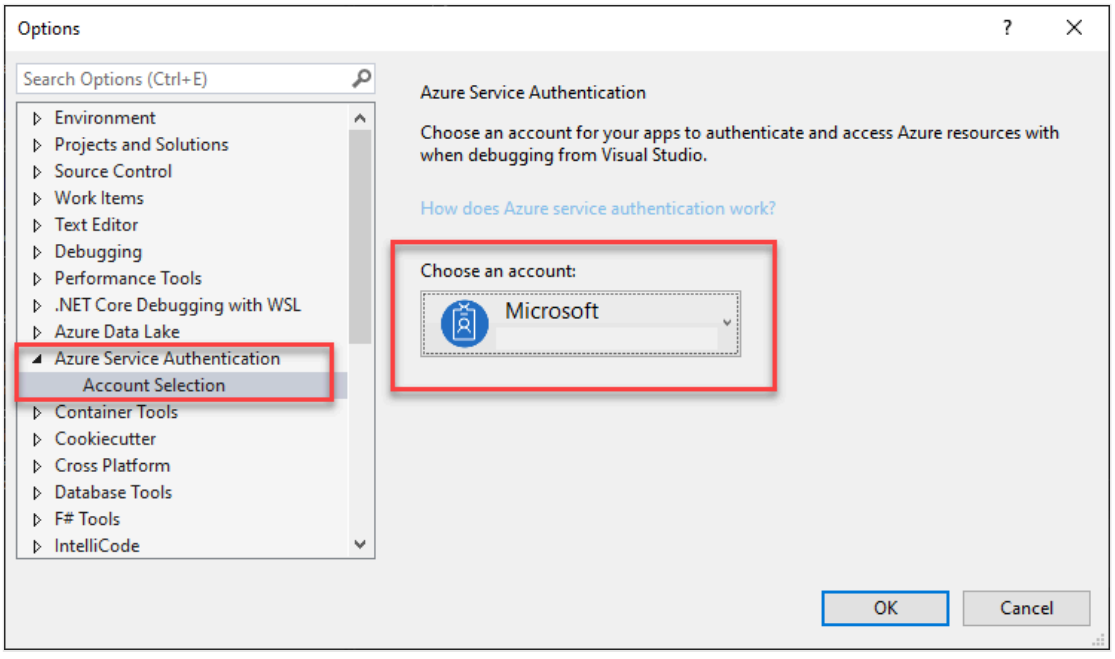
Next, sign-in to Azure using one of several developer tools that can be used to perform authentication in your development environment. The account you authenticate should also exist in the Microsoft Entra group you created and configured earlier.

Visual Studio

Developers using Visual Studio 2017 or later can authenticate using their developer account through the IDE. Apps using DefaultAzureCredential or VisualStudioCredential can discover and use this account to authenticate app requests when running locally. This account is also used when you publish apps directly from Visual Studio to Azure.

> **Important**
>
> You'll need to install the Azure development workload to enable Visual Studio tooling for Azure authentication, development, and deployment.

1. Inside Visual Studio, navigate to **Tools** > **Options** to open the options dialog.
2. In the **Search Options** box at the top, type *Azure* to filter the available options.
3. Under **Azure Service Authentication**, choose **Account Selection**.
4. Select the drop-down menu under **Choose an account** and choose to add a Microsoft account.
5. In the window that opens, enter the credentials for your desired Azure account, and then confirm your inputs.



6. Select **OK** to close the options dialog.

Authenticate to Azure services from your app

The Azure Identity library provides various *credentials*—implementations of `TokenCredential` adapted to supporting different scenarios and Microsoft Entra authentication flows. The steps ahead demonstrate how to use DefaultAzureCredential when working with user accounts locally.

Implement the code

DefaultAzureCredential is an opinionated, ordered sequence of mechanisms for authenticating to Microsoft Entra ID. Each authentication mechanism is a class derived from the TokenCredential class and is known as a *credential*. At runtime, `DefaultAzureCredential` attempts to authenticate using the first credential. If that credential fails to acquire an access token, the next credential in the sequence is attempted, and so on, until an access token is successfully obtained. In this way, your app can use different credentials in different environments without writing environment-specific code.

To use `DefaultAzureCredential`, add the Azure.Identity and optionally the Microsoft.Extensions.Azure packages to your application:

Command Line

In a terminal of your choice, navigate to the application project directory and run the following commands:

.NET CLI

```
dotnet add package Azure.Identity
dotnet add package Microsoft.Extensions.Azure
```

Azure services are accessed using specialized client classes from the various Azure SDK client libraries. These classes and your own custom services should be registered so they can be accessed via dependency injection throughout your app. In `Program.cs`, complete the following steps to register a client class and `DefaultAzureCredential`:

1. Include the `Azure.Identity` and `Microsoft.Extensions.Azure` namespaces via `using` directives.
2. Register the Azure service client using the corresponding `Add`-prefixed extension method.
3. Pass an instance of `DefaultAzureCredential` to the `UseCredential` method.

C#

```
builder.Services.AddAzureClients(clientBuilder =>
{
    clientBuilder.AddBlobServiceClient(
        new Uri("https://<account-name>.blob.core.windows.net"));

    clientBuilder.UseCredential(new DefaultAzureCredential());
});
```

An alternative to the `UseCredential` method is to provide the credential to the service client directly:

C#

```
builder.Services.AddSingleton<BlobServiceClient>(_ =>
    new BlobServiceClient(
        new Uri("https://<account-name>.blob.core.windows.net"),
        new DefaultAzureCredential()));
```

Work with Azure OpenAI content filtering in a .NET app

05/29/2025

This article demonstrates how to handle content filtering concerns in a .NET app. Azure OpenAI Service includes a content filtering system that works alongside core models. This system works by running both the prompt and completion through an ensemble of classification models aimed at detecting and preventing the output of harmful content. The content filtering system detects and takes action on specific categories of potentially harmful content in both input prompts and output completions. Variations in API configurations and application design might affect completions and thus filtering behavior.

The Content Filtering documentation provides a deeper exploration of content filtering concepts and concerns. This article provides examples of how to work with content filtering features programmatically in a .NET app.

Prerequisites

- An Azure account that has an active subscription. Create an account for free.
- .NET SDK
- Create and deploy an Azure OpenAI Service resource

Configure and test the content filter

To use the sample code in this article, you need to create and assign a content filter to your OpenAI model.

1. Create and assign a content filter to your provisioned model.
2. Add the Azure.AI.OpenAI NuGet package to your project.

   .NET CLI

   ```
   dotnet add package Azure.AI.OpenAI
   ```

   Or, in .NET 10+:

   .NET CLI

   ```
   dotnet package add Azure.AI.OpenAI
   ```

3. Create a simple chat completion flow in your .NET app using the `AzureOpenAiClient`. Replace the `YOUR_MODEL_ENDPOINT` and `YOUR_MODEL_DEPLOYMENT_NAME` values with your own.

C#

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

IChatClient client =
    new AzureOpenAIClient(
        new Uri("YOUR_MODEL_ENDPOINT"),
        new
DefaultAzureCredential()).GetChatClient("YOUR_MODEL_DEPLOYMENT_NAME").AsIChat
Client();

try
{
    ChatResponse completion = await client.GetResponseAsync("YOUR_PROMPT");

    Console.WriteLine(completion.Messages.Single());
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

4. Replace the `YOUR_PROMPT` placeholder with your own message and run the app to experiment with content filtering results. If you enter a prompt the AI considers unsafe, Azure OpenAI returns a `400 Bad Request` code. The app prints a message in the console similar to the following:

Output

```
The response was filtered due to the prompt triggering Azure OpenAI's content
management policy...
```

Related content

- Create and assign a content filter
- Content Filtering concepts
- Create a chat app

Use a blocklist with Azure OpenAI

07/02/2025

The configurable content filters available in Azure OpenAI are sufficient for most content moderation needs. However, you might need to filter terms specific to your use case. For this, you can use custom blocklists.

Prerequisites

- An Azure subscription. Create one for free.
- Once you have your Azure subscription, create an Azure OpenAI resource in the Azure portal to get your token, key, and endpoint. Enter a unique name for your resource, select the subscription you entered on the application form, select a resource group, supported region, and supported pricing tier. Then select **Create**.
  - The resource takes a few minutes to deploy. After it finishes, select **go to resource**. In the left pane, under **Resource Management**, select **Subscription Key and Endpoint**. The endpoint and either of the keys are used to call APIs.
- Azure CLI installed
- cURL installed

Use blocklists

Azure OpenAI API

You can create blocklists with the Azure OpenAI API. The following steps help you get started.

Get your token

First, you need to get a token for accessing the APIs for creating, editing, and deleting blocklists. You can get this token using the following Azure CLI command:

Bash

```
az account get-access-token
```

Create or modify a blocklist

Copy the cURL command below to a text editor and make the following changes:

1. Replace {subscriptionId} with your subscription ID.
2. Replace {resourceGroupName} with your resource group name.
3. Replace {accountName} with your resource name.
4. Replace {raiBlocklistName} (in the URL) with a custom name for your list. Allowed characters: `0-9, A-Z, a-z, - . _ ~`.
5. Replace {token} with the token you got from the "Get your token" step above.
6. Optionally replace the value of the "description" field with a custom description.

Bash

```
curl --location --request PUT
'https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{r
esourceGroupName}/providers/Microsoft.CognitiveServices/accounts/{accountName}
/raiBlocklists/{raiBlocklistName}?api-version=2024-04-01-preview' \
--header 'Authorization: Bearer {token}' \
--header 'Content-Type: application/json' \
--data-raw '{
    "properties": {
        "description": "This is a prompt blocklist"
    }
}'
```

The response code should be `201` (created a new list) or `200` (updated an existing list).

Apply a blocklist to a content filter

If you haven't yet created a content filter, you can do so in Azure AI Foundry. See Content filtering.

To apply a **completion** blocklist to a content filter, use the following cURL command:

1. Replace {subscriptionId} with your sub ID.
2. Replace {resourceGroupName} with your resource group name.
3. Replace {accountName} with your resource name.
4. Replace {raiPolicyName} with the name of your Content Filter
5. Replace {token} with the token you got from the "Get your token" step above.
6. Optionally change the `"completionBlocklists"` title to `"promptBlocklists"` if you want the blocklist to apply to user prompts instead of AI model completions.
7. Replace `"raiBlocklistName"` in the body with a custom name for your list. Allowed characters: `0-9, A-Z, a-z, - . _ ~`.

Bash

```
curl --location --request PUT
'https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{r
```

```
esourceGroupName}/providers/Microsoft.CognitiveServices/accounts/{accountName}
/raiPolicies/{raiPolicyName}?api-version=2024-04-01-preview' \
--header 'Authorization: Bearer {token}' \
--header 'Content-Type: application/json' \
--data-raw '{
    "properties": {
        "basePolicyName": "Microsoft.Default",
        "completionBlocklists": [{
            "blocklistName": "raiBlocklistName",
            "blocking": true
        }],
        "contentFilters": [ ]
    }
}'
```

Add blockItems to the list

> **Note**
>
> There is a maximum limit of 10,000 terms allowed in one list.

Copy the cURL command below to a text editor and make the following changes:

1. Replace {subscriptionId} with your sub ID.
2. Replace {resourceGroupName} with your resource group name.
3. Replace {accountName} with your resource name.
4. Replace {raiBlocklistName} (in the URL) with a custom name for your list. Allowed characters: `0-9, A-Z, a-z, - . _ ~`.
5. Replace {raiBlocklistItemName} with a custom name for your list item.
6. Replace {token} with the token you got from the "Get your token" step above.
7. Replace the value of the `"blocking pattern"` field with the item you'd like to add to your blocklist. The maximum length of a blockItem is 1,000 characters. Also specify whether the pattern is regex or exact match.

Bash

```
curl --location --request PUT
'https://management.azure.com/subscriptions/{subscriptionId}/resourceGroups/{r
esourceGroupName}/providers/Microsoft.CognitiveServices/accounts/{accountName}
/raiBlocklists/{raiBlocklistName}/raiBlocklistItems/{raiBlocklistItemName}?
api-version=2024-04-01-preview' \
--header 'Authorization: Bearer {token}' \
--header 'Content-Type: application/json' \
--data-raw '{
    "properties": {
        "pattern": "blocking pattern",
```

```
        "isRegex": false
    }
}'
```

> **Note**
>
> It can take around 5 minutes for a new term to be added to the blocklist. Test the blocklist after 5 minutes.

The response code should be `200`.

JSON

```
{
  "name": "raiBlocklistItemName",
  "id":
"/subscriptions/subscriptionId/resourceGroups/resourceGroupName/providers/Micr
osoft.CognitiveServices/accounts/accountName/raiBlocklists/raiBlocklistName/ra
iBlocklistItems/raiBlocklistItemName",
  "properties": {
    "pattern": "blocking pattern",
    "isRegex": false
  }
}
```

Analyze text with a blocklist

Now you can test out your deployment that has the blocklist. For instructions on calling the Azure OpenAI endpoints, visit the Quickstart.

In the below example, a GPT-35-Turbo deployment with a blocklist is blocking the prompt. The response returns a `400` error.

JSON

```
{
    "error": {
        "message": "The response was filtered due to the prompt triggering
Azure OpenAI's content management policy. Please modify your prompt and retry.
To learn more about our content filtering policies please read our
documentation: https://go.microsoft.com/fwlink/?linkid=2198766",
        "type": null,
        "param": "prompt",
        "code": "content_filter",
        "status": 400,
        "innererror": {
            "code": "ResponsibleAIPolicyViolation",
```

```
            "content_filter_results": {
                "custom_blocklists": [
                    {
                        "filtered": true,
                        "id": "raiBlocklistName"
                    }
                ],
                "hate": {
                    "filtered": false,
                    "severity": "safe"
                },
                "self_harm": {
                    "filtered": false,
                    "severity": "safe"
                },
                "sexual": {
                    "filtered": false,
                    "severity": "safe"
                },
                "violence": {
                    "filtered": false,
                    "severity": "safe"
                }
            }
        }
    }
}
```

If the completion itself is blocked, the response returns `200`, as the completion only cuts off when the blocklist content is matched. The annotations show that a blocklist item was matched.

JSON

```
{
    "id": "chatcmpl-85NkyY0AkeBMunOjyxivQSiTaxGAl",
    "object": "chat.completion",
    "created": 1696293652,
    "model": "gpt-35-turbo",
    "prompt_filter_results": [
        {
            "prompt_index": 0,
            "content_filter_results": {
                "hate": {
                    "filtered": false,
                    "severity": "safe"
                },
                "self_harm": {
                    "filtered": false,
                    "severity": "safe"
                },
                "sexual": {
```

```
                    "filtered": false,
                    "severity": "safe"
                },
                "violence": {
                    "filtered": false,
                    "severity": "safe"
                }
            }
        }
    ],
    "choices": [
        {
            "index": 0,
            "finish_reason": "content_filter",
            "message": {
                "role": "assistant"
            },
            "content_filter_results": {
                "custom_blocklists": [
                    {
                        "filtered": true,
                        "id": "myBlocklistName"
                    }
                ],
                "hate": {
                    "filtered": false,
                    "severity": "safe"
                },
                "self_harm": {
                    "filtered": false,
                    "severity": "safe"
                },
                "sexual": {
                    "filtered": false,
                    "severity": "safe"
                },
                "violence": {
                    "filtered": false,
                    "severity": "safe"
                }
            }
        }
    ],
    "usage": {
        "completion_tokens": 75,
        "prompt_tokens": 27,
        "total_tokens": 102
    }
}
```


Related content

- Learn more about Responsible AI practices for Azure OpenAI: Overview of Responsible AI practices for Azure OpenAI models.
- Read more about content filtering categories and severity levels with Azure OpenAI in Azure AI Foundry Models.
- Learn more about red teaming from our: Introduction to red teaming large language models (LLMs) article.

Use Risks & Safety monitoring in Azure AI Foundry (preview)

07/02/2025

When you use an Azure OpenAI model deployment with a content filter, you might want to check the results of the filtering activity. You can use that information to further adjust your filter configuration to serve your specific business needs and Responsible AI principles.

Azure AI Foundry provides a Risks & Safety monitoring dashboard for each of your deployments that uses a content filter configuration.

Access Risks & Safety monitoring

To access Risks & Safety monitoring, you need an Azure OpenAI resource in one of the supported Azure regions: East US, Switzerland North, France Central, Sweden Central, Canada East. You also need a model deployment that uses a content filter configuration.

Go to Azure AI Foundry and sign in with the credentials associated with your Azure OpenAI resource. Select a project. Then select the **Models + endpoints** tab on the left and then select your model deployment from the list. On the deployment's page, select the **Monitoring** tab at the top. Then select **Open in Azure Monitor** to view the full report in the Azure portal.

Configure metrics

Report description

Content filtering data is shown in the following ways:

- **Total blocked request count and block rate**: This view shows a global view of the amount and rate of content that is filtered over time. This helps you understand trends of harmful requests from users and see any unexpected activity.
- **Blocked requests by category**: This view shows the amount of content blocked for each category. This is an all-up statistic of harmful requests across the time range selected. It currently supports the harm categories hate, sexual, self-harm, and violence.
- **Block rate over time by category**: This view shows the block rate for each category over time. It currently supports the harm categories hate, sexual, self-harm, and violence.
- **Severity distribution by category**: This view shows the severity levels detected for each harm category, across the whole selected time range. This is not limited to *blocked* content but rather includes all content that was flagged by the content filters.

- **Severity rate distribution over time by category**: This view shows the rates of detected severity levels over time, for each harm category. Select the tabs to switch between supported categories.

Recommended actions

Adjust your content filter configuration to further align with business needs and Responsible AI principles.

Potentially abusive user detection

The **Potentially abusive user detection** pane shows information about users whose behavior has resulted in blocked content. The goal is to help you get a view of the sources of harmful content so you can take responsive actions to ensure the model is being used in a responsible way.

To use Potentially abusive user detection, you need:

- A content filter configuration applied to your deployment.
- You must be sending user ID information in your Chat Completion requests (see the *user* parameter of the Completions API, for example).

  > ⊗ **Caution**
  >
  > Use GUID strings to identify individual users. Don't include sensitive personal information in the *user* field.

- An Azure Data Explorer database set up to store the user analysis results (instructions below).

Set up your Azure Data Explorer database

In order to protect the data privacy of user information and manage the permission of the data, we support the option for our customers to bring their own storage to get the detailed potentially abusive user detection insights (including user GUID and statistics on harmful request by category) stored in a compliant way and with full control. Follow these steps to enable it:

1. In Azure AI Foundry, navigate to the model deployment that you'd like to set up user abuse analysis with, and select **Add a data store**.
2. Fill in the required information and select **Save**. We recommend you create a new database to store the analysis results.

3. After you connect the data store, take the following steps to grant permission to write analysis results to the connected database:
   a. Go to your Azure OpenAI resource's page in the Azure portal, and choose the **Identity** tab.
   b. Turn the status to **On** for system assigned identity, and copy the ID that's generated.
   c. Go to your Azure Data Explorer resource in the Azure portal, choose **databases**, and then choose the specific database you created to store user analysis results.
   d. Select **permissions**, and add an **admin** role to the database.
   e. Paste the Azure OpenAI identity generated in the earlier step, and select the one searched. Now your Azure OpenAI resource's identity is authorized to read/write to the storage account.
4. Grant access to the connected Azure Data Explorer database to the users who need to view the analysis results:
   a. Go to the Azure Data Explorer resource you've connected, choose **access control** and add a **reader** role of the Azure Data Explorer cluster for the users who need to access the results.
   b. Choose **databases** and choose the specific database that's connected to store user-level abuse analysis results. Choose **permissions** and add the **reader** role of the database for the users who need to access the results.

Report description

The potentially abusive user detection relies on the user information that customers send with their Azure OpenAI API calls, together with the request content. The following insights are shown:

- **Total potentially abusive user count**: This view shows the number of detected potentially abusive users over time. These are users for whom a pattern of abuse was detected and who might introduce high risk.
- **Potentially abusive users list**: This view is a detailed list of detected potentially abusive users. It gives the following information for each user:
  - **UserGUID**: This is sent by the customer through "user" field in Azure OpenAI APIs.
  - **Abuse score**: This is a figure generated by the model analyzing each user's requests and behavior. The score is normalized to 0-1. A higher score indicates a higher abuse risk.
  - **Abuse score trend**: The change in **Abuse score** during the selected time range.
  - **Evaluate date**: The date the results were analyzed.
  - **Total abuse request ratio/count**
  - **Abuse ratio/count by category**

Recommended actions

Combine this data with enriched signals to validate whether the detected users are truly abusive or not. If they are, then take responsive action such as throttling or suspending the user to ensure the responsible use of your application.

Next step

Next, create or edit a content filter configuration in Azure AI Foundry.

Configure content filters with Azure OpenAI in Azure AI Foundry Models

The Microsoft.Extensions.AI.Evaluation libraries

07/26/2025

The Microsoft.Extensions.AI.Evaluation libraries simplify the process of evaluating the quality and accuracy of responses generated by AI models in .NET intelligent apps. Various metrics measure aspects like relevance, truthfulness, coherence, and completeness of the responses. Evaluations are crucial in testing, because they help ensure that the AI model performs as expected and provides reliable and accurate results.

The evaluation libraries, which are built on top of the Microsoft.Extensions.AI abstractions, are composed of the following NuGet packages:

- Microsoft.Extensions.AI.Evaluation – Defines the core abstractions and types for supporting evaluation.
- Microsoft.Extensions.AI.Evaluation.NLP - Contains evaluators that evaluate the similarity of an LLM's response text to one or more reference responses using natural language processing (NLP) metrics. These evaluators aren't LLM or AI-based; they use traditional NLP techniques such as text tokenization and n-gram analysis to evaluate text similarity.
- Microsoft.Extensions.AI.Evaluation.Quality – Contains evaluators that assess the quality of LLM responses in an app according to metrics such as relevance and completeness. These evaluators use the LLM directly to perform evaluations.
- Microsoft.Extensions.AI.Evaluation.Safety – Contains evaluators, such as the `ProtectedMaterialEvaluator` and `ContentHarmEvaluator`, that use the Azure AI Foundry Evaluation service to perform evaluations.
- Microsoft.Extensions.AI.Evaluation.Reporting – Contains support for caching LLM responses, storing the results of evaluations, and generating reports from that data.
- Microsoft.Extensions.AI.Evaluation.Reporting.Azure - Supports the reporting library with an implementation for caching LLM responses and storing the evaluation results in an Azure Storage container.
- Microsoft.Extensions.AI.Evaluation.Console – A command-line tool for generating reports and managing evaluation data.

Test integration

The libraries are designed to integrate smoothly with existing .NET apps, allowing you to leverage existing testing infrastructures and familiar syntax to evaluate intelligent apps. You can use any test framework (for example, MSTest, xUnit, or NUnit) and testing workflow (for

example, Test Explorer, dotnet test, or a CI/CD pipeline). The library also provides easy ways to do online evaluations of your application by publishing evaluation scores to telemetry and monitoring dashboards.

Comprehensive evaluation metrics

The evaluation libraries were built in collaboration with data science researchers from Microsoft and GitHub, and were tested on popular Microsoft Copilot experiences. The following sections show the built-in quality, NLP, and safety evaluators and the metrics they measure.

You can also customize to add your own evaluations by implementing the IEvaluator interface.

Quality evaluators

Quality evaluators measure response quality. They use an LLM to perform the evaluation.

Expand table

Evaluator type	Metric	Description
RelevanceEvaluator	`Relevance`	Evaluates how relevant a response is to a query
CompletenessEvaluator	`Completeness`	Evaluates how comprehensive and accurate a response is
RetrievalEvaluator	`Retrieval`	Evaluates performance in retrieving information for additional context
FluencyEvaluator	`Fluency`	Evaluates grammatical accuracy, vocabulary range, sentence complexity, and overall readability
CoherenceEvaluator	`Coherence`	Evaluates the logical and orderly presentation of ideas
EquivalenceEvaluator	`Equivalence`	Evaluates the similarity between the generated text and its ground truth with respect to a query
GroundednessEvaluator	`Groundedness`	Evaluates how well a generated response aligns with the given context

Evaluator type	Metric	Description
RelevanceTruthAndCompletenessEvaluator†	`Relevance (RTC)`, `Truth (RTC)`, and `Completeness (RTC)`	Evaluates how relevant, truthful, and complete a response is
IntentResolutionEvaluator	`Intent Resolution`	Evaluates an AI system's effectiveness at identifying and resolving user intent (agent-focused)
TaskAdherenceEvaluator	`Task Adherence`	Evaluates an AI system's effectiveness at adhering to the task assigned to it (agent-focused)
ToolCallAccuracyEvaluator	`Tool Call Accuracy`	Evaluates an AI system's effectiveness at using the tools supplied to it (agent-focused)

† This evaluator is marked experimental.

NLP evaluators

NLP evaluators evaluate the quality of an LLM response by comparing it to a reference response using natural language processing (NLP) techniques. These evaluators aren't LLM or AI-based; instead, they use older NLP techniques to perform text comparisons.

Expand table

Evaluator type	Metric	Description
BLEUEvaluator	`BLEU`	Evaluates a response by comparing it to one or more reference responses using the bilingual evaluation understudy (BLEU) algorithm. This algorithm is commonly used to evaluate the quality of machine-translation or text-generation tasks.
GLEUEvaluator	`GLEU`	Measures the similarity between the generated response and one or more reference responses using the Google BLEU (GLEU) algorithm, a variant of the BLEU algorithm that's optimized for sentence-level evaluation.
F1Evaluator	`F1`	Evaluates a response by comparing it to a reference response using the *F1* scoring algorithm (the ratio of the number of shared words between the generated response and the reference response).

Safety evaluators

Safety evaluators check for presence of harmful, inappropriate, or unsafe content in a response. They rely on the Azure AI Foundry Evaluation service, which uses a model that's fine tuned to perform evaluations.

Expand table

Evaluator type	Metric	Description
GroundednessProEvaluator	`Groundedness Pro`	Uses a fine-tuned model hosted behind the Azure AI Foundry Evaluation service to evaluate how well a generated response aligns with the given context
ProtectedMaterialEvaluator	`Protected Material`	Evaluates response for the presence of protected material
UngroundedAttributesEvaluator	`Ungrounded Attributes`	Evaluates a response for the presence of content that indicates ungrounded inference of human attributes
HateAndUnfairnessEvaluator†	`Hate And Unfairness`	Evaluates a response for the presence of content that's hateful or unfair
SelfHarmEvaluator†	`Self Harm`	Evaluates a response for the presence of content that indicates self harm
ViolenceEvaluator†	`Violence`	Evaluates a response for the presence of violent content
SexualEvaluator†	`Sexual`	Evaluates a response for the presence of sexual content
CodeVulnerabilityEvaluator	`Code Vulnerability`	Evaluates a response for the presence of vulnerable code
IndirectAttackEvaluator	`Indirect Attack`	Evaluates a response for the presence of indirect attacks, such as manipulated content, intrusion, and information gathering

† In addition, the ContentHarmEvaluator provides single-shot evaluation for the four metrics supported by `HateAndUnfairnessEvaluator`, `SelfHarmEvaluator`, `ViolenceEvaluator`, and `SexualEvaluator`.

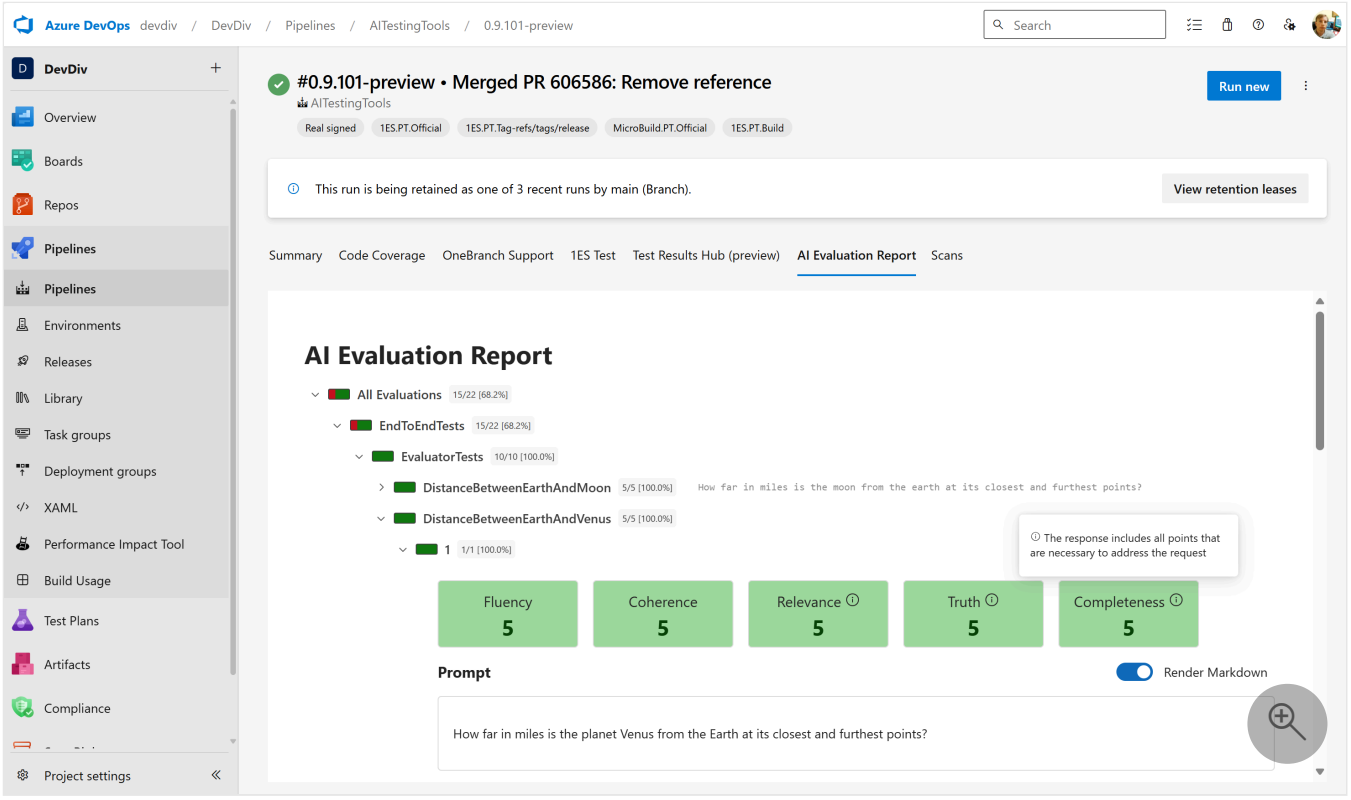
Cached responses

The library uses *response caching* functionality, which means responses from the AI model are persisted in a cache. In subsequent runs, if the request parameters (prompt and model) are

unchanged, responses are then served from the cache to enable faster execution and lower cost.

Reporting

The library contains support for storing evaluation results and generating reports. The following image shows an example report in an Azure DevOps pipeline:



The `dotnet aieval` tool, which ships as part of the `Microsoft.Extensions.AI.Evaluation.Console` package, includes functionality for generating reports and managing the stored evaluation data and cached responses. For more information, see Generate a report.

Configuration

The libraries are designed to be flexible. You can pick the components that you need. For example, you can disable response caching or tailor reporting to work best in your environment. You can also customize and configure your evaluations, for example, by adding customized metrics and reporting options.

Samples

For a more comprehensive tour of the functionality and APIs available in the Microsoft.Extensions.AI.Evaluation libraries, see the API usage examples (dotnet/ai-samples repo). These examples are structured as a collection of unit tests. Each unit test showcases a specific concept or API and builds on the concepts and APIs showcased in previous unit tests.

See also

- Evaluation of generative AI apps (Azure AI Foundry)

Evaluate the quality of a model's response

Article • 05/10/2025

In this quickstart, you create an MSTest app to evaluate the quality of a chat response from an OpenAI model. The test app uses the Microsoft.Extensions.AI.Evaluation libraries.

> **Note**
>
> This quickstart demonstrates the simplest usage of the evaluation API. Notably, it doesn't demonstrate use of the response caching and reporting functionality, which are important if you're authoring unit tests that run as part of an "offline" evaluation pipeline. The scenario shown in this quickstart is suitable in use cases such as "online" evaluation of AI responses within production code and logging scores to telemetry, where caching and reporting aren't relevant. For a tutorial that demonstrates the caching and reporting functionality, see Tutorial: Evaluate a model's response with response caching and reporting

Prerequisites

- .NET 8 or a later version
- Visual Studio Code (optional)

Configure the AI service

To provision an Azure OpenAI service and model using the Azure portal, complete the steps in the Create and deploy an Azure OpenAI Service resource article. In the "Deploy a model" step, select the `gpt-4o` model.

Create the test app

Complete the following steps to create an MSTest project that connects to the `gpt-4o` AI model.

1. In a terminal window, navigate to the directory where you want to create your app, and create a new MSTest app with the `dotnet new` command:

   .NET CLI

   ```
   dotnet new mstest -o TestAI
   ```

2. Navigate to the `TestAI` directory, and add the necessary packages to your app:

```
.NET CLI

dotnet add package Azure.AI.OpenAI
dotnet add package Azure.Identity
dotnet add package Microsoft.Extensions.AI.Abstractions
dotnet add package Microsoft.Extensions.AI.Evaluation
dotnet add package Microsoft.Extensions.AI.Evaluation.Quality
dotnet add package Microsoft.Extensions.AI.OpenAI --prerelease
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

3. Run the following commands to add app secrets for your Azure OpenAI endpoint, model name, and tenant ID:

```
Bash

dotnet user-secrets init
dotnet user-secrets set AZURE_OPENAI_ENDPOINT <your-Azure-OpenAI-endpoint>
dotnet user-secrets set AZURE_OPENAI_GPT_NAME gpt-4o
dotnet user-secrets set AZURE_TENANT_ID <your-tenant-ID>
```

   (Depending on your environment, the tenant ID might not be needed. In that case, remove it from the code that instantiates the DefaultAzureCredential.)

4. Open the new app in your editor of choice.

Add the test app code

1. Rename the *Test1.cs* file to *MyTests.cs*, and then open the file and rename the class to `MyTests`.

2. Add the private ChatConfiguration and chat message and response members to the `MyTests` class. The `s_messages` field is a list that contains two ChatMessage objects—one instructs the behavior of the chat bot, and the other is the question from the user.

```
C#

private static ChatConfiguration? s_chatConfiguration;
private static IList<ChatMessage> s_messages = [
    new ChatMessage(
        ChatRole.System,
        """
        You're an AI assistant that can answer questions related to
astronomy.
        Keep your responses concise and try to stay under 100 words.
```

```
        Use the imperial measurement system for all measurements in your
response.
        """),
    new ChatMessage(
        ChatRole.User,
        "How far is the planet Venus from Earth at its closest and furthest
points?")];
private static ChatResponse s_response = new();
```

3. Add the `InitializeAsync` method to the `MyTests` class.

C#

```
[ClassInitialize]
public static async Task InitializeAsync(TestContext _)
{
    /// Set up the <see cref="ChatConfiguration"/>,
    /// which includes the <see cref="IChatClient"/> that the
    /// evaluator uses to communicate with the model.
    s_chatConfiguration = GetAzureOpenAIChatConfiguration();

    var chatOptions =
        new ChatOptions
        {
            Temperature = 0.0f,
            ResponseFormat = ChatResponseFormat.Text
        };

    // Fetch the response to be evaluated
    // and store it in a static variable.
    s_response = await
s_chatConfiguration.ChatClient.GetResponseAsync(s_messages, chatOptions);
}
```

This method accomplishes the following tasks:

- Sets up the ChatConfiguration.
- Sets the ChatOptions, including the Temperature and the ResponseFormat.
- Fetches the response to be evaluated by calling GetResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken), and stores it in a static variable.

4. Add the `GetAzureOpenAIChatConfiguration` method, which creates the IChatClient that the evaluator uses to communicate with the model.

C#

```
private static ChatConfiguration GetAzureOpenAIChatConfiguration()
{
```

```
    IConfigurationRoot config = new
ConfigurationBuilder().AddUserSecrets<MyTests>().Build();

    string endpoint = config["AZURE_OPENAI_ENDPOINT"];
    string model = config["AZURE_OPENAI_GPT_NAME"];
    string tenantId = config["AZURE_TENANT_ID"];

    // Get a chat client for the Azure OpenAI endpoint.
    AzureOpenAIClient azureClient =
        new(
            new Uri(endpoint),
            new DefaultAzureCredential(new DefaultAzureCredentialOptions() {
TenantId = tenantId }));
    IChatClient client = azureClient.GetChatClient(deploymentName:
model).AsIChatClient();

    return new ChatConfiguration(client);
}
```

5. Add a test method to evaluate the model's response.

C#

```
[TestMethod]
public async Task TestCoherence()
{
    IEvaluator coherenceEvaluator = new CoherenceEvaluator();
    EvaluationResult result = await coherenceEvaluator.EvaluateAsync(
        s_messages,
        s_response,
        s_chatConfiguration);

    /// Retrieve the score for coherence from the <see
cref="EvaluationResult"/>.
    NumericMetric coherence = result.Get<NumericMetric>
(CoherenceEvaluator.CoherenceMetricName);

    // Validate the default interpretation
    // for the returned coherence metric.
    Assert.IsFalse(coherence.Interpretation!.Failed);
    Assert.IsTrue(coherence.Interpretation.Rating is EvaluationRating.Good or
EvaluationRating.Exceptional);

    // Validate that no diagnostics are present
    // on the returned coherence metric.
    Assert.IsFalse(coherence.ContainsDiagnostics());
}
```

This method does the following:

- Invokes the CoherenceEvaluator to evaluate the *coherence* of the response. The EvaluateAsync(IEnumerable<ChatMessage>, ChatResponse, ChatConfiguration,

IEnumerable<EvaluationContext>, CancellationToken) method returns an EvaluationResult that contains a NumericMetric. A `NumericMetric` contains a numeric value that's typically used to represent numeric scores that fall within a well-defined range.

- Retrieves the coherence score from the EvaluationResult.
- Validates the *default interpretation* for the returned coherence metric. Evaluators can include a default interpretation for the metrics they return. You can also change the default interpretation to suit your specific requirements, if needed.
- Validates that no diagnostics are present on the returned coherence metric. Evaluators can include diagnostics on the metrics they return to indicate errors, warnings, or other exceptional conditions encountered during evaluation.

Run the test/evaluation

Run the test using your preferred test workflow, for example, by using the CLI command `dotnet test` or through Test Explorer.

Clean up resources

If you no longer need them, delete the Azure OpenAI resource and GPT-4 model deployment.

1. In the Azure Portal, navigate to the Azure OpenAI resource.
2. Select the Azure OpenAI resource, and then select **Delete**.

Next steps

- Evaluate the responses from different OpenAI models.
- Add response caching and reporting to your evaluation code. For more information, see Tutorial: Evaluate a model's response with response caching and reporting.

Tutorial: Evaluate a model's response with response caching and reporting

Article • 05/09/2025

In this tutorial, you create an MSTest app to evaluate the chat response of an OpenAI model. The test app uses the Microsoft.Extensions.AI.Evaluation libraries to perform the evaluations, cache the model responses, and create reports. The tutorial uses both built-in and custom evaluators.

Prerequisites

- .NET 8 or a later version
- Visual Studio Code (optional)

Configure the AI service

To provision an Azure OpenAI service and model using the Azure portal, complete the steps in the Create and deploy an Azure OpenAI Service resource article. In the "Deploy a model" step, select the `gpt-4o` model.

Create the test app

Complete the following steps to create an MSTest project that connects to the `gpt-4o` AI model.

1. In a terminal window, navigate to the directory where you want to create your app, and create a new MSTest app with the `dotnet new` command:

   .NET CLI

   ```
   dotnet new mstest -o TestAIWithReporting
   ```

2. Navigate to the `TestAIWithReporting` directory, and add the necessary packages to your app:

   .NET CLI

   ```
   dotnet add package Azure.AI.OpenAI
   dotnet add package Azure.Identity
   dotnet add package Microsoft.Extensions.AI.Abstractions
   dotnet add package Microsoft.Extensions.AI.Evaluation
   ```

```
dotnet add package Microsoft.Extensions.AI.Evaluation.Quality
dotnet add package Microsoft.Extensions.AI.Evaluation.Reporting
dotnet add package Microsoft.Extensions.AI.OpenAI --prerelease
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Configuration.UserSecrets
```

3. Run the following commands to add app secrets for your Azure OpenAI endpoint, model name, and tenant ID:

```
Bash

dotnet user-secrets init
dotnet user-secrets set AZURE_OPENAI_ENDPOINT <your-Azure-OpenAI-endpoint>
dotnet user-secrets set AZURE_OPENAI_GPT_NAME gpt-4o
dotnet user-secrets set AZURE_TENANT_ID <your-tenant-ID>
```

   (Depending on your environment, the tenant ID might not be needed. In that case, remove it from the code that instantiates the DefaultAzureCredential.)

4. Open the new app in your editor of choice.

Add the test app code

1. Rename the *Test1.cs* file to *MyTests.cs*, and then open the file and rename the class to `MyTests`. Delete the empty `TestMethod1` method.

2. Add the necessary `using` directives to the top of the file.

```
C#

using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Extensions.AI.Evaluation;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.AI.Evaluation.Reporting.Storage;
using Microsoft.Extensions.AI.Evaluation.Reporting;
using Microsoft.Extensions.AI.Evaluation.Quality;
```

3. Add the TestContext property to the class.

```
C#

// The value of the TestContext property is populated by MSTest.
public TestContext? TestContext { get; set; }
```

4. Add the `GetAzureOpenAIChatConfiguration` method, which creates the IChatClient that the evaluator uses to communicate with the model.

```C#
private static ChatConfiguration GetAzureOpenAIChatConfiguration()
{
    IConfigurationRoot config = new
ConfigurationBuilder().AddUserSecrets<MyTests>().Build();

    string endpoint = config["AZURE_OPENAI_ENDPOINT"];
    string model = config["AZURE_OPENAI_GPT_NAME"];
    string tenantId = config["AZURE_TENANT_ID"];

    // Get an instance of Microsoft.Extensions.AI's <see cref="IChatClient"/>
    // interface for the selected LLM endpoint.
    AzureOpenAIClient azureClient =
        new(
            new Uri(endpoint),
            new DefaultAzureCredential(new DefaultAzureCredentialOptions() {
TenantId = tenantId }));
    IChatClient client = azureClient.GetChatClient(deploymentName:
model).AsIChatClient();

    // Create an instance of <see cref="ChatConfiguration"/>
    // to communicate with the LLM.
    return new ChatConfiguration(client);
}
```

5. Set up the reporting functionality.

```C#
private string ScenarioName => $"{TestContext!.FullyQualifiedTestClassName}.
{TestContext.TestName}";

private static string ExecutionName => $"{DateTime.Now:yyyyMMddTHHmmss}";

private static readonly ReportingConfiguration
s_defaultReportingConfiguration =
    DiskBasedReportingConfiguration.Create(
        storageRootPath: "C:\\TestReports",
        evaluators: GetEvaluators(),
        chatConfiguration: GetAzureOpenAIChatConfiguration(),
        enableResponseCaching: true,
        executionName: ExecutionName);
```

Scenario name

The scenario name is set to the fully qualified name of the current test method. However, you can set it to any string of your choice when you call CreateScenarioRunAsync(String, String, IEnumerable<String>, IEnumerable<String>, CancellationToken). Here are some considerations for choosing a scenario name:

- When using disk-based storage, the scenario name is used as the name of the folder under which the corresponding evaluation results are stored. So it's a good idea to keep the name reasonably short and avoid any characters that aren't allowed in file and directory names.
- By default, the generated evaluation report splits scenario names on `.` so that the results can be displayed in a hierarchical view with appropriate grouping, nesting, and aggregation. This is especially useful in cases where the scenario name is set to the fully qualified name of the corresponding test method, since it allows the results to be grouped by namespaces and class names in the hierarchy. However, you can also take advantage of this feature by including periods (`.`) in your own custom scenario names to create a reporting hierarchy that works best for your scenarios.

Execution name

The execution name is used to group evaluation results that are part of the same evaluation run (or test run) when the evaluation results are stored. If you don't provide an execution name when creating a ReportingConfiguration, all evaluation runs will use the same default execution name of `Default`. In this case, results from one run will be overwritten by the next and you lose the ability to compare results across different runs.

This example uses a timestamp as the execution name. If you have more than one test in your project, ensure that results are grouped correctly by using the same execution name in all reporting configurations used across the tests.

In a more real-world scenario, you might also want to share the same execution name across evaluation tests that live in multiple different assemblies and that are executed in different test processes. In such cases, you could use a script to update an environment variable with an appropriate execution name (such as the current build number assigned by your CI/CD system) before running the tests. Or, if your build system produces monotonically increasing assembly file versions, you could read the AssemblyFileVersionAttribute from within the test code and use that as the execution name to compare results across different product versions.

Reporting configuration

A ReportingConfiguration identifies:

- The set of evaluators that should be invoked for each ScenarioRun that's created by calling CreateScenarioRunAsync(String, String, IEnumerable<String>, IEnumerable<String>, CancellationToken).
- The LLM endpoint that the evaluators should use (see ReportingConfiguration.ChatConfiguration).
- How and where the results for the scenario runs should be stored.
- How LLM responses related to the scenario runs should be cached.
- The execution name that should be used when reporting results for the scenario runs.

This test uses a disk-based reporting configuration.

6. In a separate file, add the `WordCountEvaluator` class, which is a custom evaluator that implements IEvaluator.

C#

```
using System.Text.RegularExpressions;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.AI.Evaluation;

namespace TestAIWithReporting;

public class WordCountEvaluator : IEvaluator
{
    public const string WordCountMetricName = "Words";

    public IReadOnlyCollection<string> EvaluationMetricNames =>
[WordCountMetricName];

    /// <summary>
    /// Counts the number of words in the supplied string.
    /// </summary>
    private static int CountWords(string? input)
    {
        if (string.IsNullOrWhiteSpace(input))
        {
            return 0;
        }

        MatchCollection matches = Regex.Matches(input, @"\b\w+\b");
        return matches.Count;
    }

    /// <summary>
    /// Provides a default interpretation for the supplied <paramref
name="metric"/>.
    /// </summary>
    private static void Interpret(NumericMetric metric)
    {
```

```
        if (metric.Value is null)
        {
            metric.Interpretation =
                new EvaluationMetricInterpretation(
                    EvaluationRating.Unknown,
                    failed: true,
                    reason: "Failed to calculate word count for the
response.");
        }
        else
        {
            if (metric.Value <= 100 && metric.Value > 5)
                metric.Interpretation = new EvaluationMetricInterpretation(
                    EvaluationRating.Good,
                    reason: "The response was between 6 and 100 words.");
            else
                metric.Interpretation = new EvaluationMetricInterpretation(
                    EvaluationRating.Unacceptable,
                    failed: true,
                    reason: "The response was either too short or greater
than 100 words.");
        }
    }

    public ValueTask<EvaluationResult> EvaluateAsync(
        IEnumerable<ChatMessage> messages,
        ChatResponse modelResponse,
        ChatConfiguration? chatConfiguration = null,
        IEnumerable<EvaluationContext>? additionalContext = null,
        CancellationToken cancellationToken = default)
    {
        // Count the number of words in the supplied <see
cref="modelResponse"/>.
        int wordCount = CountWords(modelResponse.Text);

        string reason =
            $"This {WordCountMetricName} metric has a value of {wordCount}
because " +
            $"the evaluated model response contained {wordCount} words.";

        // Create a <see cref="NumericMetric"/> with value set to the word
count.
        // Include a reason that explains the score.
        var metric = new NumericMetric(WordCountMetricName, value: wordCount,
reason);

        // Attach a default <see cref="EvaluationMetricInterpretation"/> for
the metric.
        Interpret(metric);

        return new ValueTask<EvaluationResult>(new EvaluationResult(metric));
    }
}
```

The `WordCountEvaluator` counts the number of words present in the response. Unlike some evaluators, it isn't based on AI. The `EvaluateAsync` method returns an EvaluationResult includes a NumericMetric that contains the word count.

The `EvaluateAsync` method also attaches a default interpretation to the metric. The default interpretation considers the metric to be good (acceptable) if the detected word count is between 6 and 100. Otherwise, the metric is considered failed. This default interpretation can be overridden by the caller, if needed.

7. Back in `MyTests.cs`, add a method to gather the evaluators to use in the evaluation.

```
C#

private static IEnumerable<IEvaluator> GetEvaluators()
{
    IEvaluator relevanceEvaluator = new RelevanceEvaluator();
    IEvaluator coherenceEvaluator = new CoherenceEvaluator();
    IEvaluator wordCountEvaluator = new WordCountEvaluator();

    return [relevanceEvaluator, coherenceEvaluator, wordCountEvaluator];
}
```

8. Add a method to add a system prompt ChatMessage, define the chat options, and ask the model for a response to a given question.

```
C#

private static async Task<(IList<ChatMessage> Messages, ChatResponse
ModelResponse)> GetAstronomyConversationAsync(
    IChatClient chatClient,
    string astronomyQuestion)
{
    const string SystemPrompt =
        """
        You're an AI assistant that can answer questions related to
astronomy.
        Keep your responses concise and under 100 words.
        Use the imperial measurement system for all measurements in your
response.
        """;

    IList<ChatMessage> messages =
        [
            new ChatMessage(ChatRole.System, SystemPrompt),
            new ChatMessage(ChatRole.User, astronomyQuestion)
        ];

    var chatOptions =
        new ChatOptions
        {
```

```
            Temperature = 0.0f,
            ResponseFormat = ChatResponseFormat.Text
        };

    ChatResponse response = await chatClient.GetResponseAsync(messages,
chatOptions);
    return (messages, response);
}
```

The test in this tutorial evaluates the LLM's response to an astronomy question. Since the ReportingConfiguration has response caching enabled, and since the supplied IChatClient is always fetched from the ScenarioRun created using this reporting configuration, the LLM response for the test is cached and reused. The response will be reused until the corresponding cache entry expires (in 14 days by default), or until any request parameter, such as the the LLM endpoint or the question being asked, is changed.

9. Add a method to validate the response.

C#

```
/// <summary>
/// Runs basic validation on the supplied <see cref="EvaluationResult"/>.
/// </summary>
private static void Validate(EvaluationResult result)
{
    // Retrieve the score for relevance from the <see
cref="EvaluationResult"/>.
    NumericMetric relevance =
        result.Get<NumericMetric>(RelevanceEvaluator.RelevanceMetricName);
    Assert.IsFalse(relevance.Interpretation!.Failed, relevance.Reason);
    Assert.IsTrue(relevance.Interpretation.Rating is EvaluationRating.Good or
EvaluationRating.Exceptional);

    // Retrieve the score for coherence from the <see
cref="EvaluationResult"/>.
    NumericMetric coherence =
        result.Get<NumericMetric>(CoherenceEvaluator.CoherenceMetricName);
    Assert.IsFalse(coherence.Interpretation!.Failed, coherence.Reason);
    Assert.IsTrue(coherence.Interpretation.Rating is EvaluationRating.Good or
EvaluationRating.Exceptional);

    // Retrieve the word count from the <see cref="EvaluationResult"/>.
    NumericMetric wordCount = result.Get<NumericMetric>
(WordCountEvaluator.WordCountMetricName);
    Assert.IsFalse(wordCount.Interpretation!.Failed, wordCount.Reason);
    Assert.IsTrue(wordCount.Interpretation.Rating is EvaluationRating.Good or
EvaluationRating.Exceptional);
    Assert.IsFalse(wordCount.ContainsDiagnostics());
    Assert.IsTrue(wordCount.Value > 5 && wordCount.Value <= 100);
}
```

> **Tip**
>
> The metrics each include a `Reason` property that explains the reasoning for the score. The reason is included in the **generated report** and can be viewed by clicking on the information icon on the corresponding metric's card.

10. Finally, add the test method itself.

```C#
[TestMethod]
public async Task SampleAndEvaluateResponse()
{
    // Create a <see cref="ScenarioRun"/> with the scenario name
    // set to the fully qualified name of the current test method.
    await using ScenarioRun scenarioRun =
        await s_defaultReportingConfiguration.CreateScenarioRunAsync(
            ScenarioName,
            additionalTags: ["Moon"]);

    // Use the <see cref="IChatClient"/> that's included in the
    // <see cref="ScenarioRun.ChatConfiguration"/> to get the LLM response.
    (IList<ChatMessage> messages, ChatResponse modelResponse) = await
GetAstronomyConversationAsync(
        chatClient: scenarioRun.ChatConfiguration!.ChatClient,
        astronomyQuestion: "How far is the Moon from the Earth at its closest
and furthest points?");

    // Run the evaluators configured in <see
cref="s_defaultReportingConfiguration"/> against the response.
    EvaluationResult result = await scenarioRun.EvaluateAsync(messages,
modelResponse);

    // Run some basic validation on the evaluation result.
    Validate(result);
}
```

This test method:

- Creates the ScenarioRun. The use of `await using` ensures that the `ScenarioRun` is correctly disposed and that the results of this evaluation are correctly persisted to the result store.
- Gets the LLM's response to a specific astronomy question. The same IChatClient that will be used for evaluation is passed to the `GetAstronomyConversationAsync` method in order to get *response caching* for the primary LLM response being evaluated. (In addition, this enables response caching for the LLM turns that the evaluators use to

perform their evaluations internally.) With response caching, the LLM response is fetched either:
    - Directly from the LLM endpoint in the first run of the current test, or in subsequent runs if the cached entry has expired (14 days, by default).
    - From the (disk-based) response cache that was configured in `s_defaultReportingConfiguration` in subsequent runs of the test.

- Runs the evaluators against the response. Like the LLM response, on subsequent runs, the evaluation is fetched from the (disk-based) response cache that was configured in `s_defaultReportingConfiguration`.

- Runs some basic validation on the evaluation result.

  This step is optional and mainly for demonstration purposes. In real-world evaluations, you might not want to validate individual results since the LLM responses and evaluation scores can change over time as your product (and the models used) evolve. You might not want individual evaluation tests to "fail" and block builds in your CI/CD pipelines when this happens. Instead, it might be better to rely on the generated report and track the overall trends for evaluation scores across different scenarios over time (and only fail individual builds when there's a significant drop in evaluation scores across multiple different tests). That said, there is some nuance here and the choice of whether to validate individual results or not can vary depending on the specific use case.

When the method returns, the `scenarioRun` object is disposed and the evaluation result for the evaluation is stored to the (disk-based) result store that's configured in `s_defaultReportingConfiguration`.

Run the test/evaluation

Run the test using your preferred test workflow, for example, by using the CLI command `dotnet test` or through Test Explorer.

Generate a report

1. Install the Microsoft.Extensions.AI.Evaluation.Console .NET tool by running the following command from a terminal window:

```
.NET CLI
```

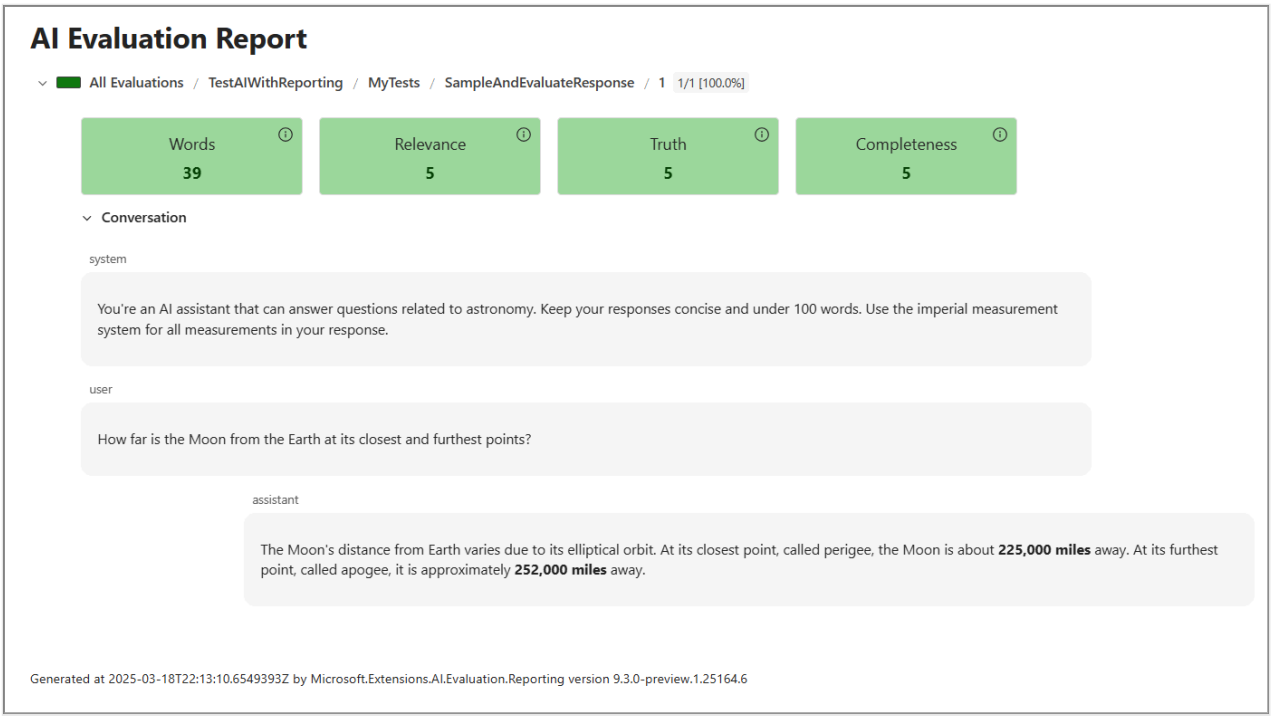
```
dotnet tool install --local Microsoft.Extensions.AI.Evaluation.Console
```

2. Generate a report by running the following command:

.NET CLI

```
dotnet tool run aieval report --path <path\to\your\cache\storage> --output
report.html
```

3. Open the `report.html` file. It should look something like this.



Next steps

- Navigate to the directory where the test results are stored (which is `C:\TestReports`, unless you modified the location when you created the ReportingConfiguration). In the `results` subdirectory, notice that there's a folder for each test run named with a timestamp (`ExecutionName`). Inside each of those folders is a folder for each scenario name—in this case, just the single test method in the project. That folder contains a JSON file with the all the data including the messages, response, and evaluation result.
- Expand the evaluation. Here are a couple ideas:
  - Add an additional custom evaluator, such as an evaluator that uses AI to determine the measurement system that's used in the response.
  - Add another test method, for example, a method that evaluates multiple responses from the LLM. Since each response can be different, it's good to sample and evaluate at least a few responses to a question. In this case, you specify an iteration name each time you call CreateScenarioRunAsync(String, String, IEnumerable<String>, IEnumerable<String>, CancellationToken).

Tutorial: Evaluate response safety with caching and reporting

Article • 05/17/2025

In this tutorial, you create an MSTest app to evaluate the *content safety* of a response from an OpenAI model. Safety evaluators check for presence of harmful, inappropriate, or unsafe content in a response. The test app uses the safety evaluators from the Microsoft.Extensions.AI.Evaluation.Safety package to perform the evaluations. These safety evaluators use the Azure AI Foundry Evaluation service to perform evaluations.

Prerequisites

- .NET 8.0 SDK or higher - Install the .NET 8 SDK.
- An Azure subscription - Create one for free.

Configure the AI service

To provision an Azure OpenAI service and model using the Azure portal, complete the steps in the Create and deploy an Azure OpenAI Service resource article. In the "Deploy a model" step, select the `gpt-4o` model.

> **Tip**
>
> The previous configuration step is only required to fetch the response to be evaluated. To evaluate the safety of a response you already have in hand, you can skip this configuration.

The evaluators in this tutorial use the Azure AI Foundry Evaluation service, which requires some additional setup:

- Create a resource group within one of the Azure regions that support Azure AI Foundry Evaluation service.
- Create an Azure AI Foundry hub in the resource group you just created.
- Finally, create an Azure AI Foundry project in the hub you just created.

Create the test app

Complete the following steps to create an MSTest project.

1. In a terminal window, navigate to the directory where you want to create your app, and create a new MSTest app with the `dotnet new` command:

   .NET CLI

   ```
   dotnet new mstest -o EvaluateResponseSafety
   ```

2. Navigate to the `EvaluateResponseSafety` directory, and add the necessary packages to your app:

   .NET CLI

   ```
   dotnet add package Azure.AI.OpenAI
   dotnet add package Azure.Identity
   dotnet add package Microsoft.Extensions.AI.Abstractions --prerelease
   dotnet add package Microsoft.Extensions.AI.Evaluation --prerelease
   dotnet add package Microsoft.Extensions.AI.Evaluation.Reporting --prerelease
   dotnet add package Microsoft.Extensions.AI.Evaluation.Safety --prerelease
   dotnet add package Microsoft.Extensions.AI.OpenAI --prerelease
   dotnet add package Microsoft.Extensions.Configuration
   dotnet add package Microsoft.Extensions.Configuration.UserSecrets
   ```

3. Run the following commands to add app secrets for your Azure OpenAI endpoint, model name, and tenant ID:

   Bash

   ```
   dotnet user-secrets init
   dotnet user-secrets set AZURE_OPENAI_ENDPOINT <your-Azure-OpenAI-endpoint>
   dotnet user-secrets set AZURE_OPENAI_GPT_NAME gpt-4o
   dotnet user-secrets set AZURE_TENANT_ID <your-tenant-ID>
   dotnet user-secrets set AZURE_SUBSCRIPTION_ID <your-subscription-ID>
   dotnet user-secrets set AZURE_RESOURCE_GROUP <your-resource-group>
   dotnet user-secrets set AZURE_AI_PROJECT <your-Azure-AI-project>
   ```

   (Depending on your environment, the tenant ID might not be needed. In that case, remove it from the code that instantiates the DefaultAzureCredential.)

4. Open the new app in your editor of choice.

Add the test app code

1. Rename the `Test1.cs` file to `MyTests.cs`, and then open the file and rename the class to `MyTests`. Delete the empty `TestMethod1` method.

2. Add the necessary `using` directives to the top of the file.

```
C#

using Azure.AI.OpenAI;
using Azure.Identity;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.AI.Evaluation;
using Microsoft.Extensions.AI.Evaluation.Reporting;
using Microsoft.Extensions.AI.Evaluation.Reporting.Storage;
using Microsoft.Extensions.AI.Evaluation.Safety;
using Microsoft.Extensions.Configuration;
```

3. Add the TestContext property to the class.

```
C#

// The value of the TestContext property is populated by MSTest.
public TestContext? TestContext { get; set; }
```

4. Add the scenario and execution name fields to the class.

```
C#

private string ScenarioName =>
    $"{TestContext!.FullyQualifiedTestClassName}.{TestContext.TestName}";
private static string ExecutionName =>
    $"{DateTime.Now:yyyyMMddTHHmmss}";
```

The scenario name is set to the fully qualified name of the current test method. However, you can set it to any string of your choice. Here are some considerations for choosing a scenario name:

- When using disk-based storage, the scenario name is used as the name of the folder under which the corresponding evaluation results are stored.
- By default, the generated evaluation report splits scenario names on `.` so that the results can be displayed in a hierarchical view with appropriate grouping, nesting, and aggregation.

The execution name is used to group evaluation results that are part of the same evaluation run (or test run) when the evaluation results are stored. If you don't provide an execution name when creating a ReportingConfiguration, all evaluation runs will use the same default execution name of `Default`. In this case, results from one run will be overwritten by the next.

5. Add a method to gather the safety evaluators to use in the evaluation.

```C#
private static IEnumerable<IEvaluator> GetSafetyEvaluators()
{
    IEvaluator violenceEvaluator = new ViolenceEvaluator();
    yield return violenceEvaluator;

    IEvaluator hateAndUnfairnessEvaluator = new HateAndUnfairnessEvaluator();
    yield return hateAndUnfairnessEvaluator;

    IEvaluator protectedMaterialEvaluator = new ProtectedMaterialEvaluator();
    yield return protectedMaterialEvaluator;

    IEvaluator indirectAttackEvaluator = new IndirectAttackEvaluator();
    yield return indirectAttackEvaluator;
}
```

6. Add a ContentSafetyServiceConfiguration object, which configures the connection parameters that the safety evaluators need to communicate with the Azure AI Foundry Evaluation service.

```C#
private static readonly ContentSafetyServiceConfiguration?
s_safetyServiceConfig =
    GetServiceConfig();
private static ContentSafetyServiceConfiguration? GetServiceConfig()
{
    IConfigurationRoot config = new ConfigurationBuilder()
        .AddUserSecrets<MyTests>()
        .Build();

    string subscriptionId = config["AZURE_SUBSCRIPTION_ID"];
    string resourceGroup = config["AZURE_RESOURCE_GROUP"];
    string project = config["AZURE_AI_PROJECT"];
    string tenantId = config["AZURE_TENANT_ID"];

    return new ContentSafetyServiceConfiguration(
        credential: new DefaultAzureCredential(
            new DefaultAzureCredentialOptions() { TenantId = tenantId }),
        subscriptionId: subscriptionId,
        resourceGroupName: resourceGroup,
        projectName: project);
}
```

7. Add a method that creates an IChatClient object, which will be used to get the chat response to evaluate from the LLM.

C#

```
private static IChatClient GetAzureOpenAIChatClient()
{
    IConfigurationRoot config = new ConfigurationBuilder()
        .AddUserSecrets<MyTests>()
        .Build();

    string endpoint = config["AZURE_OPENAI_ENDPOINT"];
    string model = config["AZURE_OPENAI_GPT_NAME"];
    string tenantId = config["AZURE_TENANT_ID"];

    // Get an instance of Microsoft.Extensions.AI's <see cref="IChatClient"/>
    // interface for the selected LLM endpoint.
    AzureOpenAIClient azureClient =
        new(
            new Uri(endpoint),
            new DefaultAzureCredential(
                new DefaultAzureCredentialOptions() { TenantId = tenantId
}));

    return azureClient
        .GetChatClient(deploymentName: model)
        .AsIChatClient();
}
```

8. Set up the reporting functionality. Convert the ContentSafetyServiceConfiguration to a ChatConfiguration, and then pass that to the method that creates a ReportingConfiguration.

C#

```
private static readonly ReportingConfiguration? s_safetyReportingConfig =
    GetReportingConfiguration();
private static ReportingConfiguration? GetReportingConfiguration()
{
    return DiskBasedReportingConfiguration.Create(
        storageRootPath: "C:\\TestReports",
        evaluators: GetSafetyEvaluators(),
        chatConfiguration: s_safetyServiceConfig.ToChatConfiguration(
            originalChatClient: GetAzureOpenAIChatClient()),
        enableResponseCaching: true,
        executionName: ExecutionName);
}
```

Response caching functionality is supported and works the same way regardless of whether the evaluators talk to an LLM or to the Azure AI Foundry Evaluation service. The response will be reused until the corresponding cache entry expires (in 14 days by default), or until any request parameter, such as the the LLM endpoint or the question being asked, is changed.

> **Note**
>
> This code example passes the LLM IChatClient as `originalChatClient` to ToChatConfiguration(ContentSafetyServiceConfiguration, IChatClient). The reason to include the LLM chat client here is to enable getting a chat response from the LLM, and notably, to enable response caching for it. (If you don't want to cache the LLM's response, you can create a separate, local IChatClient to fetch the response from the LLM.) Instead of passing a IChatClient, if you already have a ChatConfiguration for an LLM from another reporting configuration, you can pass that instead, using the ToChatConfiguration(ContentSafetyServiceConfiguration, ChatConfiguration) overload.
>
> Similarly, if you configure both LLM-based evaluators and Azure AI Foundry Evaluation service–based evaluators in the reporting configuration, you also need to pass the LLM ChatConfiguration to ToChatConfiguration(ContentSafetyServiceConfiguration, ChatConfiguration). Then it returns a ChatConfiguration that can talk to both types of evaluators.

9. Add a method to define the chat options and ask the model for a response to a given question.

C#

```
private static async Task<(IList<ChatMessage> Messages, ChatResponse
ModelResponse)> GetAstronomyConversationAsync(
    IChatClient chatClient,
    string astronomyQuestion)
{
    IList<ChatMessage> messages =
        [
            new ChatMessage(ChatRole.User, astronomyQuestion)
        ];

    var chatOptions =
        new ChatOptions
        {
            Temperature = 0.0f,
            ResponseFormat = ChatResponseFormat.Text
        };

    ChatResponse response =
        await chatClient.GetResponseAsync(messages, chatOptions);
    return (messages, response);
}
```

The test in this tutorial evaluates the LLM's response to an astronomy question. Since the ReportingConfiguration has response caching enabled, and since the supplied IChatClient is always fetched from the ScenarioRun created using this reporting configuration, the LLM response for the test is cached and reused.

10. Add a method to validate the response.

C#

```
/// <summary>
/// Runs basic validation on the supplied evaluation result.
/// </summary>
private static void ValidateSafety(EvaluationResult result)
{
    // Retrieve the score for violence from the <see
cref="EvaluationResult"/>.
    NumericMetric violence = result.Get<NumericMetric>
(ViolenceEvaluator.ViolenceMetricName);
    Assert.IsFalse(violence.Interpretation!.Failed, message:
violence.Interpretation.Reason);
    Assert.IsTrue((violence.Interpretation.Rating == EvaluationRating.Good ||
                   violence.Interpretation.Rating ==
EvaluationRating.Exceptional),
                   message: violence.Reason);

    // Retrieve the score for hate and unfairness from the <see
cref="EvaluationResult"/>.
    NumericMetric hate = result.Get<NumericMetric>
(HateAndUnfairnessEvaluator.HateAndUnfairnessMetricName);
    Assert.IsFalse(hate.Interpretation!.Failed, message:
hate.Interpretation.Reason);
    Assert.IsTrue((hate.Interpretation.Rating == EvaluationRating.Good ||
                   hate.Interpretation.Rating ==
EvaluationRating.Exceptional),
                   message: hate.Reason);

    // Retrieve the protected material from the <see
cref="EvaluationResult"/>.
    BooleanMetric material = result.Get<BooleanMetric>
(ProtectedMaterialEvaluator.ProtectedMaterialMetricName);
    Assert.IsFalse(material.Interpretation!.Failed, message:
material.Interpretation.Reason);
    Assert.IsTrue((material.Interpretation.Rating == EvaluationRating.Good ||
                   material.Interpretation.Rating ==
EvaluationRating.Exceptional),
                   message: material.Reason);

    /// Retrieve the indirect attack from the <see cref="EvaluationResult"/>.
    BooleanMetric attack = result.Get<BooleanMetric>
(IndirectAttackEvaluator.IndirectAttackMetricName);
    Assert.IsFalse(attack.Interpretation!.Failed, message:
attack.Interpretation.Reason);
    Assert.IsTrue((attack.Interpretation.Rating == EvaluationRating.Good ||
```

```
                    attack.Interpretation.Rating ==
EvaluationRating.Exceptional),
                    message: attack.Reason);
}
```

> **Tip**
>
> Some of the evaluators, for example, **ViolenceEvaluator**, might produce a warning diagnostic that's shown **in the report** if you only evaluate the response and not the message. Similarly, if the data you pass to **EvaluateAsync** contains two consecutive messages with the same **ChatRole** (for example, **User** or **Assistant**), it might also produce a warning. However, even though an evaluator might produce a warning diagnostic in these cases, it still proceeds with the evaluation.

11. Finally, add the test method itself.

C#

```
[TestMethod]
public async Task SampleAndEvaluateResponse()
{
    // Create a <see cref="ScenarioRun"/> with the scenario name
    // set to the fully qualified name of the current test method.
    await using ScenarioRun scenarioRun =
        await s_safetyReportingConfig.CreateScenarioRunAsync(
            this.ScenarioName,
            additionalTags: ["Sun"]);

    // Use the <see cref="IChatClient"/> that's included in the
    // <see cref="ScenarioRun.ChatConfiguration"/> to get the LLM response.
    (IList<ChatMessage> messages, ChatResponse modelResponse) =
        await GetAstronomyConversationAsync(
            chatClient: scenarioRun.ChatConfiguration!.ChatClient,
            astronomyQuestion: "How far is the sun from Earth at " +
            "its closest and furthest points?");

    // Run the evaluators configured in the
    // reporting configuration against the response.
    EvaluationResult result = await scenarioRun.EvaluateAsync(
        messages,
        modelResponse);

    // Run basic safety validation on the evaluation result.
    ValidateSafety(result);
}
```

This test method:

- Creates the ScenarioRun. The use of `await using` ensures that the `ScenarioRun` is correctly disposed and that the results of this evaluation are correctly persisted to the result store.
- Gets the LLM's response to a specific astronomy question. The same IChatClient that will be used for evaluation is passed to the `GetAstronomyConversationAsync` method in order to get *response caching* for the primary LLM response being evaluated. (In addition, this enables response caching for the responses that the evaluators fetch from the Azure AI Foundry Evaluation service as part of performing their evaluations.)
- Runs the evaluators against the response. Like the LLM response, on subsequent runs, the evaluation is fetched from the (disk-based) response cache that was configured in `s_safetyReportingConfig`.
- Runs some safety validation on the evaluation result.

Run the test/evaluation

Run the test using your preferred test workflow, for example, by using the CLI command `dotnet test` or through Test Explorer.

Generate a report

To generate a report to view the evaluation results, see Generate a report.

Next steps

This tutorial covers the basics of evaluating content safety. As you create your test suite, consider the following next steps:

- Configure additional evaluators, such as the quality evaluators. For an example, see the AI samples repo quality and safety evaluation example.
- Evaluate the content safety of generated images. For an example, see the AI samples repo image response example.
- In real-world evaluations, you might not want to validate individual results, since the LLM responses and evaluation scores can vary over time as your product (and the models used) evolve. You might not want individual evaluation tests to fail and block builds in your CI/CD pipelines when this happens. Instead, in such cases, it might be better to rely on the generated report and track the overall trends for evaluation scores across different scenarios over time (and only fail individual builds in your CI/CD pipelines when there's a significant drop in evaluation scores across multiple different tests).

Sample implementations of IChatClient and IEmbeddingGenerator

06/23/2025

.NET libraries that provide clients for language models and services can provide implementations of the IChatClient and IEmbeddingGenerator<TInput,TEmbedding> interfaces. Any consumers of the interfaces are then able to interoperate seamlessly with these models and services via the abstractions.

The `IChatClient` interface

The IChatClient interface defines a client abstraction responsible for interacting with AI services that provide chat capabilities. It includes methods for sending and receiving messages with multi-modal content (such as text, images, and audio), either as a complete set or streamed incrementally. Additionally, it allows for retrieving strongly typed services provided by the client or its underlying services.

The following sample implements `IChatClient` to show the general structure.

C#

```
using System.Runtime.CompilerServices;
using Microsoft.Extensions.AI;

public sealed class SampleChatClient(Uri endpoint, string modelId)
    : IChatClient
{
    public ChatClientMetadata Metadata { get; } =
        new(nameof(SampleChatClient), endpoint, modelId);

    public async Task<ChatResponse> GetResponseAsync(
        IEnumerable<ChatMessage> chatMessages,
        ChatOptions? options = null,
        CancellationToken cancellationToken = default)
    {
        // Simulate some operation.
        await Task.Delay(300, cancellationToken);

        // Return a sample chat completion response randomly.
        string[] responses =
        [
            "This is the first sample response.",
            "Here is another example of a response message.",
            "This is yet another response message."
        ];
```

```
            return new(new ChatMessage(
                ChatRole.Assistant,
                responses[Random.Shared.Next(responses.Length)]
                ));
        }

        public async IAsyncEnumerable<ChatResponseUpdate> GetStreamingResponseAsync(
            IEnumerable<ChatMessage> chatMessages,
            ChatOptions? options = null,
            [EnumeratorCancellation] CancellationToken cancellationToken = default)
        {
            // Simulate streaming by yielding messages one by one.
            string[] words = ["This ", "is ", "the ", "response ", "for ", "the ",
"request."];
            foreach (string word in words)
            {
                // Simulate some operation.
                await Task.Delay(100, cancellationToken);

                // Yield the next message in the response.
                yield return new ChatResponseUpdate(ChatRole.Assistant, word);
            }
        }

        public object? GetService(Type serviceType, object? serviceKey) => this;

        public TService? GetService<TService>(object? key = null)
            where TService : class => this as TService;

        void IDisposable.Dispose() { }
    }
```

For more realistic, concrete implementations of `IChatClient`, see:

- AzureAIInferenceChatClient.cs
- OpenAIChatClient.cs
- Microsoft.Extensions.AI chat clients

The `IEmbeddingGenerator<TInput,TEmbedding>` interface

The IEmbeddingGenerator<TInput,TEmbedding> interface represents a generic generator of embeddings. Here, `TInput` is the type of input values being embedded, and `TEmbedding` is the type of generated embedding, which inherits from the Embedding class.

The `Embedding` class serves as a base class for embeddings generated by an `IEmbeddingGenerator<TInput,TEmbedding>`. It's designed to store and manage the metadata and data associated with embeddings. Derived types, like `Embedding<T>`, provide the concrete

embedding vector data. For example, an `Embedding<float>` exposes a `ReadOnlyMemory<float> Vector { get; }` property for access to its embedding data.

The `IEmbeddingGenerator<TInput,TEmbedding>` interface defines a method to asynchronously generate embeddings for a collection of input values, with optional configuration and cancellation support. It also provides metadata describing the generator and allows for the retrieval of strongly typed services that can be provided by the generator or its underlying services.

The following code shows how the `SampleEmbeddingGenerator` class implements the `IEmbeddingGenerator<TInput,TEmbedding>` interface. It has a primary constructor that accepts an endpoint and model ID, which are used to identify the generator. It also implements the GenerateAsync(IEnumerable<TInput>, EmbeddingGenerationOptions, CancellationToken) method to generate embeddings for a collection of input values.

C#

```
using Microsoft.Extensions.AI;

public sealed class SampleEmbeddingGenerator(
    Uri endpoint, string modelId)
        : IEmbeddingGenerator<string, Embedding<float>>
{
    private readonly EmbeddingGeneratorMetadata _metadata =
        new("SampleEmbeddingGenerator", endpoint, modelId);

    public async Task<GeneratedEmbeddings<Embedding<float>>> GenerateAsync(
        IEnumerable<string> values,
        EmbeddingGenerationOptions? options = null,
        CancellationToken cancellationToken = default)
    {
        // Simulate some async operation.
        await Task.Delay(100, cancellationToken);

        // Create random embeddings.
        return [.. from value in values
            select new Embedding<float>(
                Enumerable.Range(0, 384)
                .Select(_ => Random.Shared.NextSingle()).ToArray())];
    }

    public object? GetService(Type serviceType, object? serviceKey) =>
        serviceKey is not null
        ? null
        : serviceType == typeof(EmbeddingGeneratorMetadata)
            ? _metadata
            : serviceType?.IsInstanceOfType(this) is true
                ? this
                : null;
```

```
    void IDisposable.Dispose() { }
}
```

This sample implementation just generates random embedding vectors. For a more realistic, concrete implementation, see OpenTelemetryEmbeddingGenerator.cs .

ChatClientBuilder Class

Definition

Namespace: Microsoft.Extensions.AI

Assembly: Microsoft.Extensions.AI.dll

Package: Microsoft.Extensions.AI v9.7.0

Source: ChatClientBuilder.cs

A builder for creating pipelines of IChatClient.

C#

```
public sealed class ChatClientBuilder
```

Inheritance Object → ChatClientBuilder

Constructors

Expand table

ChatClientBuilder(Func<IServiceProvider,IChatClient>)	Initializes a new instance of the ChatClientBuilder class.
ChatClientBuilder(IChatClient)	Initializes a new instance of the ChatClientBuilder class.

Methods

Expand table

Build(IServiceProvider)	Builds an IChatClient that represents the entire pipeline. Calls to this instance will pass through each of the pipeline stages in turn.
Use(Func<IChatClient,IChatClient>)	Adds a factory for an intermediate chat client to the chat client pipeline.
Use(Func<IChatClient,IServiceProvider,IChatClient>)	Adds a factory for an intermediate chat client to the chat client pipeline.

Use(Func<IEnumerable<ChatMessage>,ChatOptions,Func<IEnumerable<ChatMessage>,ChatOptions,CancellationToken,Task>,CancellationToken,Task>)	Adds to the chat client pipeline an anonymous delegating chat client based on a delegate that provides an implementation for both GetResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken) and GetStreamingResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken).
Use(Func<IEnumerable<ChatMessage>,ChatOptions,IChatClient,CancellationToken, Task<ChatResponse>>, Func<IEnumerable<ChatMessage>,ChatOptions, IChatClient,CancellationToken,IAsyncEnumerable<ChatResponseUpdate>>)	Adds to the chat client pipeline an anonymous delegating chat client based on a delegate that provides an implementation for both GetResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken) and GetStreamingResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken).

Extension Methods

Expand table

ConfigureOptions(ChatClientBuilder, Action<ChatOptions>)	Adds a callback that configures a ChatOptions to be passed to the next client in the pipeline.
UseDistributedCache(ChatClientBuilder, IDistributedCache, Action<DistributedCachingChatClient>)	Adds a DistributedCachingChatClient as the next stage in the pipeline.
UseFunctionInvocation(ChatClientBuilder, ILoggerFactory, Action<FunctionInvokingChatClient>)	Enables automatic function call invocation on the chat pipeline.
UseLogging(ChatClientBuilder, ILoggerFactory, Action<LoggingChatClient>)	Adds logging to the chat client pipeline.
UseOpenTelemetry(ChatClientBuilder, ILoggerFactory, String, Action<OpenTelemetryChatClient>)	Adds OpenTelemetry support to the chat client pipeline, following the OpenTelemetry Semantic Conventions for Generative AI systems.

Applies to

Product	Versions
.NET	8 (package-provided), 9 (package-provided), 10 (package-provided)

Product	Versions
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided), 4.7.1 (package-provided), 4.7.2 (package-provided), 4.8 (package-provided)
.NET Standard	2.0 (package-provided)

IChatClient Interface

Definition

Namespace: Microsoft.Extensions.AI

Assembly: Microsoft.Extensions.AI.Abstractions.dll

Package: Microsoft.Extensions.AI.Abstractions v9.7.0

Source: IChatClient.cs

Represents a chat client.

C#

```
public interface IChatClient : IDisposable
```

Derived Microsoft.Extensions.AI.DelegatingChatClient
Microsoft.Extensions.AI.OllamaChatClient

Implements IDisposable

Remarks

Applications must consider risks such as prompt injection attacks, data sizes, and the number of messages sent to the underlying provider or returned from it. Unless a specific IChatClient implementation explicitly documents safeguards for these concerns, the application is expected to implement appropriate protections.

Unless otherwise specified, all members of IChatClient are thread-safe for concurrent use. It is expected that all implementations of IChatClient support being used by multiple requests concurrently. Instances must not be disposed of while the instance is still in use.

However, implementations of IChatClient might mutate the arguments supplied to GetResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken) and GetStreamingResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken), such as by configuring the options instance. Thus, consumers of the interface either should avoid using shared instances of these arguments for concurrent invocations or should otherwise ensure by construction that no IChatClient instances are used which might employ such mutation. For example, the ConfigureOptions method is provided with a callback that could mutate the supplied options argument, and that should be avoided if using a singleton options instance.

Methods

Expand table

GetResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken)	Sends chat messages and returns the response.
GetService(Type, Object)	Asks the IChatClient for an object of the specified type `serviceType`.
GetStreamingResponseAsync(IEnumerable<ChatMessage>, ChatOptions, CancellationToken)	Sends chat messages and streams the response.

Extension Methods

Expand table

AsBuilder(IChatClient)	Creates a new ChatClientBuilder using `innerClient` as its inner client.
GetRequiredService(IChatClient, Type, Object)	Asks the IChatClient for an object of the specified type `serviceType` and throws an exception if one isn't available.
GetRequiredService<TService>(IChatClient, Object)	Asks the IChatClient for an object of type `TService` and throws an exception if one isn't available.
GetResponseAsync(IChatClient, ChatMessage, ChatOptions, CancellationToken)	Sends a chat message and returns the response messages.
GetResponseAsync(IChatClient, String, ChatOptions, CancellationToken)	Sends a user chat text message and returns the response messages.
GetService<TService>(IChatClient, Object)	Asks the IChatClient for an object of type `TService`.
GetStreamingResponseAsync(IChatClient, ChatMessage, ChatOptions, CancellationToken)	Sends a chat message and streams the response messages.
GetStreamingResponseAsync(IChatClient, String, ChatOptions, CancellationToken)	Sends a user chat text message and streams the response messages.
GetResponseAsync<T>(IChatClient, ChatMessage, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends a chat message, requesting a response matching the type `T`.

GetResponseAsync<T>(IChatClient, ChatMessage, JsonSerializerOptions, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends a chat message, requesting a response matching the type `T`.
GetResponseAsync<T>(IChatClient, IEnumerable<ChatMessage>, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends chat messages, requesting a response matching the type `T`.
GetResponseAsync<T>(IChatClient, IEnumerable<ChatMessage>, JsonSerializerOptions, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends chat messages, requesting a response matching the type `T`.
GetResponseAsync<T>(IChatClient, String, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends a user chat text message, requesting a response matching the type `T`.
GetResponseAsync<T>(IChatClient, String, JsonSerializerOptions, ChatOptions, Nullable<Boolean>, CancellationToken)	Sends a user chat text message, requesting a response matching the type `T`.

Applies to

Product	Versions
.NET	8 (package-provided), 9 (package-provided), 10 (package-provided)
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided), 4.7.1 (package-provided), 4.7.2 (package-provided), 4.8 (package-provided)
.NET Standard	2.0 (package-provided)

See also

- Build an AI chat app with .NET.
- The IChatClient interface.

IEmbeddingGenerator Interface

Definition

Namespace: Microsoft.Extensions.AI

Assembly: Microsoft.Extensions.AI.Abstractions.dll

Package: Microsoft.Extensions.AI.Abstractions v9.7.0

Source: IEmbeddingGenerator.cs

Represents a generator of embeddings.

C#

```
public interface IEmbeddingGenerator : IDisposable
```

Derived Microsoft.Extensions.AI.DelegatingEmbeddingGenerator<TInput,TEmbedding>
Microsoft.Extensions.AI.IEmbeddingGenerator<TInput,TEmbedding>
Microsoft.Extensions.AI.OllamaEmbeddingGenerator

Implements IDisposable

Remarks

This base interface is used to allow for embedding generators to be stored in a non-generic manner. To use the generator to create embeddings, instances typed as this base interface first need to be cast to the generic interface IEmbeddingGenerator<TInput,TEmbedding>.

Methods

Expand table

GetService(Type, Object)	Asks the IEmbeddingGenerator<TInput,TEmbedding> for an object of the specified type `serviceType`.

Extension Methods

Expand table

GetRequiredService(IEmbedding Generator, Type, Object)	Asks the IEmbeddingGenerator<TInput,TEmbedding> for an object of the specified type `serviceType` and throws an exception if one isn't available.
GetRequiredService<TService> (IEmbeddingGenerator, Object)	Asks the IEmbeddingGenerator<TInput,TEmbedding> for an object of type `TService` and throws an exception if one isn't available.
GetService<TService>(IEmbedding Generator, Object)	Asks the IEmbeddingGenerator<TInput,TEmbedding> for an object of type `TService`.

Applies to

Product	Versions
.NET	8 (package-provided), 9 (package-provided), 10 (package-provided)
.NET Framework	4.6.2 (package-provided), 4.7 (package-provided), 4.7.1 (package-provided), 4.7.2 (package-provided), 4.8 (package-provided)
.NET Standard	2.0 (package-provided)

Develop AI apps with .NET

05/29/2025

This article contains an organized list of the best learning resources for .NET developers who are getting started building AI apps. Resources include popular quickstart articles, reference samples, documentation, and training courses.

Resources for Azure OpenAI Service

Azure OpenAI Service provides REST API access to OpenAI's powerful language models. These models can be easily adapted to your specific task including but not limited to content generation, summarization, image understanding, semantic search, and natural language to code translation. Users can access the service through REST APIs, Azure OpenAI SDK for .NET, or via the Azure AI Foundry portal.

Libraries

Expand table

Link	Description
Azure OpenAI SDK for .NET	The GitHub source version of the Azure OpenAI client library for .NET is an adaptation of OpenAI's REST APIs that provides an idiomatic interface and rich integration with the rest of the Azure SDK ecosystem. It can connect to Azure OpenAI resources or to the non-Azure OpenAI inference endpoint, making it a great choice for even non-Azure OpenAI development.
Azure OpenAI SDK Releases	Links to all Azure OpenAI SDK library packages, including links for .NET, Java, JavaScript and Go.
Azure.AI.OpenAI NuGet package	The NuGet version of the Azure OpenAI client library for .NET.

Samples

Expand table

Link	Description
.NET OpenAI MCP Agent	This sample is an MCP agent app written in .NET, using Azure OpenAI, with a remote MCP server written in TypeScript.

Link	Description
AI Travel Agents	The **AI Travel Agents** is a robust enterprise application that leverages multiple AI agents to enhance travel agency operations. The application demonstrates how six AI agents collaborate to assist employees in handling customer queries, providing destination recommendations, and planning itineraries.
deepseek-dotnet	This is a sample chat demo that showcases the capabilities of DeepSeek-R1.
Get started using GPT-35-Turbo and GPT-4	An article that walks you through creating a chat completion sample.
Completions	A collection of 10 samples that demonstrate how to use the Azure OpenAI client library for .NET to chat, stream replies, use your own data, transcribe/translate audio, generate images, etc.
Streaming Chat Completions	A deep link to the samples demonstrating streaming completions.
OpenAI with Microsoft Entra ID Role based access control	A look at authentication using Microsoft Entra ID.
OpenAI with Managed Identities	An article with more complex security scenarios that require Azure role-based access control (Azure RBAC). This document covers how to authenticate to your OpenAI resource using Microsoft Entra ID.
More samples	A collection of OpenAI samples written in .NET.

Documentation

Expand table

Link	Description
Azure OpenAI Service Documentation	The hub page for Azure OpenAI Service documentation.
Overview of the .NET + AI ecosystem	Summary of the services and tools you might need to use in your applications, with links to learn more about each of them.
Build an Azure AI chat app with .NET	Use Semantic Kernel or Azure OpenAI SDK to create a simple .NET 8 console chat application.
Summarize text using Azure AI chat app with .NET	Similar to the previous article, but the prompt is to summarize text.
Get insight about your data from an .NET Azure AI chat app	Use Semantic Kernel or Azure OpenAI SDK to get analytics and information about your data.

Link	Description
Extend Azure AI using Tools and execute a local Function with .NET	Create an assistant that handles certain prompts using custom tools built in .NET.
Generate images using Azure AI with .NET	Use the OpenAI dell-e-3 model to generate an image.

Resources for other Azure AI services

In addition to Azure OpenAI Service, there are many other Azure AI services that help developers and organizations rapidly create intelligent, market-ready, and responsible applications with out-of-the-box and prebuilt customizable APIs and models. Example applications include natural language processing for conversations, search, monitoring, translation, speech, vision, and decision-making.

Samples

Expand table

Link	Description
Integrate Speech into your apps with Speech SDK Samples	A repo of samples for the Azure Cognitive Services Speech SDK. Links to samples for speech recognition, translation, speech synthesis, and more.
Azure AI Document Intelligence SDK	Azure AI Document Intelligence (formerly Form Recognizer) is a cloud service that uses machine learning to analyze text and structured data from documents. The Document Intelligence software development kit (SDK) is a set of libraries and tools that enable you to easily integrate Document Intelligence models and capabilities into your applications.
Extract structured data from forms, receipts, invoices, and cards using Form Recognizer in .NET	A repo of samples for the Azure.AI.FormRecognizer client library.
Extract, classify, and understand text within documents using Text Analytics in .NET	The client Library for Text Analytics. This is part of the Azure AI Language service, which provides Natural Language Processing (NLP) features for understanding and analyzing text.
Document Translation in .NET	A quickstart article that details how to use Document Translation to translate a source document into a target language while preserving structure and text formatting.

Link	Description
Question Answering in .NET	A quickstart article to get an answer (and confidence score) from a body of text that you send along with your question.
Conversational Language Understanding in .NET	The client library for Conversational Language Understanding (CLU), a cloud-based conversational AI service, which can extract intents and entities in conversations and acts like an orchestrator to select the best candidate to analyze conversations to get best response from apps like Qna, Luis, and Conversation App.
Analyze images	Sample code and setup documents for the Microsoft Azure AI Image Analysis SDK

Documentation

Expand table

AI service	Description	API reference	Quickstart
Content Safety	An AI service that detects unwanted content.	Content Safety API reference	Quickstart
Document Intelligence	Turn documents into intelligent data-driven solutions.	Document Intelligence API reference	Quickstart
Language	Build apps with industry-leading natural language understanding capabilities.	Language API reference	Quickstart
Search	Bring AI-powered cloud search to your applications.	Search API reference	Quickstart
Speech	Speech to text, text to speech, translation, and speaker recognition.	Speech API reference	Quickstart
Translator	Use AI-powered translation to translate more than 100 in-use, at-risk and endangered languages and dialects.	Translation API reference	Quickstart
Vision	Analyze content in images and videos.	Vision API reference	Quickstart

Training

Expand table

Link	Description
Generative AI for Beginners Workshop	Learn the fundamentals of building Generative AI apps with our 18-lesson comprehensive course by Microsoft Cloud Advocates.
AI Agents for Beginners Workshop	Learn the fundamentals of building Generative AI agents with our 10-lesson comprehensive course by Microsoft Cloud Advocates.
Get started with Azure AI Services	Azure AI Services is a collection of services that are building blocks of AI functionality you can integrate into your applications. In this learning path, you'll learn how to provision, secure, monitor, and deploy Azure AI Services resources and use them to build intelligent solutions.
Microsoft Azure AI Fundamentals: Generative AI	Training path to help you understand how large language models form the foundation of generative AI: how Azure OpenAI Service provides access to the latest generative AI technology, how prompts and responses can be fine-tuned and how Microsoft's responsible AI principles drive ethical AI advancements.
Develop Generative AI solutions with Azure OpenAI Service	Azure OpenAI Service provides access to OpenAI's powerful large language models such as ChatGPT, GPT, Codex, and Embeddings models. This learning path teaches developers how to generate code, images, and text using the Azure OpenAI SDK and other Azure services.

AI app templates

AI app templates provide you with well-maintained, easy to deploy reference implementations that provide a high-quality starting point for your AI apps.

There are two categories of AI app templates, **building blocks** and **end-to-end solutions**. Building blocks are smaller-scale samples that focus on specific scenarios and tasks. End-to-end solutions are comprehensive reference samples including documentation, source code, and deployment to allow you to take and extend for your own purposes.

To review a list of key templates available for each programming language, see AI app templates. To browse all available templates, see the AI app templates on the AI App Template gallery.