

AZ-2005

# Azure OpenAI と Semantic Kernel SDK を使用して AI エージェントを開発する



## Azure OpenAI と Semantic Kernel SDK を使用して AI エージェントを開発する

4 時間 20 分 • ラーニングパス • 6 モジュール

中級 開発者 .NET Visual Studio Code Azure OpenAI Service

Semantic Kernel SDK を使用して、タスクを自動化し、自然言語処理を実行するインテリジェントなアプリケーションを構築する方法について学習します。

### 前提条件

- C# でのプログラミングの経験。
- Visual Studio Code IDE がインストールされていること。
- Azure と Azure portal に関する知識。
- Azure OpenAI Service へのアクセス。

# 本コースご受講の前提条件

- ・本コースの前提条件は以下の通りです。



## 前提条件

- C# でのプログラミングの経験。
- Visual Studio Code IDE がインストールされていること。
- Azure と Azure portal に関する知識。
- Azure OpenAI Serviceへのアクセス。

C#については必要に応じて別途入門書などで学習してください

Azure OpenAI Serviceについては、本コースでは概要のみ説明します

# 本日の到達目標

- Azure OpenAI Serviceの基礎を理解する
  - リソースの作成と生成AIモデルのデプロイ
  - アプリからのデプロイの利用
- Semantic Kernelの基礎を理解する
  - プロンプトの実行
  - チャットの実装
  - 会話履歴の管理
  - プラグインの利用
  - 独自のプラグインの作成
  - プラグインの関数の呼び出し方法（明示的・自動的）の理解
- Semantic Kernel を使用するAIエージェントの構築方法を理解する
  - 家電をコントロールするAIエージェントの作成

# AZ-2005



## Azure OpenAI と Semantic Kernel SDK を 使用して AI エージェントを開発する

- ・モジュール1 カーネルの構築
  - ・モジュール2 Semantic Kernel プラグインの作成
  - ・モジュール3 AIエージェントにスキルを与える
  - ・モジュール4 プロンプトと関数を組み合わせる
  - ・モジュール5 関数の自動呼び出し
  - ・モジュール6 AIエージェントの作成例
- 
- The diagram illustrates the structure of the development process. It shows six numbered modules on the left, each associated with one of four categories on the right via blue curly braces. The categories are: '概要' (Overview) for module 1; 'Plugin (Function) Creation Method' for modules 2 and 3; 'Plugin (Function) Invocation Method' for modules 4 and 5; and 'PoC (Concept Proof)' for module 6.
- ・モジュール1 カーネルの構築
  - ・モジュール2 Semantic Kernel プラグインの作成
  - ・モジュール3 AIエージェントにスキルを与える
  - ・モジュール4 プロンプトと関数を組み合わせる
  - ・モジュール5 関数の自動呼び出し
  - ・モジュール6 AIエージェントの作成例
- 概要
- Plugin (Function) Creation Method
- Plugin (Function) Invocation Method
- PoC (Concept Proof)



## カーネルを構築する

42 分 • モジュール • 8 ユニット

[△ フィードバック](#)

中級

開発者

.NET

Visual Studio Code

Azure OpenAI Service

このモジュールでは、Semantic Kernel SDK を紹介します。カーネルがコードを大規模な言語モデルに接続して、生成人工知能を使用して機能を拡張する方法について学習します。

### 学習の目的

- セマンティック カーネルの目的を理解する。
- プロンプトの基本を理解する。
- より効果的なプロンプトの手法を学習する。

# モジュール1

- 「Azure OpenAI Service」とは？
- 「エージェント」とは？
- 「Semantic Kernel」とは？
- Semantic Kernelの「カーネル」とは？
- Semantic Kernelの「コネクター」とは？
- Semantic Kernelの基本的な使い方
- Semantic Kernelでの会話履歴の管理
- Semantic Kernelによる画像の生成

# Azure OpenAI Serviceとは

- OpenAI社が開発した生成AIモデルをAzure上で利用するためのサービス
- GPTやDALL-Eなどの生成AIモデルを利用できる
  - GPT (Generative Pre-trained Transformer): テキストを生成
  - DALL-E: 画像を生成
- ユーザーは「プロンプト」で指示を与える

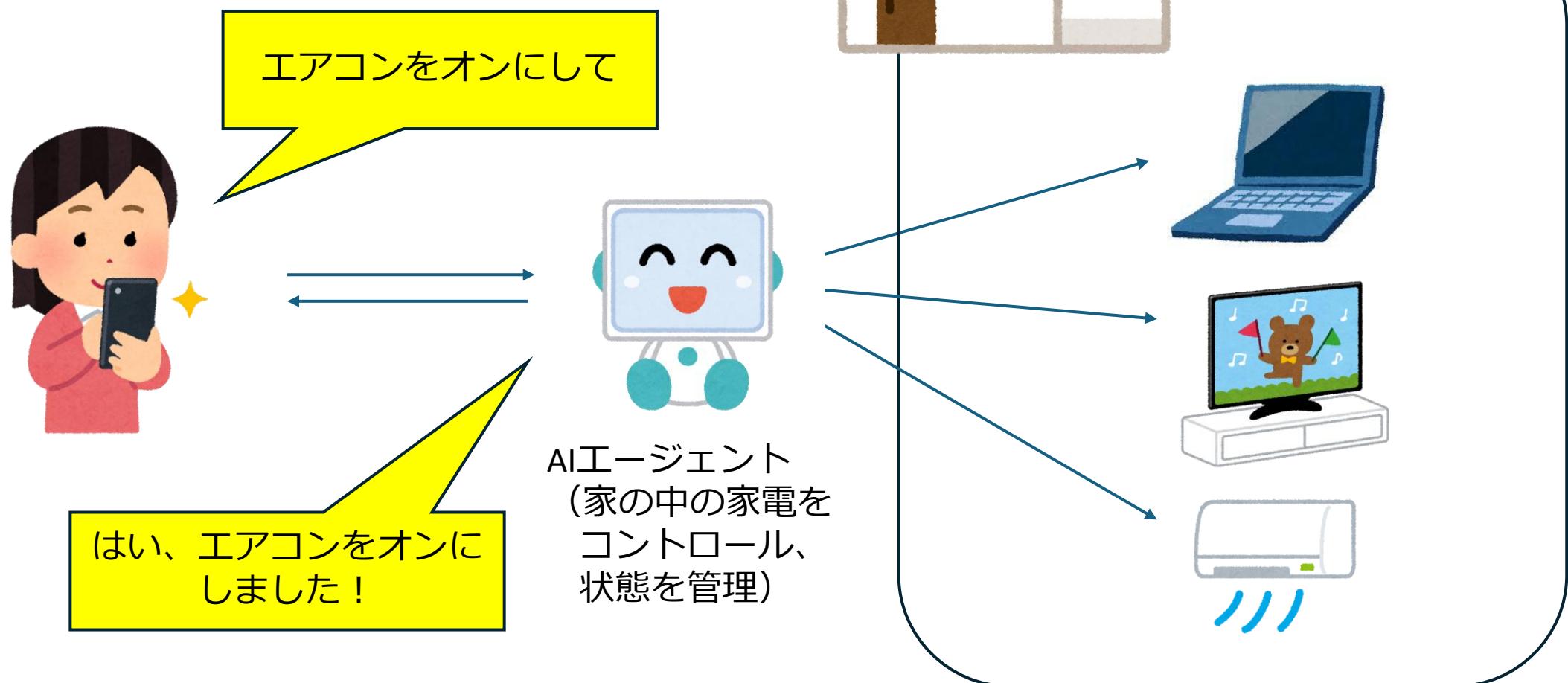
# モジュール1

- ・「Azure OpenAI Service」とは？
- ・「エージェント」とは？
- ・「Semantic Kernel」とは？
- ・Semantic Kernelの「カーネル」とは？
- ・Semantic Kernelの「コネクター」とは？
- ・Semantic Kernelの基本的な使い方
- ・Semantic Kernelでの会話履歴の管理
- ・Semantic Kernelによる画像の生成

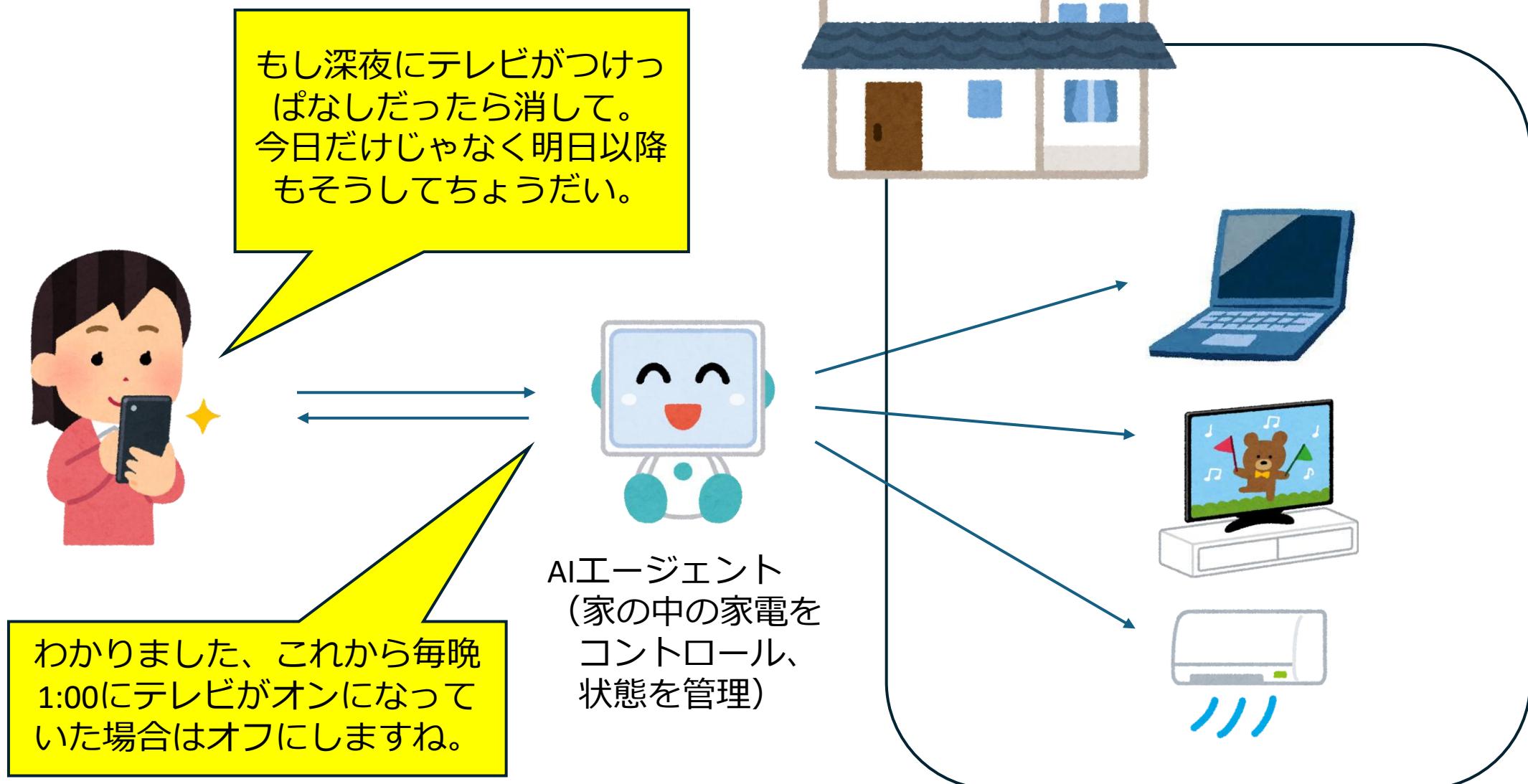
# 「エージェント」とは？

- agent: 代理人、仲介業者
- 人間の仕事の一部などを代行できるソフトウェア
- 専門的な知識や技術を持ち、複雑な作業を実行できる
- GPTなどの生成AI技術を使用して実装される

# 「エージェント」のイメージ



## 自律的に稼働する「エージェント」のイメージ



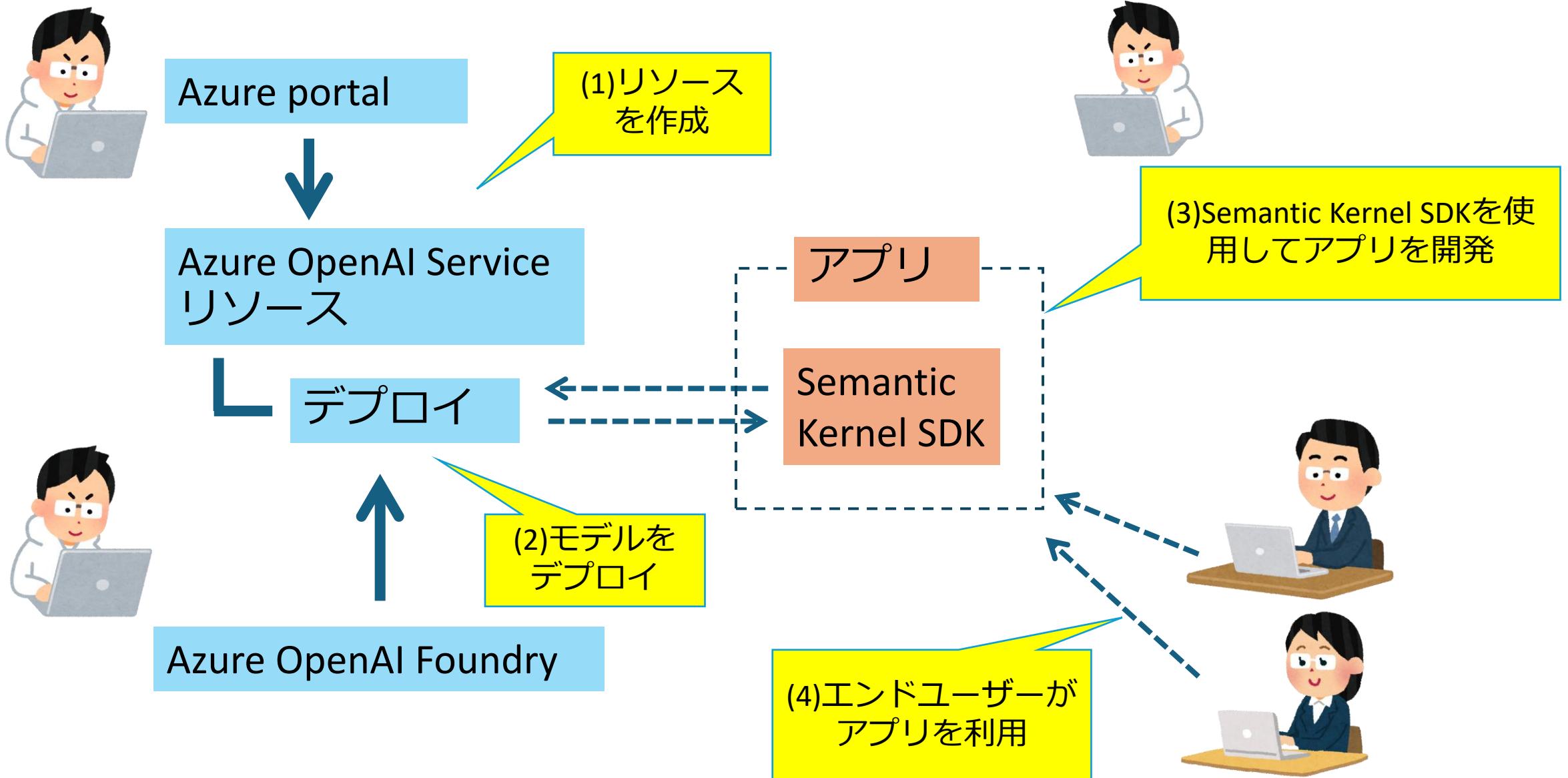
# モジュール1

- ・「Azure OpenAI Service」とは？
  - ・「エージェント」とは？
  - ・「Semantic Kernel」とは？
- 
- ・Semantic Kernelの「カーネル」とは？
  - ・Semantic Kernelの「コネクター」とは？
  - ・Semantic Kernelの基本的な使い方
  - ・Semantic Kernelでの会話履歴の管理
  - ・Semantic Kernelによる画像の生成

# Semantic Kernel とは？

- ・生成AIアプリや**AIエージェント**の開発に役立つフレームワーク
- ・生成AIモデルを利用するアプリを素早く開発できる
- ・オープンソースで開発されている（Microsoftが主導）
- ・**Microsoftによるサポートが提供される**
- ・無料で利用できる
- ・C#/Python/Javaに対応
- ・**プラグイン**による機能追加が可能
- ・様々な生成AIモデルを利用できる

# Semantic Kernelを使用したアプリの開発



## ■ Semantic Kernel (C# / Python) のGitHubリポジトリ

README Code of conduct MIT license Security

[+ 948](#)

## Semantic Kernel

### Status

- Python  
[pypi v1.18.2](#)
- .NET  
[nuget v1.33.0](#) [dotnet-ci-docker no status](#) [dotnet-ci-windows no status](#)

### Overview

license [MIT](#) [Discord](#) [440 online](#)

[Semantic Kernel](#) is an SDK that integrates Large Language Models (LLMs) like [OpenAI](#), [Azure OpenAI](#), and [Hugging Face](#) with conventional programming languages like C#, Python, and Java. Semantic Kernel achieves this by allowing you to define [plugins](#) that can be chained together in just a [few lines of code](#).

### Contributors 344



[+ 330 contributors](#)

### Deployments 500+

[integration inactive](#)

[+ more deployments](#)

### Languages



C# 70.4%	Python 27.4%
Jupyter Notebook 2.1%	
Handlebars 0.1%	PowerShell 0.0%
F# 0.0%	

## ■ Semantic Kernel (Java) のGitHubリポジトリ

README    Code of conduct    MIT license    Security     

---

 Build Java Semantic Kernel passing  license MIT  Discord 440 online

# Semantic Kernel for Java

---

Welcome to the Semantic Kernel for Java. For detailed documentation, visit [Microsoft Learn](#).

[Semantic Kernel](#) is an SDK that integrates Large Language Models (LLMs) like [OpenAI](#), [Azure OpenAI](#), and [Hugging Face](#) with conventional programming languages like C#, Python, and Java. Semantic Kernel achieves this by allowing you to define [plugins](#) that can be chained together in just a [few lines of code](#).

What makes Semantic Kernel *special*, however, is its ability to *automatically* orchestrate plugins with AI. With Semantic Kernel [planners](#), you can ask an LLM to generate a plan that achieves a user's unique goal. Afterwards, Semantic Kernel will execute the plan for the user.

■各言語におけるAIサービスのサポート状況（現在、C#が最も充実している。本コースではC#コードサンプルを使用）

# Adding AI services to Semantic Kernel

Article • 06/25/2024 • 4 contributors

 Feedback

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

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

[+] Expand table

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

# モジュール1

- ・「Azure OpenAI Service」とは？
- ・「エージェント」とは？
- ・「Semantic Kernel」とは？
- ・Semantic Kernelの「カーネル」とは？
- ・Semantic Kernelの「コネクター」とは？
- ・Semantic Kernelの基本的な使い方
- ・Semantic Kernelでの会話履歴の管理
- ・Semantic Kernelによる画像の生成

# カーネル (Kernel) とは

- kernel: 中心部、核心、種子、穀粒（米、麦、とうもろこし）
- Semantic Kernelのカーネル:
  - Semantic Kernelによるプログラミングの中心となるオブジェクト
  - 生成AIモデルとの通信を行う「サービス」の管理を行う
    - ITextGenerationService
    - IChatCompletionService
    - ITextToImageService
  - 会話履歴の管理を行う
    - ChatHistory
  - プラグイン（追加機能）の登録や呼び出しを行う
    - コア（組み込み）プラグイン: TimePlugin, ConversationSummaryPlugin など
    - 自作のプラグイン: 家電のコントロールを行うプラグインなど

## ■ カーネルの作成

```
using Microsoft.SemanticKernel;

var builder = Kernel.CreateBuilder();

// ... カーネル (kernel) と生成AIモデルを接続

// ... カーネル (kernel) にプラグインを追加

var kernel = builder.Build();

// ... カーネル (kernel) を利用
```

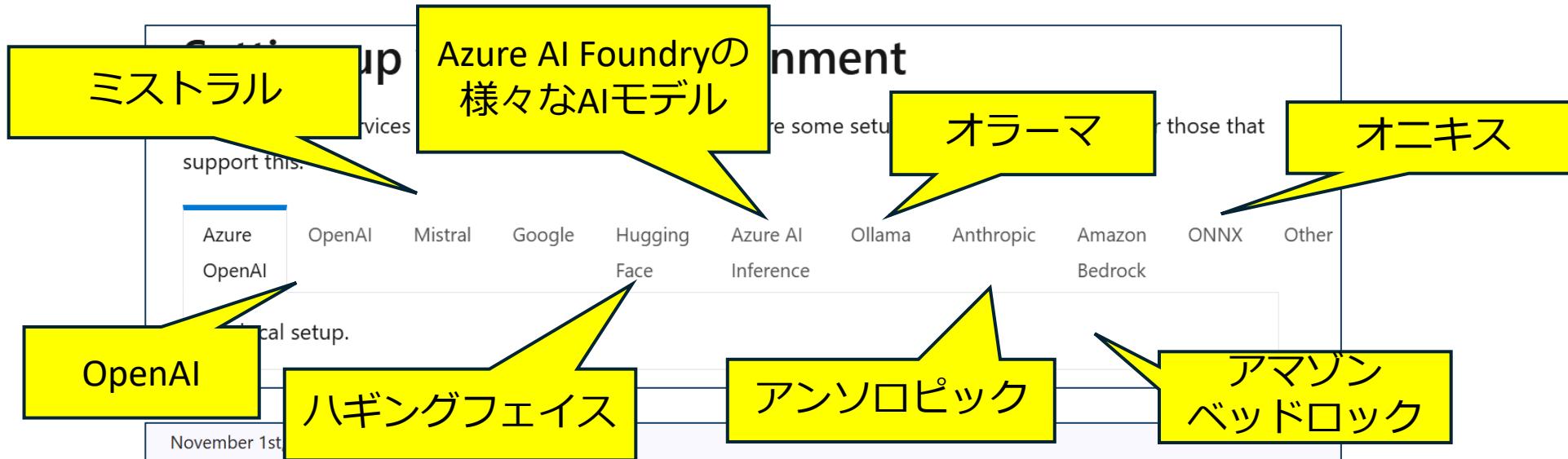
# モジュール1

- ・「Azure OpenAI Service」とは？
- ・「エージェント」とは？
- ・「Semantic Kernel」とは？
- ・Semantic Kernelの「カーネル」とは？
- ・Semantic Kernelの「コネクター」とは？
- ・Semantic Kernelの基本的な使い方
- ・Semantic Kernelでの会話履歴の管理
- ・Semantic Kernelによる画像の生成

# Semantic Kernelのコネクター

- 「コネクター」 (connectors) を使用して、様々なAIモデル、様々な外部サービスを利用できる
    - AIモデル
      - Azure OpenAIコネクター
      - OpenAIコネクター
    - Azureのサービス
      - Cosmos DBコネクター
    - Microsoftのサービス
      - Graphコネクター
    - その他
      - Redisコネクター
      - SQLiteコネクター
      - PostgreSQLコネクター
- Microsoft.SemanticKernel.Connectors.AzureAllInference  
➤ Microsoft.SemanticKernel.Connectors.AzureAISeach  
➤ Microsoft.SemanticKernel.Connectors.  
AzureCosmosDBMongoDB  
➤ Microsoft.SemanticKernel.Connectors.  
AzureCosmosDBNoSQL  
➤ Microsoft.SemanticKernel.Connectors.AzureOpenAI  
➤ Microsoft.SemanticKernel.Connectors.Chroma  
➤ Microsoft.SemanticKernel.Connectors.DuckDB  
➤ Microsoft.SemanticKernel.Connectors.Google  
➤ Microsoft.SemanticKernel.Connectors.HuggingFace  
➤ Microsoft.SemanticKernel.Connectors.Kusto  
➤ Microsoft.SemanticKernel.Connectors.Milvus  
➤ Microsoft.SemanticKernel.Connectors.MistralAI  
➤ Microsoft.SemanticKernel.Connectors.MistralAI.Client  
➤ Microsoft.SemanticKernel.Connectors.MongoDB  
➤ Microsoft.SemanticKernel.Connectors.Onnx  
➤ Microsoft.SemanticKernel.Connectors.OpenAI  
➤ Microsoft.SemanticKernel.Connectors.Pinecone  
➤ Microsoft.SemanticKernel.Connectors.Postgres  
➤ Microsoft.SemanticKernel.Connectors.Qdrant  
➤ Microsoft.SemanticKernel.Connectors.Redis  
➤ Microsoft.SemanticKernel.Connectors.Sqlite  
➤ Microsoft.SemanticKernel.Connectors.SqlServer  
➤ Microsoft.SemanticKernel.Connectors.Weaviate
- <https://learn.microsoft.com/en-us/dotnet/api/microsoft.semantickernel?view=semantic-kernel-dotnet>

■ Semantic Kernelでは、コネクターを介して、Azure OpenAIを含む様々な生成AIモデルの利用が可能



例: Amazon Bedrock上の「タイタン」モデルに接続するコネクターが利用可能

## Introducing AWS Bedrock with Semantic Kernel



Tao, Roger

One of the principal features of Semantic Kernel is its ability to integrate various AI services seamlessly. We are pleased to announce that this capability now extends to [AWS Bedrock](#). With AWS Bedrock, you can access foundational models such as the [Amazon Titan models](#).

The new connector supports Chat Completion, Text Generation, and Text Embeddings, depending on the chosen model.

<https://learn.microsoft.com/en-us/semantic-kernel/concepts/ai-services/chat-completion/?tabs=csharp-AzureOpenAI%2Cpython-AzureOpenAI%2Cjava-AzureOpenAI&pivots=programming-language-csharp#setting-up-your-local-environment>

<https://devblogs.microsoft.com/semantic-kernel/introducing-aws-bedrock-with-semantic-kernel/>

# モジュール1

- ・「Azure OpenAI Service」とは？
  - ・「エージェント」とは？
  - ・「Semantic Kernel」とは？
  - ・Semantic Kernelの「カーネル」とは？
  - ・Semantic Kernelの「コネクター」とは？
  - ・Semantic Kernelの基本的な使い方
- 
- ・Semantic Kernelでの会話履歴の管理
  - ・Semantic Kernelによる画像の生成

# Semantic Kernelの基本的な使い方

- Azure OpenAI Serviceリソースの作成
- キーとエンドポイントの取得
- GPT-4oモデルのデプロイ
- キー、エンドポイント、デプロイ名を環境変数にセット
- C#プロジェクトの作成
- Semantic Kernelパッケージの追加
- Semantic Kernelを使用したコードの記述
- 実行

# Azure OpenAI Service リソースの作成

- Azureサブスクリプションでは、事前に利用申請を済ませておく。
- Azureの他のリソースと同様に、 Azure portal、 Azure CLI、 Azure PowerShell、 ARMテンプレート、 Bicepなどを使用して、 リソースを作成できる。



Azureサブスクリプション



リソースグループ



Azure OpenAI Service リソース

- リソースの作成
- ホーム
- ダッシュボード
- すべてのサービス
- お気に入り
- すべてのリソース
- リソース グループ
- App Service
- 関数アプリ
- SQL データベース
- Azure Cosmos DB
- Virtual Machines
- ロード バランサー
- ストレージ アカウント

Microsoft Azure リソース、サービス、ドキュメントの検索 (G+/)

## すべてのサービス | AI + Machine Learning

すべて サービスのフィルター

お気に入り 最近使用したもの 推奨

カテゴリ

- AI + Machine Learning
- 分析
- コンピューティング
- コンテナー
- データベース
- DevOps
- 全般
- ハイブリッド + マルチクラウド
- ID
- 統合
- モノのインターネット
- Management and governance
- 移行
- Mixed Reality

### Azure AI サービス

- Azure AI services
- Azure AI Video Indexer
- Bot Service
- Computer Vision
- Custom Vision
- Face API
- Language
- Azure OpenAI
- 音声サービス

### 機械学習

- Bonsai
- Azure Machine Learning

Azure AI services multi-service account  
Anomaly Detector  
Cognitive Search  
Content Moderator  
Document intelligences  
Immersive Reader  
Metrics Advisor  
Personalizer  
翻訳  
Intelligent Recommendations アカウント  
Azure Synapse Analytics

[すべてのサービス > Azure AI services](#)

# Azure AI services | Azure OpenAI



Azure AI services



検索



+ 作成



削除されたリソースの管理



ビューの管理



更新



CSV



概要



All Azure AI services

## Azure AI services



Azure OpenAI



Cognitive Search



Computer Vision



Face API



Global AI services

任意のフィールドのフィルター...

サブスクリプション 次の値と等しい すべて

種類

2 件中 1 ~ 2 件のレコードを表示しています。

<input type="checkbox"/> 名前 ↑↓	サブタイプ ↑↓	場所 ↑↓
<input type="checkbox"/> aoaigpt3eastus92837...	OpenAI	East US
<input type="checkbox"/> aoaigpt4canadaeast9...	OpenAI	Canada East

# Azure OpenAI の作成

...

- ① 基本    ② ネットワーク    ③ タグ    ④ レビューおよび送信

GPT-3 モデルを利用した OpenAI の言語生成機能により、新しいビジネス ソリューションを実現します。そのモデルは、何兆もの単語で事前トレーニングされており、推論時にいくつかの短い例を示すことで、シナリオに簡単に適応できます。概要作成からコンテンツ、コード生成まで、さまざまなシナリオに適用できます。

## 詳細情報

### プロジェクトの詳細

サブスクリプション \* ⓘ

リソース グループ \* ⓘ

aoairg3

新規作成

サブスクリプションを選択

リソースグループを選択または作成

### インスタンスの詳細

リージョン ⓘ

East US

リージョンを選択

名前 \* ⓘ

myopenaiservice928374

世界中で一意となる（重複しない）リソース名を入力

価格レベル \* ⓘ

Standard S0

価格レベルを選択（現在 Standard S0のみ）

# モデルのデプロイ



Azureサブスクリプション



リソースグループ



Azure OpenAI Service リソース



gpt-4o

dall-e-3



デプロイ(gpt-4o)

デプロイ(dall-e-3)

ベースモデル

デプロイ: ベースモデルを選択して、どのモデルをどのように使用するかという**設定を作ること**。

Azure OpenAI Serviceのモデルについては、デプロイを行っても特に仮想マシンなどは作成されず、デプロイを行っただけではコストも発生しない。

# ■ Azure AI Foundry (以前のAzure AI Studio) を使用して、GPT-4oのデプロイを行う

Azure AI Foundry / admin-5008 / モデル カタログ

すべてのハブとプロジェクト 5 ヘルプ プロジェクト admin-5008 SA

## カスタム AI ソリューションを構築するための適切なモデルを見つける

新着情報

Realtime Audio Updates

Gretel Navigator Tabular is now available!

Codestral 2501 from Mistral AI

メタからの Llama 3.3

Phi-4 がやって来た!

モデルのチェックアウト ブログを読む

モデルのチェックアウト ブログを読む

Check out models ブログを読む

Check out model ブログを読む

New model benchmarks available now in model catalog

Model benchmarks are integrated into model catalog for easier navigation. Compare benchmarks across models and datasets available in the industry to assess which one meets your business scenario.

Compare with benchmarks How model benchmarks are scored

コレクション 業界 デプロイオプション 推論タスク タスクの微調整 ライセンス モデルの比較 モデル 1825

検索

o1 チャットの完了 gpt-4o チャットの完了 Phi-4 チャットの完了 Gretel-Navigator-Tabular チャットの完了, data-generation o1-preview チャットの完了 o1-mini チャットの完了

gpt-4o-realtime-... チャットの完了 gpt-4o-mini チャットの完了 Llama-3.3-70B-Instruct チャットの完了 tsuzumi-7b チャットの完了 Bria-2.3-Fast テキストから画像へ Minstral-3B チャットの完了

Virchow image-feature-extraction Virchow2 image-feature-extraction Cohere-embed-v3-english 埋め込み Cohere-embed-v3-multi... 埋め込み Llama-3.2-11B-Vision-Ins... チャットの完了

使用したいモデルをクリック

管理センター

## ← gpt-4o

▷ デプロイ □ 微調整

詳細 ベンチマーク 既存のデプロイ コードサンプル ライセンス

gpt-4o offers a shift in how AI models interact with multimodal inputs. By seamlessly combining text, images, and audio, gpt-4o provides a richer, more engaging user experience.

Matching the intelligence of gpt-4 turbo, it is remarkably more efficient, delivering text at twice the speed and at half the cost. Additionally, GPT-4o exhibits the highest vision performance and excels in non-English languages compared to previous OpenAI models.

gpt-4o is engineered for speed and efficiency. Its advanced ability to handle complex queries with minimal resources can translate into cost savings and performance.

The introduction of gpt-4o opens numerous possibilities for businesses in various sectors:

1. Enhanced customer service: By integrating diverse data inputs, gpt-4o enables more dynamic and comprehensive customer support interactions.
2. Advanced analytics: Leverage gpt-4o's capability to process and analyze different types of data to enhance decision-making and uncover deeper insights.

詳細表示

### モデルバージョン

リージョンの可用性に関する詳細情報 □

Sweden Central

モデルID	可用性	ライフサイクル	最大要求	廃止の日付
2024-05-13	Standard, グローバル標準, プロビジョニング管理, グローバルバッチ, データゾーン標準, データ	④ 一般提供 [タイトルなし]	Input: 128000 Output: 4096	2025年5月20日(火)

詳細表示

### データ、メディア、言語

Property	説明	
サポートされるデータ型	入力 text, image, audio	出力 text

### クイックファクト

OpenAI

gpt-4o  
chat-completion

最後のトレーニング  
**10月 2023**

価格  
[価格を見る □](#)

ベンチマーク

0.85 品質指数  
AI品質

7.5 1MトークンあたりのUSD  
推定コスト

モデルID  
コードでモデルを配置するときにこのモデルIDを参照する  
azureml://registries/azure-openai/models/gpt-4o/versions/2024-11-20

## モデル gpt-4o をデプロイする

デプロイ名\*

gpt-4o

デプロイ名（任意）を指定。デフォルトではモデル名と同じ名前となる

### デプロイの種類

グローバル標準

Global Standard: 最も高い制限が適用された API 呼び出しあたりの支払。グローバル展開の種類 の詳細をご覧ください。

データはリソースの Azure 地域の外部でグローバルに処理される可能性がありますが、データストレージは AI リソースの Azure 地域に残ります。データ所在地 の詳細をご覧ください。

#### ▼ デプロイの詳細

カスタマイズ

モデルバージョン

2024-11-20

接続されている AI リソース

ai-hub5324523343671299464\_aoui

プロジェクト

admin-5008

認証の種類

キー

容量

10K 1 分あたりのトークン数 (TPM)

リソースの場所

Sweden Central

コンテンツの安全性

DefaultV2

デプロイ

取り消し



## モデルとサービスのデプロイを管理

モデルデプロイ サービスエンドポイント

+ モデルのデプロイ

最新の情報に更新

編集

プレイグラウンドで開く

ビューのリセット

概要

モデルカタログ

プレイグラウンド

AIサービス

ビルトとカスタマイズ

コード プレビュー

微調整

プロンプトフロー

評価と改善

トレース プレビュー

評価

安全性とセキュリティ

マイアセット

モデル + エンドポイント

データとインデックス

Web Apps

名前

モデル名

モデルバージョン

状態

モデルの廃止日

ai-hub5324523343671299464\_aoai Azure AI サービス

エンドポイントの取得

gpt-4o

2024-11-20

成功

名前	モデル名	モデルバージョン	状態
ai-hub5324523343671299464_aoai	Azure AI サービス	エンドポイントの取得	

Azure AI Foundry / admin-5008 / プレイグラウンド

## サンプルコード

次のコードを使用して、アプリケーションへの現在のプロンプトと設定の統合を開始できます。

```
https://ai-hub5324523343671299464.openai.azure.com/ python
```

Entra ID 認証 キー認証

```
1 import os
2 import base64
3 from openai import AzureOpenAI
4
5 endpoint = os.getenv("ENDPOINT_URL", "https://ai-hub5324523343671299464.openai.azure.com/")
6 deployment = os.getenv("DEPLOYMENT_NAME", "gpt-4o")
7 subscription_key = os.getenv("AZURE_OPENAI_API_KEY", "REPLACE_WITH_YOUR_KEY_VALUE_HERE")
8
9 # Initialize Azure OpenAI Service client with key-based authentication
10 client = AzureOpenAI(
11     azure_endpoint=endpoint,
12     api_key=subscription_key,
13     api_version="2024-05-01-preview",
14 )
15
16
17
18 IMAGE_PATH = "YOUR_IMAGE_PATH"
19 encoded_image = base64.b64encode(open(IMAGE_PATH, 'rb').read()).decode('ascii')
20
21 # チャットプロンプトを準備する
22 chat_prompt = []
23
24 # 音声認識が有効になっている場合は音声結果を含める
25 messages = chat_prompt
```

アプリケーションでキーが誤って公開されないようにするには、環境[タイトルなし] Azure Key Vaultなどのシークレット管理ツールを使用する必要があります。  
環境の設定に関する詳細情報

エンドポイント ① https://ai-hub5324523343671299464.openai.azure.com/ エンドポイント

API キー ① ..... キー

プロジェクト 5 ジェクト admin-5008 ヘルプ ヘルプ

応答形式 テキスト

Recipe creation

0/128000 送信するトークン

アプリを実行するコンピュータの「環境変数」に、キー・エンドポイント・デプロイ名を設定。



## ■ GPT-4oにプロンプトを送信してテキストを生成（InvokePromptAsyncを使用、1ターン）

```
using Microsoft.SemanticKernel;

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

var response = await kernel.InvokePromptAsync("こんにちは");

Console.WriteLine(response);
```

内部的にはITextGenerationServiceのインスタンスが作成されカーネルに登録される

内部的にはカーネルのITextGenerationServiceを使用して、プロンプトからテキストを生成

こんにちは！ 😊 何かお手伝いできますか？

実行結果例

■参考：メソッド名が「～Async」の場合は、その呼び出しの前に「await」を付けてください

await を忘れている！

```
var response = kernel.InvokePromptAsync("こんにちは");
```

await を付けてください

```
var response = await kernel.InvokePromptAsync("こんにちは");
```

これは、時間のかかる処理（生成AIによるコンテンツ生成など）を**非同期**（バッ  
クグラウンド）で実行し、その結果を受け取る、という意味です。

本トレーニングでは詳しく解説しませんが、詳しくは「**非同期プログラミング**」について学習してください

## ■GPTにプロンプトを送信してテキストを生成（ITextGenerationServiceを使用、1ターン）

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

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

var textGenerationService = kernel.GetRequiredService<ITextGenerationService>();
var response = await textGenerationService.GetTextContentAsync("こんにちは");

Console.WriteLine(response);
```

カーネルから  
ITextGenerationServiceのオブ  
ジェクトを取り出して利用

こんにちは！ 😊 何かお手伝いできますか？

実行結果はInvokePromptAsync()の  
場合と同じ

# モジュール1

- ・「Azure OpenAI Service」とは？
- ・「エージェント」とは？
- ・「Semantic Kernel」とは？
- ・Semantic Kernelの「カーネル」とは？
- ・Semantic Kernelの「コネクター」とは？
- ・Semantic Kernelの基本的な使い方
- ・Semantic Kernelでの会話履歴の管理
- ・Semantic Kernelによる画像の生成

# 会話の履歴とは

最初のプロンプト

あなた: 私の名前は山田です

アシスタントの回答

アシスタント: 山田さん、こんにちは！ 😊 お話しできて嬉しいです。今日はどんなことについてお話ししましょうか？

次のプロンプト

あなた: 私の名前は？

アシスタントの回答

アシスタント: あなたのお名前は「山田さん」です！ 😊 お間違いありませんよね？

あなた:

アシスタントが、以前の会話の情報を踏まえて回答している  
(会話の履歴が記録がされている)

# 会話の履歴の管理

- ・このような複数ターンからなる会話を実現するには、**会話の履歴をアプリケーション側で管理する**必要がある
- ・Semantic Kernelの場合は「**ChatHistory**」クラスを使用して、会話の履歴を管理できる
- ・この場合、**ITextGenerationService**ではなく、**IChatCompletionService**を利用する

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

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

...
```

# ChatHistoryクラス

- ・会話の履歴を管理するオブジェクト
- ・以下のものを含めることができる
  - ・システムメッセージ
  - ・ユーザーメッセージ（ユーザーが入力したプロンプト）
  - ・アシスタントの出力

## ■ Semantic Kernel の「ChatHistory」クラスの利用例（基本的な使い方）

システムメッセージの設定  
ユーザーメッセージ（ユーザーのプロンプト）の設定

```
using Microsoft.SemanticKernel.ChatCompletion;

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

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

アシスタント（生成AI）が  
出力したコンテンツの設定

## ■ChatHistoryを使用した複数ターンのチャット例

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

var chatCompletionService = kernel.GetRequiredService<IChatCompletionService>();
var history = new ChatHistory(); ————— ChatHistoryオブジェクトを作成

while (true)
{
    Console.Write("あなた: ");
    var input = Console.ReadLine() ?? "";
    history.AddUserMessage(input); ————— ユーザーのプロンプトをChatHistoryに追加

    var response = await chatCompletionService.GetChatMessageContentAsync(history);
    Console.WriteLine($"アシスタント: {response}");
    history.AddAssistantMessage(response.ToString()); ————— 生成AIの出力をChatHistoryに追加
}
```

あなた: 私の名前は山田です  
アシスタント: 山田さん、こんにちは！ 😊 お話してきて嬉しいです。今日はどんなことについてお話ししましょうか？  
あなた: 私の名前は？  
アシスタント: あなたの名前は「山田さん」です！ 😊 お間違いありませんよね？  
あなた:

■参考: ChatHistoryを使用しない場合、会話の履歴を踏まえた回答ができなくなる

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

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

while (true)
{
    Console.Write("あなた: ");
    var input = Console.ReadLine() ?? "";
    var response = await chatCompletionService.GetChatMessageContentAsync(input);
    Console.WriteLine($"アシスタント: {response}");
}
```

あなた: 私は山田です  
アシスタント: こんにちは、山田さん！ 😊 何かお手伝いできることがあれば、ぜひ教えてください！  
あなた: 私の名前は？  
アシスタント: 申し訳ありませんが、こちらではあなたの名前を特定することはできません。教えていただければ、その名前を使ってお話しすることができますよ！ 😊  
あなた:

# ここまでまとめ

```
var response = await kernel.InvokePromptAsync("こんにちは");
```

プロンプトからのコンテンツ生成（1ターン）

```
var textGenerationService = kernel.GetRequiredService<ITextGenerationService>();  
var response = await textGenerationService.GetTextContentAsync("こんにちは");
```

プロンプトからのコンテンツ生成（1ターン）

```
var chatCompletionService = kernel.GetRequiredService<IChatCompletionService>();  
var history = new ChatHistory();  
history.AddUserMessage(input);  
var response = await chatCompletionService.GetChatMessageContentAsync(history);  
history.AddAssistantMessage(response.ToString());
```

プロンプトからのコンテンツ生成（複数ターン）

# モジュール1

- ・「Azure OpenAI Service」とは？
- ・「エージェント」とは？
- ・「Semantic Kernel」とは？
- ・Semantic Kernelの「カーネル」とは？
- ・Semantic Kernelの「コネクター」とは？
- ・Semantic Kernelの基本的な使い方
- ・Semantic Kernelでの会話履歴の管理
- ・Semantic Kernelによる画像の生成

# Semantic Kernelによる画像の生成

- DALL-E (ダリ) などのモデルを使用すると、プロンプトから画像を生成することができる

## カスタム AI ソリューションを構築するための適切なモデルを見つける

? ヘルプ

概要

モデル カタログ

プレイグラウンド

AI サービス

ビルトとカスタマイズ

&lt;/&gt; コード プレビュー

微調整

プロンプト フロー

評価と改善

トレース プレビュー

評価

安全性とセキュリティ

マイ アセット

モデル + エンドポイント

データとインデックス

Web Apps

管理センター

コレクション

業界

デプロイ オプション

推論タスク

タスクの微調整

ライセンス

モデルの比較

dall-e

×

モデル 341

dall-e-3 テキストから画像へ	dall-e-2 テキストから画像へ	richieIo-small-e-czech-finetuned... トークンの分類	intfloat-multilingual-e5-large feature-extraction
text-embedding-3-small 埋め込み	text-embedding-3-large 埋め込み	baai-bge-large-en-v1.5 feature-extraction	OpenAI-CLIP-Image-Text-E... 埋め込み
OpenAI-CLIP-Image-Text-Embedding 埋め込み	Facebook-DinoV2-Image-Embedding 埋め込み	Facebook-DinoV2-Image-Embedding 埋め込み	Cohere-embed-v3-multi... 埋め込み
Cohere-embed-v3-english 埋め込み	text-embedding-ada-002 埋め込み	E.L.Y.Crop-Protection チャットの完了	Cerence-CaLLM-Edge チャットの完了
gokaygokay-flux-prompt-english テキストからテキストへの生成	Cohere-rerank-v3-english テキストの分類	prometheus-eval-prometheus... テキストからテキストへの生成	prometheus-eval-prometheus... テキストからテキストへの生成
equall-saul-7b-instruct-v1 テキスト生成	alibaba-nlp-gte-large-en-v1.5 sentence-similarity	distilbert-base-uncased-fine-tuned... テキストの分類	helsinki-nlp-opus-mt-de-en トランсл레이ター
distilbert-base-uncased-fine-tuned... テキストの分類	j-hartmann-emotion-english... テキストの分類	helsinki-nlp-opus-mt-en-de トランсл레이ター	helsinki-nlp-opus-mt-en-es トランсл레이ター

概要
モデルカタログ
プレイグラウンド
AI サービス
ビルトとカスタマイズ ^
コード プレビュー
微調整
プロンプトフロー
評価と改善 ^
トレース プレビュー
評価
安全性とセキュリティ
マイアセット ^
モデル + エンドポイント
データとインデックス
Web Apps

## ← dall-e-3

ヘルプ

▷ デプロイ

△ 微調整

## 詳細 既存のデプロイ

DALL-E 3 generates images from text prompts that are provided by the user. DALL-E 3 is generally available for use on Azure OpenAI.

The image generation API creates an image from a text prompt. It does not edit existing images or create variations.

Learn more at: <https://learn.microsoft.com/azure/ai-services/openai/concepts/models#dall-e>

## モデルバージョン

リージョンの可用性に関する詳細情報 □

Sweden Central ▼

## クイックファクト

OpenAI



text-to-image

最後のトレーニング  
使用できません価格  
[価格を見る](#) □

## モデルID

コードでモデルを配置するときにこのモデル ID を参照する

azureml://registries/azure-openai/models/dall-e-3/versions/3.0 □

モデル ID	可用性	ライフサイクル	最大要求	廃止の日付
3.0	Standard	◆ プレビュー 運用環境での使用には適していません。注意して続行してください	Input: N/A Output: N/A	2025年4月30 日(水)

詳細表示

## データ、メディア、言語

Property	説明
----------	----

## モデル dall-e-3 をデプロイする

デプロイ名\*

dall-e-3



デプロイの種類

Standard



Standard: API 呼び出しごとに支払い、より低いレート制限が適用されます。Azure データ所在地の確約に従います。量が小規模から中規模の、間欠的に発生するワークロードに最適です。  
[Standard 展開](#) の詳細をご覧ください。

▽ デプロイの詳細

カスタマイズ

モデルバージョン

3.0

接続されている AI リソース

ai-hub5324523343671299464\_aoai

プロジェクト

admin-5008

認証の種類

キー

容量

1 容量ユニット (CU)

リソースの場所

Sweden Central

コンテンツの安全性

既定

デプロイ

取り消し

## Manage deployments of your models and services

Model deployments Service endpoints

+ Deploy model ▾

⟳ Refresh

✎ Edit

↗ Open in playground

⟲ Reset view

Name	Model name	Model version	State
ai-hub5324523343671299464_aoai	Azure AI Services	 Get endpoint	
dall-e-3	dall-e-3	3.0	Succeeded
gpt-4o	gpt-4o	2024-11-20	Succeeded

## ■画像の生成の例

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.TextToImage;
```

環境変数からdall-e-3のデ  
プロイ名、エンドポイン  
ト、キーを取り出す

```
var deployName = Environment.GetEnvironmentVariable("AOAI_DALLE_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";
```

ITextToImageServiceのイ  
ンスタンスを作成しカ  
ネルに登録

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAITextToImage(deployName, endpoint, apiKey);
var kernel = builder.Build();
```

ITextToImageServiceのイ  
ンスタンスを取得

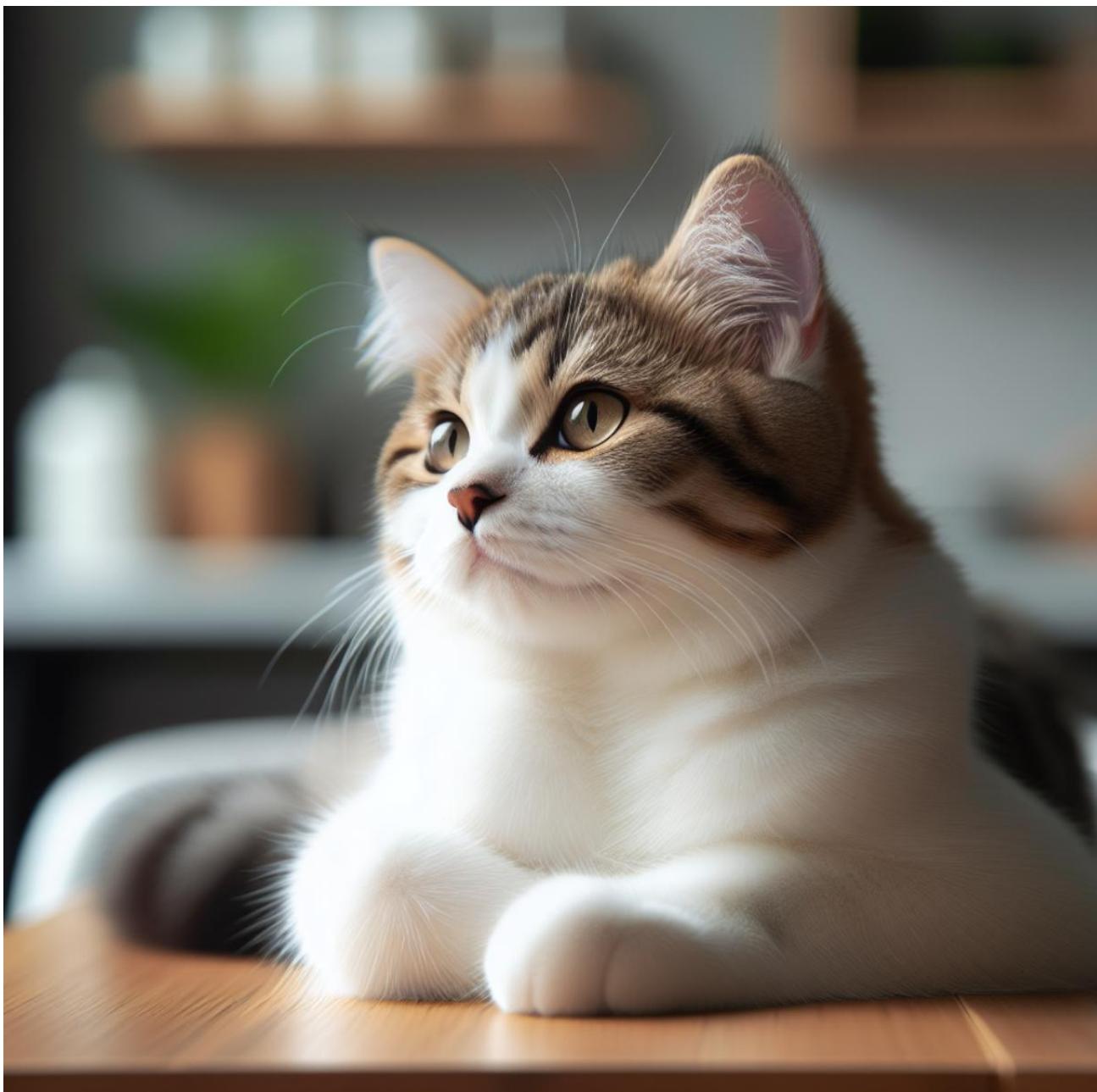
```
var service = kernel.GetRequiredService<ITextToImageService>();
// size: '1024x1024', '1792x1024', '1024x1792'
var image = await service.GenerateImageAsync("A cat sitting on a table", 1024, 1024);
Console.WriteLine(image);
```

「テーブルの上に座ってい  
る猫」というプロンプトか  
ら画像を生成

実行するとURLが返される（このURLはAzure  
Blob StorageのオブジェクトのSASであり、生  
成された画像の読み取りが可能）

[https://dalleprodsec.blob.core.windows.net/private/images/5cf1ec52-e805-45eb-9fd1-bcd55ce4c32a/generated\\_00.png?se=2025-01-19T08%3A50%3A38Z&sig=hX1TzHj1SHGx1bqvoq3SfaMVzCDKsxZdtgnleNesBd0%3D&ske=2025-01-22T14%3A40%3A06Z&skoid=e52d5ed7-0657-4f62-bc12-7e5dbb260a96&skt=2025-01-15T14%3A40%3A06Z&sktid=33e01921-4d64-4f8c-a055-5bdaffd5e33d&skv=2020-10-02&sp=r&spr=https&sr=b&sv=2020-10-02](https://dalleprodsec.blob.core.windows.net/private/images/5cf1ec52-e805-45eb-9fd1-bcd55ce4c32a/generated_00.png?se=2025-01-19T08%3A50%3A38Z&sig=hX1TzHj1SHGx1bqvoq3SfaMVzCDKsxZdtgnleNesBd0%3D&ske=2025-01-22T14%3A40%3A06Z&skoid=e52d5ed7-0657-4f62-bc12-7e5dbb260a96&skt=2025-01-15T14%3A40%3A06Z&sktid=33e01921-4d64-4f8c-a055-5bdaffd5e33d&skv=2020-10-02&sp=r&spr=https&sr=b&sv=2020-10-02)

■表示されたSAS URLをWebブラウザーで開くと、生成された画像が表示される



# モジュール1 まとめ

- Azure OpenAI Serviceでは、gpt-4oやdall-e-3などの生成AIモデルをAzure上にデプロイして利用できる
- Semantic Kernelを使用して、生成AIアプリやAIエージェントを作成できる
- Semantic Kernelの「カーネル」(Kernel)は、生成AIモデルへの接続(コネクション)や、様々な機能(プラグイン)を管理する
- Semantic Kernelの「サービス」を利用して、プロンプトからのテキストコンテンツ生成や、プロンプトからの画像の生成などを実行できる
  - `kernel.InvokePromptAsync`(プロンプト)
  - `kernel.GetService<ITextGenerationService>().GetTextContentAsync`(プロンプト)
  - `kernel.GetService<ITextToImageService>().GenerateImageAsync`(プロンプト)
- Semantic Kernelの「ChatHistory」は複数ターンの会話の履歴を管理する
  - `var chatHistory = new ChatHistory();`
  - `kernel.GetService<IChatCompletionService>().GetChatMessageContentAsync(chatHistory)`

# モジュール2 + モジュール3



## セマンティック カーネル用のプラグインを作成する

6 分 残り • モジュール • 8/10 ユニットが完了しました

[△ フィードバック](#)

中級 開発者 .NET Visual Studio Code Azure OpenAI Service

このモジュールでは、Semantic Kernel SDK プラグインについて説明します。SDK のプラグインを使用して、カスタマイズされたタスクを実行し、インテリジェントなアプリケーションを作成する方法について学習します。



## AI エージェントにスキルを与える

29 分 • モジュール • 5 ユニット

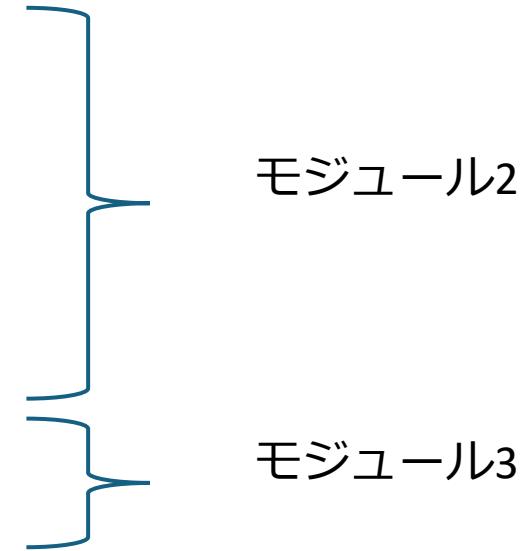
[△ フィードバック](#)

中級 開発者 .NET Visual Studio Code Azure OpenAI Service

このモジュールでは、セマンティック カーネル SDK のネイティブ関数について調べます。ネイティブ関数でカスタマイズされたタスクを実行し、AI エージェントに "スキル" を効果的に与える方法を学習します。

# モジュール2 + モジュール3

- ・プラグインと関数
- ・組み込みプラグイン（コアプラグイン）
- ・プラグインの自作
  - ・プロンプト関数（セマンティック関数）
  - ・フォルダからプラグインを読み込む
  - ・メソッド関数（ネイティブ関数）

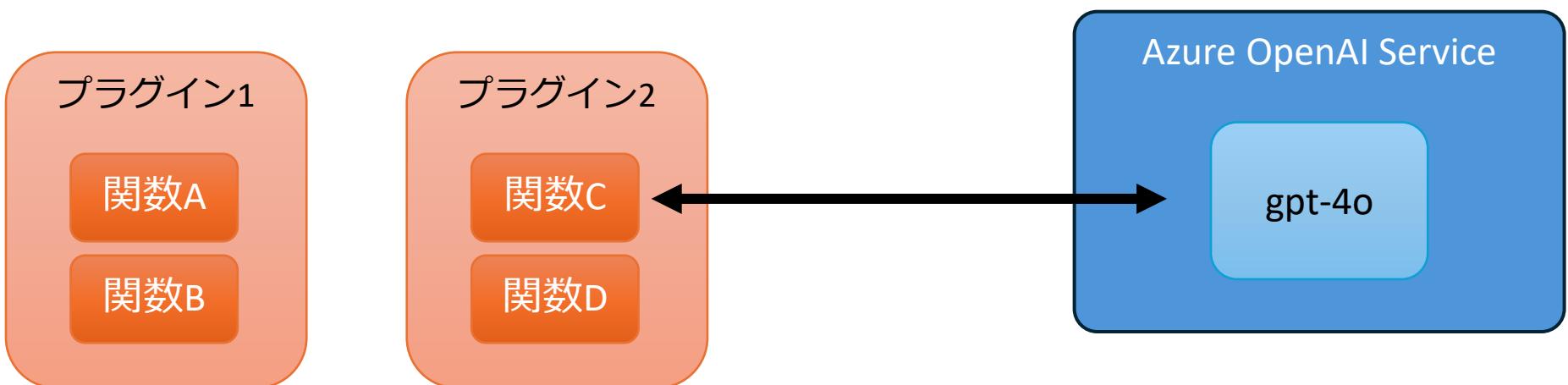


# モジュール2 + モジュール3

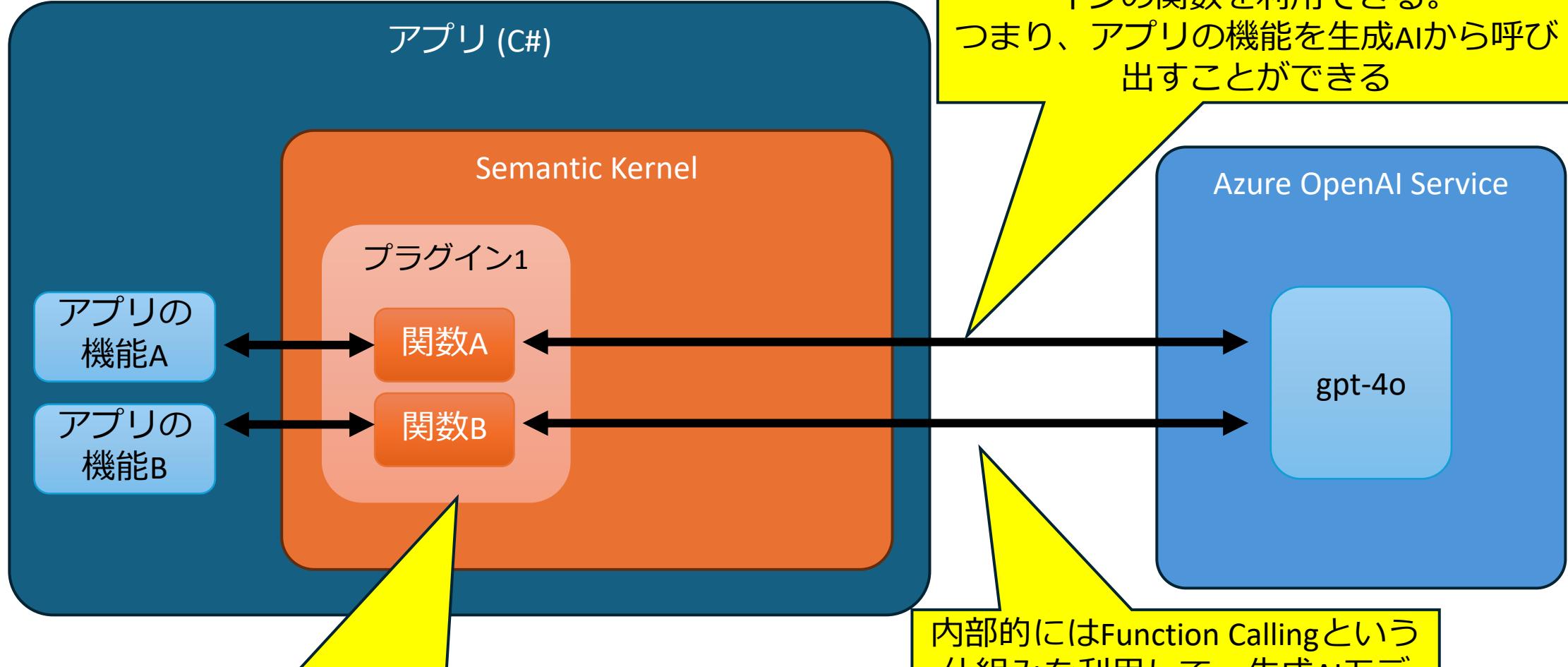
- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

# プラグインと関数

- ・プラグイン: アプリケーションに独自の機能を追加する仕組み
- ・プラグインの中には1つ～複数の「関数」(Function)が含まれる
  - ・プラグインは複数の関数をまとめた「入れ物」「箱」と考えるとよい
- ・**関数は生成AIモデルに利用させることができる**
  - ・つまり、生成AIモデルから**任意の処理**を実行できるようになる!!



# プラグイン: アプリに機能を追加

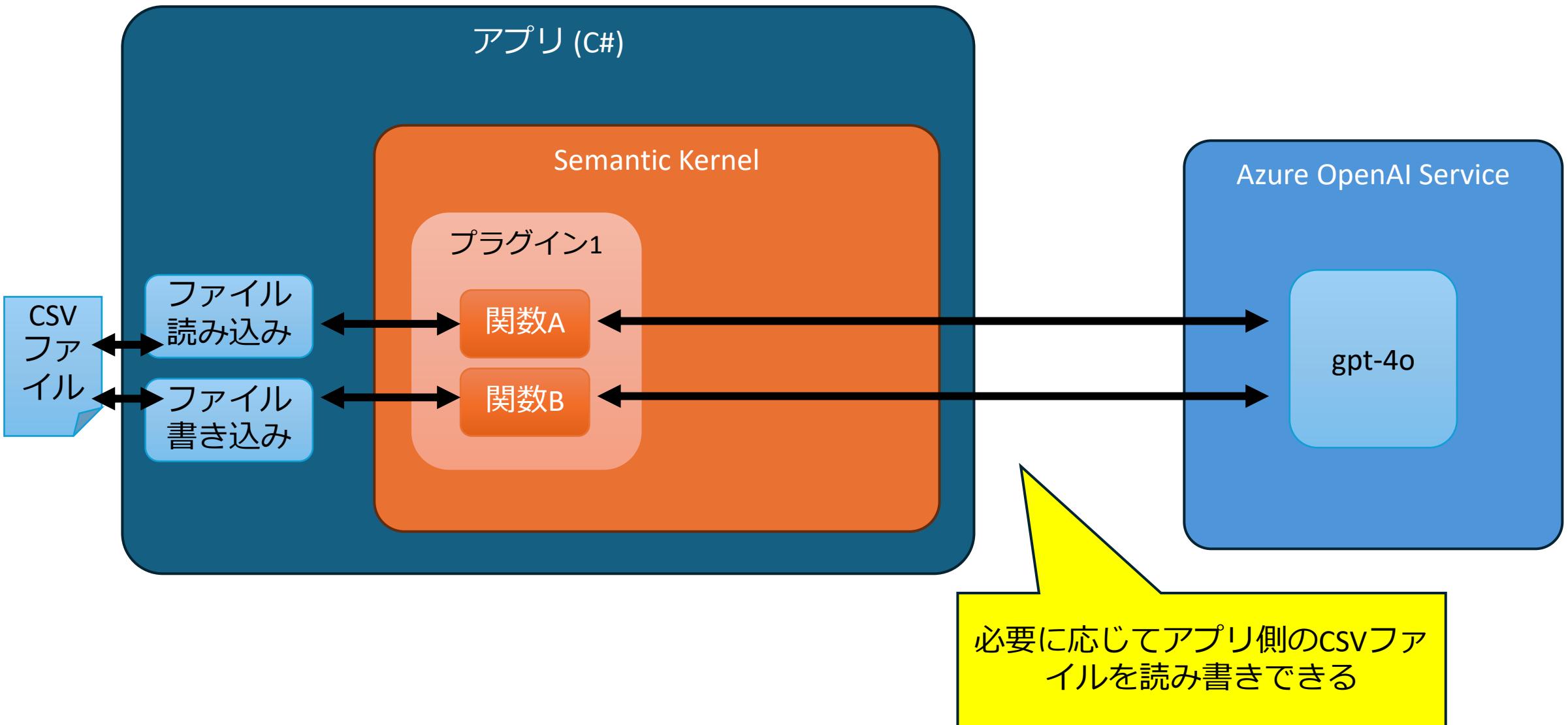


アプリの機能を、  
プラグインの関数  
として公開できる

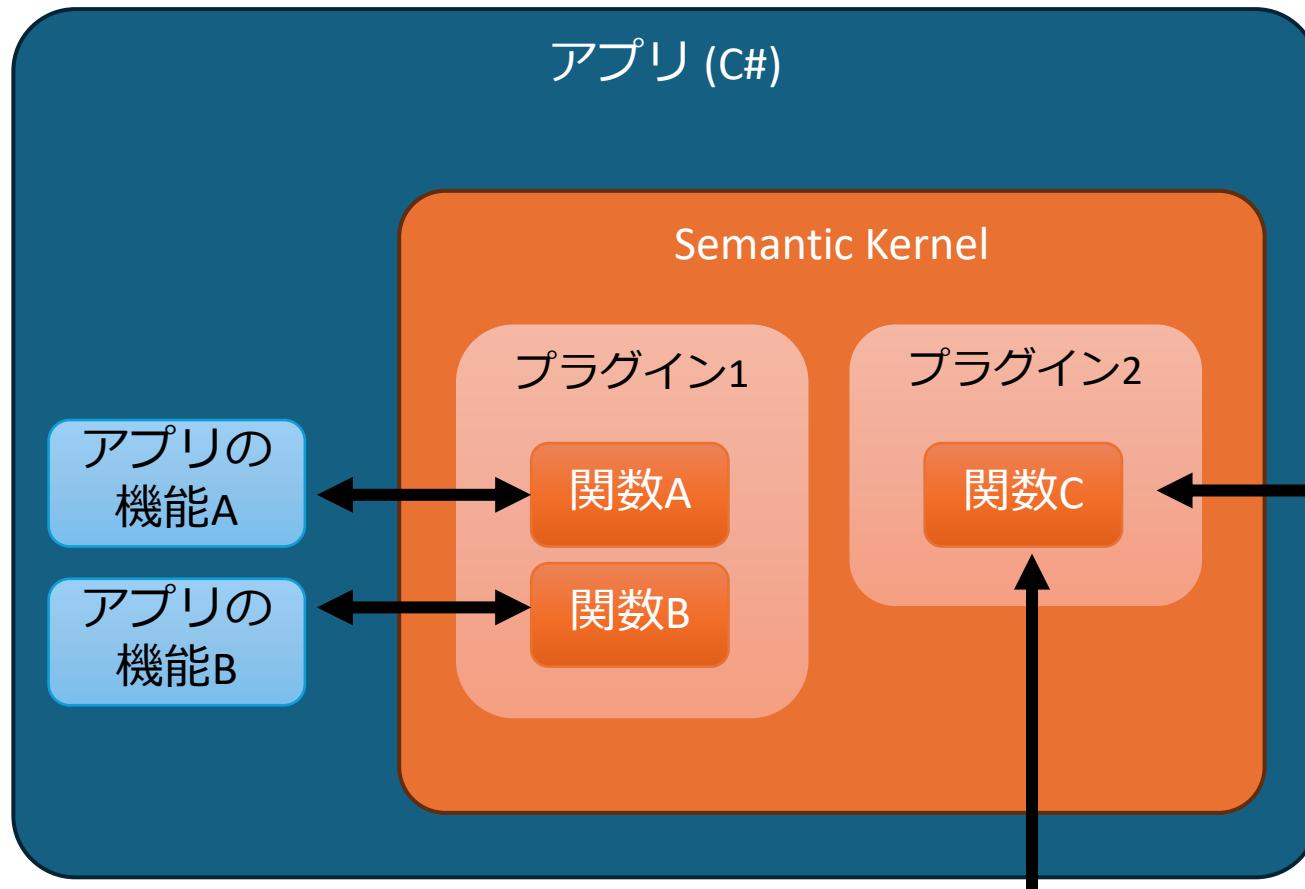
生成AIモデルは、必要に応じて、  
プラグインの関数を利用できる。  
つまり、アプリの機能を生成AIから呼び  
出すことができる

内部的にはFunction Callingという  
仕組みを利用して、生成AIモ  
デルが、Semantic Kernelに対して、  
関数の呼び出しを依頼する

# プラグインの実装例



# アプリの外部のサービスの利用



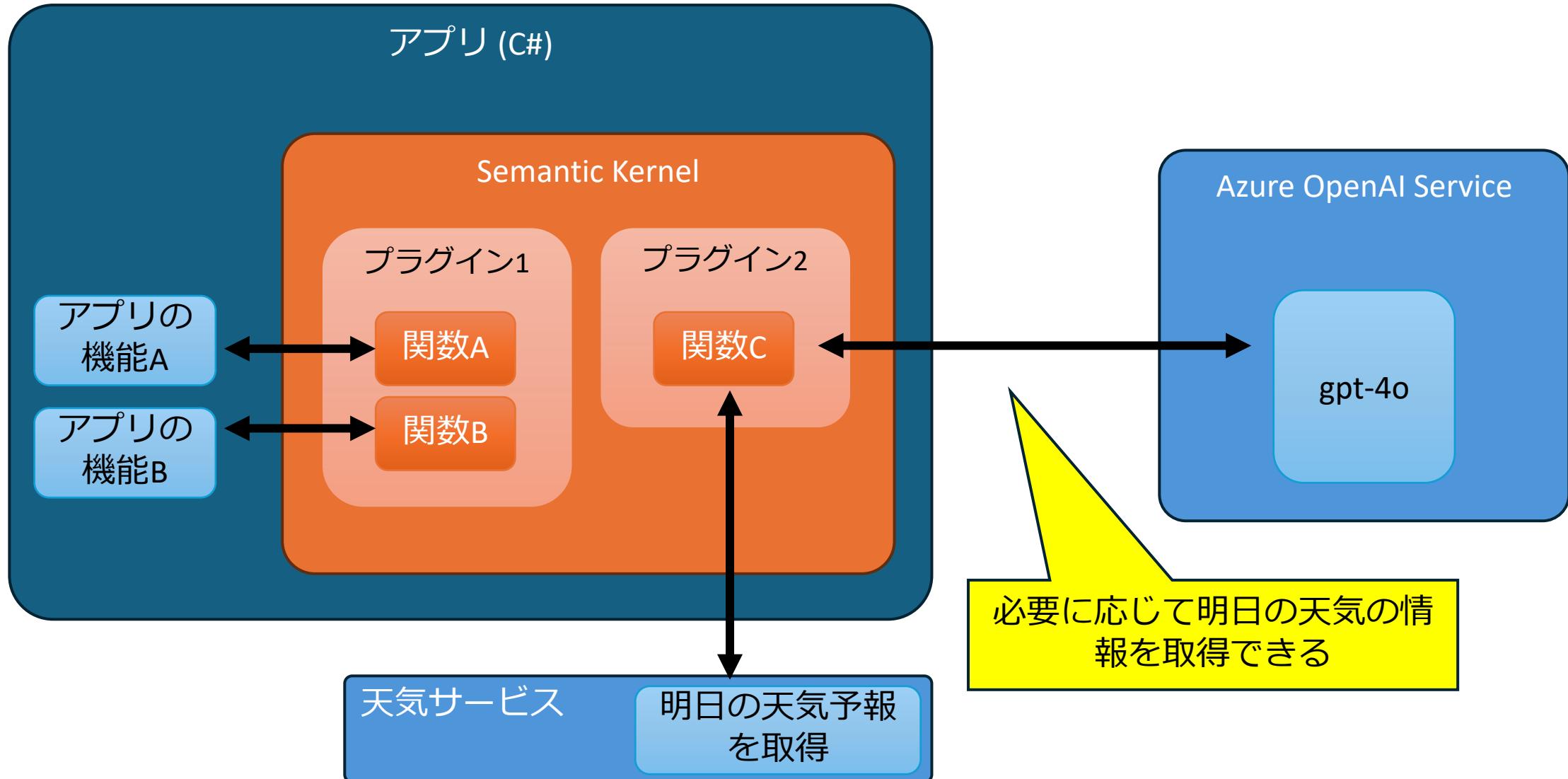
生成AIは必要に応じて外部サービスを呼び出すことが可能となる

外部サービスの機能を提供するプラグインも作成できる

外部サービス

機能C

# プラグインの実装例



# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

# 「組み込みプラグイン」とは？

- ・「コアプラグイン」とも
- ・Semantic Kernelに付随する、すぐに使える作成済みのプラグイン
- ・C#の場合、NuGetパッケージ「Microsoft.SemanticKernel.Plugins.Core」として提供される
- ・このパッケージには以下のプラグインが含まれる
  - ・現在の日時などの情報の取得: TimePlugin
  - ・テキストからの情報の抽出: ConversationSummaryPlugin
  - ・ファイルの読み書き: FileIOPPlugin
  - ・HTTPを使用したWebアクセス（GET/POST/PUT/DELETE等）: HttpPlugin
  - ・加算・減算などの算術: MathPlugin
  - ・テキストのトリム（文字列前後の空白除去）などの処理: TextPlugin
  - ・指定した秒数のウェイト: WaitPlugin

## ■組み込みプラグインのパッケージの追加

The screenshot shows the NuGet package page for `Microsoft.SemanticKernel.Plugins.Core`. The page includes the following details:

- Downloads:** Total 492.0K, Current version 981, Per day average 1.0K.
- About:** Last updated 13 days ago, Project website, Source repository, MIT license.
- Actions:** Copy command-line, Download package (210.84 KB), Download symbols (47.97 KB), Open in NuGet Package Explorer, Open in NuGet Trends.
- Dependencies:** A table showing versions, downloads, and last update dates for three prerelease versions:

Version	Downloads	Last updated
1.33.0-alpha	981	13 days ago
1.32.0-alpha	2,740	a month ago
1.31.0-alpha	2,800	2 months ago

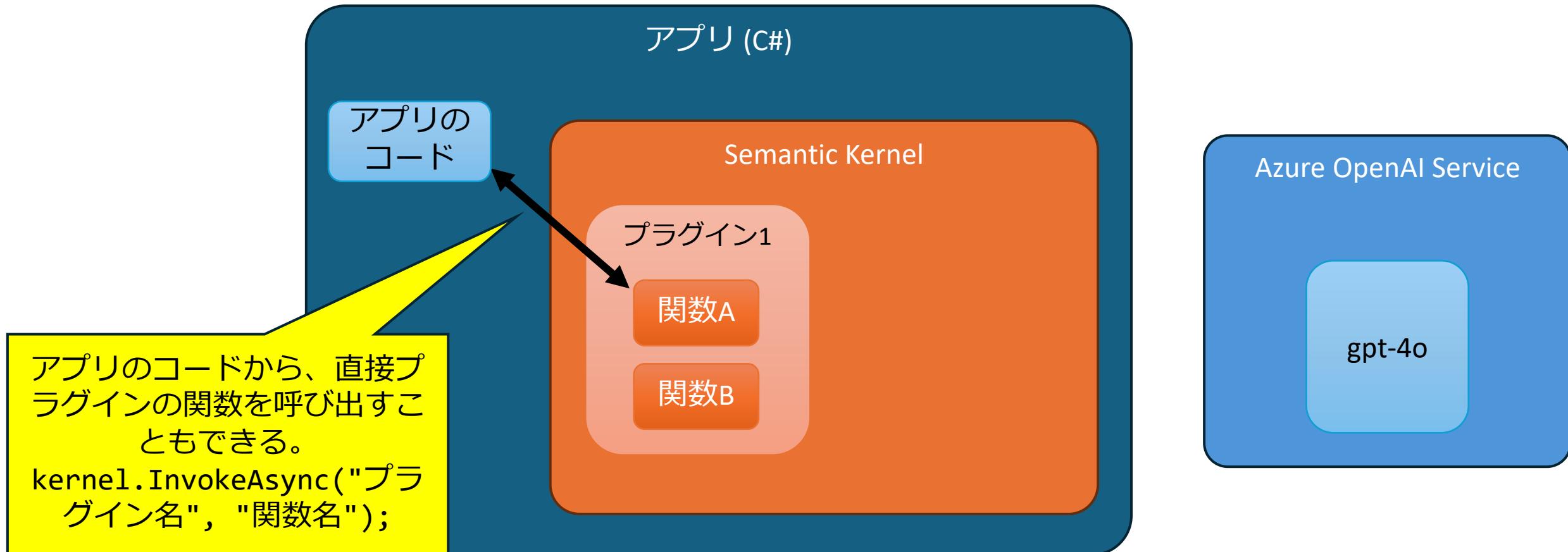
```
dotnet add package Microsoft.SemanticKernel.Plugins.Core -version <バージョン番号>
```

```
dotnet add package Microsoft.SemanticKernel.Plugins.Core --prerelease
```

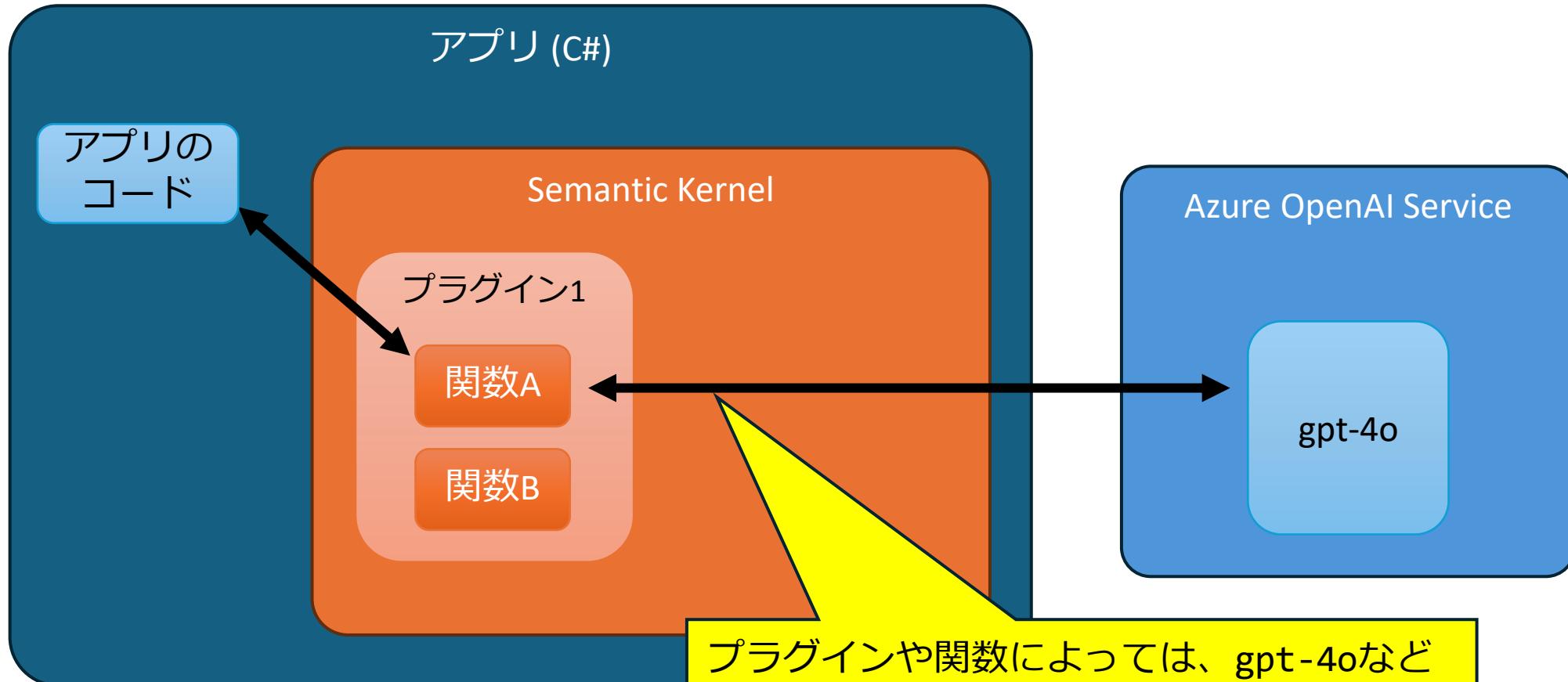
```
dotnet add package Microsoft.SemanticKernel.Plugins.Core
```

プレリリース版しかないパッケ時  
の場合、これはエラーとなるので  
注意

# プラグインの直接呼び出し



# プラグインの直接呼び出し



プラグインや関数によっては、gpt-4oなどの生成AIモデルへの接続が必要となる。  
例：ConversationSummaryPluginの関数は、入力されたテキストの要約文を生成するために生成AIモデルを使用する（具体的にはITextGenerationServiceを使用する）

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

# 組み込みプラグイン利用時の警告について

- 組み込みプラグインは現在「評価目的」で提供されているため、使用しようとすると警告が出る。

```
var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<TimePlugin>();
var kernel = builder.Build();
```

赤い波線が引かれている。  
このままではコンパイルできない。

```
'Microsoft.SemanticKernel.Plugins.Core.TimePlugin' is for evaluation purposes only and is
subject to change or removal in future updates. Suppress this diagnostic to proceed.
(SKEXP0050)
```

'Microsoft.SemanticKernel.Plugins.Core.TimePlugin' は評価目的のみであり、**将来の更新で変更または削除される可能性があります**。続行するには、この診断を抑制してください。

# 組み込みプラグインの位置づけ

- ・現在のところ、組み込みプラグイン（`Microsoft.SemanticKernel.Plugins.Core` パッケージに含まれるプラグイン）は、Semantic Kernel の正式な機能の一部というよりは、**評価目的で、暫定的に提供されているに過ぎない。**
- ・プラグインの簡易的なサンプル実装のようなもの。
- ・**将来変更されるかもしれないし、なくなるかもしれない**
- ・必要に応じて組み込みプラグイン同等の機能を持つ**自作のプラグイン**を作つてそれを利用することも検討してください
  - ・（プラグインの自作についてはこのあとで解説）

- 警告を消す（=警告を理解し、それでも利用する）には、コード冒頭で「#pragma warning disable SKEXP0050」を使用する

```
#pragma warning disable SKEXP0050

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;

var builder = Kernel.CreateBuilder();

builder.Plugins.AddFromType<TimePlugin>();

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

組み込みプラグイン利用時の  
警告を抑制

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

## ■組み込みプラグイン「TimePlugin」の利用例

```
#pragma warning disable SKEXP0050

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<TimePlugin>();

var kernel = builder.Build();

var response = await kernel.InvokeAsync("TimePlugin", "Now");
Console.WriteLine(response);
```

組み込みプラグイン利用時の  
警告を抑制

TimePlugin利用時は、カーネルに  
Azure OpenAIを組み込む必要はない。

InvokeAsync (プラグイン名、関数  
名) で、プラグインを直接呼び出す  
ことができる。

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

## ■組み込みプラグイン「ConversationalSummaryPlugin」の利用例

```
#pragma warning disable SKEXP0050
```

```
using Microsoft.SemanticKernel;
```

```
using Microsoft.SemanticKernel.Plugins.Core;
```

```
var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
```

```
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
```

```
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";
```

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

```
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
```

```
builder.Plugins.AddFromType<ConversationSummaryPlugin>();
```

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

```
var kernelArguments = new KernelArguments
```

```
{
```

```
    ["input"] = File.ReadAllText("sample.txt")
```

```
};
```

```
var response = await kernel.InvokeAsync("ConversationSummaryPlugin", "SummarizeConversation",  
kernelArguments);
```

```
Console.WriteLine(response);
```

組み込みプラグイン利用時の  
警告を抑制

ConversationalSummaryPlugin利  
用時は、カーネルにAzure  
OpenAIを組み込む必要がある

ConversationalSummaryPluginの  
組み込み

## ■組み込みプラグイン「ConversationalSummaryPlugin」の利用例

```
#pragma warning disable SKEXP0050

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<ConversationSummaryPlugin>();
var kernel = builder.Build();

var kernelArguments = new KernelArguments {
    ["input"] = File.ReadAllText("sample.txt")
};

var response = await kernel.InvokeAsync("ConversationSummaryPlugin", "SummarizeConversation",
kernelArguments);

Console.WriteLine(response);
```

ここでは、対象データ（要約を行う文章）をファイル「sample.txt」から読み込んでいる

## ■sample.txt（「竹取物語」の冒頭）

今は昔竹取の翁といふものありけり。野山にまじりて、竹をとりつゝ、萬の事につかひけり。  
名をば讚岐造磨となんいひける。その竹の中に、本光る竹ひとつすぢありけり。  
怪しがりて寄りて見るに、筒の中ひかりたり。  
それを見れば、三寸ばかりなる人いと美しうて居たり。  
翁いふやう、「われ朝ごと夕ごとに見る、  
竹の中におはするにて知りぬ、子になり給ふべき人なんめり。」とて、手にうち入れて家にもてきぬ。  
妻の嫗にあづけて養はす。  
美しきこと限なし。いと幼ければ籠に入れて養ふ。  
竹取の翁この子を見つけて後に、竹をとるに、節をへだてゝよ毎に、金ある竹を見つくること重りぬ。  
かくて翁やう／＼豊になりゆく。  
この兒養ふほどに、すぐ／＼と大になります。  
三月ばかりになる程に、よきほどなる人になりぬれば、  
髪上などさだして、髪上せさせ裳着もぎす。  
帳ちやうの内よりも出さず、いつきかしづき養ふほどに、  
この兒のかたち清けうらなること世になく、家の内は暗き處なく光満ちたり。  
翁心地あしく苦しき時も、  
この子を見れば苦しき事も止みぬ。腹だしきことも慰みけり。  
翁竹をとること久しくなりぬ。  
勢猛の者になりにけり。  
この子いと大になりぬれば、名をば三室戸齋部秋田を呼びつけさす。  
秋田なよ竹のかぐや姫とつけつ。  
このほど三日うちあげ遊ぶ。萬の遊をぞしける。  
男女をとこをうなきらはず呼び集へて、いとかしこくあそぶ。

## ■組み込みプラグイン「ConversationalSummaryPlugin」の利用例

```
#pragma warning disable SKEXP0050

using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<ConversationSummaryPlugin>();
var kernel = builder.Build();

var kernelArguments = new KernelArguments
{
    ["input"] = File.ReadAllText("sample.txt")
};

var response = await kernel.InvokeAsync("ConversationSummaryPlugin", "SummarizeConversation",
kernelArguments);

Console.WriteLine(response);
```

KernelArgumentsオブジェクトを使用して、関数に対する入力パラメータを与える

InvokeAsync(Plugin名、関数名、  
kernelArguments)でPluginを呼び出し。ここでは  
文章の要約を行うSummarizeConversation関数を呼び  
出している。第3引数でkernelArgumentsオブジェクト  
を渡している。

## ■組み込みプラグイン「ConversationalSummaryPlugin」の「SummarizeConversation」関数による要約の例

Once upon a time, there was an old bamboo cutter named Sanuki no Miyatsuko. He lived by gathering bamboo from the mountains and using it for various purposes. One day, he discovered a glowing bamboo stalk, and upon cutting it open, he found a tiny, beautiful girl inside, about three inches tall. The old man believed she was destined to be his child and brought her home, entrusting her care to his wife. The girl grew rapidly and became extraordinarily beautiful, bringing light and joy to their home. The old man also began finding gold inside bamboo stalks, which made him wealthy. When the girl reached an appropriate age, they named her "Nayotake no Kaguya-hime" (Princess Kaguya of the Bamboo). To celebrate, they held a grand three-day feast, inviting people of all kinds to join in the festivities.

(参考訳) 昔、讃岐造という竹取の老人がいました。彼は山から竹を採り、それを様々な用途に使って暮らしていました。ある日、光る竹を見つけ、それを割ってみると、身長三寸ほどの小さな美しい女の子がいました。老人は、この女の子が自分の子になる運命にあると信じ、連れて帰り、妻に託しました。女の子はすくすくと成長し、並外れて美しくなり、家に光と喜びをもたらしました。老人はまた、竹の中から黄金を見つけるようになり、お金持ちになりました。女の子が適齢期になると、彼らは彼女を「なよたけのかぐや姫」と名付けました。お祝いに、人々は三日間、大宴会を開き、あらゆる人々を招いて祭りに参加させました。

# ここまでまとめ

- ・カーネルをビルドする途中で、プラグインを追加する
  - ・`builder.AddFromType<プラグインのクラス名>();`
- ・カーネルに組み込まれたプラグインの関数を直接呼び出す
  - ・入力パラメータなしの場合
    - ・`kernel.InvokeAsync("プラグイン名", "関数名");`
  - ・入力パラメータありの場合
    - ・`var kernelArguments = new KernelArguments { {パラメータ名, 値} };`
    - ・`kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments);`

# 参考: KernelArgumentsの書き方 (パラメータが複数個ある場合)

```
var kernelArguments = new KernelArguments
{
    {パラメータ1, 値1},
    {パラメータ2, 値2},
};
```

コレクションのAddメソッド  
を使う方法

```
var kernelArguments = new KernelArguments
{
    [パラメータ1] = 値1,
    [パラメータ2] = 値2,
};
```

インデックス初期化子を使う  
方法

# 参考: KernelArgumentsの書き方 (パラメータが1つだけの場合)

```
var kernelArguments = new KernelArguments
{
    {パラメータ1, 値1}
};
```

コレクションのAddメソッド  
を使う方法

```
var kernelArguments = new KernelArguments
{
    [パラメータ1] = 値1
};
```

インデックス初期化子を使う  
方法

# 参考: KernelArgumentsの書き方（1行で記述）

```
var kernelArguments = new KernelArguments { {パラメータ1, 値1} };
```

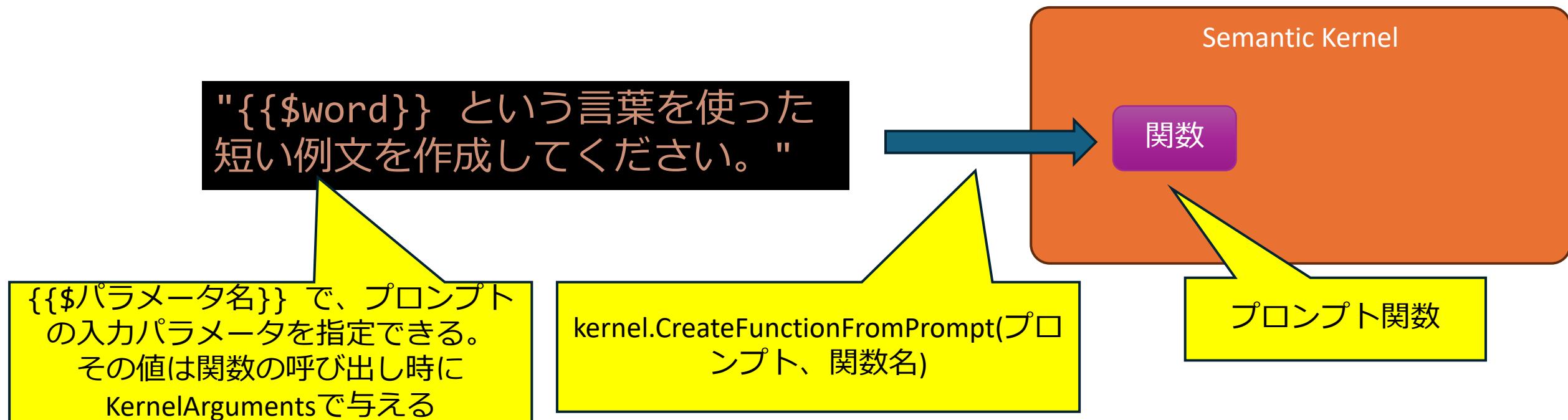
```
var kernelArguments = new KernelArguments { [パラメータ1] = 値1 };
```

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

# プロンプトから関数を作る

- ・プロンプト（生成AIに対する指示文）を関数化できる。
- ・これを「プロンプト関数」と呼ぶ
  - ・※以前のSemantic Kernelでは「セマンティック関数」とも呼ばれていた



# プロンプトから関数を作る

`{{$パラメータ名}}` で、プロンプトのパラメータ（入力）を指定できる。  
 値は関数の呼び出し時に  
 KernelArguments で与える

" `{{$word}}` という言葉を使った  
 短い例文を作成してください。 "



CreateFunctionFromPrompt  
 で作成されたプロンプト  
 関数はプラグインには属  
 していない

# プロンプト関数の作成と呼び出しの例

プロンプト関数の作成

```
var prompt = "{$word}" という言葉を使った短い例文を作成してください。";
```

```
var function = kernel.CreateFunctionFromPrompt(prompt, functionName: "GenerateSampleSentence");
```

プロンプト関数の呼び出し  
方法(1)プロンプト関数の  
InvokeAsyncメソッドを使用

```
var kernelArguments = new KernelArguments() { { "word", "肃々" } };  
var result = await function.InvokeAsync(kernel, kernelArguments);  
Console.WriteLine(result);
```

プロンプト関数の呼び出し  
方法(2)カーネルの  
InvokeAsyncメソッドを使用

```
var kernelArguments = new KernelArguments() { { "word", "肃々" } };  
var result = await kernel.InvokeAsync(function, kernelArguments);  
Console.WriteLine(result);
```

出力例（方法1・2どちらも同じ）

1. 式典は肃々と進行した。
2. 彼は肃々と仕事を片付けている。
3. 授業中、生徒たちは肃々と先生の話を聞いていた。

# プロンプト関数のプラグイン化

"{{\\$word}} という言葉を使った  
短い例文を作成してください。"

ImportPluginFromFunction  
を使用すると、関数をプ  
ラグインにインポートで  
きる



```
var function = kernel.CreateFunctionFromPrompt("プロンプト", "関数名");
kernel.ImportPluginFromFunction("プラグイン名", [function]);
```

## ■プロンプト関数の作成、**プラグインとしての登録（インポート）**、呼び出しの例

プロンプト関数の作成

```
var prompt = "{$word} という言葉を使った短い例文を作成してください。";
var function = kernel.CreateFunctionFromPrompt(prompt, functionName: "GenerateSampleSentence");
```

```
kernel.ImportPluginFromFunctions("SamplePlugin", [function]);
```

プラグイン「SamplePlugin」が作成され、そこにプロンプト関数が追加される。またこのプラグインがカーネルの管理下に置かれる

```
var kernelArguments = new KernelArguments() { { "word", "肃々" } };
var result = await kernel.InvokeAsync("SamplePlugin", "GenerateSampleSentence", kernelArguments);
Console.WriteLine(result);
```

出力例

すると、組み込みプラグインと同様にInvokeAsync()で呼び出しできる

1. 式典は肃々と進行した。
2. 彼は肃々と仕事を片付けている。
3. 授業中、生徒たちは肃々と先生の話を聞いていた。

# ここまでまとめ

- ・プロンプトから関数を作成できる。
- ・「プロンプト関数」と呼ばれる。
- ・プロンプト関数には**入力パラメータ**をもたせることができる

" **{{\$word}}**" という言葉を使った短い例文を作成してください。"

入力パラメータ

- ・**入力パラメータ**の値はKernelArgumentを使用して指定する。

```
var kernelArguments = new KernelArguments() { { "word", "肃々" } };
```

# ここまでまとめ

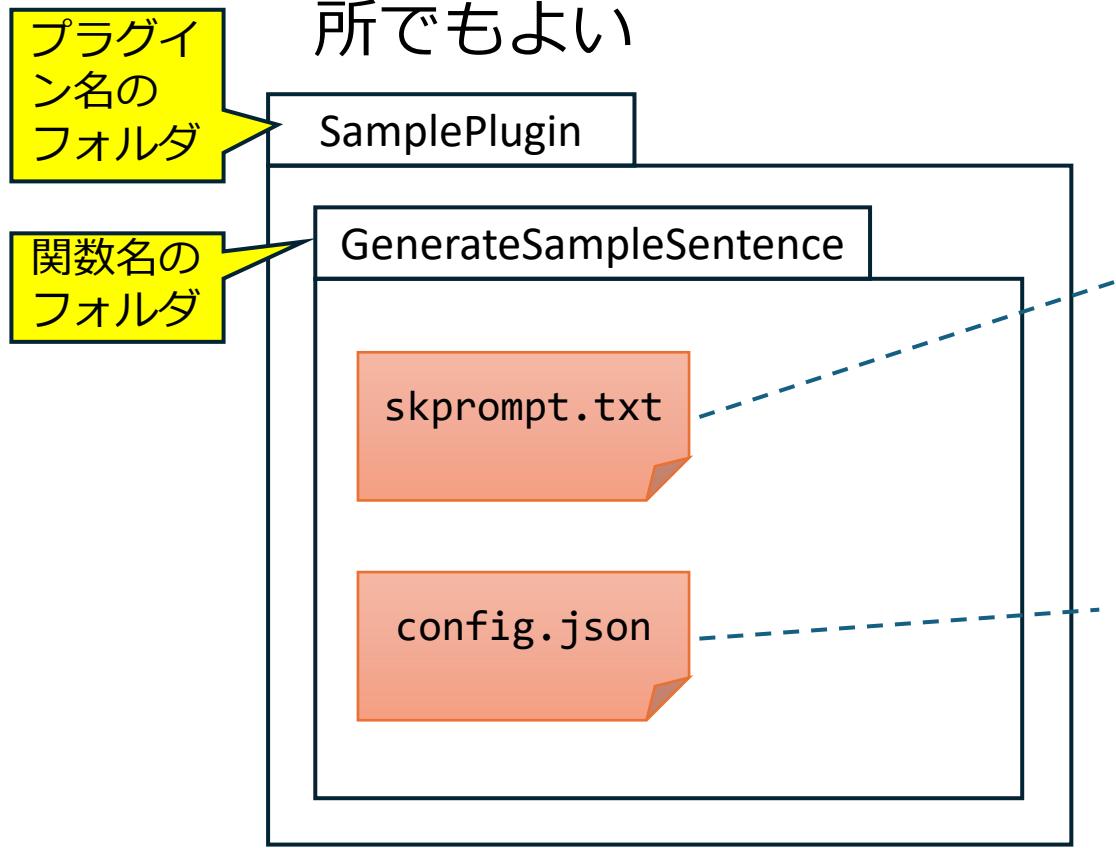
- プロンプト関数の作成
  - `var function = kernel.CreateFunctionFromPrompt("プロンプト", functionName: "関数名")`
- プロンプト関数の呼び出し（プラグイン化せずに呼び出す）
  - `var kernelArguments = new KernelArguments { {"パラメータ名", "パラメータの値"} };`
  - `function.InvokeAsync(kernel, kernelArguments);`  
or
  - `kernel.InvokeAsync(function, kernelArguments);`
- プロンプト関数の呼び出し（プラグイン化してから呼び出す）
  - `kernel.ImportPluginFromFunctions("プラグイン名", [function]);`
  - `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments);`

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
      - フォルダからファイルを読み込んで関数を作成する
    - メソッド関数 (C#の属性を使って関数を作成する)

# プラグインをフォルダから読み込む

- 以下のようなフォルダ・ファイルを用意する
  - このフォルダ自体の配置場所はどこでもよい。プロジェクト直下の `plugins` といった場所でもよいし、`C:\plugins` といったプロジェクト外の場所でもよい



`\{$word\}` という言葉を使った短い例文を作成してください。

```
{  
    "schema": 1,  
    "description" : "指定された単語を含む短い例文を作成",  
    "input_variables": [  
        {  
            "name": "word",  
            "description": "例文に含める単語"  
        }  
    ]  
}
```

## ■ フォルダからのプラグインのインポートと利用

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);

builder.Plugins.AddFromPromptDirectory("plugins/SamplePlugin");

var kernel = builder.Build();

var kernelArguments = new KernelArguments
{
    ["word"] = "肃々"
};
var response = await kernel.InvokeAsync("SamplePlugin", "GenerateSampleSentence", kernelArguments);

Console.WriteLine(response);
```

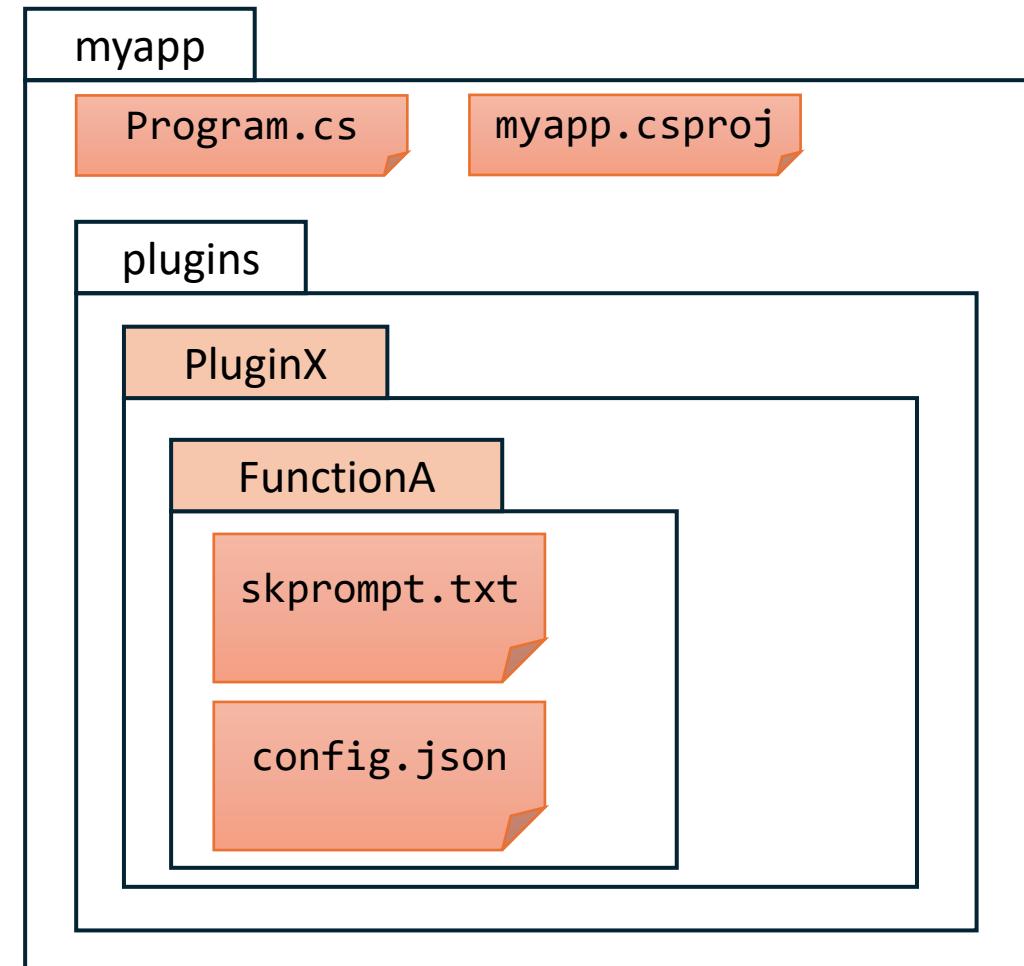
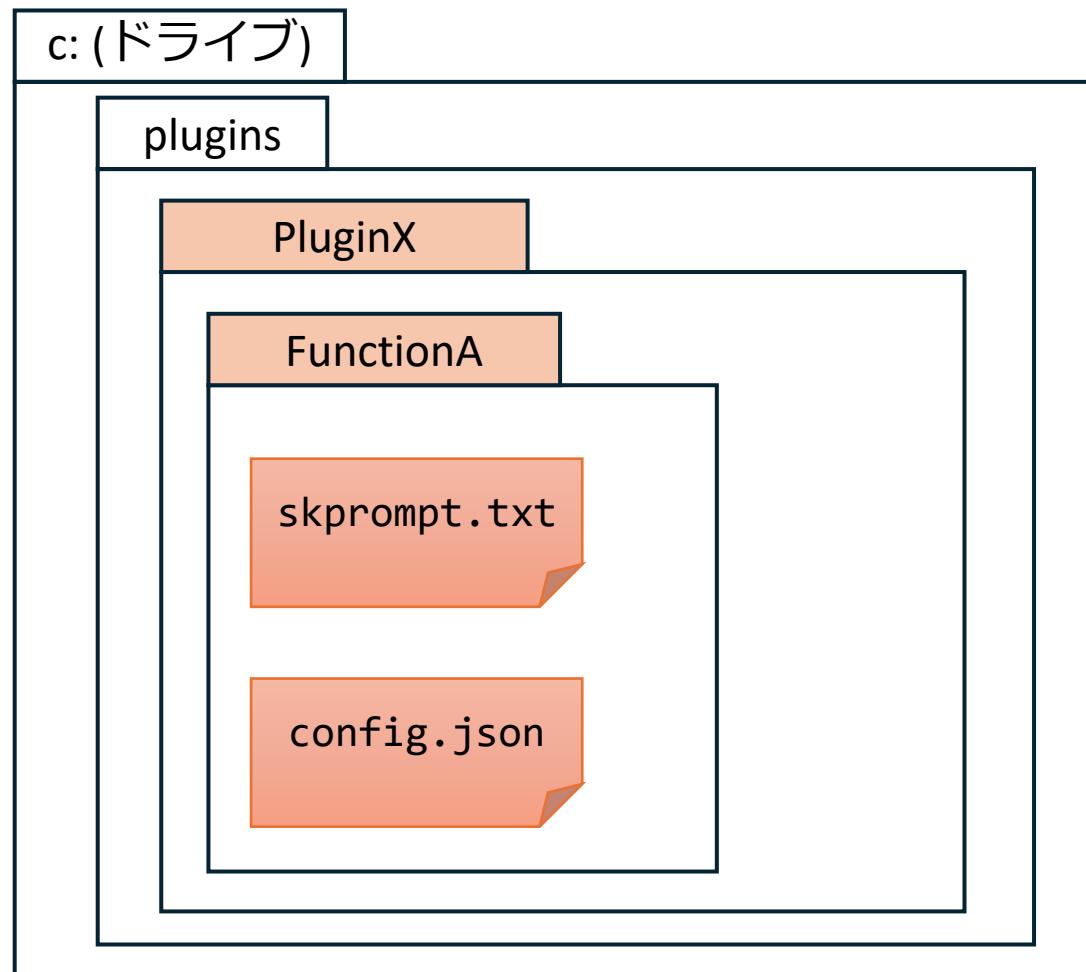
AddFromPromptDirectory(プラグイン名のフォルダ)で、そこにある関数をすべて読み込む。

### 出力例

1. 式典は肃々と進行した。
2. 彼は肃々と仕事を片付けている。
3. 授業中、生徒たちは肃々と先生の話を聞いていた。

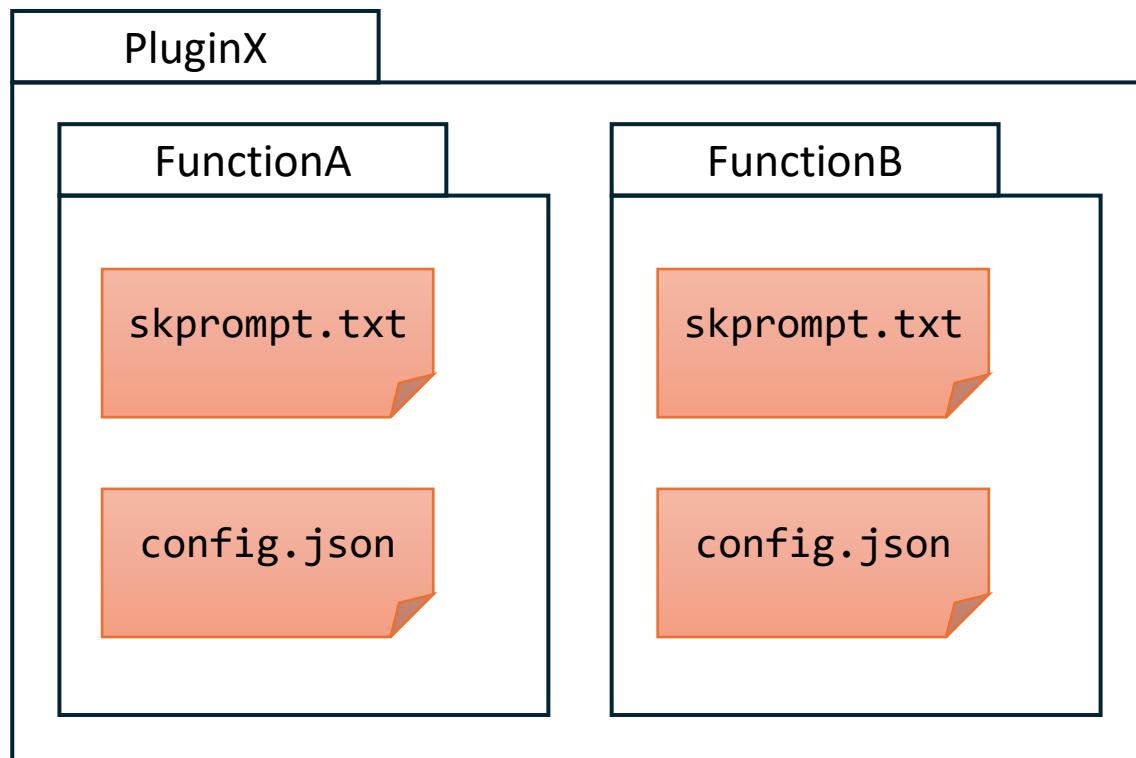
# フォルダ構成

- ・ プラグインのフォルダ自体の配置場所はどこでもかまわない。
- ・ C:\plugins といったプロジェクト外の場所でもよいし、プロジェクト直下の plugins といった場所でもよい。



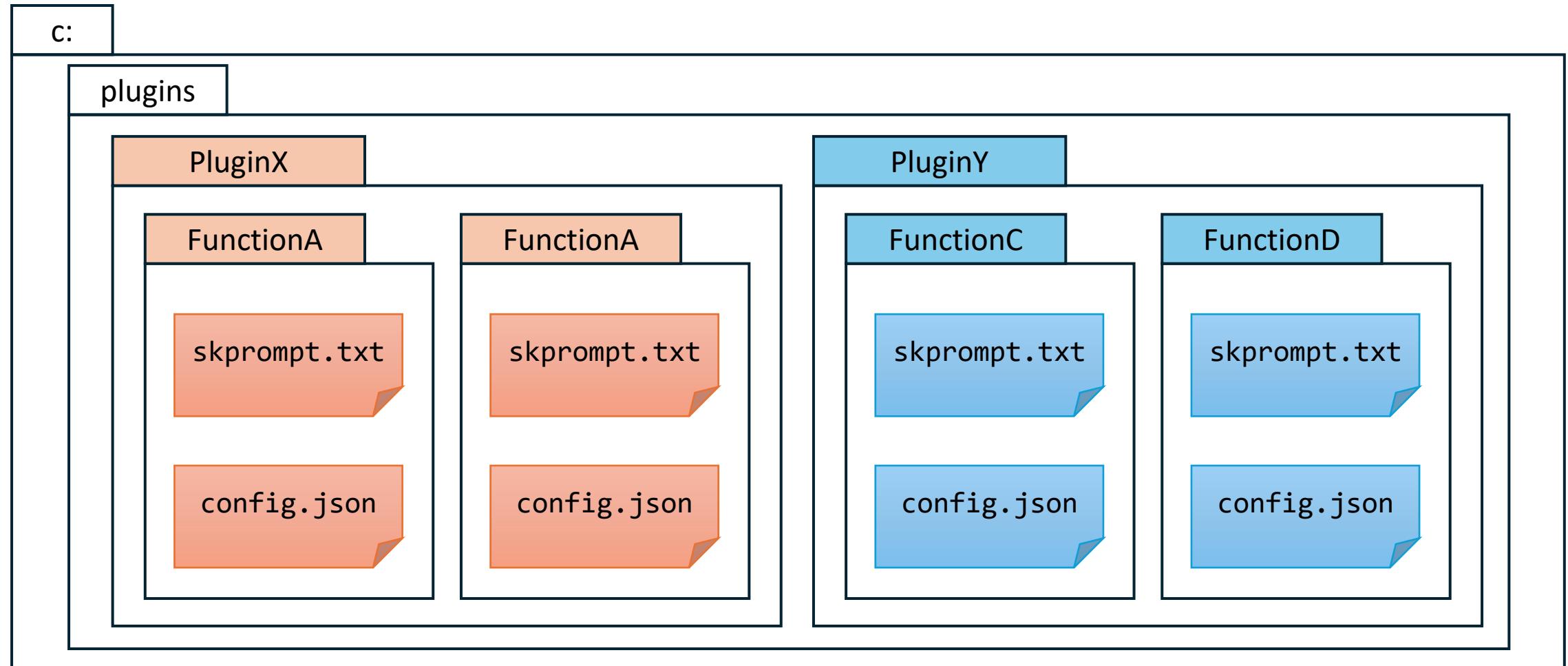
# プラグインをフォルダから読み込む

- 1つのプラグインには複数の関数を配置することもできる



```
builder.Plugins.AddFromPromptDirectory("plugins/PluginX");
```

# 複数のプラグインがある場合



```
builder.Plugins.AddFromPromptDirectory("c:/plugins/PluginX");
```

PluginX(FunctionA,  
FunctionB)を読み込む

```
builder.Plugins.AddFromPromptDirectory("c:/plugins/PluginY");
```

PluginY(FunctionC,  
FunctionD)を読み込む

## ■少し複雑な関数（「Coding関数」）の定義例

### Coding

skprompt.txt

以下の要件に従ってコード例を出力してください。  
コード例はなるべくシンプルで短いものとしてください。

要件: {{\$requirements}}

使用するプログラミング言語:  
{{\$language}}

2つの入力パラメータを使用

入力パラメータ「requirements」の仕様を定義

入力パラメータ「language」の仕様を定義

config.json

```
{  
    "schema": 1,  
    "description": "Generate a program source code",  
    "execution_settings": {  
        "default": {  
            "max_tokens": 1000,  
            "temperature": 0.1  
        }  
    },  
    "input_variables": [  
        {  
            "name": "requirements",  
            "description": "requirements for the program",  
            "default": "output 'hello world'"  
        },  
        {  
            "name": "language",  
            "description": "the programming language to use",  
            "default": "C#"  
        }  
    ]  
}
```

GPT-4oに対する指定  
・最大出力トークン数を1000に制限  
・温度を設定（小さい値に設定すると毎回同じような結果が生成される）

## ■ フォルダからのプラグインのインポートと利用

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

builder.Plugins.AddFromPromptDirectory("plugins/ProgrammerPlugin");

builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
var kernel = builder.Build();

var kernelArguments = new KernelArguments
{
    {"requirements", "1から100までの間の乱数を1つ生成"},
    {"language", "Python"}
};
var response = await kernel.InvokeAsync("ProgrammerPlugin", "Coding", kernelArguments);

Console.WriteLine(response);
```

以下は、Pythonで1から100までの間の乱数を1つ生成するシンプルなコード例です：

```
```python
import random

print(random.randint(1, 100))
```
```

# モジュール2 + モジュール3

- 「プラグイン」とは？「関数」とは？
- 組み込みプラグインとは？
- 組み込みプラグイン利用時に表示される警告について
- 組み込みプラグインの利用例
  - TimePlugin
  - ConversationalSummaryPlugin
- プラグインの自作
  - プロンプト関数
    - コード内のプロンプトから関数を作成する
    - フォルダからファイルを読み込んで関数を作成する
  - メソッド関数 (C#の属性を使って関数を作成する)

# 「メソッド関数」とは

- C#などの言語を使用して書かれた関数
- 以前のSemantic Kernelでは「ネイティブ関数」とも呼ばれていた
- このクラスはどこに配置されていてもかまわない

```
class SamplePlugin
{
    [KernelFunction]
    public static string Hello() => "Hello from SamplePlugin!";
}
```

クラス名=プラグイン名  
となる

メソッド関数

# メソッド関数でできること

- ・メソッド関数内では**任意の処理**を実行できる！
  - ・システム日時の取得
  - ・ファイル入出力
  - ・ネットワーク入出力（外部APIの呼び出し、Web検索など）
  - ・OSコマンドの実行
  - ・データベースへのアクセス
  - ・IoTデバイスのコントロール/情報取得

# C#の「属性」（attributes）を使用して、 メソッド関数を作る

- **[KernelFunction("...")]**
  - この属性を付けたメソッドが、メソッド関数となる

```
class SamplePlugin
{
    [KernelFunction]
    public static string Hello() => "Hello from SamplePlugin!";
}
```

# 一つのクラス（プラグイン）で 複数のメソッド関数を定義できる

```
class SamplePlugin
{
    [KernelFunction]
    public static string Hello() => "Hello from SamplePlugin!";

    [KernelFunction]
    public static string Now() => DateTime.Now.ToString();

    public static int Helper(int x) => x + 1;
}
```

クラス名 = プラグイン名  
となる

メソッド関数 Hello

メソッド関数 Now

これは属性が付いていない  
のでメソッド関数ではない

## ■ネイティブ関数を含むプラグインの例（ネイティブ関数の定義と呼び出し）

```
using Microsoft.SemanticKernel;  
  
var builder = Kernel.CreateBuilder();  
builder.Plugins.AddFromType<SamplePlugin>();  
var kernel = builder.Build();  
  
var result = await kernel.InvokeAsync("SamplePlugin", "Hello");  
Console.WriteLine(result);  
  
class SamplePlugin  
{  
    [KernelFunction]  
    public static string Hello() => "Hello from SamplePlugin!";  
  
    [KernelFunction]  
    public static string Now() => DateTime.Now.ToString();  
}
```

AddFromType<プラグインのクラス>()で、  
プラグインをカーネルに追加

プラグイン名と関数名を指定し  
て関数を呼び出し  
※組み込みプラグインと同様

「SamplePlugin」プラグインと  
関数の定義

Hello from SamplePlugin!

## ■複数のメソッド関数を含むプラグインの例

```
using Microsoft.SemanticKernel;

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var result = await kernel.InvokeAsync("SamplePlugin", "Now");
Console.WriteLine(result);

class SamplePlugin
{
    [KernelFunction]
    public static string Hello() => "Hello from SamplePlugin!";

    [KernelFunction]
    public static string Now() => DateTime.Now.ToString();
}
```

プラグインには複数の関数を含  
めることができる

Hello関数

Now関数

# メソッド関数に説明を加える

- **[Description("...")]**
  - 関数、関数の引数、関数の戻り値に説明を加える。
  - **生成AIモデルは説明、関数名、引数名などの情報を使用して、呼び出すべき関数や、関数に渡すべき引数を決定する**
  - **つまり生成AIモデルが適切な関数を選択したり、関数に適切な引数（入力パラメータ）を与えるためのヒントとなる**
- この属性を使用する場合は、ソースコードの頭の部分に **using System.ComponentModel;** を追加する

## ■ メソッド関数の例（関数の説明を追加）

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

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var hiArgs = new KernelArguments { ["name"] = "太郎" };
var hiResult = await kernel.InvokeAsync("SamplePlugin", "Hi", hiArgs);
Console.WriteLine(hiResult);

class SamplePlugin {
    [KernelFunction("Hi")]
    [Description("Says hi to the given name.")]
    [return: Description("The greeting.")]
    public static string Hi([Description("the name of the user")]string name) => $"{name}さん、こんにちは";
}
```

関数の説明

太郎さん、こんにちは

## ■ メソッド関数の例（戻り値の説明を追加）

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

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var hiArgs = new KernelArguments { ["name"] = "太郎" };
var hiResult = await kernel.InvokeAsync("SamplePlugin", "Hi", hiArgs);
Console.WriteLine(hiResult);

class SamplePlugin {
    [KernelFunction("Hi")]
    [Description("Says hi to the given name.")]
    [return: Description("The greeting.")]
    public static string Hi([Description("the name of the user")]string name) => $"{name}さん、こんにちは";
}
```

関数の戻り値の説明

太郎さん、こんにちは

## ■ メソッド関数の例（引数の説明を追加）

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

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var hiArgs = new KernelArguments { ["name"] = "太郎" };
var hiResult = await kernel.InvokeAsync("SamplePlugin", "Hi", hiArgs);
Console.WriteLine(hiResult);

class SamplePlugin {
    [KernelFunction("Hi")]
    [Description("Says hi to the given name.")]
    [return: Description("The greeting.")]
    public static string Hi([Description("the name of the user")]string name) => $"{name}さん、こんにちは";
}
```

関数の引数の説明

太郎さん、こんにちは

## ■ メソッド関数の呼び出し例

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

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var kernelArguments = new KernelArguments { ["name"] = "太郎" };
var result = await kernel.InvokeAsync("SamplePlugin", "Hi", kernelArguments);
Console.WriteLine(result);

class SamplePlugin {
    [KernelFunction("Hi")]
    [Description("Says hi to the given name.")]
    [return: Description("The greeting.")]
    public static string Hi([Description("the name of the user")]string name) => $"{name}さん、こんにちは";
}
```

引数はKernelArgumentsで指定

InvokeAsyncに  
KernelArgumentsを渡す

太郎さん、こんにちは

## ■ メソッド関数の呼び出し例（複数の引数）

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

var builder = Kernel.CreateBuilder();
builder.Plugins.AddFromType<SamplePlugin>();
var kernel = builder.Build();

var arguments = new KernelArguments { {"x", 1}, {"y", 2} };
var result = await kernel.InvokeAsync("SamplePlugin", "Add", arguments);

Console.WriteLine(result);

class SamplePlugin {
    [KernelFunction("Add")]
    [Description("Adds two numbers.")]
    [return: Description("The sum of the two numbers.")]
    public static int Add([Description("the value of X")]int x, [Description("the value of Y")]int y)
        => x + y;
}
```

2つの引数を渡すことも  
できる

# モジュール2・3まとめ

- 組み込みプラグインの利用方法

- `builder.Plugins.AddFromType<プラグインのクラス>()`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- プロンプト関数の利用方法

- `var function = kernel.CreateFunctionFromPrompt("プロンプト", functionName: "関数名")`
- `kernel.ImportPluginFromFunctions("プラグイン名", [function])`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- フォルダからのプラグインの読み込み方法

- `skprompt.txt`にプロンプト、`config.json`に関数の仕様を記述
- `builder.Plugins.AddFromPromptDirectory("フォルダのパス")`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- メソッド関数の利用方法

- メソッドに`[KernelFunction]`属性を付けたクラスを定義
- `builder.Plugins.AddFromType<プラグインのクラス>()`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

# モジュール4 + モジュール5



## プロンプトと関数を組み合わせる

29分・モジュール・5ユニット

[△フィードバック](#)

中級 開発者 .NET Visual Studio Code Azure OpenAI Service

このモジュールでは、関数とプロンプトをセマンティック カーネル SDK と組み合わせる方法を示します。プロンプト内で関数を入れ子にすると、通常、大規模な言語モデルが単独では完了できないタスクを自分のコードを使って完了できます。

**学習の目的**

- セマンティック カーネル SDK を使用してプラグインを作成する練習。
- プロンプトとネイティブ関数を組み合わせる方法を学習します。



## 関数の自動呼び出し

26分 残り・モジュール・1/5ユニットが完了しました

[△フィードバック](#)

中級 開発者 .NET Visual Studio Code Azure OpenAI Service

このモジュールでは、Semantic Kernel SDK を使用して関数を自動的に呼び出す設定について説明します。Semantic Kernel を使用して関数を自動的に呼び出し、ユーザーの要求を完了する方法について説明します。

**学習の目的**

- Semantic Kernel SDK を使用して関数を自動的に呼び出す方法を学習します。

# 概要

- ・モジュール1では、プロンプトからコンテンツを生成する方法について学習しました
- ・モジュール2・3では、プラグインと関数の利用方法について学習しました
- ・モジュール4・5では、これらを**組み合わせて**使う方法を学習します
  - ・つまり「**生成AIがプロンプトからコンテンツを生成する際にプラグインの関数を利用してもらう方法**」を学習します
  - ・さらに言い換えると「**生成AIにプラグインを与えて仕事をしてもらう方法**」あるいは「**AIエージェントの作成方法**」を学習します

# (復習) モジュール1: プロンプトからのコンテンツ生成

```
var response = await kernel.InvokePromptAsync("こんにちは");
```

プロンプトからのコンテン  
ツ生成（1ターン）

```
var textGenerationService = kernel.GetRequiredService<ITextGenerationService>();  
var response = await textGenerationService.GetTextContentAsync("こんにちは");
```

プロンプトからのコンテン  
ツ生成（1ターン）

```
var chatCompletionService = kernel.GetRequiredService<IChatCompletionService>();  
var history = new ChatHistory();  
history.AddUserMessage(input);  
var response = await chatCompletionService.GetChatMessageContentAsync(history);  
history.AddAssistantMessage(response.ToString());
```

プロンプトからのコンテン  
ツ生成（複数ターン）

# 概要

- ・モジュール1では、プロンプトからコンテンツを生成する方法について学習しました
  - ・モジュール2・3では、プラグインと関数の利用方法について学習しました
- 
- ・モジュール4・5では、これらを**組み合わせて**使う方法を学習します
    - ・つまり「**生成AIがプロンプトからコンテンツを生成する際にプラグインの関数を利用してもらう方法**」を学習します
    - ・さらに言い換えると「**生成AIにプラグインを与えて仕事をしてもらう方法**」あるいは「**AIエージェントの作成方法**」を学習します

# (復習) モジュール2・3: プラグイン(関数)の利用方法

- 組み込みプラグインの利用方法

- `builder.Plugins.AddFromType<プラグインのクラス>()`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- プロンプト関数の利用方法

- `var function = kernel.CreateFunctionFromPrompt("プロンプト", functionName: "関数名")`
- `kernel.ImportPluginFromFunctions("プラグイン名", [function])`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- フォルダからのプラグインの読み込み方法

- `skprompt.txt`にプロンプト、`config.json`に関数の仕様を記述
- `builder.Plugins.AddFromPromptDirectory("フォルダのパス")`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

- メソッド関数の利用方法

- メソッドに`[KernelFunction]`属性を付けたクラスを定義
- `builder.Plugins.AddFromType<プラグインのクラス>()`
- `kernel.InvokeAsync("プラグイン名", "関数名", kernelArguments)`

# 概要

- ・モジュール1では、プロンプトからコンテンツを生成する方法について学習しました
- ・モジュール2・3では、プラグインと関数の利用方法について学習しました

- ・モジュール4・5では、これらを**組み合わせて**使う方法を学習します
  - ・つまり「**生成AIがプロンプトからコンテンツを生成する際にプラグインの関数を利用してもらう方法**」を学習します
  - ・さらに言い換えると「**生成AIにプラグインを与えて仕事をしてもらう方法**」あるいは「**AIエージェントの基礎となる技術**」を学習します

# モジュール4 + モジュール5

- ・プロンプトから関数を利用する
    - ・(1)プロンプト内での関数の明示的な呼び出し
    - ・(2)関数の自動呼び出し
      - ・kernel.InvokePromptAsyncから
      - ・chatCompletionService.GetChatMessageContentAsyncから
- 
- モジュール4
- モジュール5

# モジュール4 + モジュール5

- プロンプトから関数を利用する
    - (1)プロンプト内での関数の明示的な呼び出し
    - (2)関数の自動呼び出し
      - kernel.InvokePromptAsyncから
      - chatCompletionService.GetChatMessageContentAsyncから
- 
- モジュール4
- モジュール5

# 参考: GPTは現在時刻を取得できない

The screenshot shows the user interface of ChatGPT 4o mini. At the top left is the logo and text "ChatGPT 4o mini ▾". On the right are "ログイン" (Login) and "サインアップ" (Sign Up) buttons. In the center, a message bubble contains the question "今何時ですか？" (What is the current time?). Below it, the AI's response is shown: "私はリアルタイムでの時刻を確認することができませんが、現在の時刻はお使いのデバイスや時計でご確認いただけます。" (I cannot check the current time in real-time, but you can check it on your device or clock). There is a reply icon (a square with a dot) next to the response. At the bottom, there is a message input field with "ChatGPT にメッセージを送信する" (Send message to ChatGPT) and a send button with a paper plane icon. A note at the bottom states "ChatGPT の回答は必ずしも正しいとは限りません。重要な情報は確認するようにしてください。" (The answer from ChatGPT is not necessarily always correct. Please verify important information). There are also icons for a help question mark and an upward arrow.

# kernel.InvokePromptAsync("プロンプト")で 明示的にプロンプトの関数を呼び出す

```
var response = await kernel.InvokePromptAsync(  
    "現在の時刻は {{TimePlugin.Now}} です。適切な挨拶をしてください");
```

プロンプトの文字列内に「{{プラグイン名.関数名}}」  
を含めることで、その関数を呼び出し、その結果をこの  
プロンプトに含めて、プロンプトを実行することができる  
(動的プロンプト)。  
この例では、現在の時刻がこのプロンプトに埋め込まれ  
て実行される

## ■プロンプト内でプラグインの関数を明示的に呼び出す例

```
#pragma warning disable SKEXP0050
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Plugins.Core;

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<TimePlugin>(); TimePluginを追加
var kernel = builder.Build();

var response = await kernel.InvokePromptAsync("現在の時刻は {{TimePlugin.Now}} です。適切な挨拶をしてください");
Console.WriteLine(response);
```

TimePluginのNow関数を明示的に呼び出して、その結果をプロンプトに含めて、プロンプトを実行

こんにちは！金曜日の夕方ですね。1週間お疲れ様でした！そろそろリラックスして週末に向けてゆったりとした時間を過ごせるころでしょうか？何かお手伝いがあればお気軽にお知らせくださいね！ 😊

現在の時刻に応じた適切なテキストが生成される。

# モジュール4 + モジュール5

- ・プロンプトから関数を利用する

- ・(1)プロンプト内での関数の明示的な呼び出し

- ・(2)関数の自動呼び出し

- kernel.InvokePromptAsyncから

- chatCompletionService.GetChatMessageContentAsyncから



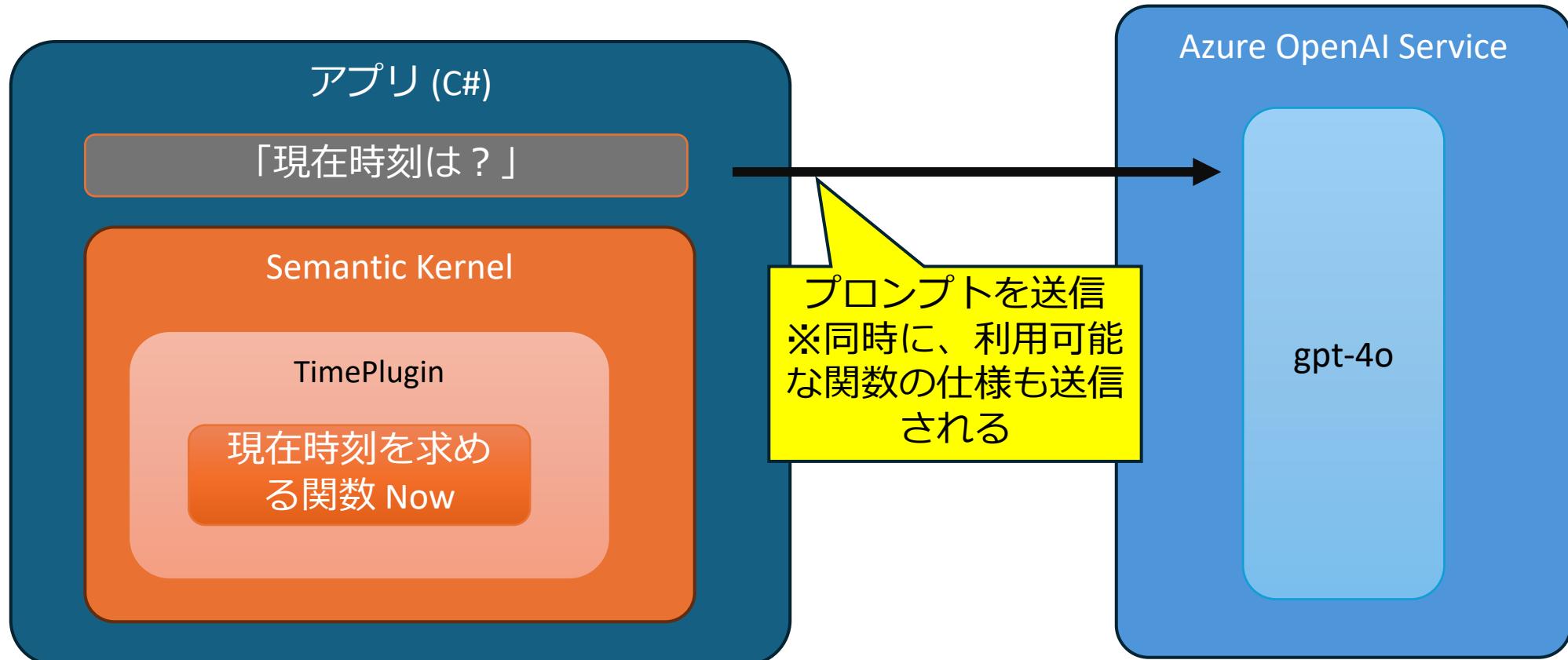
モジュール4



モジュール5

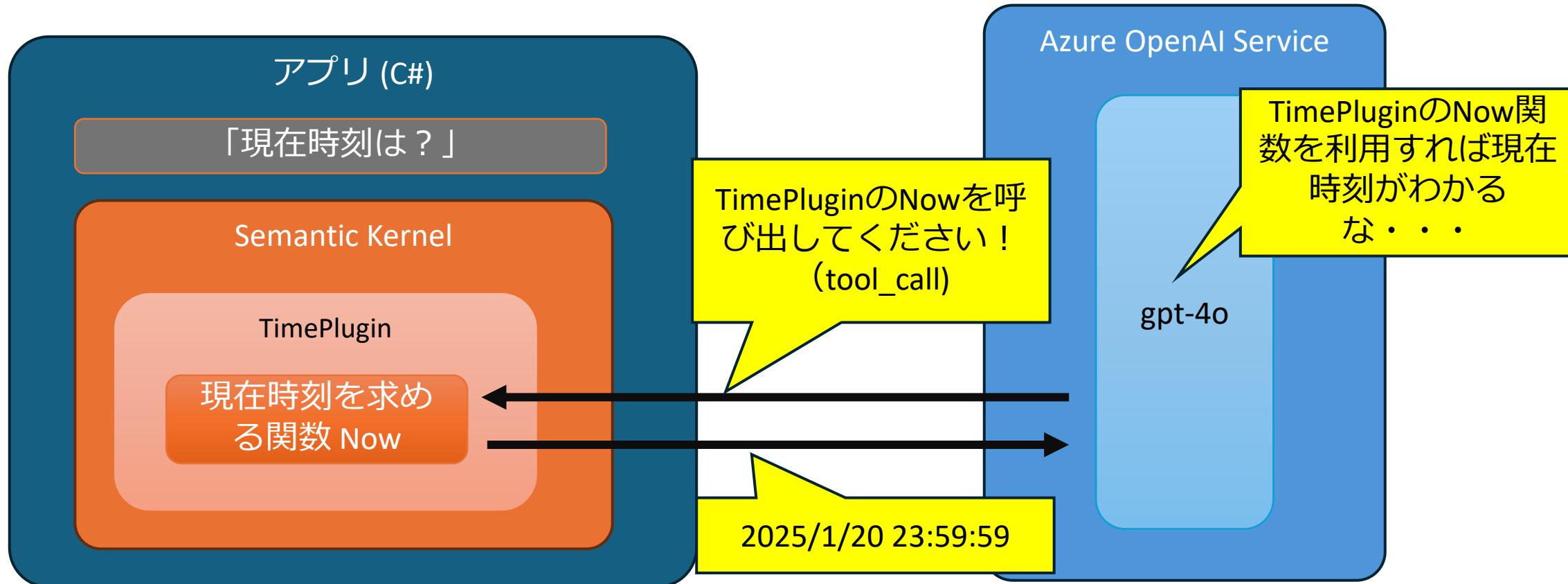
# 関数の自動呼び出し

- 生成AIモデル側で、プロンプトの実行に必要な関数を決定し、関数を利用させることができる



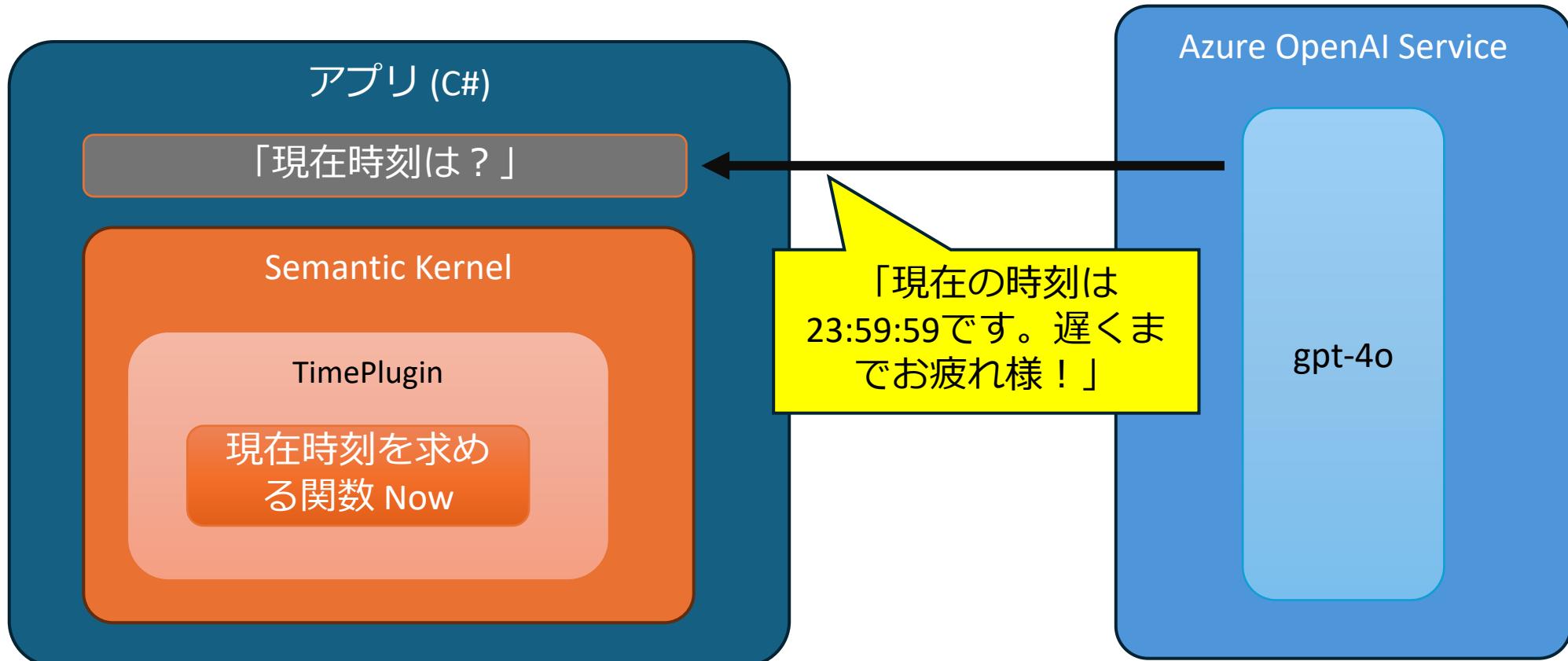
# 関数の自動呼び出し

- 生成AIモデル側で、プロンプトの実行に必要な関数を決定し、関数を利用させることができる



# 関数の自動呼び出し

- 生成AIモデル側で、プロンプトの実行に必要な関数を決定し、関数を利用させることができる



# モジュール4 + モジュール5

- ・プロンプトから関数を利用する

- ・(1)プロンプト内での関数の明示的な呼び出し

- ・(2)関数の自動呼び出し

- ・kernel.InvokePromptAsyncから

- ・chatCompletionService.GetChatMessageContentAsyncから



モジュール4



モジュール5

# 関数の自動呼び出し

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

```
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};
```

```
var arg = new KernelArguments(openAIPromptExecutionSettings);
```

```
var response = await kernel.InvokePromptAsync("現在の時刻を表示し、適切な挨拶をしてください", arg);
Console.WriteLine(response);
```

プロンプト実行時の設定を保持するOpenAIPromptExecutionSettingsを使用する際はこの記述が必要

プロンプト実行時の設定を保持するオブジェクトを作成

必要に応じて、関数の呼び出しを自動的に行うように設定

# 関数の自動呼び出し

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

```
OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};

var arg = new KernelArguments(openAIPromptExecutionSettings);  
KernelArgumentsにプロンプト実行時の設定を組み込む  
var response = await kernel.InvokePromptAsync("現在の時刻を表示し、適切な挨拶をしてください", arg);
Console.WriteLine(response);
```

InvokePromptAsyncの第2引数としてKernelArgumentsを渡す

## ■ プラグインの関数を自動的に呼び出す例 (InvokePromptAsync)

```
#pragma warning disable SKEXP0050
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.OpenAI;
using Microsoft.SemanticKernel.Plugins.Core;

var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<TimePlugin>();
var kernel = builder.Build();

OpenAIPromptExecutionSettings openAIPromptExecutionSettings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};

var arg = new KernelArguments(openAIPromptExecutionSettings);
var response = await kernel.InvokePromptAsync("現在の時刻を表示し、適切な挨拶をしてください", arg);
Console.WriteLine(response);
```

現在の時刻は、2025年1月17日（金）17時25分です。  
こんにちは！夕方ですね。お疲れ様です！

# モジュール4 + モジュール5

- ・プロンプトから関数を利用する

- ・(1)プロンプト内での関数の明示的な呼び出し

- ・(2)関数の自動呼び出し

- ・kernel.InvokePromptAsyncから

- ・chatCompletionService.GetChatMessageContentAsyncから



モジュール4



モジュール5

## ■ プラグインの関数を自動的に呼び出す例 (IChatCompletionService.GetChatMessageContentAsync)

```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<TimePlugin>();
var kernel = builder.Build();
OpenAIPromptExecutionSettings settings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};

var chatCompoletionService = kernel.GetRequiredService<IChatCompletionService>();
var history = new ChatHistory();

while (true)
{
    Console.Write("あなた: ");
    var input = Console.ReadLine() ?? "";
    history.AddUserMessage(input);
    var response = await chatCompoletionService.GetChatMessageContentAsync(history, settings, kernel);
    Console.WriteLine($"アシスタント: {response}");
    history.AddAssistantMessage(response.ToString());
}
```

必要に応じて、  
関数の呼び出しを自動  
的に行うように設定

GetChatMessageContent  
Asyncの引数にsettings  
とkernelを追加

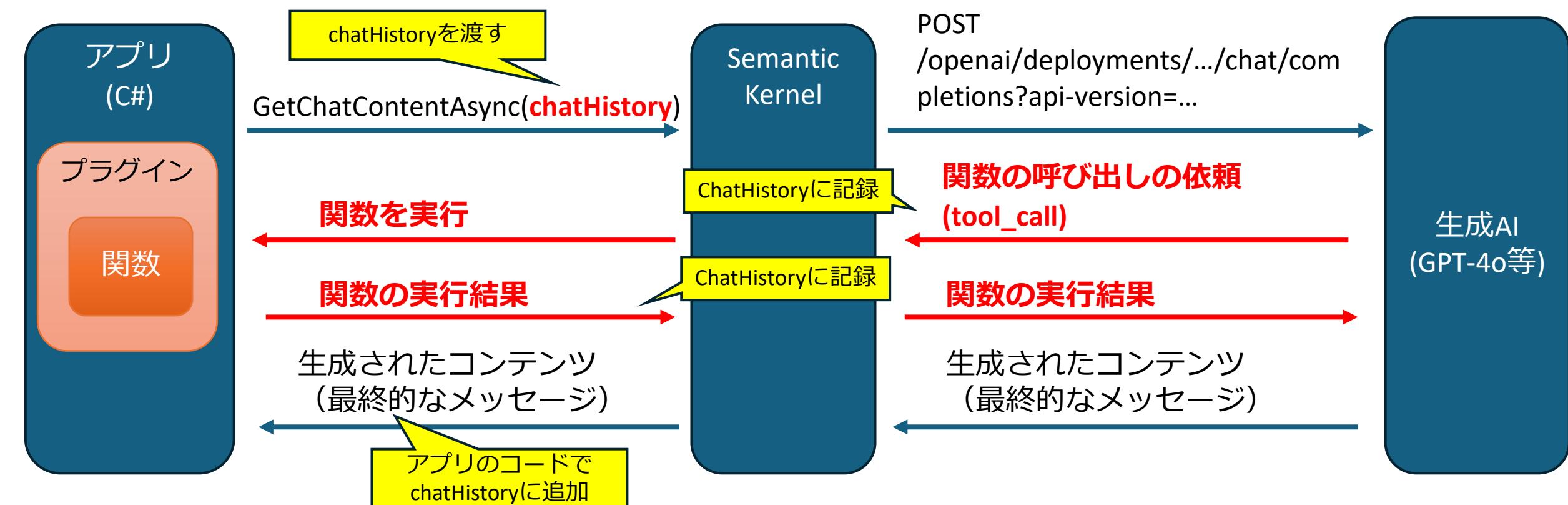
あなた: 現在の時刻を表示し、適切な挨拶をしてください

アシスタント: 現在の時刻は、2025年1月20日（月）21時45分です。こんばんは！今日はどんな一日でしたか？

あなた:

## ■ IChatCompletionServiceでのChatHistoryの使い方

- 自動関数呼び出しが有効になっているChatCompletionServiceに ChatHistoryオブジェクトを渡すと、ChatHistoryオブジェクトには**関数呼び出しと結果が含まれるように操作される**
- ただし、最終的なメッセージは、アプリケーション側で、 ChatHistory オブジェクトに追加する必要がある



```
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<TimePlugin>();
var kernel = builder.Build();
OpenAIPromptExecutionSettings settings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};

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

while (true)
{
    Console.Write("あなた: ");
    var input = Console.ReadLine() ?? "";
    history.AddUserMessage(input);
    var response = await chatCompletionService.GetChatMessageContentAsync(history, settings, kernel);
    Console.WriteLine($"アシスタント: {response}");
    history.AddAssistantMessage(response.ToString());
}
```

GetChatMessageContentAsync()の結果（生成AIが生成したテキスト）はアプリ側のコードでChatHistoryに追加する必要がある（が、関数呼び出しの履歴に関してはSemantic Kernelで制御されるため、特にその部分のコードを記述する必要はない！）

# モジュール4・5まとめ

- プラグインの関数をプロンプト内で明示的に呼び出しできる
  - `InvokePromptAsync("... {{プラグイン名.関数名}} ...")`
- プラグインの関数がプロンプト実行時に自動で呼び出されるよう設定できる
  - `kernel.InvokePromptAsync()`を利用する場合
    - `var settings = new OpenAIPromptExecutionSettings {  
 FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() };`
    - `var arg = new KernelArguments(setting);`
    - `InvokePromptAsync("プロンプト", arg);`
  - `chatCompletionService.GetChatMessageContentAsync()`を利用する場合
    - `var settings = new OpenAIPromptExecutionSettings {  
 FunctionChoiceBehavior = FunctionChoiceBehavior.Auto() };`
    - `var response = await chatCompletionService.GetChatCompletionAsync(history, settings, kernel);`



## ガイド付きプロジェクト - AIエージェントを作成する

14 分 残り • モジュール • 3/7 ユニットが完了しました

△ フィードバック

中級

開発者

.NET

Visual Studio Code

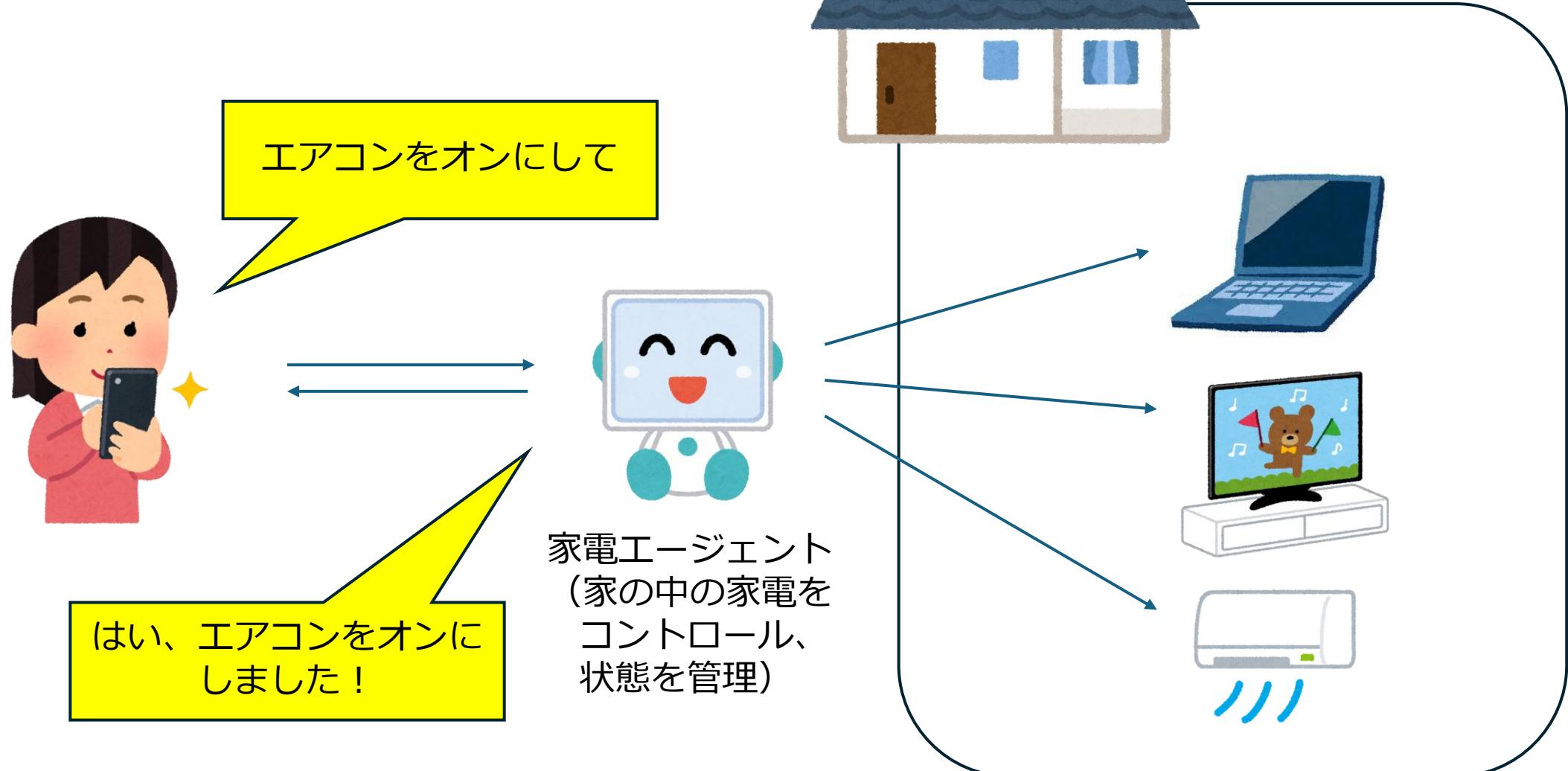
Azure OpenAI Service

このモジュールでは、Semantic Kernel SDKを使用して、概念実証（Proof of Concept、PoC）を行います。ここでは例として家電を制御するAIエージェントを開発します。

# モジュール6

- ・ここまでで学習した知識をすべて使用して、AIエージェント「家電エージェント」を作成します

# 「家電エージェント」のイメージ



## ■ 「家電エージェント」の利用例

あなた： こんにちは

エージェント： こんにちは！ 家電の管理が必要であれば、何でもお手伝いしますよ。どのようなサポートが必要ですか？

あなた： 現在のすべての家電の状態を教えて

エージェント： 現在の家電の状態は以下の通りです：

1. パソコン： オフ
2. テレビ： オフ
3. エアコン： オフ

あなた： エアコンの電源をオンにして

エージェント： エアコンの電源をオンにしました。

あなた： 現在オンになっている家電はどれ？

エージェント： 現在オンになっている家電は以下の通りです：

- エアコン



あなた： テレビの電源をつけて

エージェント： 現在オンになっている家電は、エアコンとテレビです。

あなた： 現在の家電の状態は？

エージェント： 現在の家電の状態は以下の通りです：

1. パソコン： オフ
2. テレビ： オン
3. エアコン： オン

あなた： 全部オフにして

エージェント： すべての家電の電源をオフにしました。現在の状態は以下の通りです：

- パソコン： オフ
- テレビ： オフ
- エアコン： オフ

あなた：



# 備考

- ・今回作成する「家電エージェント」は、家電を実際にコントロールするのではなく、3つの家電（パソコン、テレビ、エアコン）のオン・オフ状態をメモリ上でシミュレーションするものとします。
- ・ただし、家電を制御可能な赤外線リモコンを用意し、今回開発するエージェントに接続することで、家電を実際に制御することも技術的には可能です。



## ■C#プロジェクトの作成

```
> mkdir proj01  
  
> cd proj01  
  
> dotnet new console  
  
> dotnet add package Microsoft.SemanticKernel
```

## ■家電（アプライアンス）を表すクラスの作成

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

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

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

## ■家電（アプライアンス）を制御するプラグインの作成

```
public class AppliancesPlugin
{
    private readonly List<ApplianceModel> appliances = new()
    {
        new ApplianceModel { Id = 1, Name = "パソコン", IsOn = false },
        new ApplianceModel { Id = 2, Name = "テレビ", IsOn = false },
        new ApplianceModel { Id = 3, Name = "エアコン", IsOn = false }
    };

    [KernelFunction]
    public List<ApplianceModel> GetAppliances() { ... }

    [KernelFunction]
    public ApplianceModel? ChangeStateAsync(int id, bool isOn) { ... }
}
```

すべての家電の情報を取得

idで指定された家電をオンまたはオフに設定

## ■ すべての家電の情報を取得する関数

```
[KernelFunction("get_appliances")]
[Description("Gets a list of appliances and their current state")]
public List<ApplianceModel> GetAppliances()
{
    return appliances;
}
```

■ idで指定された家電をオンまたはオフに設定する関数

```
[KernelFunction("change_state")]
[Description("Changes the state of the appliance with the given ID")]
public ApplianceModel? ChangeStateAsync(int id, bool isOn)
{
    var appliance = appliances.FirstOrDefault(appliance => appliance.Id == id);

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

    appliance.IsOn = isOn;

    return appliance;
}
```

## ■メインプログラム（Program.cs）

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

// 環境変数から設定を取得
var deployName = Environment.GetEnvironmentVariable("AOAI_DEPLOY_NAME") ?? "";
var endpoint = Environment.GetEnvironmentVariable("AOAI_ENDPOINT") ?? "";
var apiKey = Environment.GetEnvironmentVariable("AOAI_KEY") ?? "";

// カーネルをビルド
var builder = Kernel.CreateBuilder();
builder.AddAzureOpenAIChatCompletion(deployName, endpoint, apiKey);
builder.Plugins.AddFromType<AppliancesPlugin>();
Kernel kernel = builder.Build();

// (続く)
```

## ■メインプログラム (Program.cs)

```
// チャット補完サービスを取得
var chatCompletionService = kernel.GetRequiredService<IChatCompletionService>();

// 関数の自動呼び出しを可能にする
OpenAIPromptExecutionSettings settings = new()
{
    FunctionChoiceBehavior = FunctionChoiceBehavior.Auto()
};

// チャット履歴
var history = new ChatHistory("あなたは家電の管理者です。");

// (続く)
```

## ■メインプログラム (Program.cs)

```
// メインループ（チャット）
while (true)
{
    // ユーザーのプロンプトを取得
    Console.Write("あなた: ");
    string userInput = Console.ReadLine() ?? "";

    // チャット履歴にユーザーのプロンプトを追加
    history.AddUserMessage(userInput);

    // エージェントの応答を表示
    Console.Write("エージェント: ");

    // プロンプトを送信してコンテンツを生成（プロンプトに従って家電を制御）
    var result = await chatCompletionService.GetChatMessageContentAsync(history, settings, kernel);

    // エージェントが生成したコンテンツを表示
    Console.WriteLine($"エージェント: {result}");

    // チャット履歴にエージェントのコンテンツを追加
    history.AddAssistantMessage($"{result}");

}
```

## ■ 「家電エージェント」の実行例

あなた： こんにちは

エージェント： こんにちは！ 家電の管理が必要であれば、何でもお手伝いしますよ。どのようなサポートが必要ですか？

あなた： 現在のすべての家電の状態を教えて

エージェント： 現在の家電の状態は以下の通りです：

1. パソコン： オフ
2. テレビ： オフ
3. エアコン： オフ

あなた： エアコンの電源をオンにして

エージェント： エアコンの電源をオンにしました。

あなた： 現在オンになっている家電はどれ？

エージェント： 現在オンになっている家電は以下の通りです：

- エアコン



あなた： テレビの電源をつけて

エージェント： 現在オンになっている家電は、エアコンとテレビです。

あなた： 現在の家電の状態は？

エージェント： 現在の家電の状態は以下の通りです：

1. パソコン： オフ
2. テレビ： オン
3. エアコン： オン

あなた： 全部オフにして

エージェント： すべての家電の電源をオフにしました。現在の状態は以下の通りです：

- パソコン： オフ
- テレビ： オフ
- エアコン： オフ

あなた：



# モジュール6 まとめ

- Semantic Kernelとgpt-4oモデルを使用して、ユーザーと自然言語（日本語）で対話ができるAIエージェントを比較的短いコードで作成できる
- 独自に開発したプラグインを組み合わせることで、AIエージェントに、家電の制御などの仕事を代行してもらうことができる

# 全体のまとめ（本日の到達目標の確認）

- Azure OpenAI Serviceの基礎を理解する
  - リソースの作成と生成AIモデルのデプロイ
  - アプリからのデプロイの利用
- Semantic Kernelの基礎を理解する
  - プロンプトの実行
  - チャットの実装
  - 会話履歴の管理
  - プラグインの利用
  - 独自のプラグインの作成
  - プラグインの関数の呼び出し方法（明示的・自動的）の理解
- Semantic Kernel を使用するAIエージェントの構築方法を理解する
  - 家電をコントロールするAIエージェントの作成

# AZ-2005 全体のまとめ



Azure OpenAI と Semantic Kernel SDK を  
使用して AI エージェントを開発する

- ・モジュール1 カーネルの構築
  - ・モジュール2 Semantic Kernel プラグインの作成
  - ・モジュール3 AIエージェントにスキルを与える
  - ・モジュール4 プロンプトと関数を組み合わせる
  - ・モジュール5 関数の自動呼び出し
  - ・モジュール6 AIエージェントの作成例
- 
- The diagram illustrates the organization of the six modules into four main categories, each represented by a blue curly brace:
- 概要** (Overview): Groups the first module.
  - Plugin(関数)の作り方** (Plugin Function Creation Methods): Groups the second and third modules.
  - Plugin(関数)の呼び出し方** (Plugin Function Calling Methods): Groups the fourth and fifth modules.
  - PoC (概念実証)** (Concept Proof): Groups the sixth module.