Semantic Kernel C# Hands-On Lab Guide - Azure Government

> **Note**: This hands-on lab has been updated to use the latest Semantic Kernel C# requirements and API structures as of August 2025. The code examples reflect the current recommended practices and package versions.

This comprehensive lab guide will walk you through building a complete Semantic Kernel application in C#. You'll learn to create kernels, add services, implement plugins, configure vector stores, and add filters using Azure OpenAI in Azure Government Cloud.

Prerequisites

Development Environment

- Visual Studio Code
- .NET 8.0 SDK or later
- C# extension for VS Code

Required Azure OpenAI Resources

- Azure OpenAI resource deployed in Azure Government Cloud
- Azure OpenAI service endpoint (e.g., `https://your-resource.openai.usgovcloudapi.net/`)
- Azure OpenAI API key
- Deployed models: `gpt-35-turbo` and `text-embedding-ada-002`

Required NuGet Packages

> **Important**: This lab requires Semantic Kernel .NET 1.0+ packages. The package structure and API have been updated to reflect the latest recommended patterns from Microsoft's official documentation.

Before starting the C# lab, you'll need to install the following NuGet packages. Each package serves a specific purpose in the Semantic Kernel ecosystem:

Package Name	Version	Purpose
`Microsoft.SemanticKernel`	Latest	Core Semantic Kernel framework and orchestration engine
`Microsoft.SemanticKernel.Connectors.AzureOpenAI`	Latest	Azure OpenAI integration for chat completion and embeddings
`Microsoft.SemanticKernel.Plugins.OpenApi`	Latest	Support for OpenAPI/REST API plugins
`Microsoft.SemanticKernel.Connectors.Sqlite`	Latest	SQLite vector store provider for persistent storage

Package Name	Version	Purpose
`Microsoft.Extensions.Logging.Console`	Latest	Console logging support for debugging
`Azure.Identity`	Latest	Azure authentication and identity management

Installation Command:

```
# Run this single command to install all required packages
dotnet add package Microsoft.SemanticKernel && \
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI && \
dotnet add package Microsoft.SemanticKernel.Plugins.OpenApi && \
dotnet add package Microsoft.SemanticKernel.Connectors.Sqlite && \
dotnet add package Microsoft.Extensions.Logging.Console && \
dotnet add package Azure.Identity
```

Alternative Individual Installation: If you prefer to install packages one by one, use these commands in the VS Code terminal:

```
dotnet add package Microsoft.SemanticKernel
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI
dotnet add package Microsoft.SemanticKernel.Plugins.OpenApi
dotnet add package Microsoft.SemanticKernel.Connectors.Sqlite
dotnet add package Microsoft.Extensions.Logging.Console
dotnet add package Azure.Identity
```

Step 1: Project Setup

1. **Open VS Code** and create a new folder for your project:

   ```
   mkdir SemanticKernelLab
   cd SemanticKernelLab
   code .
   ```

2. **Open the integrated terminal** in VS Code (`Ctrl+`` or `View > Terminal`) and create a new console application:

   ```
   dotnet new console
   ```

3. **Install the required NuGet packages** using the installation command from the Required NuGet Packages section above, or run them individually:

```
dotnet add package Microsoft.SemanticKernel
dotnet add package Microsoft.SemanticKernel.Connectors.AzureOpenAI
dotnet add package Microsoft.SemanticKernel.Plugins.OpenApi
dotnet add package Microsoft.SemanticKernel.Plugins.Memory
dotnet add package Microsoft.SemanticKernel.Connectors.Memory.Sqlite
dotnet add package Microsoft.Extensions.Logging.Console
dotnet add package Azure.Identity
```

> **What you're doing:** These packages provide the core Semantic Kernel functionality, Azure OpenAI connectors, plugin support, memory capabilities, and logging infrastructure needed for the lab.

Step 2: Create the Kernel and Add Chat Completion Service

4. **Replace the contents of** `Program.cs` with the following code. This sets up the kernel with Azure OpenAI in Azure Government:

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;
using Microsoft.Extensions.Logging;

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

// Add Azure OpenAI chat completion service for Azure Government
builder.AddAzureOpenAIChatCompletion(
    deploymentName: "gpt-35-turbo", // Your deployed model name
    endpoint: "https://your-resource.openai.usgovcloudapi.net/", // Azure Gov
endpoint
    apiKey: "your-azure-openai-api-key"); // Replace with your actual API key

// Add console logging
builder.Services.AddLogging(c =>
c.AddConsole().SetMinimumLevel(LogLevel.Information));

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

Console.WriteLine("✅ Kernel created successfully!");
Console.WriteLine("✅ Azure OpenAI chat completion service added for Azure
Government!");

// Test basic chat completion
var chatCompletionService = kernel.GetRequiredService<IChatCompletionService>();
var response = await chatCompletionService.GetChatMessageContentAsync("Hello,
what's the capital of France?");
Console.WriteLine($"AI Response: {response.Content}");
```

> **What you're doing:** This code creates a Semantic Kernel instance and adds an Azure OpenAI chat completion service specifically configured for Azure Government Cloud. Note the `.usgovcloudapi.net` endpoint which is specific to Azure Government.

Important Configuration Notes:

- **Endpoint:** Azure Government uses `.usgovcloudapi.net` instead of `.openai.azure.com`
- **Deployment Name:** Use the exact name you gave your model deployment in Azure OpenAI Studio
- **API Key:** Get this from your Azure OpenAI resource in the Azure Government portal
- **Model:** Azure Government typically has `gpt-35-turbo` rather than `gpt-3.5-turbo`

5. **Run the application** to test the basic setup:

```
dotnet run
```

Step 3: Add Native Plugin

6. **Create a new file** called `MathPlugin.cs` in your project folder. Add the following native plugin code:

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

public class MathPlugin
{
    [KernelFunction]
    [Description("Adds two numbers together")]
    public double Add(
        [Description("The first number")] double number1,
        [Description("The second number")] double number2)
    {
        return number1 + number2;
    }

    [KernelFunction]
    [Description("Multiplies two numbers together")]
    public double Multiply(
        [Description("The first number")] double number1,
        [Description("The second number")] double number2)
    {
        return number1 * number2;
    }

    [KernelFunction]
    [Description("Calculates the square root of a number")]
    public double SquareRoot([Description("The number to calculate square root
for")] double number)
    {
        if (number < 0)
            throw new ArgumentException("Cannot calculate square root of negative
number");
```

```
        return Math.Sqrt(number);
    }
}
```

> **What you're doing:** This creates a native plugin with mathematical functions. Native plugins are C# classes with methods decorated with `[KernelFunction]` attributes that the AI can call.

Step 4: Add Prompt-Based Plugin

7. **Create a new folder** called `Plugins` in your project:

```
mkdir Plugins
mkdir Plugins/WritingPlugin
```

8. **Create a file** `Plugins/WritingPlugin/Summarize.txt` with this prompt template:

```
Summarize the following text in a concise way:

{{$input}}

Summary:
```

9. **Create a file** `Plugins/WritingPlugin/config.json` with plugin configuration:

```
{
  "schema": 1,
  "description": "Plugin for writing assistance",
  "functions": [
    {
      "name": "Summarize",
      "description": "Summarizes text content",
      "is_semantic": true
    }
  ]
}
```

> **What you're doing:** Prompt-based plugins use text templates with placeholders (like `{{$input}}`) that the AI fills in. These are useful for consistent AI behaviors.

Step 5: Add OpenAPI Plugin

10. **Update your** `Program.cs` to include all plugins and demonstrate their usage:

```
```

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;
using Microsoft.SemanticKernel.Plugins.OpenApi;
using Microsoft.Extensions.Logging;
using System.Text.Json;

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

// Add Azure OpenAI chat completion service for Azure Government
builder.AddAzureOpenAIChatCompletion(
    deploymentName: "gpt-35-turbo", // Your deployed model name
    endpoint: "https://your-resource.openai.usgovcloudapi.net/", // Azure Gov
endpoint
    apiKey: "your-azure-openai-api-key"); // Replace with your actual API key

// Add console logging
builder.Services.AddLogging(c =>
c.AddConsole().SetMinimumLevel(LogLevel.Information));

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

Console.WriteLine("✅ Kernel created successfully!");

// Add native plugin
kernel.Plugins.AddFromType<MathPlugin>("MathPlugin");
Console.WriteLine("✅ Native MathPlugin added!");

// Add prompt-based plugin
var pluginsDirectory = Path.Combine(Directory.GetCurrentDirectory(), "Plugins");
kernel.Plugins.AddFromPromptDirectory(pluginsDirectory);
Console.WriteLine("✅ Prompt-based WritingPlugin added!");

// Add OpenAPI plugin (using a public API - note: ensure API is accessible from
Azure Gov)
try
{
    await kernel.Plugins.AddFromOpenApiAsync(
        "WeatherPlugin",
        new Uri("https://api.weatherapi.com/v1/openapi.json"),
        new OpenApiFunctionExecutionParameters()
        {
            HttpClient = new HttpClient()
        });
    Console.WriteLine("✅ OpenAPI WeatherPlugin added!");
}
catch (Exception ex)
{
    Console.WriteLine($"⚠️ OpenAPI plugin failed to load: {ex.Message}");
    Console.WriteLine("Continuing without OpenAPI plugin...");
}
```

```
// Test native plugin
Console.WriteLine("\n🧮 Testing Native Plugin:");
var mathResult = await kernel.InvokeAsync("MathPlugin", "Add", new() { ["number1"]
= "5", ["number2"] = "3" });
Console.WriteLine($"5 + 3 = {mathResult}");

// Test prompt-based plugin
Console.WriteLine("\n📝 Testing Prompt-based Plugin:");
var longText = "Artificial Intelligence (AI) is a branch of computer science that
aims to create intelligent machines that can perform tasks that typically require
human intelligence. These tasks include learning, reasoning, problem-solving,
perception, and language understanding. AI has applications in many fields
including healthcare, finance, transportation, and entertainment.";
var summaryResult = await kernel.InvokeAsync("WritingPlugin", "Summarize", new() {
["input"] = longText });
Console.WriteLine($"Summary: {summaryResult}");

// Test function calling with chat completion
Console.WriteLine("\n💬 Testing Function Calling:");
var chatHistory = new ChatHistory();
chatHistory.AddUserMessage("Can you calculate the square root of 16 and then
multiply it by 3?");

var chatFunction = kernel.GetRequiredService<IChatCompletionService>();
var executionSettings = new AzureOpenAIPromptExecutionSettings()
{
    ToolCallBehavior = ToolCallBehavior.AutoInvokeKernelFunctions
};

var response = await chatFunction.GetChatMessageContentAsync(chatHistory,
executionSettings, kernel);
Console.WriteLine($"AI Response: {response.Content}");
```

> **What you're doing:** This code demonstrates loading all three types of plugins and using them with Azure OpenAI in Azure Government. Note the use of `AzureOpenAIPromptExecutionSettings` instead of the OpenAI equivalent.

Step 6: Add Vector Store (Updated from Legacy Memory)

11. **Create a new file** called `VectorStoreService.cs`:

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.Connectors.Sqlite;
using Microsoft.SemanticKernel.Data;
using Microsoft.SemanticKernel.Embeddings;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;

public class VectorStoreService
{
    private readonly IVectorStore _vectorStore;
```

```
    private readonly ITextEmbeddingGenerationService _embeddingService;
    private readonly IVectorStoreRecordCollection<string, DataModel> _collection;
    private const string CollectionName = "GeneralKnowledge";

    public VectorStoreService(string endpoint, string apiKey, string
embeddingDeploymentName)
    {
        // Create vector store with SQLite storage
        _vectorStore = new SqliteVectorStore("vectors.db");

        // Create Azure OpenAI embedding service
        _embeddingService = new AzureOpenAITextEmbeddingGenerationService(
            deploymentName: embeddingDeploymentName,
            endpoint: endpoint,
            apiKey: apiKey);

        // Get or create collection
        _collection = _vectorStore.GetCollection<string, DataModel>
(CollectionName);
    }

    public async Task SaveInformationAsync(string id, string text, string?
description = null)
    {
        // Generate embedding for the text
        var embedding = await _embeddingService.GenerateEmbeddingAsync(text);

        var record = new DataModel
        {
            Id = id,
            Text = text,
            Description = description ?? "",
            Embedding = embedding
        };

        await _collection.UpsertAsync(record);
        Console.WriteLine($"💾 Saved to vector store: {id}");
    }

    public async Task<string> SearchInformationAsync(string query, double minScore
= 0.7)
    {
        // Generate embedding for the query
        var queryEmbedding = await
_embeddingService.GenerateEmbeddingAsync(query);

        // Search for similar vectors
        var searchOptions = new VectorSearchOptions
        {
            Top = 3,
            VectorPropertyName = nameof(DataModel.Embedding)
        };

        var results = await _collection.VectorizedSearchAsync(queryEmbedding,
```

```
searchOptions);

        var matches = new List<string>();
        await foreach (var result in results)
        {
            if (result.Score >= minScore)
            {
                matches.Add($"[Score: {result.Score:F2}] {result.Record.Text}");
            }
        }

        return matches.Count > 0 ? string.Join("\n", matches) : "No relevant
information found in vector store.";
    }
}

public class DataModel
{
    [VectorStoreRecordKey]
    public string Id { get; set; } = string.Empty;

    [VectorStoreRecordData]
    public string Text { get; set; } = string.Empty;

    [VectorStoreRecordData]
    public string Description { get; set; } = string.Empty;

    [VectorStoreRecordVector(1536)] // Ada-002 embedding dimension
    public ReadOnlyMemory<float> Embedding { get; set; }
}
```

> **What you're doing:** This creates a vector store service using the modern Vector Store approach instead of the legacy memory system. Vector stores provide better performance and more flexible data management with Azure OpenAI embeddings.

Step 7: Add Filters

12. **Create a new file** called `LoggingFilter.cs`:

```
using Microsoft.SemanticKernel;

public class LoggingFilter : IFunctionInvocationFilter
{
    public async Task OnFunctionInvocationAsync(FunctionInvocationContext context,
Func<FunctionInvocationContext, Task> next)
    {
        Console.WriteLine($"🔍 Filter: Executing function
'{context.Function.Name}'");
        Console.WriteLine($"📥 Input: {string.Join(", ",
context.Arguments.Select(arg => $"{arg.Key}={arg.Value}"))}");
```

```
        var startTime = DateTime.UtcNow;

        try
        {
            await next(context);
            var duration = DateTime.UtcNow - startTime;
            Console.WriteLine($"✅ Filter: Function '{context.Function.Name}'
completed in {duration.TotalMilliseconds}ms");
            Console.WriteLine($"📤 Output: {context.Result}");
        }
        catch (Exception ex)
        {
            var duration = DateTime.UtcNow - startTime;
            Console.WriteLine($"❌ Filter: Function '{context.Function.Name}'
failed after {duration.TotalMilliseconds}ms: {ex.Message}");
            throw;
        }
    }
}

public class SecurityFilter : IFunctionInvocationFilter
{
    private readonly HashSet<string> _blockedWords =
new(StringComparer.OrdinalIgnoreCase)
    {
        "password", "secret", "apikey", "token"
    };

    public async Task OnFunctionInvocationAsync(FunctionInvocationContext context,
Func<FunctionInvocationContext, Task> next)
    {
        // Check arguments for sensitive information
        foreach (var arg in context.Arguments)
        {
            if (_blockedWords.Any(blocked =>
arg.Value?.ToString()?.Contains(blocked, StringComparison.OrdinalIgnoreCase) ==
true))
            {
                Console.WriteLine($"🛡️ Security Filter: Blocked potentially
sensitive input in argument '{arg.Key}'");
                throw new UnauthorizedAccessException($"Argument '{arg.Key}'
contains potentially sensitive information");
            }
        }

        await next(context);
    }
}
```

> **What you're doing:** Filters intercept function calls before and after execution. The logging filter tracks performance and inputs/outputs, while the security filter prevents sensitive data from being processed.

13. **Update** `Program.cs` to include vector store and filters:

```
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.AzureOpenAI;
using Microsoft.Extensions.Logging;

// Configuration for Azure Government
const string azureEndpoint = "https://your-resource.openai.usgovcloudapi.net/";
const string apiKey = "your-azure-openai-api-key"; // Replace with your actual API
key
const string chatDeploymentName = "gpt-35-turbo"; // Your chat model deployment
name
const string embeddingDeploymentName = "text-embedding-ada-002"; // Your embedding
model deployment name

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

// Add Azure OpenAI chat completion service for Azure Government
builder.AddAzureOpenAIChatCompletion(
    deploymentName: chatDeploymentName,
    endpoint: azureEndpoint,
    apiKey: apiKey);

// Add console logging
builder.Services.AddLogging(c =>
c.AddConsole().SetMinimumLevel(LogLevel.Information));

// Add filters
builder.Services.AddSingleton<IFunctionInvocationFilter, LoggingFilter>();
builder.Services.AddSingleton<IFunctionInvocationFilter, SecurityFilter>();

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

Console.WriteLine("☑ Kernel created with Azure OpenAI for Azure Government!");
Console.WriteLine("☑ Filters enabled!");

// Add plugins
kernel.Plugins.AddFromType<MathPlugin>("MathPlugin");
var pluginsDirectory = Path.Combine(Directory.GetCurrentDirectory(), "Plugins");
kernel.Plugins.AddFromPromptDirectory(pluginsDirectory);

// Initialize vector store service with Azure OpenAI embeddings
var vectorStoreService = new VectorStoreService(azureEndpoint, apiKey,
embeddingDeploymentName);

// Save some information to vector store
await vectorStoreService.SaveInformationAsync("fact1", "The capital of France is
Paris");
await vectorStoreService.SaveInformationAsync("fact2", "Python is a programming
language created by Guido van Rossum");
```

```
await vectorStoreService.SaveInformationAsync("fact3", "The Semantic Kernel is a
Microsoft framework for AI orchestration");
await vectorStoreService.SaveInformationAsync("fact4", "Azure Government provides
cloud services for US government agencies");

Console.WriteLine("\n🔍 Testing Vector Store with Azure OpenAI Embeddings:");
var vectorSearchResult = await vectorStoreService.SearchInformationAsync("What is
the capital of France?");
Console.WriteLine($"Vector search result:\n{vectorSearchResult}");

Console.WriteLine("\n🧮 Testing Filters with Math Plugin:");
try
{
    var result = await kernel.InvokeAsync("MathPlugin", "Add", new() { ["number1"]
= "10", ["number2"] = "5" });
    Console.WriteLine($"Math result: {result}");
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

Console.WriteLine("\n🛡️ Testing Security Filter:");
try
{
    // This should trigger the security filter
    await kernel.InvokeAsync("MathPlugin", "Add", new() { ["number1"] =
"mypassword123", ["number2"] = "5" });
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine($"Security filter activated: {ex.Message}");
}

Console.WriteLine("\n🎉 C# Lab with Azure Government completed successfully!");
```

> **What you're doing:** This final version demonstrates the complete integration with Azure OpenAI in Azure Government, including vector store services using Azure OpenAI embeddings and all filtering capabilities.

Running the C# Lab

1. **Navigate to your C# project folder** in VS Code
2. **Ensure your Azure OpenAI credentials are set** in the code:
   - Replace `"https://your-resource.openai.usgovcloudapi.net/"` with your actual Azure Government endpoint
   - Replace `"your-azure-openai-api-key"` with your actual API key
   - Verify your deployment names match (`gpt-35-turbo`, `text-embedding-ada-002`)
3. **Run the application:**

```
dotnet run
```

4. **Expected output:** You should see successful creation of kernel, plugins, memory operations with Azure OpenAI embeddings, and filter demonstrations

Key Learning Points

What You've Built

Kernel Creation:

- The kernel is the central orchestration engine that manages AI services, plugins, and execution
- Configured specifically for Azure Government Cloud compliance requirements

Chat Completion Service:

- Integrated Azure OpenAI's GPT models deployed in Azure Government for natural language processing
- Configured for function calling to enable AI to use your plugins automatically
- Uses Azure Government specific endpoints (`.usgovcloudapi.net`) for compliance requirements

Plugin Types:

- **Native Plugins:** Code-based functions (C# methods) that AI can call
- **Prompt-based Plugins:** Template-driven AI behaviors using structured prompts
- **OpenAPI Plugins:** Integration with external REST APIs (weather API example)

Vector Store:

- Semantic search using Azure OpenAI vector embeddings deployed in Azure Government
- Modern Vector Store approach (replaces legacy Memory Store)
- Persistent storage with improved performance and flexibility
- Enables AI to remember and retrieve relevant context while maintaining government compliance requirements

Filters:

- Pre/post-processing of function calls
- Logging, security, performance monitoring
- Extensible pipeline for custom behaviors

Development Workflow in VS Code

File Organization:

```
C# Project:
├── Program.cs (main application with Azure Government config)
├── MathPlugin.cs (native plugin)
├── VectorStoreService.cs (vector store operations with Azure OpenAI embeddings)
├── LoggingFilter.cs (filters)
└── Plugins/WritingPlugin/ (prompt-based plugin)
```

Key VS Code Features Used:

- Integrated terminal for package management and running applications
- IntelliSense for code completion and error detection
- File explorer for organizing plugin structures
- Extensions for C# development support

Package Version Compatibility

Important Notes:

- **Semantic Kernel Versioning:** Uses Semantic Kernel .NET 1.0+ with current stable APIs and improved Azure Government support.
- **Vector Store Migration:** Legacy Memory Store packages have been replaced by modern Vector Store connectors for better performance.
- **Azure Package Dependencies:** The Azure OpenAI connectors automatically handle compatible versions of Azure SDKs.
- **.NET Version:** Requires .NET 8.0 or later for optimal Semantic Kernel support.

Verifying Installation: After installing packages, verify they're working correctly:

C# Verification:

```
dotnet list package
```

Key Updates in This Version

- **Updated to Semantic Kernel .NET 1.0+**: Leveraging the latest stable APIs and improvements for Azure Government
- **.NET 8.0+ requirement**: Ensuring compatibility with modern .NET features and performance optimizations
- **Vector Store approach**: Replaced legacy Memory Store with modern Vector Store connectors for improved performance
- **Simplified package structure**: Updated package dependencies to reflect current Semantic Kernel organization
- **Current API patterns**: Updated service registration and kernel building to follow latest recommended practices

Troubleshooting Tips

Common Issues:

1. **Azure OpenAI Credential Errors:** Ensure your Azure OpenAI resource is properly deployed in Azure Government and API key is valid
2. **Endpoint Configuration:** Verify you're using the correct Azure Government endpoint (`.usgovcloudapi.net`)

3. **Model Deployment Names:** Check that your deployment names exactly match those in Azure OpenAI Studio
4. **Package Versions:** Semantic Kernel is rapidly evolving; ensure Azure connector packages are compatible
5. **Path Issues:** Verify plugin directory structures match exactly as shown
6. **Async/Await:** Both implementations use asynchronous patterns; ensure proper async handling
7. **Government Cloud Access:** Ensure your subscription has access to Azure Government and required services are enabled

Azure Government Specific Considerations:

- **Compliance:** Azure Government provides FedRAMP High and DoD IL2-IL5 compliance
- **Data Residency:** All data processing occurs within US government data centers
- **Network Isolation:** Government cloud services are isolated from commercial Azure
- **Model Availability:** Check Azure Government documentation for available AI models and regions

Best Practices:

- Use environment variables for Azure OpenAI credentials (never hardcode secrets)
- Implement proper error handling for AI service calls
- Structure plugins in separate files/modules for maintainability
- Test each component individually before integration
- Use logging to debug function call flows
- Follow Azure Government security and compliance guidelines
- Regularly update Semantic Kernel packages for latest Azure Government compatibility
- Monitor usage and costs through Azure Government portal

Troubleshooting Package Issues:

- **C#:** If package conflicts occur, try `dotnet clean` and `dotnet restore`
- **Version Conflicts:** Check the Semantic Kernel GitHub releases page for compatible package versions

Additional Azure Government Resources:

- Azure Government Documentation
- Azure OpenAI in Government
- Government Cloud Compliance

This lab provides a comprehensive foundation for building AI-powered applications with Semantic Kernel using Azure OpenAI in Azure Government, demonstrating core concepts that scale to production scenarios while maintaining government compliance requirements. The Azure Government configuration ensures data sovereignty and regulatory compliance for government and defense applications.