# Advanced Topic 1: Generative AI and AI Agents with Azure AI and Semantic Kernel

## 1. Concept Explanation

Generative AI has revolutionized the way we interact with technology. At its core, it’s about creating new, original content—be it text, images, or code—that is coherent and contextually relevant. When combined with the concept of AI agents, we move beyond simple content generation to creating autonomous systems that can perform complex tasks, reason about problems, and interact with their environment. Azure AI, particularly through the Azure AI Foundry, provides a comprehensive platform for building these sophisticated AI solutions. A key component in this ecosystem is the Semantic Kernel, an open-source SDK that acts as an orchestration layer, allowing developers to integrate large language models (LLMs) with conventional programming languages like C# and Python.

**Key Components:**

* **Azure AI Foundry:** A unified platform for building, managing, and deploying AI solutions. It provides access to a wide range of models from the model catalog, tools for prompt engineering (Prompt Flow), and capabilities for fine-tuning and evaluating models.
* **Semantic Kernel:** An open-source SDK that simplifies the integration of LLMs into applications. It provides a framework for creating and chaining AI skills, enabling developers to build intelligent agents that can perform tasks like summarization, question answering, and code generation. It supports various LLMs, including those from Azure OpenAI Service.
* **Retrieval Augmented Generation (RAG):** A technique that enhances the capabilities of LLMs by grounding their responses in external, authoritative data sources. Instead of relying solely on the knowledge embedded during training, RAG systems retrieve relevant information from a knowledge base and use it to augment the prompt given to the LLM, leading to more accurate, relevant, and up-to-date responses. This is particularly useful for enterprise applications where LLMs need to access proprietary or real-time data.
* **Fine-tuning Language Models:** The process of further training a pre-trained language model on a smaller, task-specific dataset. This allows the model to adapt to specific domains, styles, or tasks, improving its performance for particular use cases without having to train a model from scratch.
* **Multi-Agent Solutions:** Architectures where multiple AI agents collaborate to achieve a common goal. Each agent might specialize in a particular task or domain, and they communicate and coordinate their actions to solve complex problems that a single agent might struggle with. This approach can lead to more robust and intelligent systems.

## 2. Relevant Use Cases and Implementation Scenarios

Generative AI and AI agents, especially when powered by Azure AI and Semantic Kernel, open up a vast array of innovative use cases across various industries:

* **Customer Service and Support:** Intelligent chatbots and virtual assistants that can understand complex queries, retrieve information from internal knowledge bases (RAG), and provide personalized responses. Multi-agent systems can route complex issues to specialized agents or human operators, ensuring efficient resolution.
* **Content Creation and Curation:** Automated generation of marketing copy, product descriptions, news articles, or even code snippets. Agents can be designed to curate content based on user preferences or trending topics, summarizing information from multiple sources.
* **Healthcare:** AI agents can assist medical professionals by summarizing patient records, answering clinical questions based on medical literature (RAG), and even generating initial drafts of medical reports. Multi-agent systems could collaborate on diagnosing rare diseases by combining expertise from different medical domains.
* **Financial Services:** Fraud detection systems that analyze transaction patterns and generate alerts. Financial advisors powered by AI agents can provide personalized investment advice, analyze market trends, and generate financial reports.
* **Education:** Personalized learning experiences where AI agents adapt to student progress, generate practice questions, and provide tailored feedback. Agents can also assist educators in creating course materials and grading assignments.
* **Software Development:** Code generation, debugging assistance, and automated testing. AI agents can act as coding copilots, suggesting code completions, refactoring code, and even generating entire functions based on natural language descriptions. Multi-agent systems could manage entire development workflows, from requirements gathering to deployment.

## 3. Key Considerations and Best Practices

Implementing Generative AI and AI agents effectively requires careful consideration of several factors:

* **Data Quality and Governance:** The performance of RAG systems heavily depends on the quality and relevance of the external data. Establishing robust data governance policies, ensuring data accuracy, and maintaining up-to-date knowledge bases are crucial.
* **Prompt Engineering:** Crafting effective prompts is an art and a science. Trainers should emphasize techniques for clear, concise, and unambiguous prompt design to elicit desired responses from LLMs. This includes few-shot learning, chain-of-thought prompting, and role-playing.
* **Model Selection and Fine-tuning:** Choosing the right base model for a specific task is important. Trainers should guide students on when to use pre-trained models, when fine-tuning is beneficial, and how to evaluate the trade-offs between model size, performance, and cost.
* **Responsible AI:** Generative AI models can produce biased, harmful, or inaccurate content. Trainers must stress the importance of implementing responsible AI principles, including fairness, accountability, transparency, and privacy. This involves content moderation, bias detection, and human oversight.
* **Scalability and Cost Management:** Deploying and managing LLMs and AI agents at scale can be resource-intensive. Best practices include optimizing model inference, leveraging Azure services for scalability (e.g., Azure Kubernetes Service for agent orchestration), and monitoring costs.
* **Security:** Protecting sensitive data and ensuring the secure operation of AI agents is paramount. This includes secure API access, data encryption, and adherence to compliance regulations.
* **Observability and Monitoring:** Implementing robust logging, monitoring, and evaluation mechanisms to track agent performance, identify issues, and continuously improve the system. This includes tracking metrics like response accuracy, latency, and user satisfaction.
* **Human-in-the-Loop:** For critical applications, maintaining a human-in-the-loop approach is essential. This allows for human review and intervention, especially when dealing with ambiguous or high-stakes scenarios, ensuring ethical and accurate outcomes.

## 4. Sample Implementations

To illustrate the practical application of Generative AI and AI Agents, we will provide sample implementations in both C# and Python. These examples will focus on a scenario inspired by real-world customer cases, specifically leveraging RAG for enhanced customer service in a complex domain. We will consider a scenario where a large manufacturing company, like Alstom (as seen in Microsoft customer stories), uses Azure AI to provide advanced technical support for their complex machinery. The AI agent will answer technical questions by retrieving information from a vast internal documentation repository.

### Customer Case Inspiration: Alstom

Alstom, a global leader in sustainable mobility, has integrated Generative AI with Azure OpenAI Service to accelerate its business processes. This case highlights the power of Azure AI in handling complex, domain-specific information, making it an ideal inspiration for demonstrating RAG capabilities. The challenge for Alstom, as for many large enterprises, is efficiently accessing and utilizing vast amounts of internal, often unstructured, data to support operations, maintenance, and customer inquiries. An AI agent capable of intelligently retrieving and synthesizing information from these sources can significantly improve efficiency and response times.

### Scenario: Intelligent Technical Support Agent for Manufacturing

Our sample implementation will simulate an intelligent technical support agent for a manufacturing company. This agent will be able to answer questions about specific machinery components, troubleshooting steps, and maintenance procedures by querying a knowledge base of technical manuals and documentation. The core of this solution will be a RAG implementation, where user queries are used to retrieve relevant document chunks, which are then fed to an LLM to generate a concise and accurate answer.

**Components:**

* **Azure OpenAI Service:** To host the large language model (e.g., gpt-3.5-turbo or gpt-4).
* **Azure AI Search (formerly Azure Cognitive Search):** To index and search the technical documentation.
* **Azure Blob Storage:** To store the raw technical documentation files.
* **Semantic Kernel:** To orchestrate the interaction between the user, the search service, and the LLM.

**Workflow:**

1. User asks a technical question.
2. The Semantic Kernel uses the user’s question to query Azure AI Search.
3. Azure AI Search retrieves relevant document snippets from the indexed technical manuals.
4. The retrieved snippets, along with the original question, are used to construct a prompt for the LLM.
5. The LLM generates a comprehensive answer based on the provided context.
6. The answer is returned to the user.

This setup ensures that the LLM’s responses are grounded in the company’s specific, accurate technical documentation, preventing hallucinations and providing reliable information to support technicians or customers.

### C# Implementation Example

This C# example demonstrates how to build a RAG-based intelligent technical support agent using Azure OpenAI, Azure AI Search, and Semantic Kernel. It assumes you have your Azure resources (Azure OpenAI, Azure AI Search) already provisioned and configured.

First, ensure you have the necessary NuGet packages installed:

<ItemGroup>  
 <PackageReference Include="Microsoft.SemanticKernel" Version="1.0.0-beta10" />  
 <PackageReference Include="Azure.AI.OpenAI" Version="1.0.0-beta.10" />  
 <PackageReference Include="Azure.Search.Documents" Version="11.4.0" />  
</ItemGroup>

using Azure;  
using Azure.AI.OpenAI;  
using Azure.Search.Documents;  
using Azure.Search.Documents.Models;  
using Microsoft.SemanticKernel;  
using Microsoft.SemanticKernel.Connectors.OpenAI;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
public class TechnicalSupportAgent  
{  
 private readonly Kernel \_kernel;  
 private readonly SearchClient \_searchClient;  
 private readonly string \_searchIndexName;  
  
 public TechnicalSupportAgent(string openAiEndpoint, string openAiKey, string openAiDeploymentName, string searchServiceEndpoint, string searchApiKey, string searchIndexName)  
 {  
 // Initialize Semantic Kernel with Azure OpenAI  
 \_kernel = Kernel.CreateBuilder()  
 .AddAzureOpenAIChatCompletion(  
 deploymentName: openAiDeploymentName,  
 endpoint: openAiEndpoint,  
 apiKey: openAiKey)  
 .Build();  
  
 // Initialize Azure AI Search client  
 \_searchClient = new SearchClient(new Uri(searchServiceEndpoint), searchIndexName, new AzureKeyCredential(searchApiKey));  
 \_searchIndexName = searchIndexName;  
 }  
  
 public async Task<string> GetAnswerAsync(string query)  
 {  
 // Step 1: Retrieve relevant documents from Azure AI Search  
 Console.WriteLine($"Searching for relevant documents for query: {query}");  
 SearchResults<SearchDocument> searchResults = await \_searchClient.SearchAsync<SearchDocument>(query, new SearchOptions { Size = 3 });  
  
 StringBuilder contextBuilder = new StringBuilder();  
 foreach (SearchResult<SearchDocument> result in searchResults.GetResults())  
 {  
 // Assuming your search documents have a 'content' field  
 if (result.Document.TryGetValue("content", out object contentValue))  
 {  
 contextBuilder.AppendLine(contentValue.ToString());  
 }  
 }  
  
 string context = contextBuilder.ToString();  
  
 if (string.IsNullOrWhiteSpace(context))  
 {  
 return "I could not find relevant information in the technical documentation. Please rephrase your question or provide more details.";  
 }  
  
 // Step 2: Augment the prompt with retrieved context and send to LLM via Semantic Kernel  
 Console.WriteLine("Generating answer with LLM using retrieved context...");  
 var prompt = $"You are a highly knowledgeable technical support agent for a manufacturing company. Use the following technical documentation to answer the user's question. If the answer is not in the provided documentation, state that you don't know.\n\nTechnical Documentation:\n{context}\n\nUser Question: {query}\n\nAnswer:";  
  
 var chatCompletionService = \_kernel.GetRequiredService<IChatCompletionService>();  
 var result = await chatCompletionService.GetChatMessageContentAsync(prompt);  
  
 return result.Content;  
 }  
  
 public static async Task Main(string[] args)  
 {  
 // Replace with your actual Azure OpenAI credentials  
 string openAiEndpoint = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_ENDPOINT") ?? "YOUR\_AZURE\_OPENAI\_ENDPOINT";  
 string openAiKey = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_KEY") ?? "YOUR\_AZURE\_OPENAI\_KEY";  
 string openAiDeploymentName = Environment.GetEnvironmentVariable("AZURE\_OPENAI\_DEPLOYMENT\_NAME") ?? "YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME";  
  
 string searchServiceEndpoint = Environment.GetEnvironmentVariable("AZURE\_SEARCH\_SERVICE\_ENDPOINT") ?? "YOUR\_AZURE\_SEARCH\_SERVICE\_ENDPOINT";  
 string searchApiKey = Environment.GetEnvironmentVariable("AZURE\_SEARCH\_API\_KEY") ?? "YOUR\_AZURE\_SEARCH\_API\_KEY";  
 string searchIndexName = Environment.GetEnvironmentVariable("AZURE\_SEARCH\_INDEX\_NAME") ?? "YOUR\_AZURE\_SEARCH\_INDEX\_NAME";  
  
 if (new[] { openAiEndpoint, openAiKey, openAiDeploymentName, searchServiceEndpoint, searchApiKey, searchIndexName }.Any(string.IsNullOrWhiteSpace))  
 {  
 Console.WriteLine("Please set the environment variables for Azure OpenAI and Azure AI Search credentials.");  
 return;  
 }  
  
 TechnicalSupportAgent agent = new TechnicalSupportAgent(openAiEndpoint, openAiKey, openAiDeploymentName, searchServiceEndpoint, searchApiKey, searchIndexName);  
  
 Console.WriteLine("Technical Support Agent Ready. Ask a question (type 'exit' to quit):\n");  
  
 while (true)  
 {  
 Console.Write("You: ");  
 string userQuery = Console.ReadLine();  
  
 if (userQuery.ToLower() == "exit")  
 {  
 break;  
 }  
  
 string answer = await agent.GetAnswerAsync(userQuery);  
 Console.WriteLine($"Agent: {answer}\n");  
 }  
 }  
}

**Explanation:**

1. **Initialization:** The TechnicalSupportAgent class is initialized with Azure OpenAI and Azure AI Search credentials. It sets up the Kernel from Semantic Kernel and the SearchClient for Azure AI Search.
2. **GetAnswerAsync Method:**
   * **Azure AI Search:** It first performs a search query against the specified Azure AI Search index using the user’s question. It retrieves the top 3 relevant documents.
   * **Context Building:** The content from the retrieved documents is concatenated to form the context that will be provided to the LLM.
   * **Prompt Augmentation:** A detailed prompt is constructed. This prompt instructs the LLM to act as a technical support agent and explicitly tells it to use the provided Technical Documentation (the retrieved context) to answer the User Question. It also includes a safeguard to state if information is not found.
   * **LLM Interaction:** The augmented prompt is sent to the Azure OpenAI chat completion service via the Semantic Kernel. The LLM then generates an answer based on the provided context and the user’s query.
3. **Main Method:** This is the entry point for the console application. It retrieves credentials from environment variables (best practice for security) and then enters a loop to continuously take user input and provide answers from the agent.

**Before running this code:**

* **Azure Resources:** Ensure you have an Azure OpenAI service and an Azure AI Search service provisioned. Deploy a chat model (e.g., gpt-3.5-turbo or gpt-4) in your Azure OpenAI service.
* **Azure AI Search Index:** You need an Azure AI Search index populated with your technical documentation. Each document in the index should have a field (e.g., content) containing the text of the technical manual or relevant snippets.
* **Environment Variables:** Set the following environment variables with your actual credentials:
  + AZURE\_OPENAI\_ENDPOINT
  + AZURE\_OPENAI\_KEY
  + AZURE\_OPENAI\_DEPLOYMENT\_NAME
  + AZURE\_SEARCH\_SERVICE\_ENDPOINT
  + AZURE\_SEARCH\_API\_KEY
  + AZURE\_SEARCH\_INDEX\_NAME

This C# example provides a solid foundation for building sophisticated RAG-based AI agents that can leverage your proprietary data for accurate and context-aware responses.

### How to Test the C# Implementation Example

This section provides step-by-step instructions on how to set up, run, and verify the C# code example for the RAG-based Technical Support Agent.

#### Prerequisites

* .NET SDK (version 6.0 or higher) installed.
* An Azure OpenAI Service instance with a chat model deployed (e.g., gpt-3.5-turbo or gpt-4). You will need the endpoint, API key, and the deployment name of your chat model.
* An Azure AI Search service with an index populated with your technical documentation. You will need the search service endpoint, API key, and the index name. Ensure your documents have a content field containing the text to be searched.

#### Setup

1. **Create a new C# Console Project:**

* dotnet new console -n TechnicalSupportRAGApp  
  cd TechnicalSupportRAGApp

1. **Add NuGet Packages:** Open the TechnicalSupportRAGApp.csproj file and add the following ItemGroup:

* <ItemGroup>  
   <PackageReference Include="Microsoft.SemanticKernel" Version="1.0.0-beta10" />  
   <PackageReference Include="Azure.AI.OpenAI" Version="1.0.0-beta.10" />  
   <PackageReference Include="Azure.Search.Documents" Version="11.4.0" />  
  </ItemGroup>
* Then, restore the packages:
* dotnet restore

1. **Create Program.cs:** Replace the content of Program.cs with the C# code provided above in the “C# Implementation Example” section. Ensure the using statements and class definitions are correct.

#### Configuration

Set the following environment variables with your Azure OpenAI and Azure AI Search credentials. Replace the placeholder values with your actual information.

* AZURE\_OPENAI\_ENDPOINT="YOUR\_AZURE\_OPENAI\_ENDPOINT"
* AZURE\_OPENAI\_KEY="YOUR\_AZURE\_OPENAI\_KEY"
* AZURE\_OPENAI\_DEPLOYMENT\_NAME="YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME"
* AZURE\_SEARCH\_SERVICE\_ENDPOINT="YOUR\_AZURE\_SEARCH\_SERVICE\_ENDPOINT"
* AZURE\_SEARCH\_API\_KEY="YOUR\_AZURE\_SEARCH\_API\_KEY"
* AZURE\_SEARCH\_INDEX\_NAME="YOUR\_AZURE\_SEARCH\_INDEX\_NAME"

**Example (for PowerShell on Windows):**

$env:AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
$env:AZURE\_OPENAI\_KEY="your-openai-api-key"  
$env:AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
$env:AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
$env:AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
$env:AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

**Example (for Bash/Zsh on Linux/macOS):**

export AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
export AZURE\_OPENAI\_KEY="your-openai-api-key"  
export AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
export AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
export AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
export AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

#### Execution

Run the application from the TechnicalSupportRAGApp directory:

dotnet run

#### Expected Results

The application will start an interactive chat session with the Technical Support Agent. You can ask questions related to the content of your Azure AI Search index. The agent will retrieve relevant information and use it to answer your questions.

**Example Interaction (assuming your index contains information about a specific machinery component):**

Technical Support Agent Ready. Ask a question (type 'exit' to quit):  
  
You: How do I troubleshoot the XYZ-123 motor?  
Searching for relevant documents for query: How do I troubleshoot the XYZ-123 motor?  
Generating answer with LLM using retrieved context...  
Agent: To troubleshoot the XYZ-123 motor, first check the power supply connections. Ensure they are securely fastened and that the motor is receiving the correct voltage as specified in the technical manual section 3.2. If the power supply is adequate, inspect the motor for any visible damage or obstructions. Refer to section 4.1 for common fault codes and their resolutions.  
  
You: What is the maximum operating temperature for the ABC-456 sensor?  
Searching for relevant documents for query: What is the maximum operating temperature for the ABC-456 sensor?  
Generating answer with LLM using retrieved context...  
Agent: The maximum operating temperature for the ABC-456 sensor is 85 degrees Celsius. This information can be found in the sensor's specifications sheet, section 2.1.  
  
You: exit

### Python Implementation Example

This Python example mirrors the C# implementation, demonstrating a RAG-based intelligent technical support agent using Azure OpenAI, Azure AI Search, and Semantic Kernel. Ensure you have the necessary Python packages installed:

pip install semantic-kernel azure-search-documents openai

import os  
import asyncio  
from azure.core.credentials import AzureKeyCredential  
from azure.search.documents.aio import SearchClient  
from semantic\_kernel import Kernel  
from semantic\_kernel.connectors.ai.open\_ai import AzureChatCompletion  
  
class TechnicalSupportAgent:  
 def \_\_init\_\_(self):  
 # Initialize Semantic Kernel with Azure OpenAI  
 self.kernel = Kernel()  
 self.kernel.add\_service(  
 AzureChatCompletion(  
 service\_id="azure\_openai\_chat\_completion",  
 deployment\_name=os.environ.get("AZURE\_OPENAI\_DEPLOYMENT\_NAME"),  
 endpoint=os.environ.get("AZURE\_OPENAI\_ENDPOINT"),  
 api\_key=os.environ.get("AZURE\_OPENAI\_KEY"),  
 )  
 )  
  
 # Initialize Azure AI Search client  
 self.search\_client = SearchClient(  
 endpoint=os.environ.get("AZURE\_SEARCH\_SERVICE\_ENDPOINT"),  
 index\_name=os.environ.get("AZURE\_SEARCH\_INDEX\_NAME"),  
 credential=AzureKeyCredential(os.environ.get("AZURE\_SEARCH\_API\_KEY"))  
 )  
  
 async def get\_answer\_async(self, query: str) -> str:  
 # Step 1: Retrieve relevant documents from Azure AI Search  
 print(f"Searching for relevant documents for query: {query}")  
 search\_results = await self.search\_client.search(search\_text=query, top=3)  
  
 context\_parts = []  
 async for result in search\_results:  
 if "content" in result:  
 context\_parts.append(result["content"])  
  
 context = "\n".join(context\_parts)  
  
 if not context.strip():  
 return "I could not find relevant information in the technical documentation. Please rephrase your question or provide more details."  
  
 # Step 2: Augment the prompt with retrieved context and send to LLM via Semantic Kernel  
 print("Generating answer with LLM using retrieved context...")  
 prompt = (  
 "You are a highly knowledgeable technical support agent for a manufacturing company. "  
 "Use the following technical documentation to answer the user's question. "  
 "If the answer is not in the provided documentation, state that you cannot find the information.\n\n"  
 f"Technical Documentation:\n{context}\n\n"  
 f"User Question: {query}\n\n"  
 "Answer:"  
 )  
  
 chat\_completion\_service = self.kernel.get\_service("azure\_openai\_chat\_completion")  
 result = await chat\_completion\_service.get\_chat\_message\_content(prompt)  
  
 return str(result)  
  
async def main():  
 # Replace with your actual Azure OpenAI and Azure AI Search credentials  
 # It's recommended to set these as environment variables  
 required\_env\_vars = [  
 "AZURE\_OPENAI\_ENDPOINT",  
 "AZURE\_OPENAI\_KEY",  
 "AZURE\_OPENAI\_DEPLOYMENT\_NAME",  
 "AZURE\_SEARCH\_SERVICE\_ENDPOINT",  
 "AZURE\_SEARCH\_API\_KEY",  
 "AZURE\_SEARCH\_INDEX\_NAME",  
 ]  
  
 for var in required\_env\_vars:  
 if not os.environ.get(var):  
 print(f"Please set the environment variable: {var}")  
 return  
  
 agent = TechnicalSupportAgent()  
  
 print("Technical Support Agent Ready. Ask a question (type 'exit' to quit):\n")  
  
 while True:  
 user\_query = input("You: ")  
  
 if user\_query.lower() == "exit":  
 break  
  
 answer = await agent.get\_answer\_async(user\_query)  
 print(f"Agent: {answer}\n")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 asyncio.run(main())

**Explanation:**

1. **Initialization:** The TechnicalSupportAgent class initializes the Kernel from Semantic Kernel and the SearchClient for Azure AI Search, similar to the C# example. It retrieves credentials from environment variables.
2. **get\_answer\_async Method:**
   * **Azure AI Search:** It performs an asynchronous search query against the Azure AI Search index using the user’s question and retrieves the top 3 relevant documents.
   * **Context Building:** The content from the retrieved documents is joined to form the context for the LLM.
   * **Prompt Augmentation:** A detailed prompt is constructed, instructing the LLM to act as a technical support agent and to use the provided Technical Documentation (the retrieved context) to answer the User Question. A safeguard is included for cases where information is not found.
   * **LLM Interaction:** The augmented prompt is sent to the Azure OpenAI chat completion service via the Semantic Kernel. The LLM then generates an answer based on the provided context and the user’s query.
3. **main Function:** This is the asynchronous entry point for the script. It checks for the presence of required environment variables and then enters an infinite loop to interact with the user, taking questions and providing answers from the agent.

**Before running this code:**

* **Azure Resources:** Ensure you have an Azure OpenAI service and an Azure AI Search service provisioned. Deploy a chat model (e.g., gpt-3.5-turbo or gpt-4) in your Azure OpenAI service.
* **Azure AI Search Index:** You need an Azure AI Search index populated with your technical documentation. Each document in the index should have a field (e.g., content) containing the text of the technical manual or relevant snippets.
* **Environment Variables:** Set the following environment variables with your actual credentials:
  + AZURE\_OPENAI\_ENDPOINT
  + AZURE\_OPENAI\_KEY
  + AZURE\_OPENAI\_DEPLOYMENT\_NAME
  + AZURE\_SEARCH\_SERVICE\_ENDPOINT
  + AZURE\_SEARCH\_API\_KEY
  + AZURE\_SEARCH\_INDEX\_NAME

This Python example provides a robust and sophisticated implementation of a RAG-based AI agent, showcasing how to integrate Azure AI services with Semantic Kernel for real-world applications.

### How to Test the C# Implementation Example

This section provides step-by-step instructions on how to set up, run, and verify the C# code example for the RAG-based Technical Support Agent.

#### Prerequisites

* .NET SDK (version 6.0 or higher) installed.
* An Azure OpenAI Service instance with a chat model deployed (e.g., gpt-3.5-turbo or gpt-4). You will need the endpoint, API key, and the deployment name of your chat model.
* An Azure AI Search service with an index populated with your technical documentation. You will need the search service endpoint, API key, and the index name. Ensure your documents have a content field containing the text to be searched.

#### Setup

1. **Create a new C# Console Project:**

* dotnet new console -n TechnicalSupportRAGApp  
  cd TechnicalSupportRAGApp

1. **Add NuGet Packages:** Open the TechnicalSupportRAGApp.csproj file and add the following ItemGroup:

* <ItemGroup>  
   <PackageReference Include="Microsoft.SemanticKernel" Version="1.0.0-beta10" />  
   <PackageReference Include="Azure.AI.OpenAI" Version="1.0.0-beta.10" />  
   <PackageReference Include="Azure.Search.Documents" Version="11.4.0" />  
  </ItemGroup>
* Then, restore the packages:
* dotnet restore

1. **Create Program.cs:** Replace the content of Program.cs with the C# code provided above in the “C# Implementation Example” section. Ensure the using statements and class definitions are correct.

#### Configuration

Set the following environment variables with your Azure OpenAI and Azure AI Search credentials. Replace the placeholder values with your actual information.

* AZURE\_OPENAI\_ENDPOINT="YOUR\_AZURE\_OPENAI\_ENDPOINT"
* AZURE\_OPENAI\_KEY="YOUR\_AZURE\_OPENAI\_KEY"
* AZURE\_OPENAI\_DEPLOYMENT\_NAME="YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME"
* AZURE\_SEARCH\_SERVICE\_ENDPOINT="YOUR\_AZURE\_SEARCH\_SERVICE\_ENDPOINT"
* AZURE\_SEARCH\_API\_KEY="YOUR\_AZURE\_SEARCH\_API\_KEY"
* AZURE\_SEARCH\_INDEX\_NAME="YOUR\_AZURE\_SEARCH\_INDEX\_NAME"

**Example (for PowerShell on Windows):**

$env:AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
$env:AZURE\_OPENAI\_KEY="your-openai-api-key"  
$env:AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
$env:AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
$env:AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
$env:AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

**Example (for Bash/Zsh on Linux/macOS):**

export AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
export AZURE\_OPENAI\_KEY="your-openai-api-key"  
export AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
export AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
export AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
export AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

#### Execution

Run the application from the TechnicalSupportRAGApp directory:

dotnet run

#### Expected Results

The application will start an interactive chat session with the Technical Support Agent. You can ask questions related to the content of your Azure AI Search index. The agent will retrieve relevant information and use it to answer your questions.

**Example Interaction (assuming your index contains information about a specific machinery component):**

Technical Support Agent Ready. Ask a question (type 'exit' to quit):  
  
You: How do I troubleshoot the XYZ-123 motor?  
Searching for relevant documents for query: How do I troubleshoot the XYZ-123 motor?  
Generating answer with LLM using retrieved context...  
Agent: To troubleshoot the XYZ-123 motor, first check the power supply connections. Ensure they are securely fastened and that the motor is receiving the correct voltage as specified in the technical manual section 3.2. If the power supply is adequate, inspect the motor for any visible damage or obstructions. Refer to section 4.1 for common fault codes and their resolutions.  
  
You: What is the maximum operating temperature for the ABC-456 sensor?  
Searching for relevant documents for query: What is the maximum operating temperature for the ABC-456 sensor?  
Generating answer with LLM using retrieved context...  
Agent: The maximum operating temperature for the ABC-456 sensor is 85 degrees Celsius. This information can be found in the sensor's specifications sheet, section 2.1.  
  
You: exit

### How to Test the Python Implementation Example

This section provides step-by-step instructions on how to set up, run, and verify the Python code example for the RAG-based Technical Support Agent.

#### Prerequisites

* Python 3.8 or higher installed.
* An Azure OpenAI Service instance with a chat model deployed (e.g., gpt-3.5-turbo or gpt-4). You will need the endpoint, API key, and the deployment name of your chat model.
* An Azure AI Search service with an index populated with your technical documentation. You will need the search service endpoint, API key, and the index name. Ensure your documents have a content field containing the text to be searched.

#### Setup

1. **Create a new Python file:** Create a file named technical\_support\_rag.py.
2. **Install required packages:**

* pip install semantic-kernel azure-search-documents openai

1. **Add the Python code:** Copy the Python code provided above in the “Python Implementation Example” section into technical\_support\_rag.py.

#### Configuration

Set the following environment variables with your Azure OpenAI and Azure AI Search credentials. Replace the placeholder values with your actual information:

* AZURE\_OPENAI\_ENDPOINT="YOUR\_AZURE\_OPENAI\_ENDPOINT"
* AZURE\_OPENAI\_KEY="YOUR\_AZURE\_OPENAI\_KEY"
* AZURE\_OPENAI\_DEPLOYMENT\_NAME="YOUR\_AZURE\_OPENAI\_DEPLOYMENT\_NAME"
* AZURE\_SEARCH\_SERVICE\_ENDPOINT="YOUR\_AZURE\_SEARCH\_SERVICE\_ENDPOINT"
* AZURE\_SEARCH\_API\_KEY="YOUR\_AZURE\_SEARCH\_API\_KEY"
* AZURE\_SEARCH\_INDEX\_NAME="YOUR\_AZURE\_SEARCH\_INDEX\_NAME"

**Example (for Bash/Zsh on Linux/macOS):**

export AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
export AZURE\_OPENAI\_KEY="your-openai-api-key"  
export AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
export AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
export AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
export AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

**Example (for PowerShell on Windows):**

$env:AZURE\_OPENAI\_ENDPOINT="https://your-openai-resource.openai.azure.com/"  
$env:AZURE\_OPENAI\_KEY="your-openai-api-key"  
$env:AZURE\_OPENAI\_DEPLOYMENT\_NAME="your-chat-model-deployment-name"  
$env:AZURE\_SEARCH\_SERVICE\_ENDPOINT="https://your-search-service.search.windows.net"  
$env:AZURE\_SEARCH\_API\_KEY="your-search-api-key"  
$env:AZURE\_SEARCH\_INDEX\_NAME="your-search-index-name"

#### Execution

Run the Python script:

python technical\_support\_rag.py

#### Expected Results

The application will start an interactive chat session with the Technical Support Agent. You can ask questions related to the content of your Azure AI Search index. The agent will retrieve relevant information and use it to answer your questions.

**Example Interaction (assuming your index contains information about a specific machinery component):**

Technical Support Agent Ready. Ask a question (type 'exit' to quit):  
  
You: How do I troubleshoot the XYZ-123 motor?  
Searching for relevant documents for query: How do I troubleshoot the XYZ-123 motor?  
Generating answer with LLM using retrieved context...  
Agent: To troubleshoot the XYZ-123 motor, first check the power supply connections. Ensure they are securely fastened and that the motor is receiving the correct voltage as specified in the technical manual section 3.2. If the power supply is adequate, inspect the motor for any visible damage or obstructions. Refer to section 4.1 for common fault codes and their resolutions.  
  
You: What is the maximum operating temperature for the ABC-456 sensor?  
Searching for relevant documents for query: What is the maximum operating temperature for the ABC-456 sensor?  
Generating answer with LLM using retrieved context...  
Agent: The maximum operating temperature for the ABC-456 sensor is 85 degrees Celsius. This information can be found in the sensor's specifications sheet, section 2.1.  
  
You: exit