# Advanced Topic: Retrieval-Augmented Generation (RAG)

## 1. Concept Explanation

Retrieval-Augmented Generation (RAG) is an advanced technique that enhances the capabilities of large language models (LLMs) by integrating them with external knowledge bases or data sources. While LLMs are powerful in generating human-like text, their responses are limited to the information they were trained on. This can lead to several issues, including generating outdated, inaccurate, or hallucinated information, especially when dealing with domain-specific or real-time data. RAG addresses these limitations by enabling LLMs to retrieve relevant, up-to-date, and authoritative information from external sources and then use this retrieved information to *ground* their generated responses.

**How RAG Works:**

The core idea behind RAG is to combine the strengths of two distinct AI paradigms:

1. **Retrieval:** Before generating a response, the system first retrieves relevant documents, passages, or data snippets from a knowledge base. This knowledge base can be anything from a company’s internal documentation, a database, a collection of research papers, or even the internet.
2. **Generation:** The retrieved information is then provided to the LLM as additional context, alongside the user’s original query. The LLM then uses this augmented input to generate a more informed, accurate, and contextually relevant response.

**The RAG Process typically involves the following steps:**

* **Indexing/Vectorization:** The external knowledge base is pre-processed and indexed. This often involves breaking down documents into smaller chunks (e.g., paragraphs, sections), converting these chunks into numerical representations called embeddings (vectorization), and storing them in a vector database or a search index (like Azure AI Search). This allows for efficient semantic search.
* **User Query:** A user submits a query or question.
* **Retrieval:** The user’s query is also converted into an embedding. This query embedding is then used to perform a similarity search against the indexed embeddings in the knowledge base. The system retrieves the top-k most relevant chunks of information.
* **Augmentation:** The retrieved chunks of information are then appended to the user’s original query, forming an augmented prompt. This augmented prompt provides the LLM with specific, relevant context.
* **Generation:** The augmented prompt is fed into the LLM. The LLM then generates a response that is grounded in the retrieved information, making it more factual and less prone to hallucinations.

**Key Benefits of RAG:**

* **Reduces Hallucinations:** By providing the LLM with authoritative information, RAG significantly minimizes the generation of incorrect or fabricated facts.
* **Access to Up-to-Date Information:** LLMs’ knowledge is static after training. RAG allows them to access and incorporate the latest information from dynamic external sources.
* **Domain Specificity:** RAG enables LLMs to answer questions about specific, proprietary, or niche domains for which they were not explicitly trained.
* **Transparency and Explainability:** Since responses are grounded in retrieved documents, it’s often possible to show the user the source of the information, increasing trust and explainability.
* **Cost-Effective:** Fine-tuning large LLMs can be expensive and data-intensive. RAG offers a more cost-effective way to adapt LLMs to new information without retraining.

## 2. Relevant Use Cases and Implementation Scenarios

RAG is a versatile technique with applications across various industries and scenarios:

* **Enterprise Knowledge Management and Q&A Systems:**
  + **Use Case:** Building intelligent chatbots or internal Q&A systems that can answer employee questions based on a company’s vast internal documentation (HR policies, technical manuals, product specifications, sales playbooks).
  + **Scenario:** An employee asks, “What is the company’s policy on remote work expenses?” The RAG system retrieves relevant sections from the HR policy document and uses an LLM to provide a concise, accurate answer, citing the policy section.
* **Customer Support and Service:**
  + **Use Case:** Empowering customer service agents or self-service portals with AI assistants that can quickly find answers to customer queries from product manuals, FAQs, and support tickets.
  + **Scenario:** A customer asks, “How do I troubleshoot error code E-205 on my new router?” The RAG system searches product documentation, retrieves the troubleshooting steps for E-205, and the LLM generates a clear, step-by-step guide for the customer.
* **Legal and Compliance:**
  + **Use Case:** Assisting legal professionals in researching case law, regulations, and contracts by providing precise answers grounded in legal databases.
  + **Scenario:** A lawyer needs to know, “What are the recent changes to data privacy regulations in California?” The RAG system queries legal databases, retrieves the latest legislative updates, and the LLM summarizes the key changes and their implications.
* **Healthcare and Medical Information:**
  + **Use Case:** Providing medical professionals with quick access to the latest research, drug information, or patient history details from vast medical literature and EHRs.
  + **Scenario:** A doctor asks, “What are the latest treatment protocols for Type 2 Diabetes based on recent clinical trials?” The RAG system retrieves relevant studies and guidelines, and the LLM synthesizes the information into a concise summary.
* **Financial Analysis and Research:**
  + **Use Case:** Enabling financial analysts to quickly extract and synthesize information from financial reports, market news, and economic indicators.
  + **Scenario:** An analyst asks, “What were the key revenue drivers for Company X in Q3 2024?” The RAG system retrieves the company’s quarterly report and relevant news articles, and the LLM extracts and presents the key drivers.
* **Education and Research:**
  + **Use Case:** Creating intelligent tutoring systems or research assistants that can answer student questions or provide summaries of academic papers from a vast corpus of educational materials.
  + **Scenario:** A student asks, “Explain the concept of quantum entanglement in simple terms.” The RAG system retrieves explanations from textbooks and scientific articles, and the LLM generates an easy-to-understand explanation.

## 3. Key Considerations and Best Practices

Implementing RAG effectively requires careful attention to several key aspects:

* **Data Preparation and Chunking:**
  + **Quality:** The quality of your source data is paramount. Ensure documents are clean, well-structured, and accurate.
  + **Chunk Size:** Experiment with different chunk sizes for your documents. Too small, and context might be lost; too large, and irrelevant information might dilute the prompt. Optimal chunking often depends on the nature of the data and the queries.
  + **Overlap:** Consider adding overlap between chunks to ensure continuity and prevent loss of context at chunk boundaries.
* **Embedding Model Selection:**
  + Choose an embedding model that is suitable for your domain and language. Models trained on general text might not perform as well for highly specialized jargon. Azure OpenAI offers various embedding models.
* **Vector Database/Search Index:**
  + Select a robust and scalable vector database or search service (like Azure AI Search) that can handle your data volume and query load efficiently. Consider features like filtering, hybrid search (keyword + vector), and scalability.
* **Retrieval Strategy:**
  + **Top-K Retrieval:** Determine the optimal number of top-k (most similar) chunks to retrieve. Too few might miss crucial information; too many might overwhelm the LLM or introduce noise.
  + **Re-ranking:** After initial retrieval, consider re-ranking the retrieved documents based on their relevance to the query using a smaller, more precise model or heuristic.
  + **Hybrid Search:** Combine keyword-based search (for exact matches) with vector similarity search (for semantic understanding) to improve retrieval accuracy.
* **Prompt Engineering for Generation:**
  + **Clear Instructions:** Instruct the LLM clearly on how to use the retrieved context. For example, “Answer the question based *only* on the provided context. If the answer is not in the context, state that you don’t know.”
  + **Context Placement:** Experiment with where to place the retrieved context in the prompt (e.g., before or after the question). Some models perform better with context at the beginning, others at the end.
  + **Role-Playing:** Assign a persona to the LLM (e.g., “You are a helpful assistant providing answers from internal documentation”) to guide its tone and style.
* **Evaluation:**
  + **Retrieval Metrics:** Evaluate the quality of your retrieval system (e.g., Recall, Precision, Mean Reciprocal Rank).
  + **Generation Metrics:** Evaluate the quality of the generated answers (e.g., faithfulness to retrieved context, relevance, fluency, conciseness). Human evaluation is often necessary for subjective quality.
  + **End-to-End Evaluation:** Test the entire RAG pipeline with real-world queries and measure its effectiveness in solving the target problem.
* **Scalability and Performance:**
  + Optimize the indexing process for large datasets.
  + Ensure your vector database/search service can handle concurrent queries.
  + Manage LLM inference costs and latency.
* **Responsible AI:**
  + **Bias:** Be aware of potential biases in your source data and how they might influence retrieved information and generated responses.
  + **Transparency:** Provide users with the sources of information when possible to build trust and allow for verification.
  + **Security and Privacy:** Ensure sensitive data in your knowledge base is handled securely and in compliance with regulations.

## 4. Sample Implementations

To illustrate the practical application of Retrieval-Augmented Generation (RAG), we will provide sample implementations in both C# and Python. Our scenario is inspired by a real customer case where a company leverages RAG to provide intelligent support for its complex products.

### Customer Case Inspiration: Intelligent Product Support for a Global Manufacturer

Many global manufacturers, like Alstom (as seen in Microsoft customer stories), deal with vast amounts of technical documentation for their complex products (e.g., trains, energy systems). Providing timely and accurate support to technicians and customers globally is a significant challenge. A RAG-based system can revolutionize this by allowing support personnel to ask natural language questions and receive precise answers grounded in the latest technical manuals, troubleshooting guides, and service bulletins. This reduces resolution times, improves consistency, and enhances customer satisfaction.

### 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:** 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 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