CDW Documentation

Agentic RAG

Agentic RAG

This document provides an overview and explanation of the code used to create a Semantic Kernel-based tool that integrates with Azure AI Search for Retrieval-Augmented Generation (RAG). In this document we have an AI agent that retrieves inflation rates from an Azure AI Search index, augments user queries with semantic search results, and streams detailed answers about the consumer price index.

A juypter notebook is used to run this

Importing the necessary packages

```
import json
import os

from typing import Annotated

from IPython.display import display, HTML

from dotenv import load_dotenv

from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import SearchIndex, SimpleField,
SearchFieldDataType, SearchableField

from openai import AsyncOpenAI

from semantic_kernel.agents import ChatCompletionAgent,
ChatHistoryAgentThread
from semantic_kernel.connectors.ai.open_ai import OpenAIChatCompletion
from semantic_kernel.contents import
FunctionCallContent,FunctionResultContent, StreamingTextContent
from semantic_kernel.functions import kernel_function
```

Creating the Semantic Kernel and AI Service

A Semantic Kernel instance is created and configured with an asynchronous OpenAI chat completion service. The service is added to the kernel for use in generating responses.

```
load_dotenv()
# Initialize the asynchronous OpenAI client
client = AsyncOpenAI(
    api_key=os.environ["GITHUB_TOKEN"],
    base_url="https://models.inference.ai.azure.com/"
)
```

```
# Create the OpenAI Chat Completion Service
chat_completion_service = OpenAIChatCompletion(
    ai_model_id="gpt-4o-mini",
    async_client=client,
)
```

Defining the Prompt Plugin

The PromptPlugin is a native plugin that defines a function to build an augmented prompt using retrieval context

```


class SearchPlugin:

    def __init__(self, search_client: SearchClient):
        self.search_client = search_client

    @kernel_function(
        name="build_augmented_prompt",
        description="Build an augmented prompt using retrieval context or
function results.",
    )
    def build_augmented_prompt(self, query: str, retrieval_context: str) ->
str:
        return (
            f"Retrieved Context:\n{retrieval_context}\n\n"
            f"User Query: {query}\n\n"
            "First review the retrieved context, if this does not answer the
query, try calling an available plugin functions that might give you an
answer. If no context is available, say so."
        )
    @kernel_function(
        name="retrieve_documents",
        description="Retrieve documents from the Azure Search service.",
    )
    def get_retrieval_context(self, query: str) -> str:
        results = self.search_client.search(query)
        context_strings = []
        for result in results:
            context_strings.append(f"Document: {result['content']}")
        return "\n\n".join(context_strings) if context_strings else "No
results found"
```

```
class EggPriceInfoPlugin:
    """A Plugin that provides the egg price in USA for this year."""

    def __init__(self):
        # Dictionary of month and the cost of egg
```

```
        self.eggprice_month = {
            "january ": "$4.95",
            "february ": "$5.90",
            "march ": "$6.23",
            "april ": "$5.12",
            "may ": "$4.548",
            "june ": "$3.775",
            "july ": "$3.599",
            "august ": "$3.49"
        }

    @kernel_function(description="Get the egg price for specific month.")
    def get_eggprice_month(self, eggprice_month: str) -> Annotated[str,
"Returns the eggprice for the month."]:
        """Get the eggprice for the month."""
        # Normalize the input month (lowercase)
        normalized_eggprice_month = eggprice_month.lower()

        # Look up the price for the month
        if normalized_eggprice_month in self.eggprice_month:
            return f"The price in {eggprice_month} is
{self.eggprice_month[normalized_eggprice_month]}."
        else:
            return f"Sorry, I don't have price information for
{eggprice_month}. Available months are: January, February, March, April,
May, June, July, August"
```

Vector Database Initialization

We initialize Azure AI Search with persistent storage and add enhanced sample documents. Azure AI Search will be used to store and retrieve documents that provide context for generating accurate responses.

```


# Initialize Azure AI Search with persistent storage
search_service_endpoint = os.getenv("AZURE_SEARCH_SERVICE_ENDPOINT")
search_api_key = os.getenv("AZURE_SEARCH_API_KEY")
index_name = "price-documents"

search_client = SearchClient(
    endpoint=search_service_endpoint,
    index_name=index_name,
    credential=AzureKeyCredential(search_api_key)
)

index_client = SearchIndexClient(
    endpoint=search_service_endpoint,
    credential=AzureKeyCredential(search_api_key)
)
```

```
# Define the index schema
fields = [
    SimpleField(name="id", type=SearchFieldDataType.String, key=True),
    SearchableField(name="content", type=SearchFieldDataType.String)
]

index = SearchIndex(name=index_name, fields=fields)

# Check if index already exists if not, create it
try:
    existing_index = index_client.get_index(index_name)
    print(f"Index '{index_name}' already exists, using the existing index.")
except Exception:
    # Create the index if it doesn't exist
    print(f"Creating new index '{index_name}'...")
    index_client.create_index(index)


# Enhanced sample documents
documents = [
    {"id": "1", "content": " In January a significant monthly increase of
0.5% was driven by a broad rise in prices for shelter, food, and energy.."},
    {"id": "2", "content": "In February The annual inflation rate started to
slow, reaching 2.8% year-over-year."},
    {"id": "3", "content": "In March The annual inflation rate cooled
sharply to 2.4%, a six-month low.."},
    {"id": "4", "content": "In April Inflation continued its decline,
reaching 2.3% year-over-year, the smallest 12-month increase since February
2021.."},
    {"id": "5", "content": "In May   The annual rate ticked up slightly to
2.4%, but core CPI was cooler than expected, suggesting minimal tariff
impact.."},
    {"id": "5", "content": "In June The annual inflation rate rose to 2.7%,
with higher food and shelter prices outpacing a decline in gasoline
costs.."},
    {"id": "5", "content": "In July  Inflation held steady at 2.7% annually,
as falling gasoline prices were offset by continued increases in core
inflation, partly due to tariffs.."},
    {"id": "5", "content": "In Aug   Economists forecast a 2.9% annual
increase, expecting tariffs to have a more apparent impact on prices.
Official data is scheduled for release on Thursday, September 11, 2025."},
]

# Add documents to the index
search_client.upload_documents(documents)
```

```
agent = ChatCompletionAgent(
    service=chat_completion_service,
    plugins=[SearchPlugin(search_client=search_client),
EggPriceInfoPlugin()],
```

```
    name="PriceAgent",
    instructions="Answer price queries using the provided tools and context.
If context is provided, do not say 'I have no context for that'",
)
```

```
async def main():
    thread: ChatHistoryAgentThread | None = None

    user_inputs = [
        "Can you share egg price in february 2025?",
        "What is the consumer price index for August 2025?",
    ]

    for user_input in user_inputs:
        html_output = (
            f"<div style='margin-bottom:10px'>"
            f"<div style='font-weight:bold'>User:</div>"
            f"<div style='margin-left:20px'>{user_input}</div></div>"
        )

        agent_name = None
        full_response: list[str] = []
        function_calls: list[str] = []

        # Buffer to reconstruct streaming function call
        current_function_name = None
        argument_buffer = ""

        async for response in agent.invoke_stream(
            messages=user_input,
            thread=thread,
        ):
            thread = response.thread
            agent_name = response.name
            content_items = list(response.items)

            for item in content_items:
                if isinstance(item, FunctionCallContent):
                    if item.function_name:
                        current_function_name = item.function_name

                    # Accumulate arguments (streamed in chunks)
                    if isinstance(item.arguments, str):
                        argument_buffer += item.arguments
                elif isinstance(item, FunctionResultContent):
                    # Finalize any pending function call before showing
result
                    if current_function_name:
                        formatted_args = argument_buffer.strip()
```

```
                        try:
                            parsed_args = json.loads(formatted_args)
                            formatted_args = json.dumps(parsed_args)
                        except Exception:
                            pass  # leave as raw string

                        function_calls.append(f"Calling function:
{current_function_name}({formatted_args})")
                        current_function_name = None
                        argument_buffer = ""

                    function_calls.append(f"\nFunction
Result:\n\n{item.result}")
                elif isinstance(item, StreamingTextContent) and item.text:
                    full_response.append(item.text)

        if function_calls:
            html_output += (
                "<div style='margin-bottom:10px'>"
                "<details>"
                "<summary style='cursor:pointer; font-weight:bold;
color:#0066cc;'>Function Calls (click to expand)</summary>"
                "<div style='margin:10px; padding:10px; background-
color:#f8f8f8; "
                "border:1px solid #ddd; border-radius:4px; white-space:pre-
wrap; font-size:14px; color:#333;'>"
                f"{chr(10).join(function_calls)}"
                "</div></details></div>"
            )

        html_output += (
            "<div style='margin-bottom:20px'>"
            f"<div style='font-weight:bold'>{agent_name or
'PriceAgent'}:</div>"
            f"<div style='margin-left:20px; white-space:pre-
wrap'>{''.join(full_response)}</div></div><hr>"
        )

        display(HTML(html_output))

await main()
```

Testing

```
async def interactive_conversation():
    """Interactive conversation loop with memory"""
    thread: ChatHistoryAgentThread | None = None
    print("🤖 PriceAgent: Hello! I can help you with price information. Type
'quit' to exit.")
    print("Ask me about egg prices, inflation rates, or consumer price index
```

```
data.")
    print("-" * 60)
    while True:
        try:
            # Get user input
            user_input = input("\n You: ").strip()
            if user_input.lower() in ['quit', 'exit', 'bye']:
                print(" PriceAgent: Goodbye! Thanks for chatting with me.")
                break
            if not user_input:
                continue
            print(" PriceAgent: ", end="", flush=True)
            # Track function calls and responses
            function_calls = []
            full_response = []
            # Buffer for function call reconstruction
            current_function_name = None
            argument_buffer = ""
            # Stream the agent response
            async for response in agent.invoke_stream(
                messages=user_input,
                thread=thread,
            ):
                thread = response.thread  # Maintain conversation memory
                content_items = list(response.items)
                for item in content_items:
                    if isinstance(item, FunctionCallContent):
                        if item.function_name:
                            current_function_name = item.function_name
                        # Accumulate arguments (streamed in chunks)
                        if isinstance(item.arguments, str):
                            argument_buffer += item.arguments
                    elif isinstance(item, FunctionResultContent):
                        # Finalize any pending function call
                        if current_function_name:
                            formatted_args = argument_buffer.strip()
                            try:
                                parsed_args = json.loads(formatted_args)
                                formatted_args = json.dumps(parsed_args,
indent=2)
                            except Exception:
                                pass
                            function_calls.append({
                                'function': current_function_name,
                                'args': formatted_args,
                                'result': str(item.result)
                            })
                            current_function_name = None
                            argument_buffer = ""
                    elif isinstance(item, StreamingTextContent) and
item.text:
```

```
                        # Print response as it streams
                        print(item.text, end="", flush=True)
                        full_response.append(item.text)
            print()  # New line after streaming response
            # Optionally show function calls in verbose mode
            if function_calls and False:  # Set to True to see function
calls
                print("\n Function calls:")
                for call in function_calls:
                    print(f"  → {call['function']}({call['args']})")
                    print(f"    Result: {call['result']}")
        except KeyboardInterrupt:
            print("\n PriceAgent: Conversation interrupted. Goodbye!")
            break
        except Exception as e:
            print(f"\n Error: {e}")
            print("Please try again.")

# Run the interactive conversation
await interactive_conversation()
```