Build Modern Apps Deep Dive

Demo Script and session preparation

This session can be delivered within one hour.

Prep:

* Open the PPT deck
* Browser open with four tabs
  + Repo page: <https://github.com/Azure/Build-Modern-AI-Apps>
  + Azure Portal on the Azure OpenAI account (if deployed separately)
  + Azure portal on the Resource group blade for the deployment
  + Web application pointing to live deployment
    - To get this to go Web app container in Azure Portal and open the Application URL on Overview blade.
* Visual Studio open to the solution
  + Set breakpoints in the following places:
    - Search
      * ChatPane.razor – Line 313
    - VectorSearchAiAssistant.Service
      * ChatService.cs –
        + GetChatCompletionAsync() Line 103, 107, 115, 192
      * CosmosDbService.cs –
        + GenericChangeFeedHandeler() Line 142
        + UpsertSessionBatchAsync() - Line 362
      * SemanticKernelRAGService.cs –
        + GetResponse () Line 135
    - VectorSearchAiAssistant.SemanticKernel
      * ChatBuilder.cs –
        + OptimizePromptSize() Line 108
      * AzureCognitiveSearchVectorMemory.cs –
        + AddMemory() line 141
      * ShortTermVolatileMemory.cs
        + Initialize() Line 34, 52
    - Recommended that classes be collapsed to definitions by right-clicking in each class, selecting outlining, collapse to definitions. This will make it easier visually to explore the source code and navigate.
* Configure PostMan to Add and Remove a product using the instructions here:
  + <https://github.com/Azure/Vector-Search-AI-Assistant/tree/cognitive-search-vector#real-time-add-and-remove-data>
  + The <chat-service-hostname> = <https://localhost:63279/>
  + Be sure to set content = application/json in headers
    - A screenshot of a computer

      Description automatically generated
    - A screenshot of a computer

      Description automatically generated
    - PS: You can also optionally configure PostMan to do this against your live deployment and demonstrate the real-time adding a new product for vector search when you do the live demo before stepping through the source code.

**Begin in PPT deck on Slide 1**

**Where to Start (Slide 7) [Have browser open to repo page]**

1. To get started you can go to our GitHub repository github.com/Azure/Build-Modern-AI-Apps.
2. Once there you can scroll down and click on the Navigate to Vector Search & AI Assistant link on this page.
3. This page has other solutions as well including Payment and Transaction Processing and Medical Claims scenarios.
4. This page also includes links to the Intelligent App Workshop created by our Apps GBB team. There is a lot of great samples and information for using Semantic Kernel for building Copilot applications in Azure.
5. [Scroll back up to Vector Search and AI Assistant link]
6. Clicking on the link for the Vector Search and AI Assistant will take you to the repository where this solution lives and includes all the information you need including an introduction to the concepts we will cover in this session as well as how to deploy and run the solution.

**Return to slide 8 on PowerShell Deployment**

**What gets deployed (Slide 9)**

Let’s go into the Azure portal and explore the services. (Return to browser)

1. Show each of the services within the portal.
2. OpenAI – click on Models, click on completions show GPT 35 Turbo, click on embeddings, show ada-002.
3. Cosmos DB – show the containers with the data. Click and show completions container with the messages within it.
4. Cog Search –
   1. Show the index and the vectorized data within it.
   2. Show the index fields and briefly touch on Retrievable, Filterable, Facetable, and Searchable. Note how the hash and salt for the Password are NOT returned in searches.
5. Storage –
   1. Click containers, show return policy, show system prompt and summarize prompt.
   2. Show memory-source, briefly touch on Cog Search faceted queries definitions and blob source definitions.
6. Azure Container Apps –
   1. Show one for API and one for Web
   2. Click into Web API, click containers, show the environment variables.

Let’s return to slides before we dive into the source for this solution. (resume with slide 10)

**Tour the UX & Live Code Walk through (Slide 13)**

Let’s walk through the application’s user experience and show you how it works.

The application frontend is an ASP.NET Blazor Server application built as a chat interface.

**Showing the UX using your LIVE Deployment (App should already be running)**

1. Browser showing the UX
2. Start a new chat
3. “What kinds of products do you sell?”
   1. Show the prompt and the completion, show the tokens used.
4. “Can you share more details about bikes?”
   1. Point out how the chat was renamed.
5. “Can you list details about your bikes?”
   1. It may not show Mountain bikes if the RAG payload is large.
   2. “Can you list details about your Mountain Bikes?”
6. Click like or dislike.
   1. A feature like this can be used to refine the experience for users. Right now, questions are answered directly from data in the database. But data from conversations can itself be mined and stored such that questions are first directed to a vectorized index of frequently answered questions first, and only then to the larger database of vectorized information if the user says the answer given wasn’t sufficient.
7. Click the **show prompt** text to show what is sent to Open AI.
8. Ask more questions to show the responses you get
   1. “What colors do the bikes come in?”,
   2. “Can you tell me what your prices are for these bikes?”
9. Optional:
   1. Highlight how you can add new data to vector search in near real-time.
   2. Start new chat session.
   3. “Can you list what socks you have?”
   4. Go to PostMan, run the Add a Product against the live deployment.
   5. Return to the Chat.
   6. “Can you list all your socks again?”
      1. This should return the new Cosmic Sock. If not, start a new session and type the question in there. Sometimes OpenAI gets confused if a previous response does not include the new sock.
   7. This is a simple example but it can apply to any kind of data. It could just as easily be user session data including what products a user recently viewed or clicked on, etc. You can easily incorporate real-time data and make available for vector search.

**Code Walk through**

Let’s now explore the solution and source code in Visual Studio

1. Show the web application Project.
   1. Open the Pages folder.
   2. Open and explore the left nav page and the chat panel razor files.
2. Show the Chat Service Web API Project.
   1. Open Chat Endpoints class. Note how the paths are mapped to the Chat Service.
3. Next move to the VSAI Service Project.
   1. Open Services folder.
   2. Open Chat Service.
      1. Slowly scroll through source here.
      2. Web API calls into this service. Main entry point to the RAG Pattern work-flow.
   3. Open Cog Search and OpenAI services.
      1. These are NOT used. Examples if not using Semantic Kernel.
      2. Show GetChatCompletion() and GetEmbeddings().
   4. Open Cosmos DB Service.
      1. Data access layer for Cosmos and stores the transactional and chat history.
      2. Show the generic Change Feed handler. Configured to monitor any container for new/updated data then vectorize and store in Cog Search.
      3. Show the Model Registry (Line 152) Right Click - Go to implementation
         1. The purpose of the model here is that it drives what data in a JSON object gets vectorized and stored in Azure Cognitive Search. You wouldn’t need to vectorize everything. So having a way to keep this more focused helps improve search accuracy.
      4. Explain the role of Embedded Entity
         1. The Embedded Entity here is what controls how Azure Cognitive Search stores the data when it is inserted. If you right click and select Definition you can see that has attributes on whether data isFilterable, IsFacetable, etc.
   5. Open Durable System prompt service
      1. This is used to retrieve the system prompts that are stored as text files from blob storage.
      2. These are both Skills within Semantic Kernel and provide instructions on how to behave and perform tasks. We’ll see those when we step through the app.
   6. Open the Semantic Kernel RAG Service.
      1. This is entry point for the Chat Service Web API and is the service layer that interfaces with the underlying Semantic Kernel and ties all the major components together.
   7. Open the constructor for RAG Service.
      1. Let’s look at the constructor for this service. Here you can see this is where the connector to OpenAI is defined for generating the embeddings with the Embedding Generation Service and here again for the Completions API where GPT will generate a response.
      2. Below that you see this is where we connect Semantic Kernel to Azure Cognitive Search. We also pass the Embedding Generator for OpenAI into this to create embeddings from the user prompts when those are passed in as a query.
   8. Open the Add Memory()
      1. This is the entry point for long-term memory generation. Describe the flow and hint at how entityType\_\_ is used for dynamic entity resolution in the indexing process.
   9. Open GetResponse() method.
      1. The main function in this class is GetResponse(). This is the most central function in the solution. It executes the query, generates the completion from OpenAI and returns the results.
      2. One thing I want to point out is that we limit the number of memories returned here to just the top ten most relevant. I’ll explain why we do this in a minute.
      3. After the query returns, we construct a new ChatBuilder object that is implemented using Semantic Kernel. This builds the payload for what Semantic Kernel will send to OpenAI to generate the chat completion. This includes everything we need including:
         1. A reference to our Semantic Kernal object.
         2. The maximum number of tokens that can be used to generate a response.
         3. Configuration values to use for token management.
         4. The system prompt text we load from blob storage.
         5. The memories from the vector query results.
         6. And finally the chat history to provide OpenAI with some context to generate a meaningful response.
      4. The code below here executes the completion and takes the text returned from OpenAI and all of the tokens used in the prompt and completion and sends in the return for the function.
4. Move to the Semantic Kernel project.
   1. Open the Memory folder and the Azure Cognitive Search folder.
      1. There are two main things to look at here.
      2. First is the memory type we create with Azure Cognitive Search. This we created before there was a connector for it, so now this is a bit easier.
   2. Open the Chat folder.
      1. The main thing I want to show here is the Chat Builder.
      2. This is a key component of the entire solution and is the responses are generated from OpenAI.
      3. It also handles the token management for the request generation to OpenAI.
   3. Open the Text Embedding folder
      1. Explain the role of the Embedding Field Attribute
      2. Describe the functionality of the Embedding Utility
   4. Open the Skills/Core folder
      1. Describe the Generic Summarizer skill
      2. Describe the Text Embedding Object Memory skill and show how the long-term memories are retrieved
5. Introduce Token Management:
   1. One aspect to building RAG Pattern solutions is managing the tokens used in the request to Open AI. Requests need to be within the maximum allowed for a model. The payloads for these requests can be rather large because you are querying for additional data and can grow beyond what a model can handle. If you pass enough data that consumes all of the tokens in the request, you’ll get an exception. However, it’s not enough to just remain under the max amount. You need to leave enough tokens for the completion. Not enough and the response may be incomplete. In this solution we implemented a way to measure token usage for the request and ensure we leave enough for the response or completion.
6. Open OptimizePromptSize() function.
   1. Token management is handled in the [OptimizePromptSize()](https://github.com/Azure/Vector-Search-AI-Assistant/blob/cognitive-search-vector/VectorSearchAiAssistant.SemanticKernel/Chat/ChatBuilder.cs" \l "L107) method in the ChatBuilder class. This method uses the SemanticKernel tokenizer, which wraps the open-source [GPT3Tokenizer](https://github.com/Azure/Vector-Search-AI-Assistant/blob/cognitive-search-vector/VectorSearchAiAssistant.SemanticKernel/Chat/SemanticKernelTokenizer.cs). The tokenizer takes text and generates an array of vectors. The number of elements in the array represent the number of tokens that will be consumed. It can also do the reverse, and take an array of vectors and output text. This is especially useful because we can equate everything in tokens and adjust numerically to get the payloads just right.
   2. In this method here we do the following.
      1. Measure the amount of tokens for the system prompt.
      2. Measure the amount of tokens for the vector search results (the memories).
      3. Measure the amount of tokens for the user prompt and conversation (messages).
      4. Finally we add up the tokens for the system prompt + RAG data + conversation + a little buffer space as well.
      5. If that is greater than the max tokens for the OpenAI Model then we reduce each of these by amounts we’ve specified in our configuration for system prompt, memories and messages.
      6. From there if we need, we just keep trimming memories and conversation history until we get under a safe limit.
      7. The resulting text is then what is sent as the prompt to Azure OpenAI.
   3. Now you should not ever get to a point where you are not sending enough information to generate a good completion. The OpenAI models can accept a decent number of tokens and these are getting larger.
   4. However, this is one aspect of design where you should focus on what is returned in your RAG Pattern queries. In our sample here we vectorize everything. But this is a simple solution we’ve built and there isn’t much data. It may become necessary as a later step to consider what data you vectorize and query in your solution.
   5. We did, however, implement one common element to gate the payload size and that is to limit the number of items returned in a query to just the top 10 most relevant results like I showed earlier
7. So those are all the major components for our solution. Let’s walk through a complete request and see it in action.

**Walking the execution flow of a RAG pattern query**

1. F5 to start the solution.
2. Talk about how the short term memories are created using a faceted search defined from text in blob storage. These do product counts for categories and count all products.
3. Allow both the application and Web API browser tabs to open. And leave in their own browser process.
4. Navigate to the browser tab running the localhost version of the chat Web UX.
5. Refresh it if it does not include the previous chat sessions using the Live version.
6. From the same chat session used before during the UX portion of the demo.
7. Ask an additional question. “What kind of socks do you have?”
8. Walk through all the breakpoints and describe what is happening.

**Add and Remove a Product**

1. Open PostMan
2. Send the PUT operation to Add a new Product
3. Walk though the breakpoints from Change Feed Processor until complete.
4. Return to the Web UX.
5. Ask question, “Please list all of your socks in your product catalog?”
   1. If it returns the list without Cosmic Socks, mention that in some cases conversation history can override vector search results.
   2. Open a new tab, ask question, “What kind of socks do you have?”
   3. This will list all, including the new Cosmic Sock.
6. Return to PostMan
7. Send DELETE operation to remove a Product
8. Walk through the breakpoints that delete the product and the item from Cog Search.

**[Return to PPT deck to close out the session]**