Retrieval Augmented Generation (RAG) in Azure AI Search

Article • 04/22/2024

Retrieval Augmented Generation (RAG) is an architecture that augments the capabilities of a Large Language Model (LLM) like ChatGPT by adding an information retrieval system that provides grounding data. Adding an information retrieval system gives you control over grounding data used by an LLM when it formulates a response. For an enterprise solution, RAG architecture means that you can constrain generative AI to *your enterprise content* sourced from vectorized documents and images, and other data formats if you have embedding models for that content.

The decision about which information retrieval system to use is critical because it determines the inputs to the LLM. The information retrieval system should provide:

- Indexing strategies that load and refresh at scale, for all of your content, at the frequency you require.
- Query capabilities and relevance tuning. The system should return *relevant* results, in the short-form formats necessary for meeting the token length requirements of LLM inputs.
- Security, global reach, and reliability for both data and operations.
- Integration with embedding models for indexing, and chat models or language understanding models for retrieval.

Azure AI Search is a proven solution for information retrieval in a RAG architecture. It provides indexing and query capabilities, with the infrastructure and security of the Azure cloud. Through code and other components, you can design a comprehensive RAG solution that includes all of the elements for generative AI over your proprietary content.

> **Note**
>
> New to copilot and RAG concepts? Watch **Vector search and state of the art retrieval for Generative AI apps** .

Approaches for RAG with Azure AI Search

Microsoft has several built-in implementations for using Azure AI Search in a RAG solution.

- Azure AI Studio, use a vector index and retrieval augmentation.
- Azure OpenAI Studio, use a search index with or without vectors.
- Azure Machine Learning, use a search index as a vector store in a prompt flow.

Curated approaches make it simple to get started, but for more control over the architecture, you need a custom solution. These templates create end-to-end solutions in:

- Python
- .NET
- JavaScript
- Java

The remainder of this article explores how Azure AI Search fits into a custom RAG solution.

Custom RAG pattern for Azure AI Search

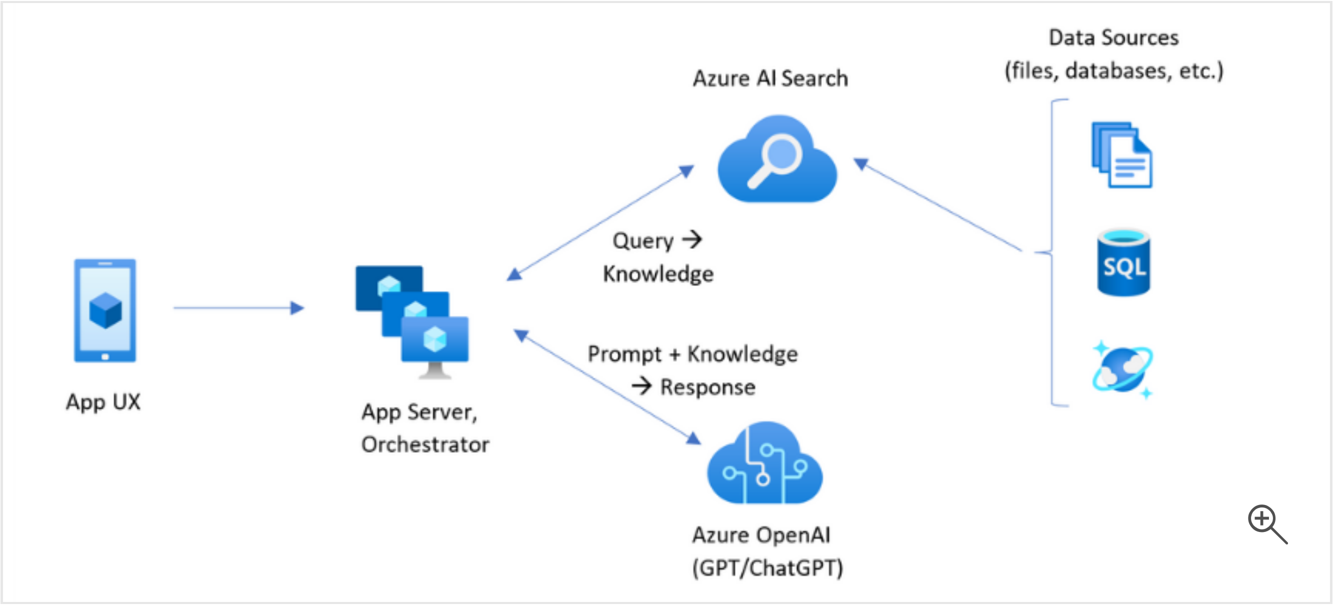
A high-level summary of the pattern looks like this:

- Start with a user question or request (prompt).

- Send it to Azure AI Search to find relevant information.
- Send the top ranked search results to the LLM.
- Use the natural language understanding and reasoning capabilities of the LLM to generate a response to the initial prompt.

Azure AI Search provides inputs to the LLM prompt, but doesn't train the model. In RAG architecture, there's no extra training. The LLM is pretrained using public data, but it generates responses that are augmented by information from the retriever.

RAG patterns that include Azure AI Search have the elements indicated in the following illustration.



- App UX (web app) for the user experience
- App server or orchestrator (integration and coordination layer)
- Azure AI Search (information retrieval system)
- Azure OpenAI (LLM for generative AI)

The web app provides the user experience, providing the presentation, context, and user interaction. Questions or prompts from a user start here. Inputs pass through the integration layer, going first to information retrieval to get the search results, but also go to the LLM to set the context and intent.

The app server or orchestrator is the integration code that coordinates the handoffs between information retrieval and the LLM. One option is to use LangChain to coordinate the workflow. LangChain integrates with Azure AI Search, making it easier to include Azure AI Search as a retriever in your workflow. Semantic Kernel is another option.

The information retrieval system provides the searchable index, query logic, and the payload (query response). The search index can contain vectors or nonvector content. Although most samples and demos include vector fields, it's not a requirement. The query is executed using the existing search engine in Azure AI Search, which can handle keyword (or term) and vector queries. The index is created in advance, based on a schema you define, and loaded with your content that's sourced from files, databases, or storage.

The LLM receives the original prompt, plus the results from Azure AI Search. The LLM analyzes the results and formulates a response. If the LLM is ChatGPT, the user interaction might be a back and forth conversation. If you're using Davinci, the prompt might be a fully composed answer. An Azure solution most likely uses Azure OpenAI, but there's no hard dependency on this specific service.

Azure AI Search doesn't provide native LLM integration for prompt flows or chat preservation, so you need to write code that handles orchestration and state. You can review demo source (Azure-Samples/azure-search-openai-demo) for a blueprint of what a full solution entails. We also recommend Azure AI Studio or Azure OpenAI Studio to create RAG-based Azure AI Search solutions that integrate with LLMs.

Searchable content in Azure AI Search

In Azure AI Search, all searchable content is stored in a search index that's hosted on your search service. A search index is designed for fast queries with millisecond response times, so its internal data structures exist to support that objective. To that end, a search index stores *indexed content*, and not whole content files like entire PDFs or images. Internally, the data structures include inverted indexes of tokenized text, vector indexes for embeddings, and unaltered text for cases where verbatim matching is required (for example, in filters, fuzzy search, regular expression queries).

When you set up the data for your RAG solution, you use the features that create and load an index in Azure AI Search. An index includes fields that duplicate or represent your source content. An index field might be simple transference (a title or description in a source document becomes a title or description in a search index), or a field might contain the output of an external process, such as vectorization or skill processing that generates a representation or text description of an image.

Since you probably know what kind of content you want to search over, consider the indexing features that are applicable to each content type:

Expand table

Content type	Indexed as	Features
text	tokens, unaltered text	Indexers can pull plain text from other Azure resources like Azure Storage and Cosmos DB. You can also push any JSON content to an index. To modify text in flight, use analyzers and normalizers to add lexical processing during indexing. Synonym maps are useful if source documents are missing terminology that might be used in a query.
text	vectors [1]	Text can be chunked and vectorized externally and then indexed as vector fields in your index.
image	tokens, unaltered text [2]	Skills for OCR and Image Analysis can process images for text recognition or image characteristics. Image information is converted to searchable text and added to the index. Skills have an indexer requirement.
image	vectors [1]	Images can be vectorized externally for a mathematical representation of image content and then indexed as vector fields in your index. You can use an open source model like OpenAI CLIP to vectorize text and images in the same embedding space.

[1] The generally available functionality of vector support requires that you call other libraries or models for data chunking and vectorization. However, integrated vectorization (preview) embeds these steps. For code samples showing both approaches, see azure-search-vectors repo .

[2] Skills are built-in support for AI enrichment. For OCR and Image Analysis, the indexing pipeline makes an internal call to the Azure AI Vision APIs. These skills pass an extracted image to Azure AI for processing, and receive the output as text that's indexed by Azure AI Search.

Vectors provide the best accommodation for dissimilar content (multiple file formats and languages) because content is expressed universally in mathematic representations. Vectors also support similarity search: matching on the coordinates that are most similar to the vector query. Compared to keyword search (or term search) that matches on tokenized terms, similarity search is more nuanced. It's a better choice if there's ambiguity or interpretation requirements in the content or in queries.

Content retrieval in Azure AI Search

Once your data is in a search index, you use the query capabilities of Azure AI Search to retrieve content.

In a non-RAG pattern, queries make a round trip from a search client. The query is submitted, it executes on a search engine, and the response returned to the client application. The response, or search results, consist exclusively of the verbatim content found in your index.

In a RAG pattern, queries and responses are coordinated between the search engine and the LLM. A user's question or query is forwarded to both the search engine and to the LLM as a prompt. The search results come back from the search engine and are redirected to an LLM. The response that makes it back to the user is generative AI, either a summation or answer from the LLM.

There's no query type in Azure AI Search - not even semantic or vector search - that composes new answers. Only the LLM provides generative AI. Here are the capabilities in Azure AI Search that are used to formulate queries:

Expand table

Query	Purpose	Why use it

feature		
Simple or full Lucene syntax	Query execution over text and nonvector numeric content	Full text search is best for exact matches, rather than similar matches. Full text search queries are ranked using the BM25 algorithm and support relevance tuning through scoring profiles. It also supports filters and facets.
Filters and facets	Applies to text or numeric (nonvector) fields only. Reduces the search surface area based on inclusion or exclusion criteria.	Adds precision to your queries.
Semantic ranking	Re-ranks a BM25 result set using semantic models. Produces short-form captions and answers that are useful as LLM inputs.	Easier than scoring profiles, and depending on your content, a more reliable technique for relevance tuning.
Vector search	Query execution over vector fields for similarity search, where the query string is one or more vectors.	Vectors can represent all types of content, in any language.
Hybrid search	Combines any or all of the above query techniques. Vector and nonvector queries execute in parallel and are returned in a unified result set.	The most significant gains in precision and recall are through hybrid queries.

Structure the query response

A query's response provides the input to the LLM, so the quality of your search results is critical to success. Results are a tabular row set. The composition or structure of the results depends on:

- Fields that determine which parts of the index are included in the response.
- Rows that represent a match from index.

Fields appear in search results when the attribute is "retrievable". A field definition in the index schema has attributes, and those determine whether a field is used in a response. Only "retrievable" fields are returned in full text or vector query results. By default all "retrievable" fields are returned, but you can use "select" to specify a subset. Besides "retrievable", there are no restrictions on the field. Fields can be of any length or type. Regarding length, there's no maximum field length limit in Azure AI Search, but there are limits on the size of an API request.

Rows are matches to the query, ranked by relevance, similarity, or both. By default, results are capped at the top 50 matches for full text search or k-nearest-neighbor matches for vector search. You can change the defaults to increase or decrease the limit up to the maximum of 1,000 documents. You can also use top and skip paging parameters to retrieve results as a series of paged results.

Rank by relevance

When you're working with complex processes, a large amount of data, and expectations for millisecond responses, it's critical that each step adds value and improves the quality of the end result. On the information retrieval side, *relevance tuning* is an activity that improves the quality of the results sent to the LLM. Only the most relevant or the most similar matching documents should be included in results.

Relevance applies to keyword (nonvector) search and to hybrid queries (over the nonvector fields). In Azure AI Search, there's no relevance tuning for similarity search and vector queries. BM25 ranking is the ranking algorithm for full text search.

Relevance tuning is supported through features that enhance BM25 ranking. These approaches include:

- Scoring profiles that boost the search score if matches are found in a specific search field or on other criteria.
- Semantic ranking that re-ranks a BM25 results set, using semantic models from Bing to reorder results for a better semantic fit to the original query.

In comparison and benchmark testing, hybrid queries with text and vector fields, supplemented with semantic ranking over the BM25-ranked results, produce the most relevant results.

Example code of an Azure AI Search query for RAG scenarios

The following code is copied from the retrievethenread.py file from a demo site. It produces `content` for the LLM from hybrid query search results. You can write a simpler query, but this example is inclusive of vector search and keyword search with semantic reranking and spell check. In the demo, this query is used to get initial content.

Python

```python
# Use semantic ranker if requested and if retrieval mode is text or
hybrid (vectors + text)
if overrides.get("semantic_ranker") and has_text:
    r = await self.search_client.search(query_text,
                                  filter=filter,
                                  query_type=QueryType.SEMANTIC,
                                  query_language="en-us",
                                  query_speller="lexicon",
                                  semantic_configuration_name="de-
fault",
                                  top=top,
                                  query_caption="extractive|high-
light-false" if use_semantic_captions else None,
                                  vector=query_vector,
                                  top_k=50 if query_vector else None,
                                  vector_fields="embedding" if
query_vector else None)
else:
    r = await self.search_client.search(query_text,
                                  filter=filter,
                                  top=top,
                                  vector=query_vector,
                                  top_k=50 if query_vector else None,
                                  vector_fields="embedding" if
query_vector else None)
if use_semantic_captions:
    results = [doc[self.sourcepage_field] + ": " + nonewlines(" .
".join([c.text for c in doc['@search.captions']])) async for doc in
r]
else:
    results = [doc[self.sourcepage_field] + ": " + nonew-
lines(doc[self.content_field]) async for doc in r]
content = "\n".join(results)
```

Integration code and LLMs

A RAG solution that includes Azure AI Search requires other components and code to create a complete solution. Whereas the previous sections covered information retrieval through Azure AI Search and which features are used to create and query searchable content, this section introduces LLM integration and interaction.

Notebooks in the demo repositories are a great starting point because they show patterns for passing search results to an LLM. Most of the code in a RAG solution consists of calls to the LLM so you need to develop an understanding of how those APIs work, which is outside the scope of this article.

The following cell block in the chat-read-retrieve-read.ipynb notebook shows search calls in the context of a chat session:

Python

```python
# Execute this cell multiple times updating user_input to accumulate
chat history
user_input = "Does my plan cover annual eye exams?"

# Exclude category, to simulate scenarios where there's a set of docs
you can't see
exclude_category = None

if len(history) > 0:
    completion = openai.Completion.create(
        engine=AZURE_OPENAI_GPT_DEPLOYMENT,
        prompt=summary_prompt_template.format(summary="\n".join(his-
tory), question=user_input),
        temperature=0.7,
        max_tokens=32,
        stop=["\n"])
    search = completion.choices[0].text
```

```
else:
    search = user_input

# Alternatively simply use search_client.search(q, top=3) if not us-
ing semantic ranking
print("Searching:", search)
print("-------------------")
filter = "category ne '{}'".format(exclude_category.replace("'",
"''")) if exclude_category else None
r = search_client.search(search,
                         filter=filter,
                         query_type=QueryType.SEMANTIC,
                         query_language="en-us",
                         query_speller="lexicon",
                         semantic_configuration_name="default",
                         top=3)
results = [doc[KB_FIELDS_SOURCEPAGE] + ": " + doc[KB_FIELDS_CON-
TENT].replace("\n", "").replace("\r", "") for doc in r]
content = "\n".join(results)

prompt = prompt_prefix.format(sources=content) + prompt_history +
user_input + turn_suffix

completion = openai.Completion.create(
    engine=AZURE_OPENAI_CHATGPT_DEPLOYMENT,
    prompt=prompt,
    temperature=0.7,
    max_tokens=1024,
    stop=["<|im_end|>", "<|im_start|>"])

prompt_history += user_input + turn_suffix +
completion.choices[0].text + "\n<|im_end|>" + turn_prefix
history.append("user: " + user_input)
history.append("assistant: " + completion.choices[0].text)

print("\n-------------------\n".join(history))
print("\n-------------------\nPrompt:\n" + prompt)
```

How to get started

- Use Azure AI Studio to create a search index.
- Use Azure OpenAI Studio and "bring your own data" to experiment with prompts

on an existing search index in a playground. This step helps you decide what model to use, and shows you how well your existing index works in a RAG scenario.

- "Chat with your data" solution accelerator , built by the Azure AI Search team, helps you create your own custom RAG solution.

- Enterprise chat app templates deploy Azure resources, code, and sample grounding data using fictitious health plan documents for Contoso and Northwind. This end-to-end solution gives you an operational chat app in as little as 15 minutes. Code for these templates is the **azure-search-openai-demo** featured in several presentations. The following links provide language-specific versions:
  - .NET
  - Python
  - JavaScript
  - Java

- Review indexing concepts and strategies to determine how you want to ingest and refresh data. Decide whether to use vector search, keyword search, or hybrid search. The kind of content you need to search over, and the type of queries you want to run, determines index design.

- Review creating queries to learn more search request syntax and requirements.

Note

Some Azure AI Search features are intended for human interaction and aren't useful in a RAG pattern. Specifically, you can skip autocomplete and suggestions. Other features like facets and orderby might be useful, but would be uncommon in a RAG scenario.

See also

- Retrieval Augmented Generation: Streamlining the creation of intelligent natural language processing models

- Retrieval Augmented Generation using Azure Machine Learning prompt flow
- Azure Cognitive Search and LangChain: A Seamless Integration for Enhanced Vector Search Capabilities

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A