Develop a RAG-based solution with your own data using Microsoft Foundry

1. Introduction

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/1-introduction

Introduction

Completed

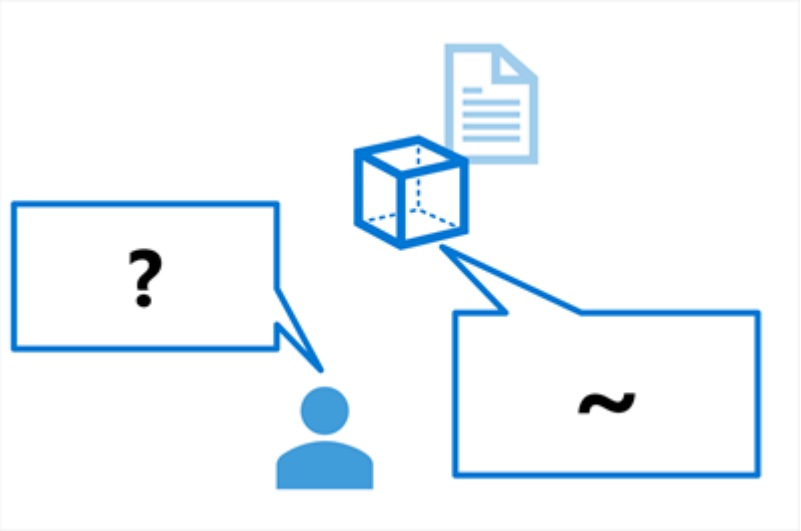
- 2 minutes

Language models are growing in popularity as they create impressive coherent answers to a user's questions. Especially when a user interacts with a language model through chat, it provides an intuitive way to get the information they need.

One prevalent challenge when implementing language models through chat is the so-called **groundedness**, which refers to whether a response is rooted, connected, or anchored in reality or a specific context. In other words, groundedness refers to whether the response of a language model is based on factual information.

Ungrounded prompts and responses

When you use a language model to generate a response to a prompt, the only information that the model has to base the answer on comes from the data on which it was trained - which is often just a large volume of uncontextualized text from the Internet or some other source.



The result will likely be a grammatically coherent and logical response to the prompt, but because it isn't grounded in relevant, factual data, it's uncontextualized; and may in fact be inaccurate and include "invented" information. For example, the question "Which product should I use to do *X*?" might include details of a fictional product.

Grounded prompts and responses

In contrast, you can use a data source to *ground* the prompt with some relevant, factual context. The prompt can then be submitted to a language model, including the grounding data, to generate a contextualized, relevant, and accurate response.

The data source can be any repository of relevant data. For example, you could use data from a product catalog database to ground the prompt "Which product should I use to do *X*?" so that the response includes relevant details of products that exist in the catalog.

In this module, you explore how to create your own chat-based language model application that is grounded, by building an agent with your own data.

2. Understand how to ground your language model

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/2-ground-language-model

Understand how to ground your language model

Completed

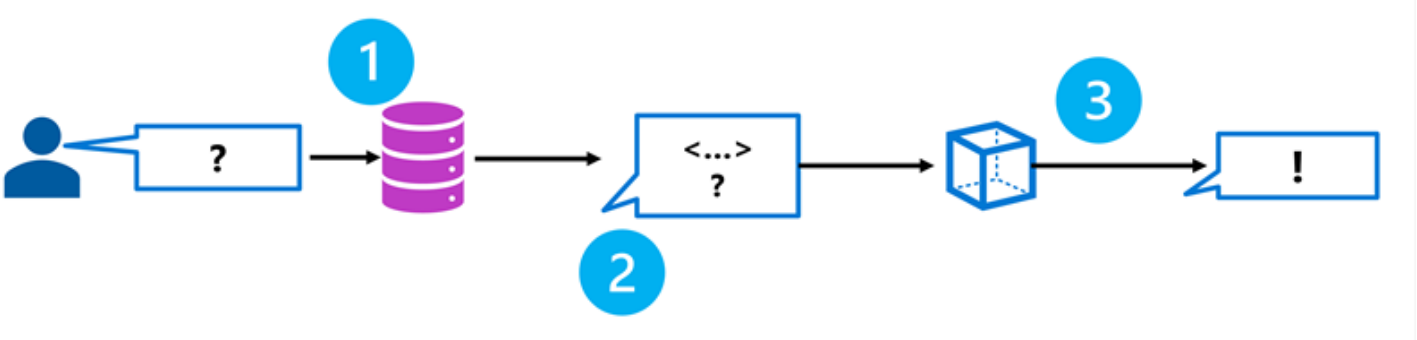
- 6 minutes

Language models excel in generating engaging text, and are ideal as the base for agents. Agents provide users with an intuitive chat-based application to receive assistance in their work. When designing an agent for a specific use case, you want to ensure your language model is grounded and uses factual information that is relevant to what the user needs.

Though language models are trained on a vast amount of data, they may not have access to the knowledge you want to make available to your users. To ensure that an agent is grounded on specific data to provide accurate and domain-specific responses, you can use **Retrieval Augmented Generation** (**RAG**).

Understanding RAG

RAG is a technique that you can use to ground a language model. In other words, it's a process for retrieving information that is relevant to the user's initial prompt. In general terms, the RAG pattern incorporates the following steps:



1. **Retrieve** grounding data based on the initial user-entered prompt.

2. **Augment** the prompt with grounding data.
3. Use a language model to **generate** a grounded response.

By retrieving context from a specified data source, you ensure that the language model uses relevant information when responding, instead of relying on its training data.

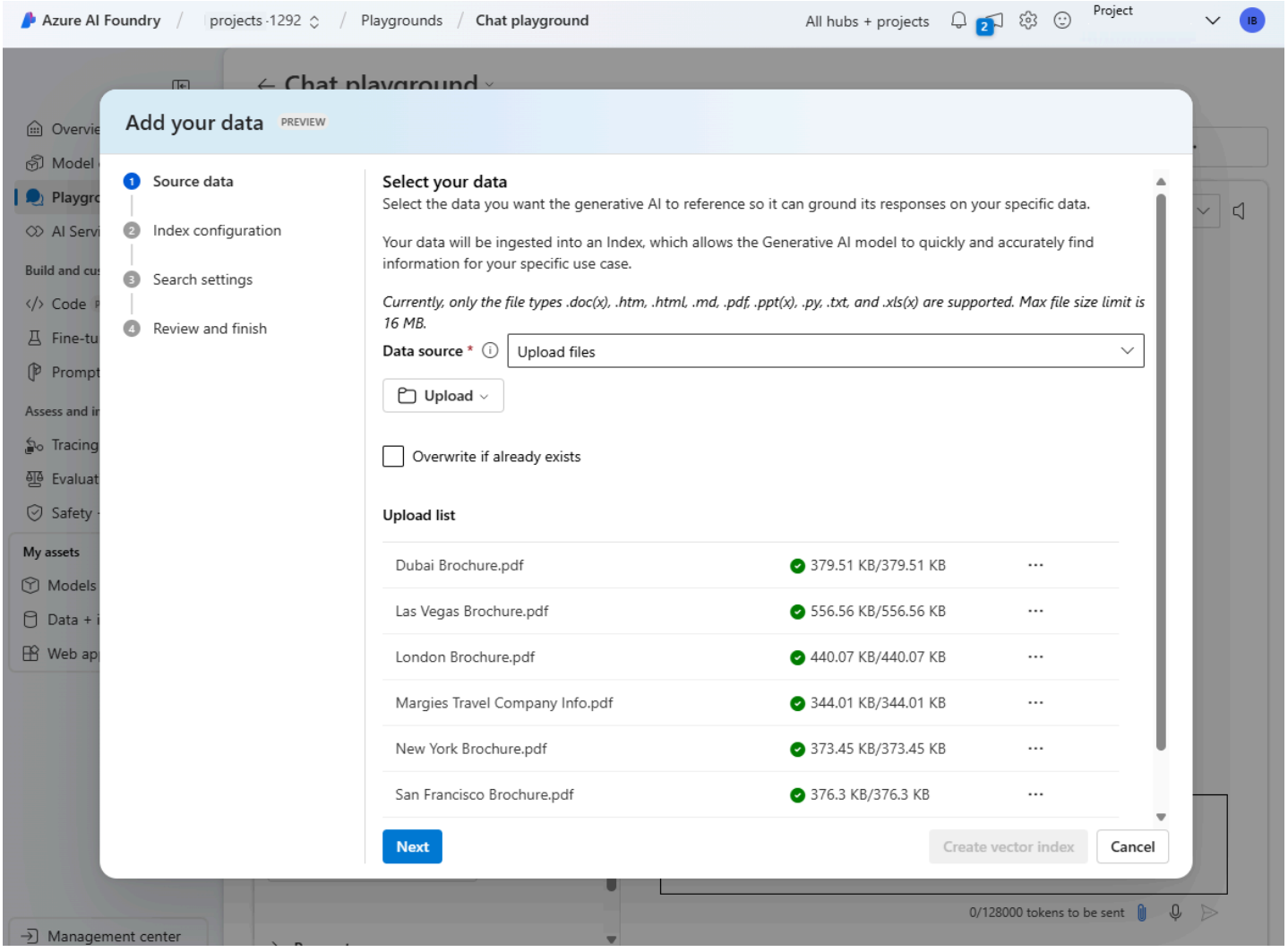
Using RAG is a powerful and easy-to-use technique for many cases in which you want to ground your language model and improve the factual accuracy of your generative AI app's responses.

Adding grounding data to an Azure AI project

You can use Microsoft Foundry to build a custom agent that uses your own data to ground prompts. Microsoft Foundry supports a range of data connections that you can use to add data to a project, including:

- Azure Blob Storage
- Azure Data Lake Storage Gen2
- Microsoft OneLake

You can also upload files or folders to the storage used by your AI Foundry project.



3. Make your data searchable

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/3-search-data

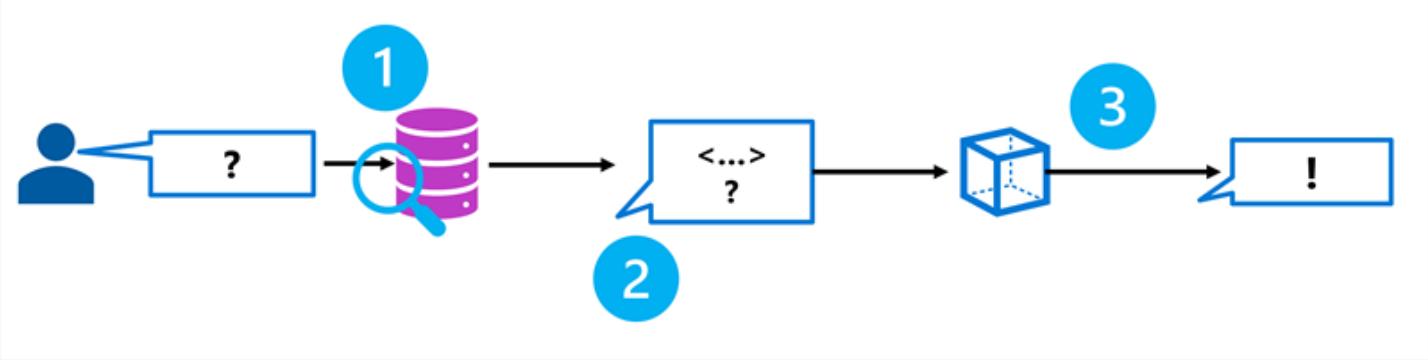
Make your data searchable

Completed

- 7 minutes

When you want to create an agent that uses your own data to generate accurate answers, you need to be able to search your data efficiently. When you build an agent with the Microsoft Foundry, you can use the integration with **Azure AI Search** to retrieve the relevant context in your chat flow.

Azure AI Search is a **retriever** that you can include when building a language model application with prompt flow. Azure AI Search allows you to bring your own data, index your data, and query the index to retrieve any information you need.



Using a *vector* index

While a text-based index will improve search efficiency, you can usually achieve a better data retrieval solution by using a *vector*-based index that contains *embeddings* that represent the text tokens in your data source.

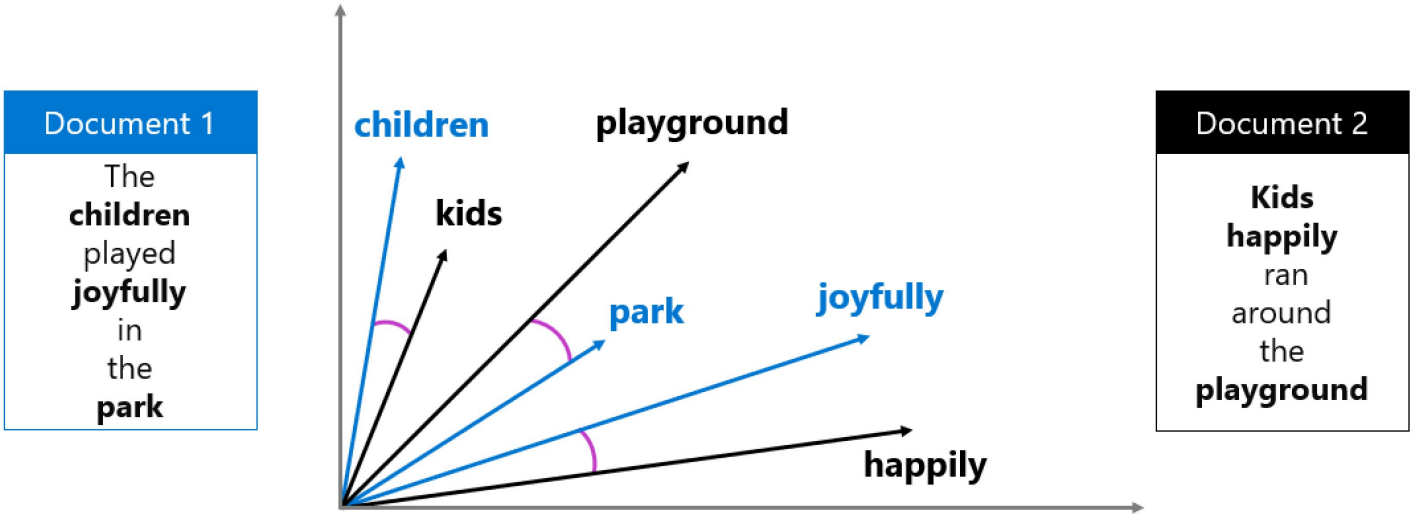
An embedding is a special format of data representation that a search engine can use to easily find the relevant information. More specifically, an embedding is a vector of floating-point numbers.

For example, imagine you have two documents with the following contents:

- "*The children played joyfully in the park.*"
- "*Kids happily ran around the playground.*"

These two documents contain texts that are semantically related, even though different words are used. By creating vector embeddings for the text in the documents, the relation between the words in the text can be mathematically calculated.

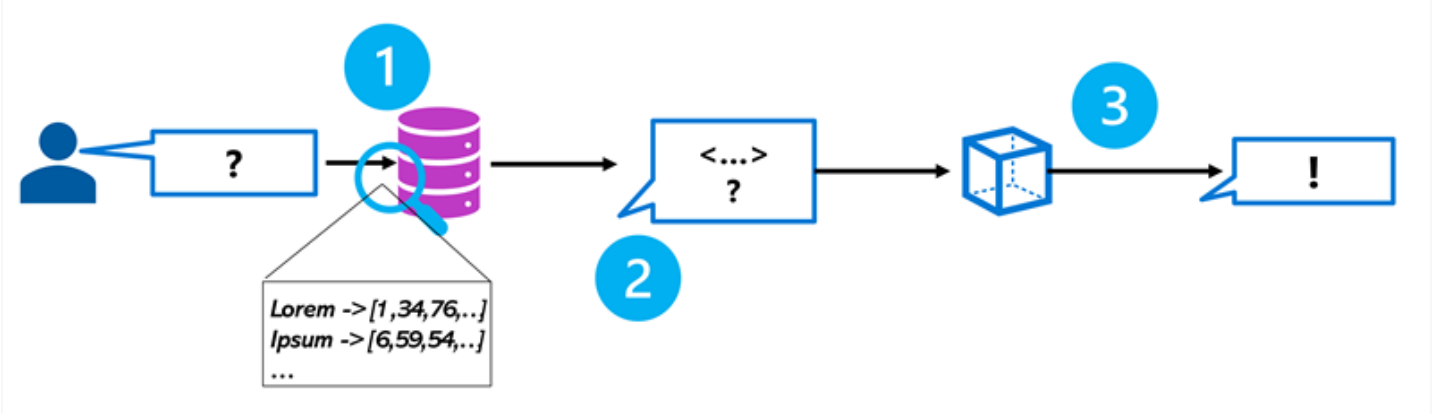
Imagine the keywords being extracted from the document and plotted as a vector in a multidimensional space:



The distance between vectors can be calculated by measuring the cosine of the angle between two vectors, also known as the *cosine similarity*. In other words, the cosine similarity computes the semantic similarity between documents and a query.

By representing words and their meanings with vectors, you can extract relevant context from your data source even when your data is stored in different formats (text or image) and languages.

When you want to be able to use vector search to search your data, you need to create embeddings when creating your search index. To create embeddings for your search index, you can use an Azure OpenAI embedding model available in Microsoft Foundry.



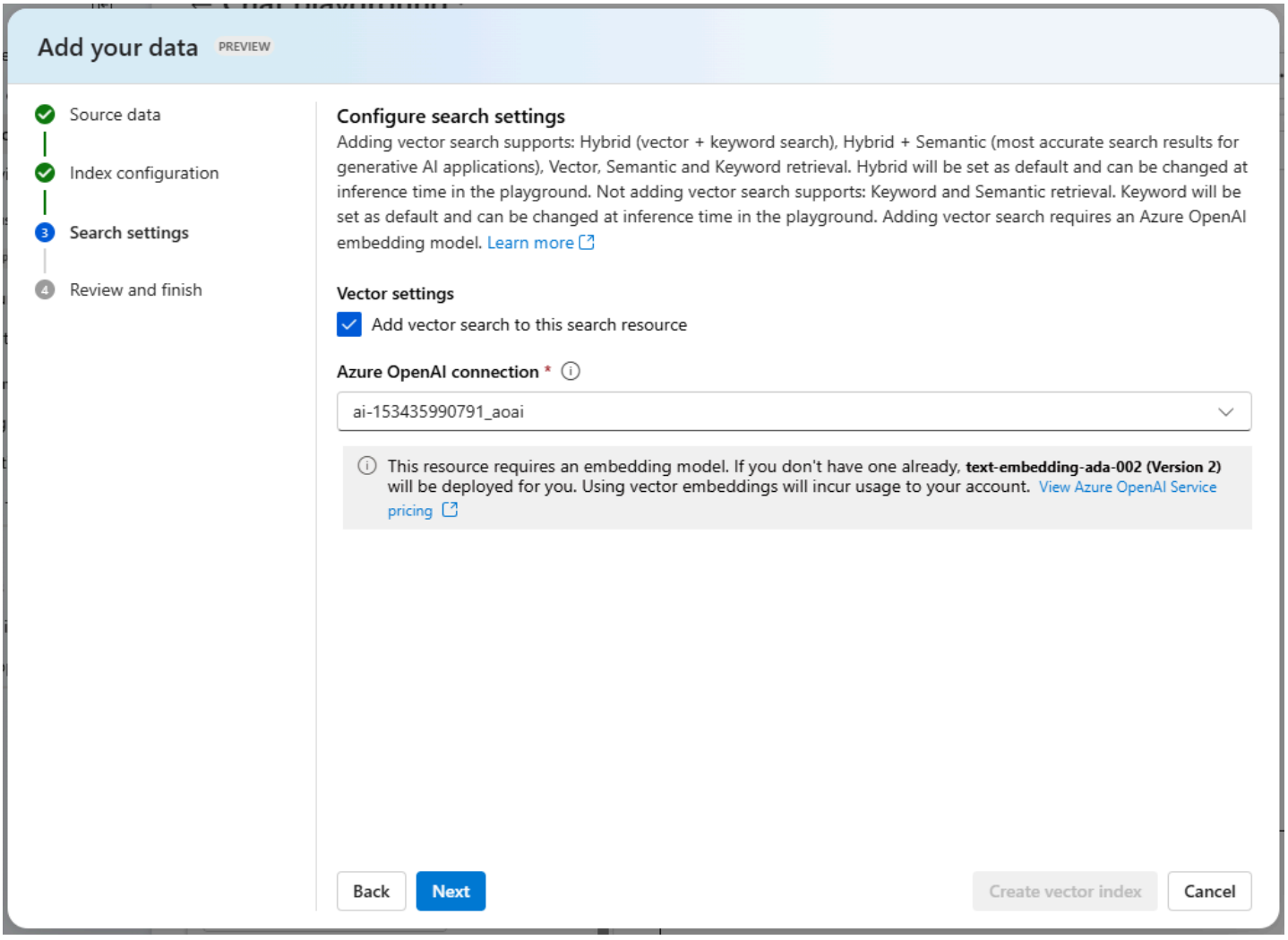
Tip

> Learn more about embeddings in the Azure OpenAI in Foundry Models.

Creating a search index

In Azure AI Search, a **search index** describes how your content is organized to make it searchable. Imagine a library containing many books. You want to be able to search through the library and retrieve the relevant book easily and efficiently. To make the library searchable, you create a catalog that contains any relevant data about books to make any book easy to find. A library's catalog serves as the search index.

Though there are different approaches to creating an index, the integration of Azure AI Search in Microsoft Foundry makes it easy for you to create an index that is suitable for language models. You can add your data to Microsoft Foundry, after which you can use Azure AI Search to create an index in the Microsoft Foundry portal using an embedding model. The index asset is stored in Azure AI Search and queried by Microsoft Foundry when used in a chat flow.



How you configure your search index depends on the data you have and the context you want your language model to use. For example, **keyword search** enables you to retrieve information that exactly matches the search query. **Semantic search** already takes it one step further by retrieving

information that matches the meaning of the query instead of the exact keyword, using semantic models. Currently, the most advanced technique is **vector search**, which creates embeddings to represent your data.

> **Tip**
>
> Learn more about vector search.

Searching an index

There are several ways that information can be queried in an index:

- **Keyword search**: Identifies relevant documents or passages based on specific keywords or terms provided as input.
- **Semantic search**: Retrieves documents or passages by understanding the meaning of the query and matching it with semantically related content rather than relying solely on exact keyword matches.
- **Vector search**: Uses mathematical representations of text (vectors) to find similar documents or passages based on their semantic meaning or context.
- **Hybrid search**: Combines any or all of the other search techniques. Queries are executed in parallel and are returned in a unified result set.

When you create a search index in Microsoft Foundry, you're guided to configuring an index that is most suitable to use in combination with a language model. When your search results are used in a generative AI application, hybrid search gives the most accurate results.

Hybrid search is a combination of keyword (and full text), and vector search, to which semantic ranking is optionally added. When you create an index that is compatible with hybrid search, the retrieved information is precise when exact matches are available (using keywords), and still relevant when only conceptually similar information can be found (using vector search).

> **Tip**
>
> Learn more about hybrid search.

4. Create a RAG-based client application

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/3b-openai-client

Create a RAG-based client application

Completed

- 7 minutes

When you've created an Azure AI Search index for your contextual data, you can use it with an OpenAI model. To ground prompts with data from your index, the Azure OpenAI SDK supports extending the request with connection details for the index.

The following Python code example shows how to implement this pattern.

```
 from openai import AzureOpenAI

# Get an Azure OpenAI chat client
chat_client = AzureOpenAI(
    api_version = "2024-12-01-preview",
    azure_endpoint = open_ai_endpoint,
    api_key = open_ai_key
)

# Initialize prompt with system message
prompt = [
    {"role": "system", "content": "You are a helpful AI assistant."}
]

# Add a user input message to the prompt
input_text = input("Enter a question: ")
prompt.append({"role": "user", "content": input_text})

# Additional parameters to apply RAG pattern using the AI Search index
rag_params = {
    "data_sources": [
        {
            "type": "azure_search",
            "parameters": {
                "endpoint": search_url,
                "index_name": "index_name",
                "authentication": {
                    "type": "api_key",
                    "key": search_key,
                }
            }
        }
    ],
```

```
}

# Submit the prompt with the index information
response = chat_client.chat.completions.create(
    model="<model_deployment_name>",
    messages=prompt,
    extra_body=rag_params
)

# Print the contextualized response
completion = response.choices[0].message.content
print(completion)
```

In this example, the search against the index is *keyword-based* - in other words, the query consists of the text in the user prompt, which is matched to text in the indexed documents. When using an index that supports it, an alternative approach is to use a *vector-based* query in which the index and the query use numeric vectors to represent text tokens. Searching with vectors enables matching based on semantic similarity as well as literal text matches.

To use a vector-based query, you can modify the specification of the Azure AI Search data source details to include an embedding model; which is then used to vectorize the query text.

```
rag_params = {
    "data_sources": [
        {
            "type": "azure_search",
            "parameters": {
                "endpoint": search_url,
                "index_name": "index_name",
                "authentication": {
                    "type": "api_key",
                    "key": search_key,
                },
                # Params for vector-based query
                "query_type": "vector",
                "embedding_dependency": {
                    "type": "deployment_name",
                    "deployment_name": "<embedding_model_deployment_name>",
                },
            }
        }
    ],
}
```

5. Implement RAG in a prompt flow

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/4-build-copilot

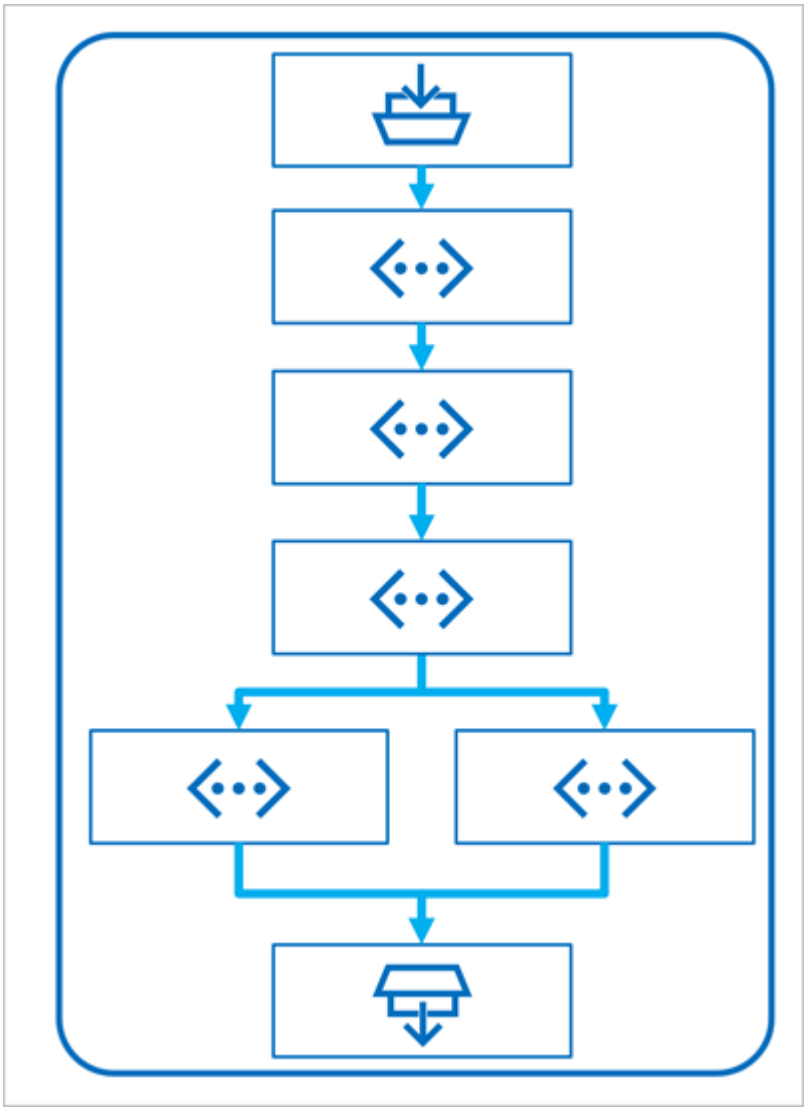
Implement RAG in a prompt flow

Completed

- 7 minutes

After uploading data to Microsoft Foundry and creating an index on your data using the integration with Azure AI Search, you can implement the RAG pattern with *Prompt Flow* to build a generative AI application.

Prompt Flow is a development framework for defining flows that orchestrate interactions with an LLM.



A flow begins with one or more *inputs*, usually a question or prompt entered by a user, and in the case of iterative conversations the chat history to this point.

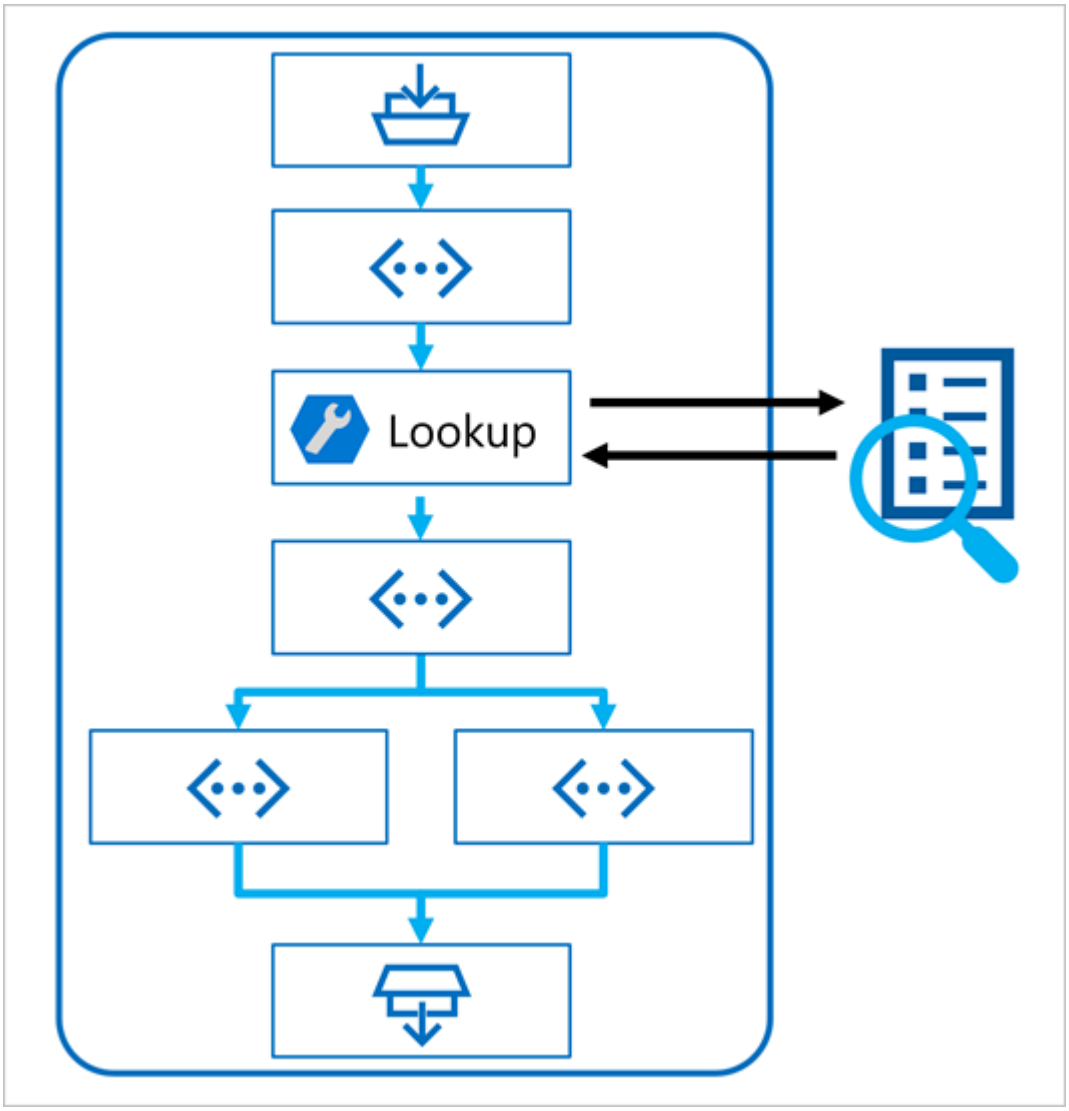
The flow is then defined as a series of connected *tools*, each of which performs a specific operation on the inputs and other environmental variables. There are multiple types of tool that you can include in a prompt flow to perform tasks such as:

- Running custom Python code
- Looking up data values in an index
- Creating prompt variants - enabling you to define multiple versions of a prompt for a large language model (LLM), varying system messages or prompt wording, and compare and evaluate the results from each variant.
- Submitting a prompt to an LLM to generate results.

Finally, the flow has one or more *outputs*, typically to return the generated results from an LLM.

Using the RAG pattern in a prompt flow

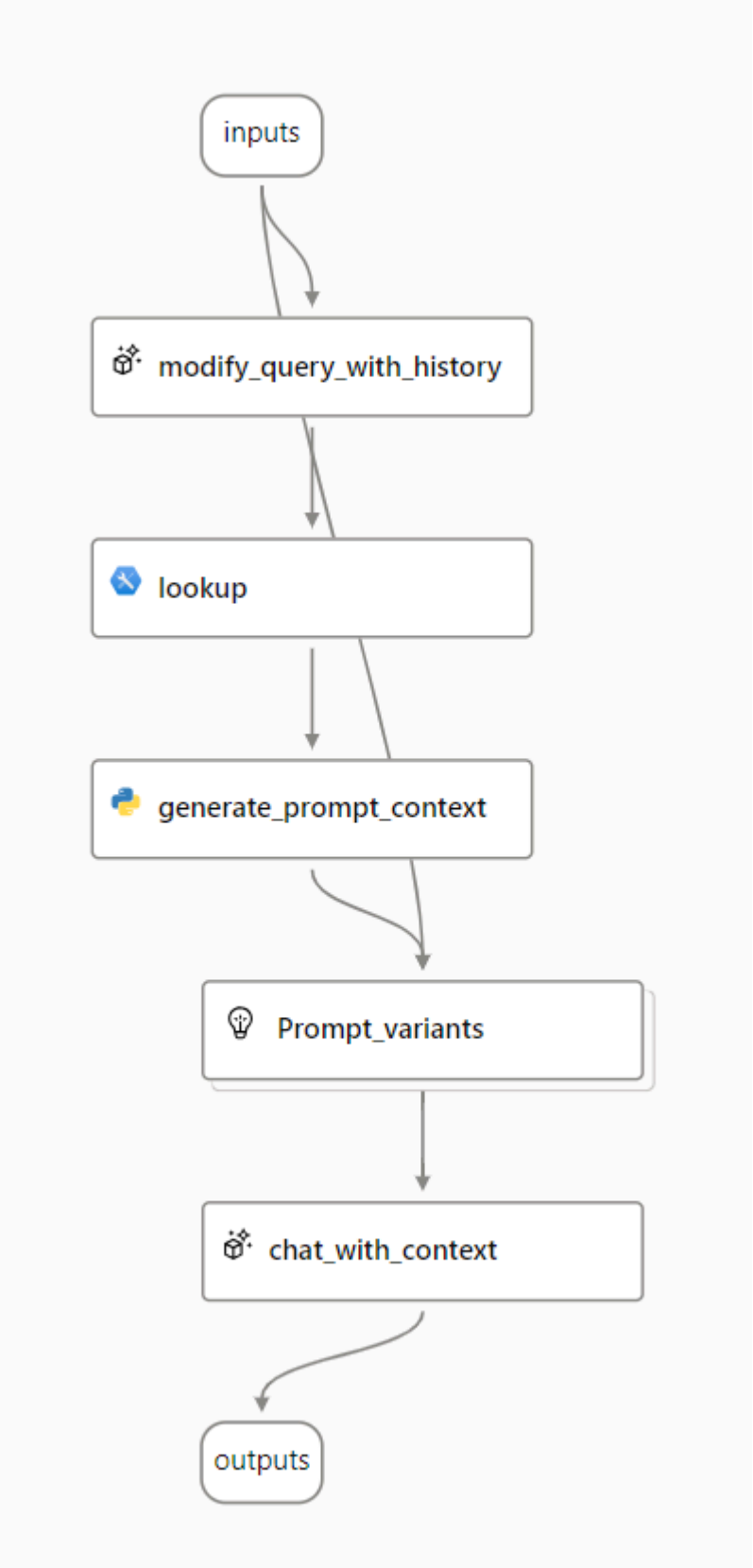
The key to using the RAG pattern in a prompt flow is to use an Index Lookup tool to retrieve data from an index so that subsequent tools in the flow can use the results to augment the prompt used to generate output from an LLM.



Use a sample to create a chat flow

Prompt flow provides various samples you can use as a starting point to create an application. When you want to combine RAG and a language model in your application, you can clone the **Multi-round Q&A on your data** sample.

The sample contains the necessary elements to include RAG and a language model:



1. Append the history to the chat input to define a prompt in the form of a contextualized form of a question.
2. Look up relevant information from your data using your search index.
3. Generate the prompt context by using the retrieved data from the index to augment the question.
4. Create prompt variants by adding a system message and structuring the chat history.
5. Submit the prompt to a language model that generates a natural language response.

Let's explore each of these elements in more detail.

Modify query with history

The first step in the flow is a Large Language Model (LLM) node that takes the chat history and the user's last question and generates a new question that includes all necessary information. By doing so, you generate more succinct input that is processed by the rest of the flow.

Look up relevant information

Next, you use the Index Lookup tool to query the search index you created with the integrated Azure AI Search feature and find the relevant information from your data source.

> **Tip**
>
> Learn more about the Index Lookup tool.

Generate prompt context

The output of the Index Lookup tool is the retrieved context you want to use when generating a response to the user. You want to use the output in a prompt that is sent to a language model, which means you want to parse the output into a more suitable format.

The output of the Index Lookup tool can include the top *n* results (depending on the parameters you set). When you generate the prompt context, you can use a Python node to iterate over the retrieved documents from your data source and combine their contents and sources into one document string. The string will be used in the prompt you send to the language model in the next step of the flow.

Define prompt variants

When you construct the prompt you want to send to your language model, you can use variants to represent different prompt contents.

When including RAG in your chat flow, your goal is to ground the chatbot's responses. Next to retrieving relevant context from your data source, you can also influence the groundedness of the chatbot's response by instructing it to use the context and aim to be factual.

With the prompt variants, you can provide varying system messages in the prompt to explore which content provides the most groundedness.

Chat with context

Finally, you use an LLM node to send the prompt to a language model to generate a response using the relevant context retrieved from your data source. The response from this node is also the output of the entire flow.

After configuring the sample chat flow to use your indexed data and the language model of your choosing, you can deploy the flow and integrate it with an application to offer users an agentic experience.

6. Exercise - Create a generative AI app that uses your own data

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/5-exercise

Exercise - Create a generative AI app that uses your own data

Completed

- 45 minutes

If you have an Azure subscription, you can use Microsoft Foundry to create a RAG-based agent for yourself.

> **Note**
>
> If you don't have an Azure subscription, and you want to explore Microsoft Foundry, you can sign up for an account, which includes credits for the first 30 days.

Launch the exercise and follow the instructions.

Launch Exercise

7. Module assessment

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/6-knowledge-check

Module assessment

Completed

- 3 minutes

8. Summary

https://learn.microsoft.com/en-us/training/modules/build-copilot-ai-studio/7-summary

Summary

Completed

- 1 minute

In this module, you learned to:

- Identify the need to ground your language model with Retrieval Augmented Generation (RAG).
- Index your data with Azure AI Search to make it searchable for language models.
- Build an agent using RAG on your own data in Microsoft Foundry.

Learn more

- Model catalog and collections in Microsoft Foundry portal
- Deploy AI models in Microsoft Foundry portal
- Prompt engineering techniques
- Microsoft Foundry Discord
- Microsoft Foundry Developer Forum