Identifying and extracting text with Document Intelligence in Microsoft Fabric

Introduction

The analysis of structured data has been well-established. However, unstructured data – including text, images, and videos – has historically posed significant analytical hurdles. The development of advanced AI models, particularly OpenAI's GPT-3 and GPT-4, is revolutionizing this landscape, enabling easier analysis and the extraction of valuable insights from unstructured data. A key illustration of this progress is the ability to query documents in natural language, a capability enabled by the synergy of information retrieval and language generation.

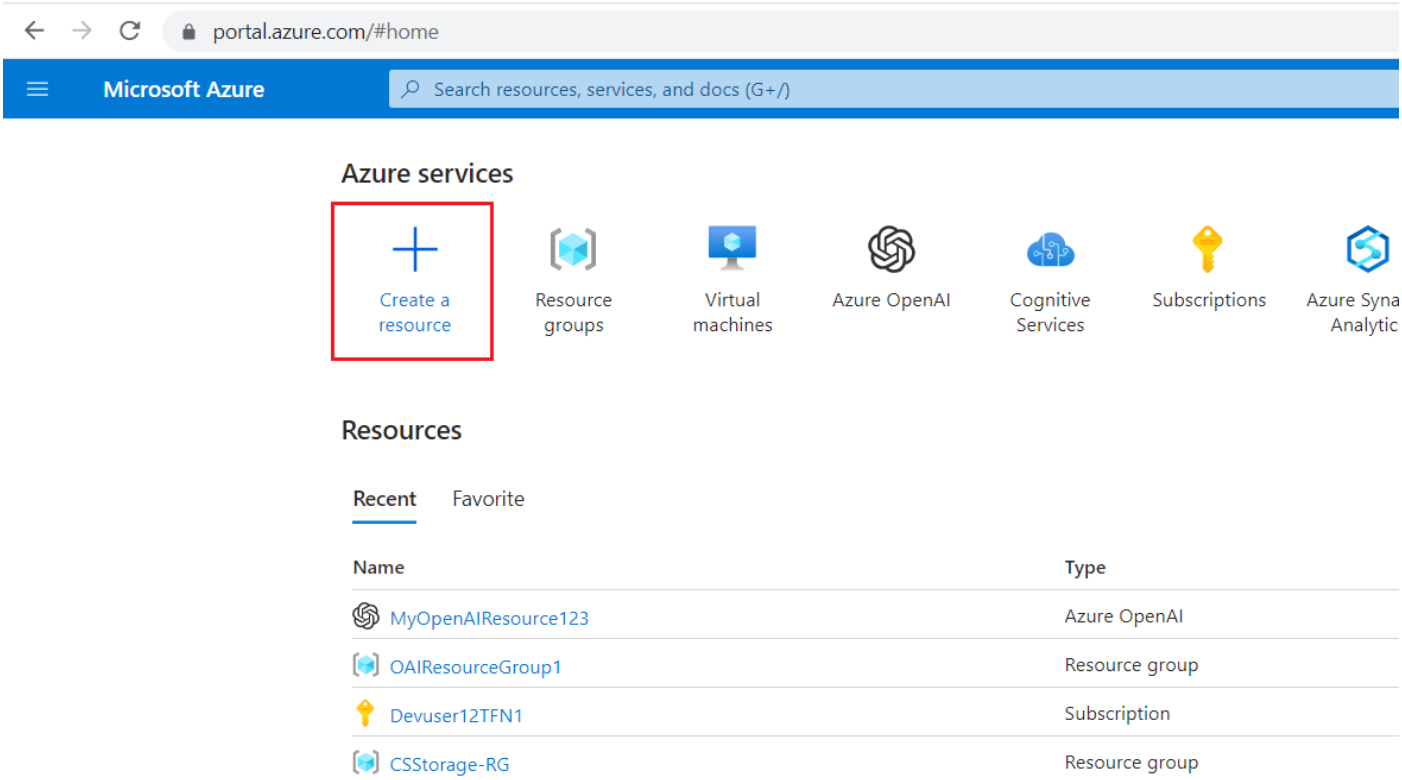
Objective

- Pre-process PDF Documents using Azure AI Document Intelligence in Azure AI Services.
- Perform text chunking using SynapseML.
- Generate embeddings for the chunks using SynapseML and Azure OpenAI Services.
- Store the embeddings in Azure AI Search.
- Build a question-answering pipeline.

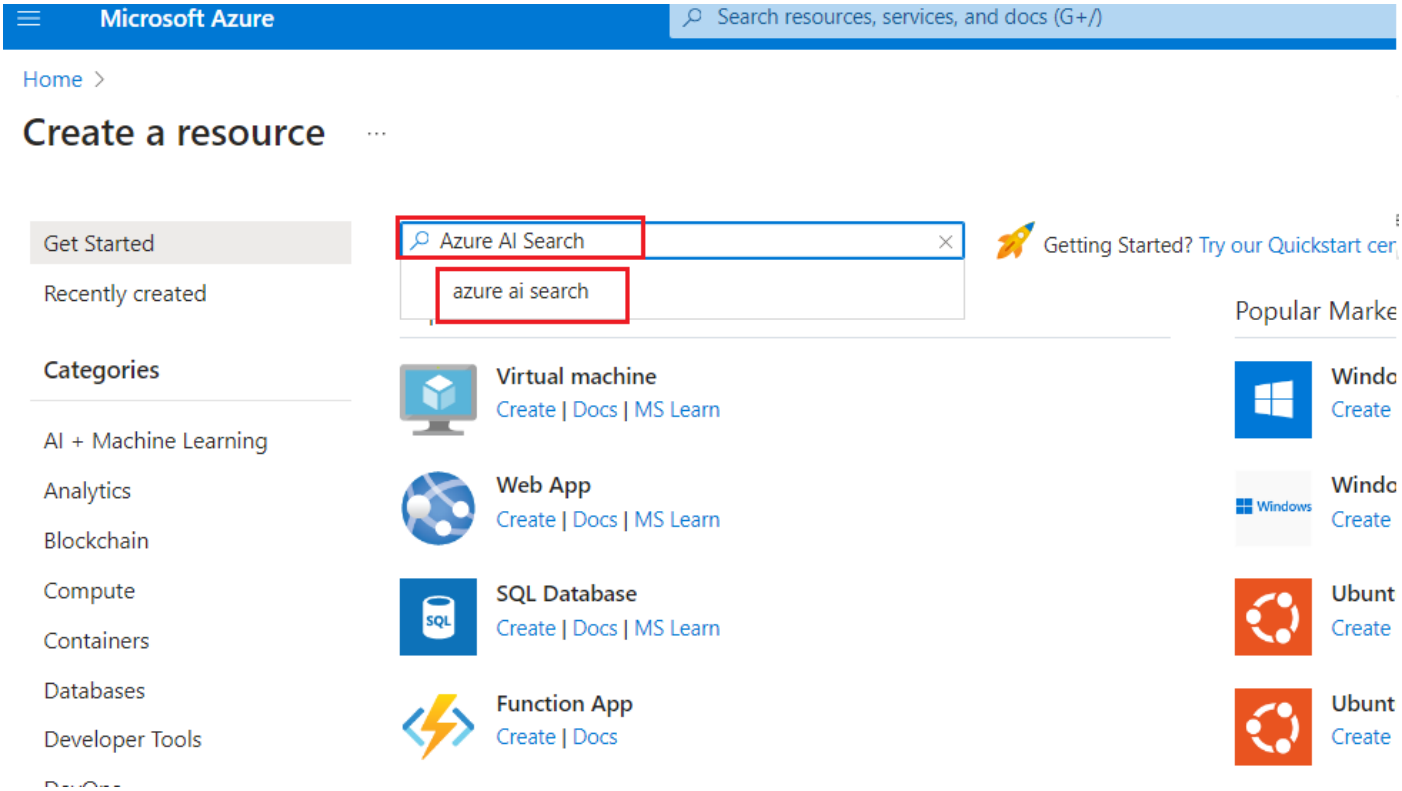
Exercise 1: Setup

Task 1: Create an Azure AI Search service in the portal

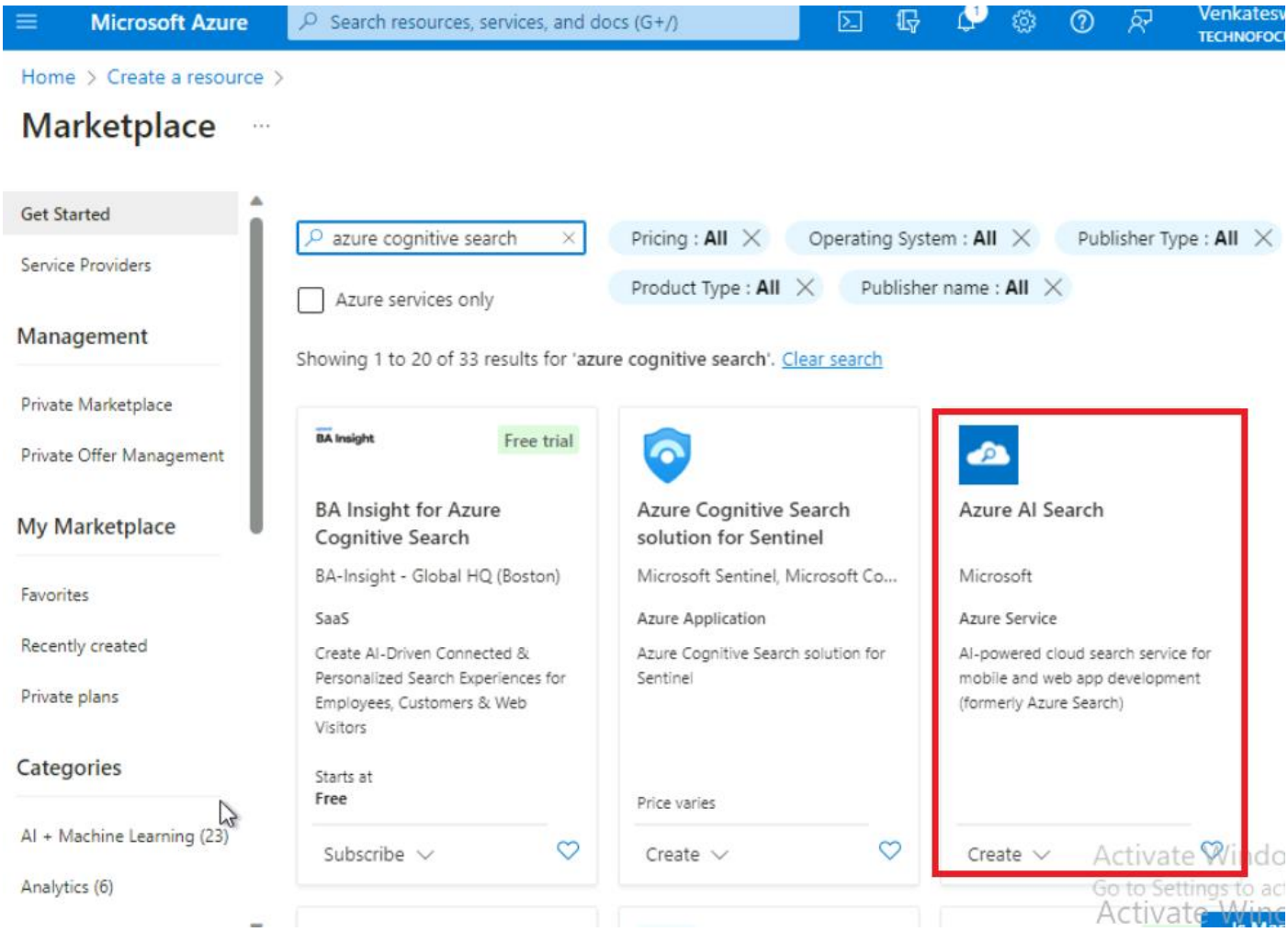
1. In Azure portal home page, click on **+ Create Resource**.



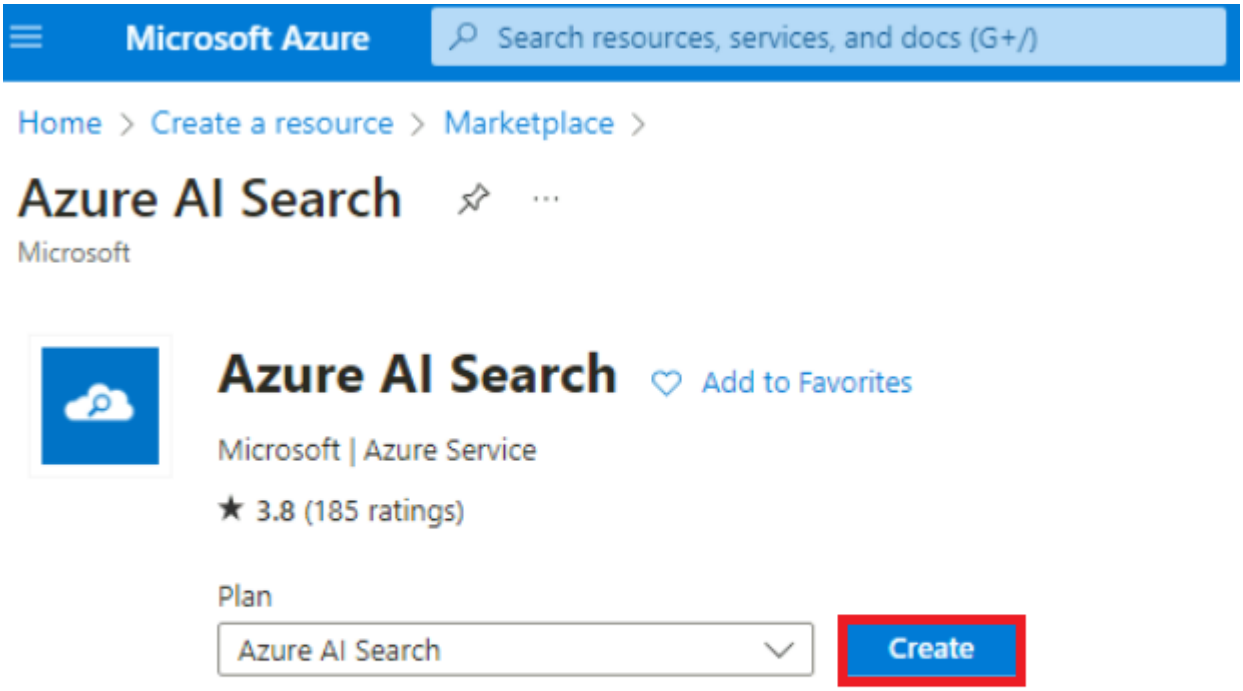
2. In the **Create a resource** page search bar, type **Azure AI Search** and click on the appeared **azure ai search**.



3. Click on **azure ai search** section.



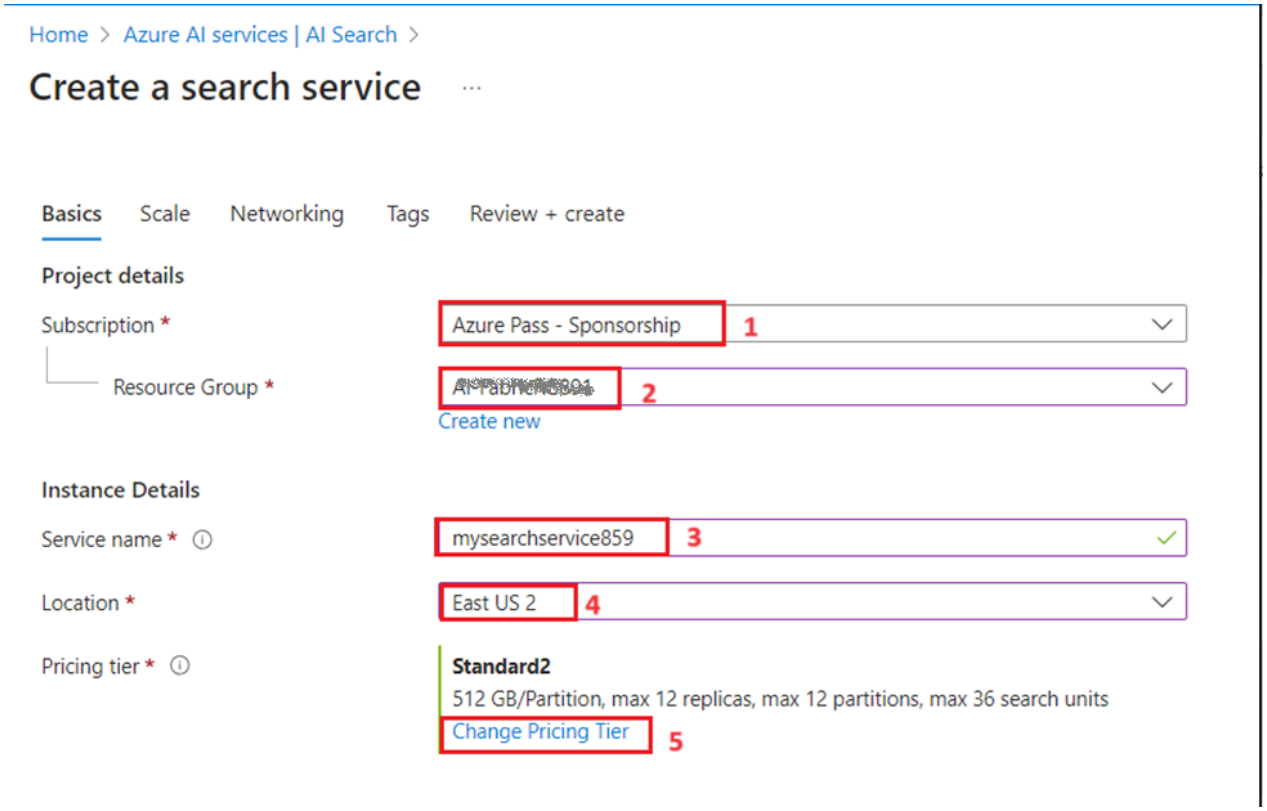
4. In the **Azure AI Search** page, click on the **Create** button.



5. On the **Create a search service** page, provide the following information and click on **Review+create** button.

Field	Description
Subscription	Select your Azure OpenAI subscription
Resource group	Select your Resource group(that you have created in **Lab 1**)
Region	EastUS 2
Name	mysearchserviceXX (XXcan be unique number)
Pricing Tier	Click on change Price Tire>select **Basic**

6.

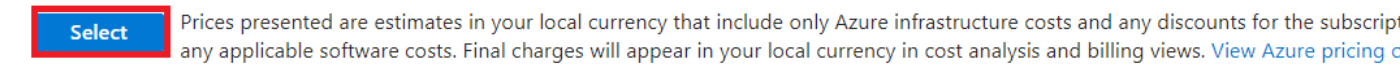


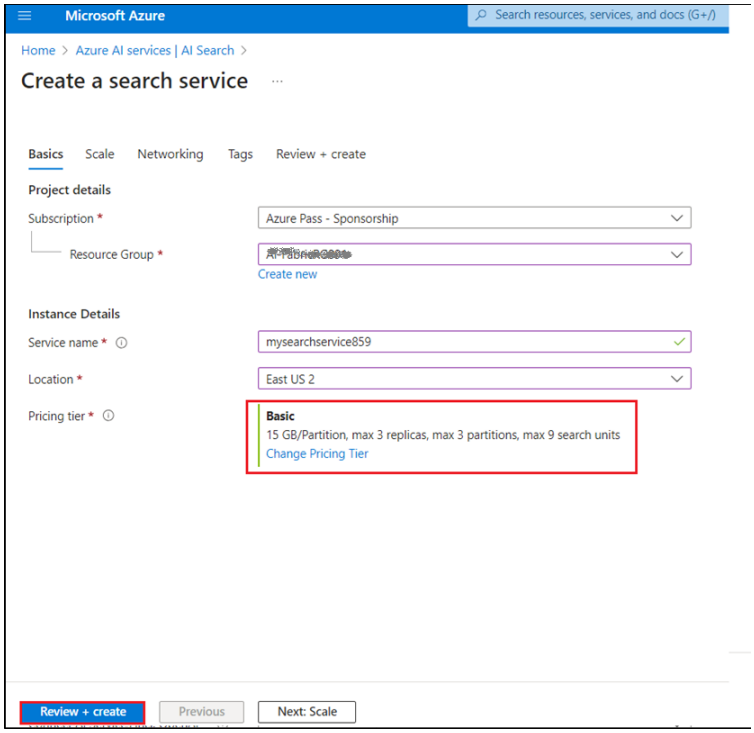
Select Pricing Tier

Browse available skus and their features

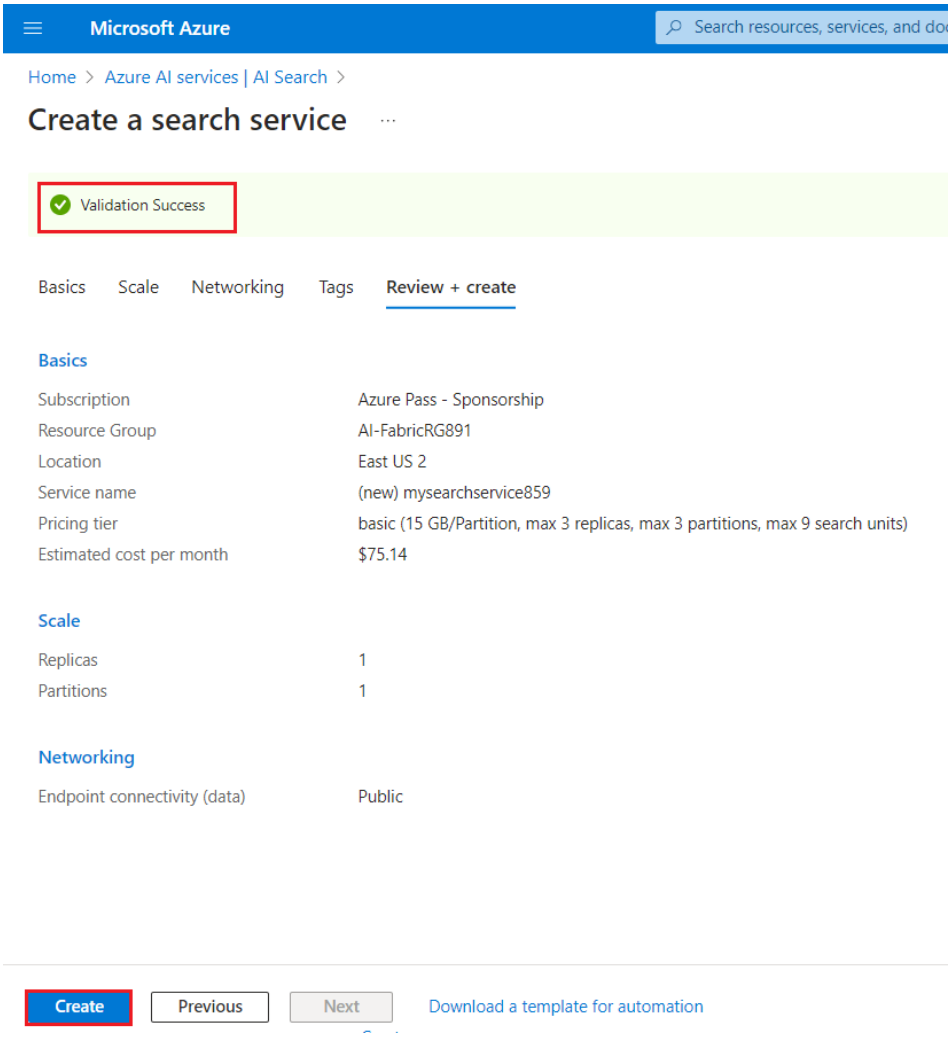
Sku	Offering	Indexes	Indexers	Storage	Search units	Replicas	Partition
F	Free	3	3	50 MB	1	1	1
B	Basic	15	15	2 GB	3	3	1
S	Standard	50	50	25 GB/Partition*	36	12	12
S2	Standard	200	200	100 GB/Partition*	36	12	12
S3	Standard	200	200	200 GB/Partition*	36	12	12
S3HD	High-density	1000	0	200 GB/Partition*	36	12	3
L1	Storage Optimized	10	10	1 TB/Partition*	36	12	12
L2	Storage Optimized	10	10	2 TB/Partition*	36	12	12

7.



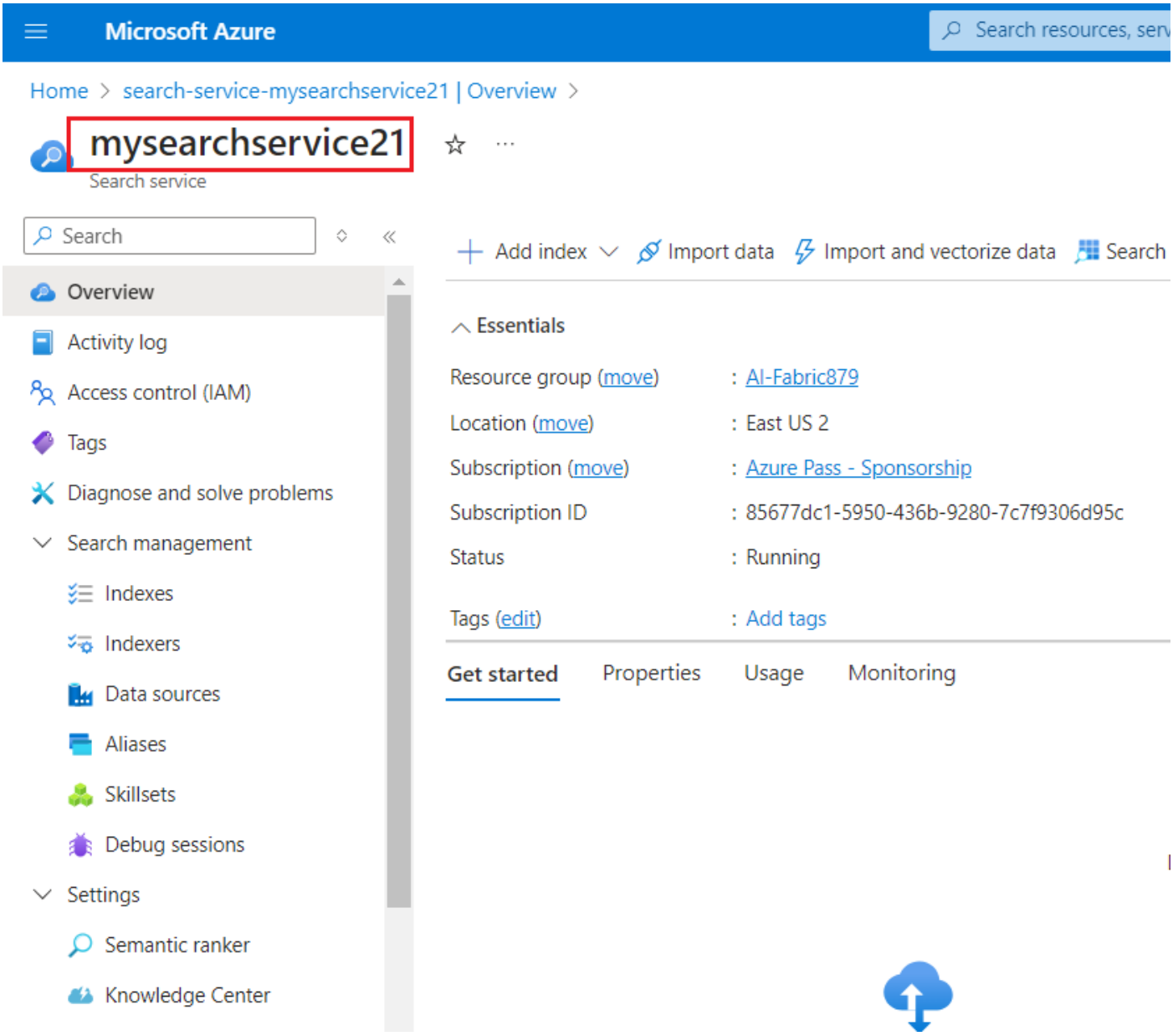


8. Once the Validation is passed, click on the **Create** button.



9. After the deployment is completed, click on the **Go to resource** button.

10. copy **AI search name** and paste them in a notepad as shown in the below image, then **Save** the notepad to use the information in the upcoming lab.



Task : Create a lakehouse

1. New Item, Select **Lakehouse** to create a lakehouse.
2. In the **New lakehouse** dialog box, enter !!**data_lakehouse**!! in the **Name** field, click on the **Create** button and open the new lakehouse.

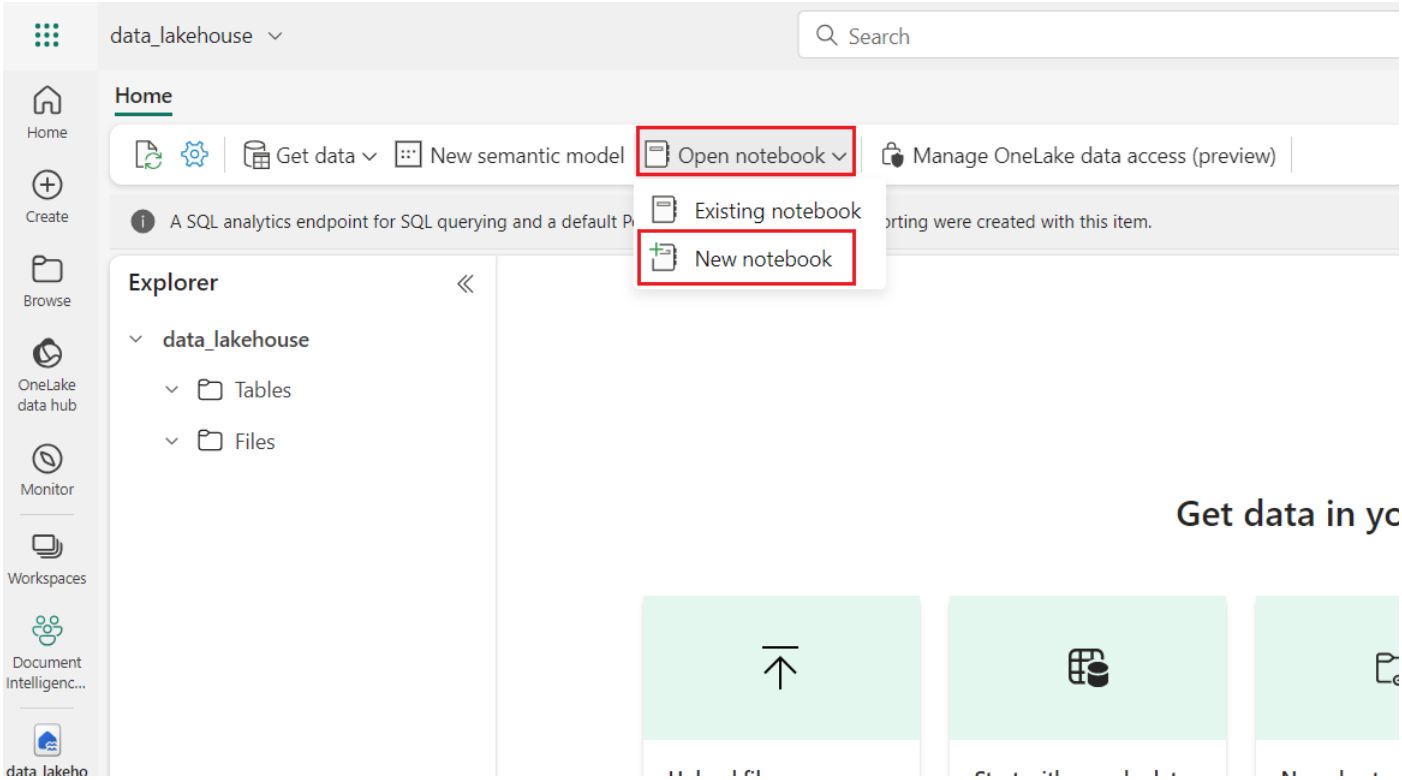
   Note: Ensure to remove space before **data_lakehouse**.
3. You will see a notification stating **Successfully created SQL endpoint**.

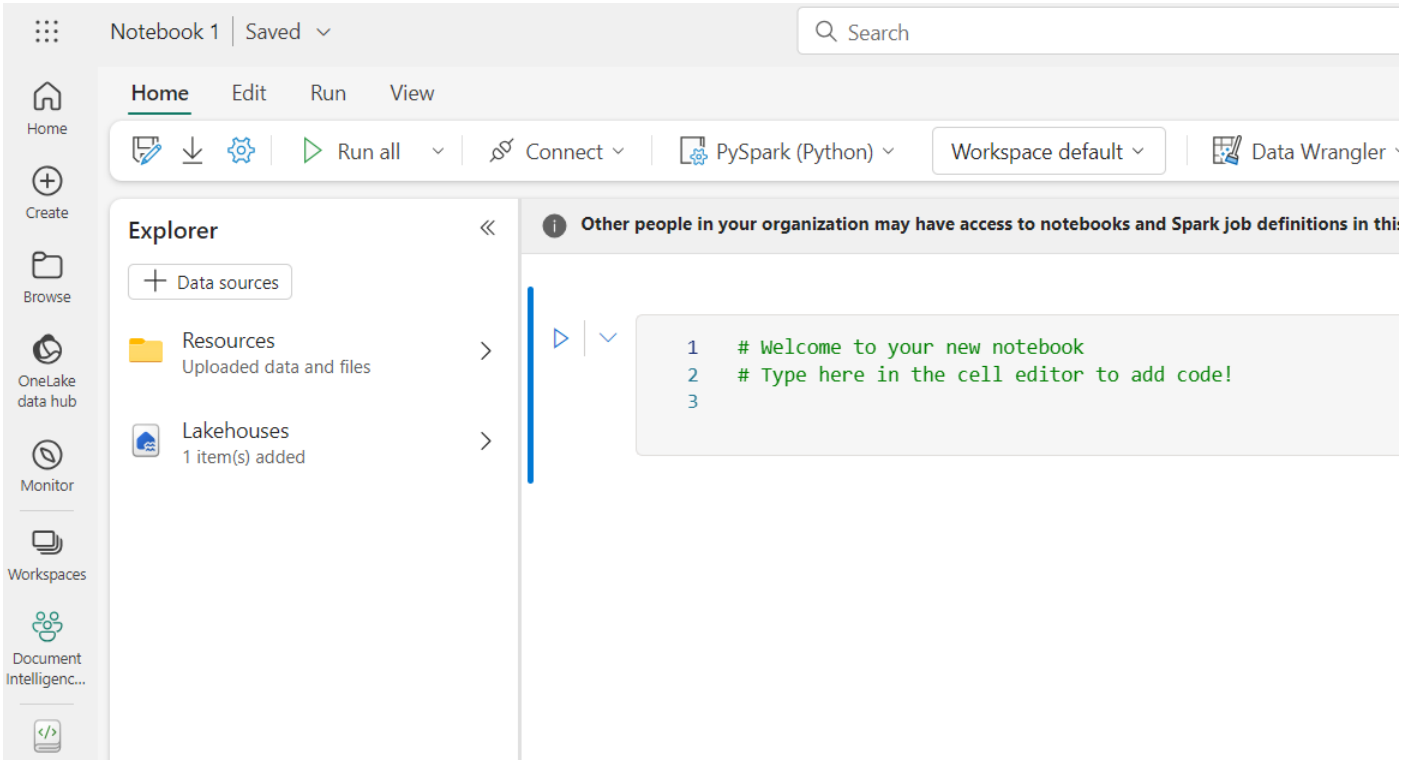
Exercise 2: Loading and Pre-processing PDF Documents

Task 1: Configure Azure API keys

To begin, navigate back to the rag_workshop Lakehouse in your workspace and create a new notebook by selecting Open Notebook and selecting New Notebook from the options.

1. In the **Lakehouse** page, navigate and click on **Open notebook** drop in the command bar, then select **New notebook**.

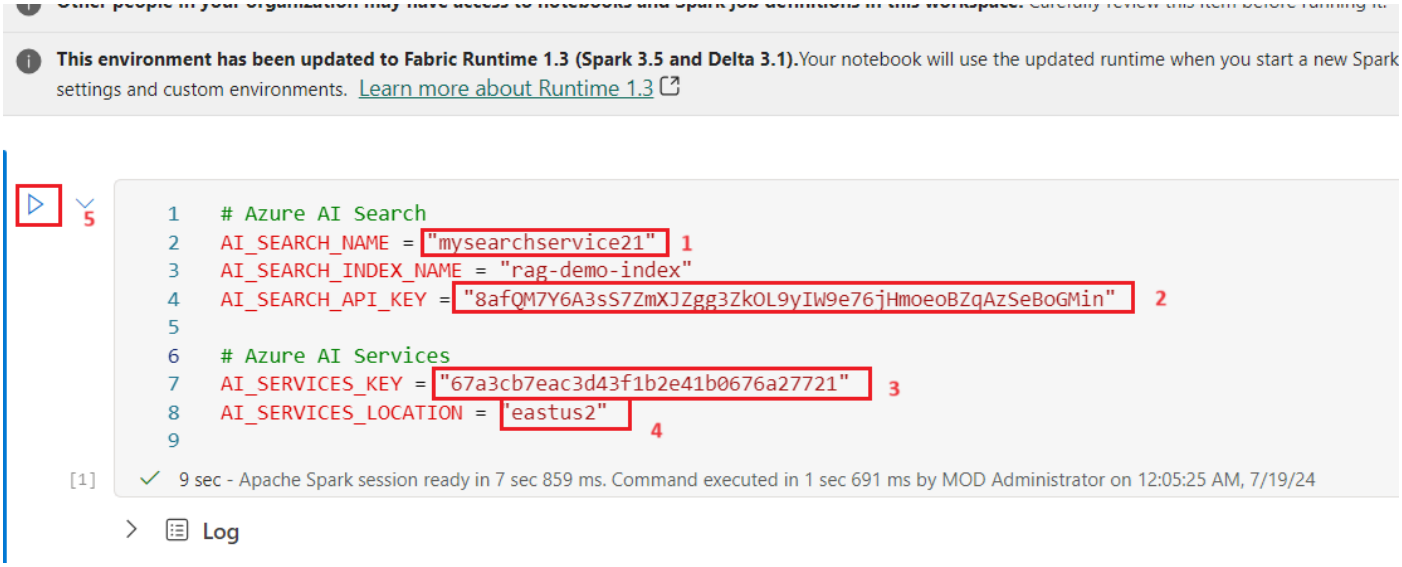




2. In the query editor, paste the following code. Provide the keys for Azure AI Services, Azure Key Vault name and secrets to access the services

Copy

```
# Azure AI Search

AI_SEARCH_NAME = ""

AI_SEARCH_INDEX_NAME = "rag-demo-index"

AI_SEARCH_API_KEY = ""


# Azure AI Services

AI_SERVICES_KEY = ""

AI_SERVICES_LOCATION = ""
```



Task 2: Loading & Analyzing the Document

1. we will be using a specific document named **support.pdf** which will be the source of our data.
2. To download the document, use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
import requests

import os


url = "https://github.com/Azure-Samples/azure-openai-rag-workshop/raw/main/data/support.pdf"

response = requests.get(url)


# Specify your path here

path = "/lakehouse/default/Files/"


# Ensure the directory exists

os.makedirs(path, exist_ok=True)
```

Write the content to a file in the specified path

filename = url.rsplit("/")[-1]

with open(os.path.join(path, filename), "wb") as f:

    f.write(response.content)

3. Now, load the PDF document into a Spark DataFrame using the spark.read.format("binaryFile") method provided by Apache Spark
4. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

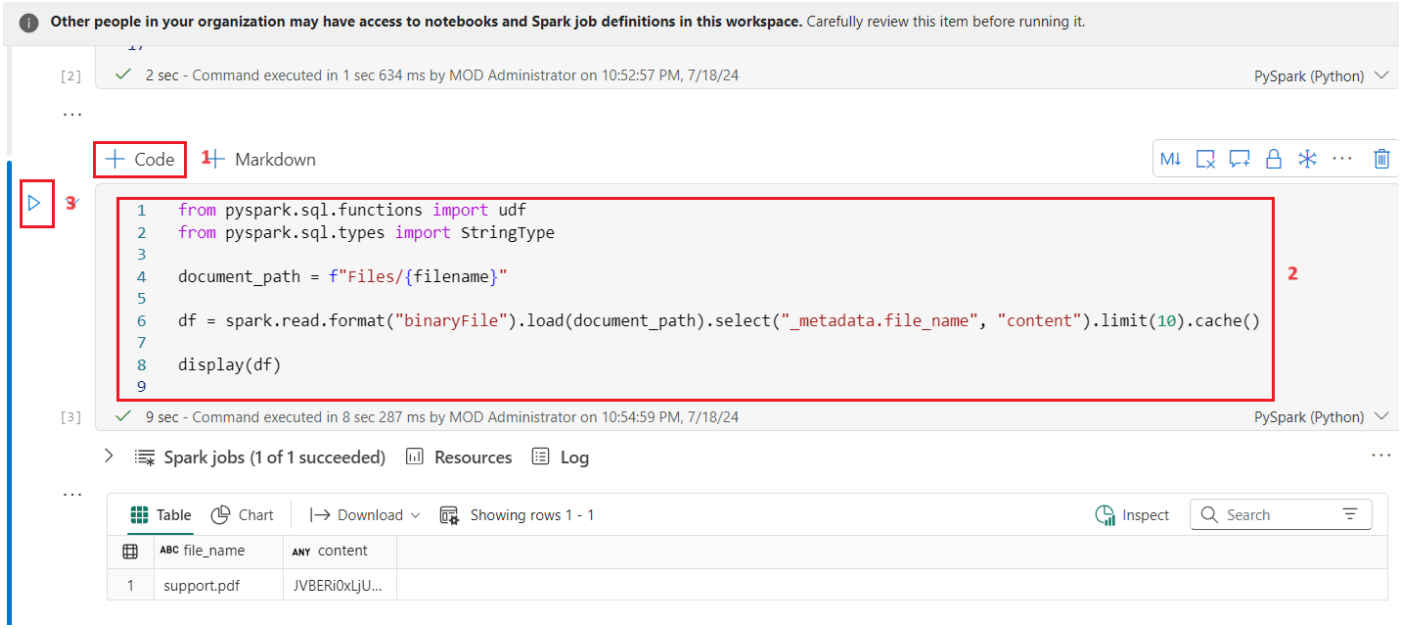
from pyspark.sql.functions import udf

from pyspark.sql.types import StringType

document_path = f"Files/{filename}"

df = spark.read.format("binaryFile").load(document_path).select("_metadata.file_name", "content").limit(10).cache()

display(df)



This code will read the PDF document and create a Spark DataFrame named df with the contents of the PDF. The DataFrame will have a schema that represents the structure of the PDF document, including its textual content.

5. Next, we'll use the Azure AI Document Intelligence to read the PDF documents and extract the text from them.
6. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

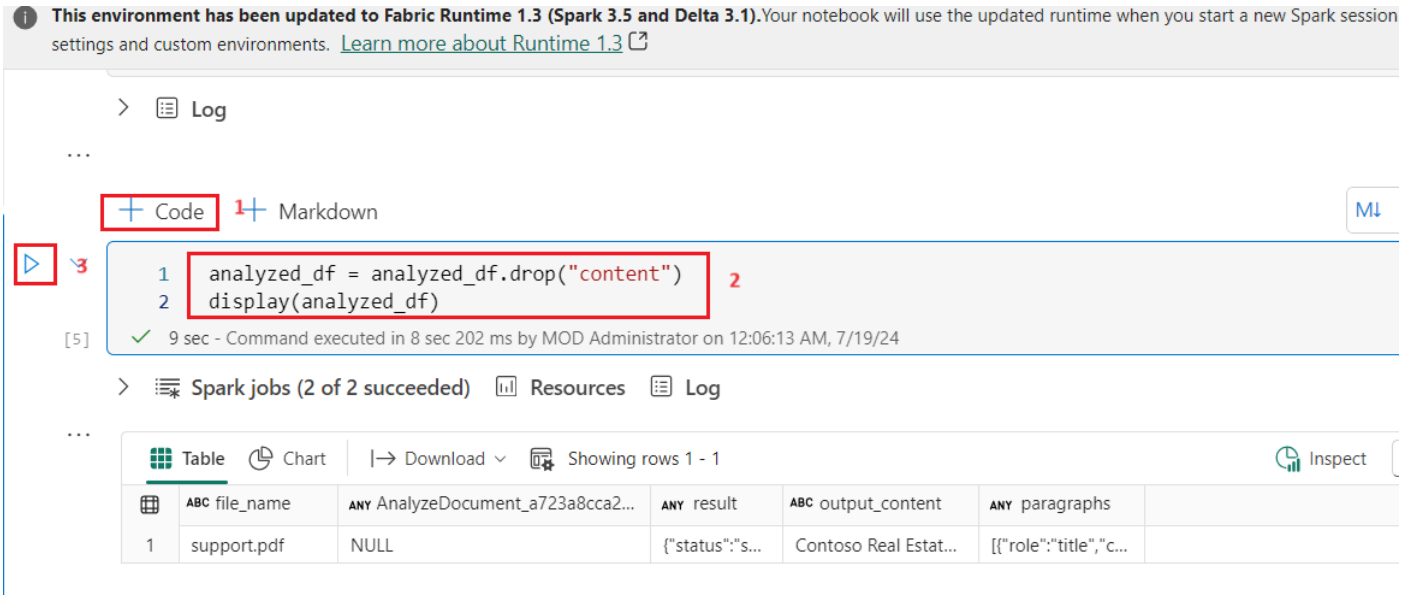
```
from synapse.ml.services import AnalyzeDocument

from pyspark.sql.functions import col


analyze_document = (

   AnalyzeDocument()

   .setPrebuiltModelId("prebuilt-layout")

   .setSubscriptionKey(AI_SERVICES_KEY)

   .setLocation(AI_SERVICES_LOCATION)

   .setImageBytesCol("content")

   .setOutputCol("result")

)


analyzed_df = (

   analyze_document.transform(df)

   .withColumn("output_content", col("result.analyzeResult.content"))

   .withColumn("paragraphs", col("result.analyzeResult.paragraphs"))

).cache()
```

7. We can observe the analyzed Spark DataFrame named analyzed_df using the following code. Note that we drop the content column as it is not needed anymore.

8. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
analyzed_df = analyzed_df.drop("content")

display(analyzed_df)
```



Exercise 3: Generating and Storing Embeddings

Task 1: Text Chunking

Before we can generate the embeddings, we need to split the text into chunks. To do this we leverage SynapseML's PageSplitter to divide the documents into smaller sections, which are subsequently stored in the chunks column. This allows for more granular representation and processing of the document content.

1. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

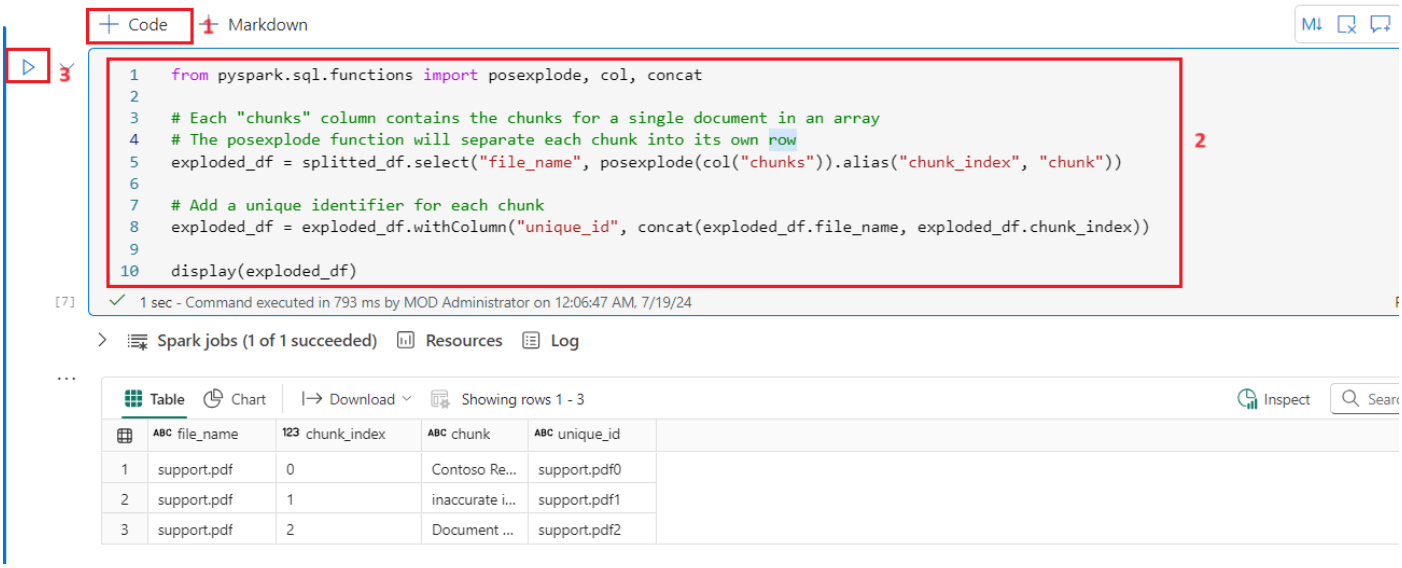
```
from synapse.ml.featurize.text import PageSplitter


ps = (
  PageSplitter()
  .setInputCol("output_content")
  .setMaximumPageLength(4000)
  .setMinimumPageLength(3000)
  .setOutputCol("chunks")
)


splitted_df = ps.transform(analyzed_df)
display(splitted_df)
```

Note that the chunks for each document are presented in a single row inside an array. In order to embed all the chunks in the following cells, we need to have each chunk in a separate row.

2. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
from pyspark.sql.functions import posexplode, col, concat

# Each "chunks" column contains the chunks for a single document in an array
# The posexplode function will separate each chunk into its own row
exploded_df = splitted_df.select("file_name", posexplode(col("chunks")).alias("chunk_index", "chunk"))

# Add a unique identifier for each chunk
exploded_df = exploded_df.withColumn("unique_id", concat(exploded_df.file_name, exploded_df.chunk_index))

display(exploded_df)
```



From this code snippet we first explode these arrays so there is only one chunk in each row, then filter the Spark DataFrame in order to only keep the path to the document and the chunk in a single row.

Task 2: Generating Embeddings

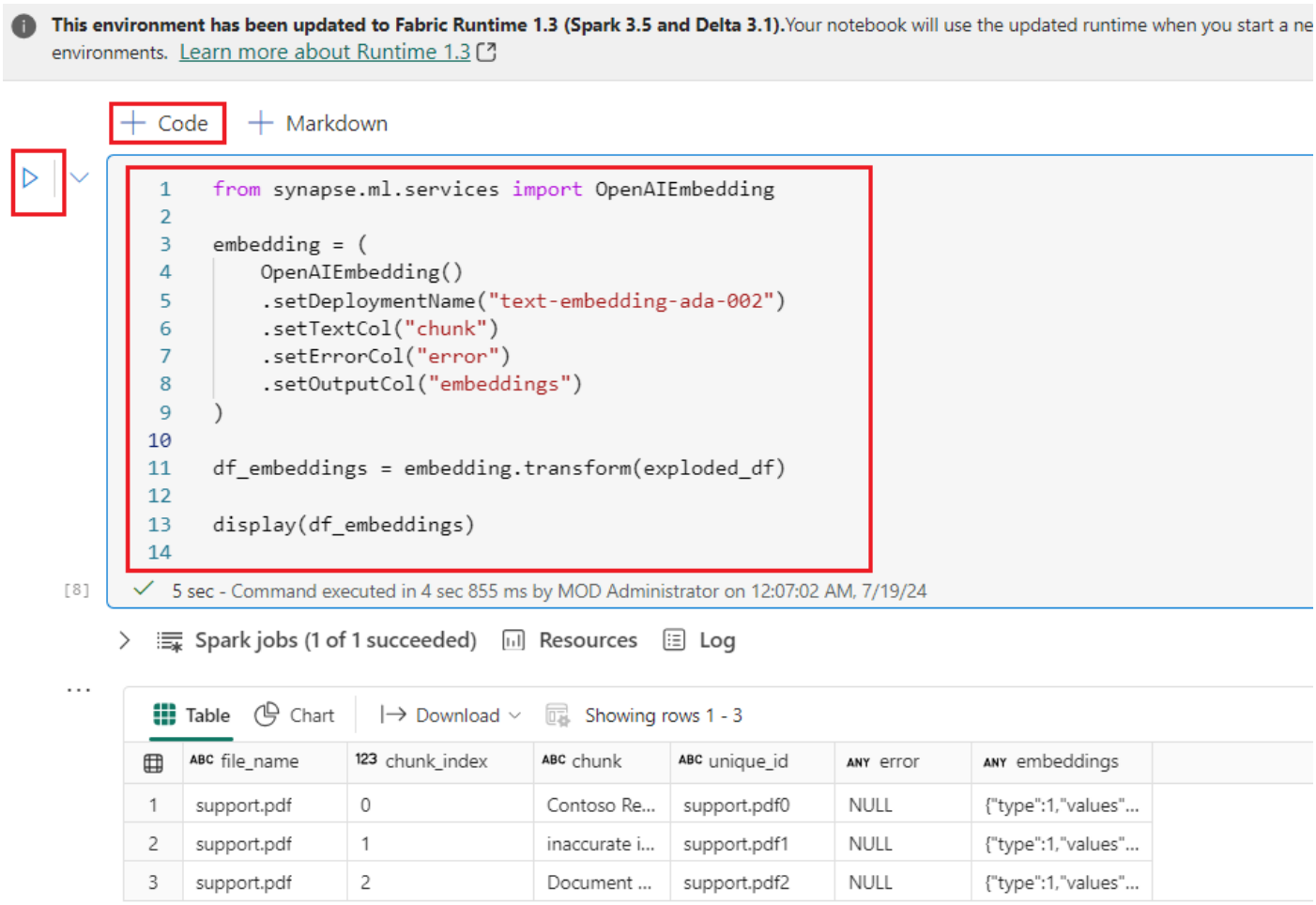
Next we'll generate the embeddings for each chunk. To do this we utilize both SynapseML and Azure OpenAI Service. By integrating the built in Azure OpenAI service with SynapseML, we can leverage the power of the Apache Spark distributed computing framework to process numerous prompts using the OpenAI service.

1. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
from synapse.ml.services import OpenAIEmbedding

embedding = (
  OpenAIEmbedding()
  .setDeploymentName("text-embedding-ada-002")
  .setTextCol("chunk")
```

```
    .setErrorCol("error")
    .setOutputCol("embeddings")
)

df_embeddings = embedding.transform(exploded_df)

display(df_embeddings)
```



This integration enables the SynapseML embedding client to generate embeddings in a distributed manner, enabling efficient processing of large volumes of data

Task 3: Storing Embeddings

Azure AI Search https://learn.microsoft.com/azure/search/search-what-is-azure-search?WT.mc_id=data-114676-jndemenge is a powerful search engine that includes the ability to perform full text search, vector search, and hybrid search. For more examples of its vector search capabilities, see the **azure-search-vector-samples repository** https://github.com/Azure/azure-search-vector-samples/.

Storing data in Azure AI Search involves two main steps:

Creating the index: The first step is to define the schema of the search index, which includes the properties of each field as well as any vector search strategies that will be used.

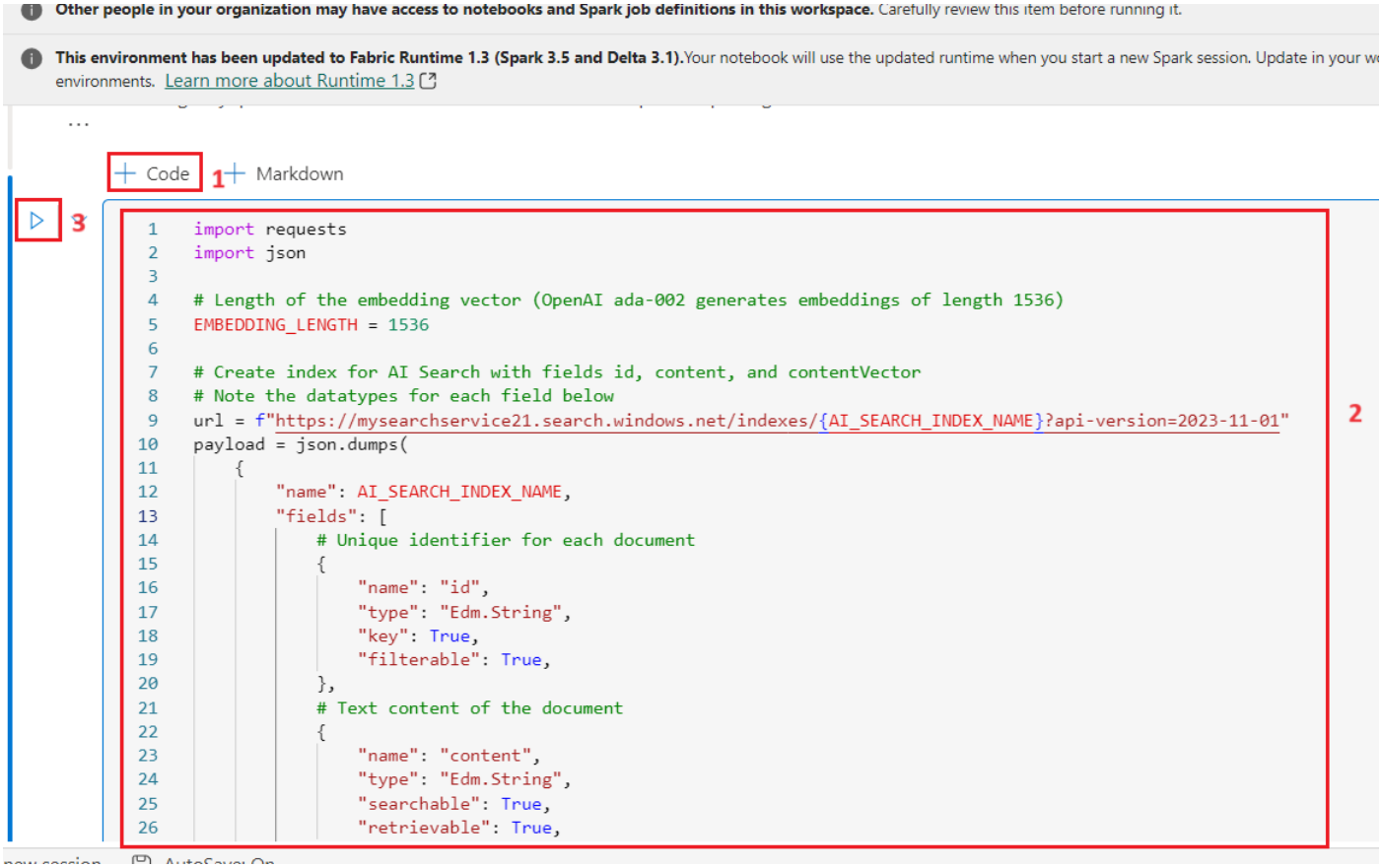
Adding chunked documents and embeddings: The second step is to upload the chunked documents, along with their corresponding embeddings, to the index. This allows for efficient storage and retrieval of the data using hybrid and vector search.

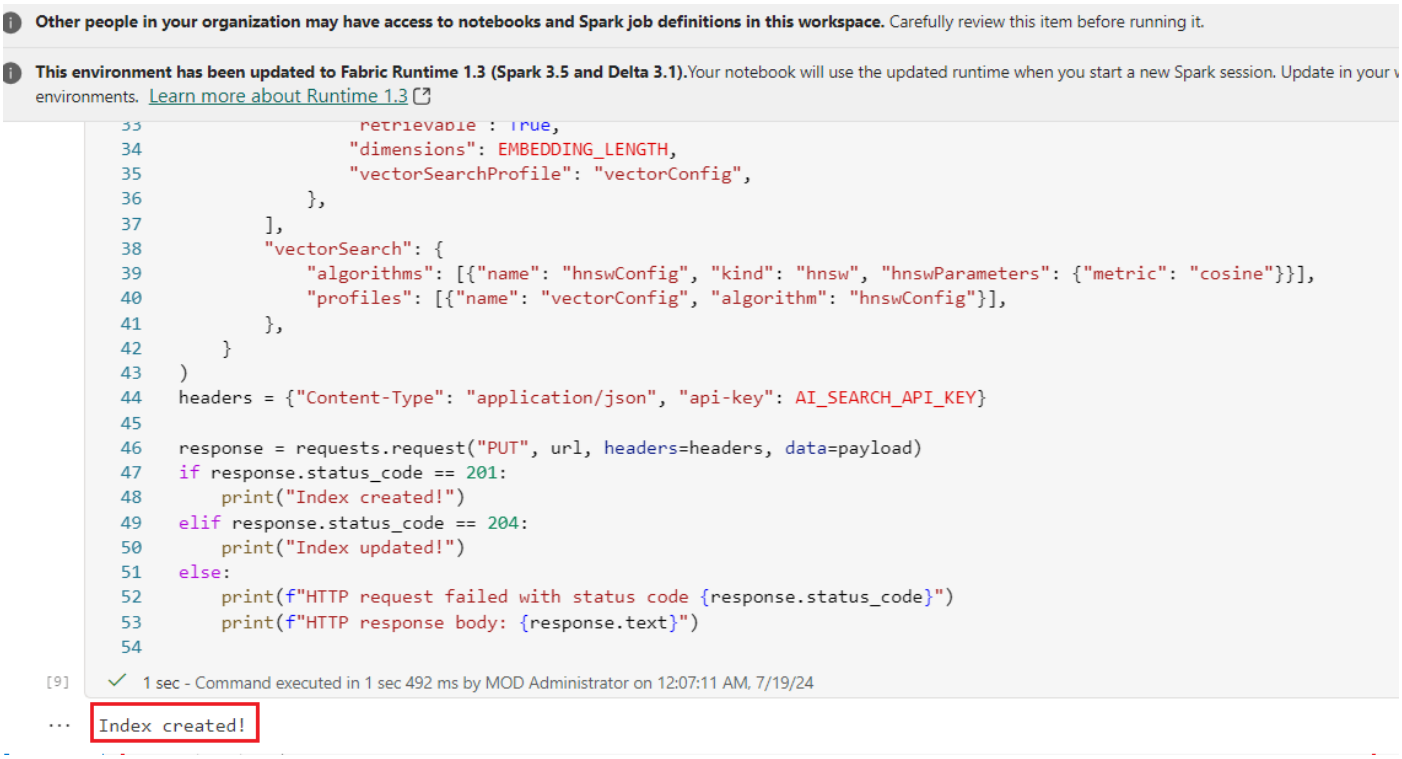
1. The following code snippet demonstrates how to create an index in Azure AI Search using the Azure AI Search REST API. This code creates an index with fields for the unique identifier of each document, the text content of the document, and the vector embedding of the text content.
2. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
import requests

import json


# Length of the embedding vector (OpenAI ada-002 generates embeddings of length 1536)

EMBEDDING_LENGTH = 1536


# Create index for AI Search with fields id, content, and contentVector

# Note the datatypes for each field below

url = f"https://{AI_SEARCH_NAME}.search.windows.net/indexes/{AI_SEARCH_INDEX_NAME}?api-version=2023-11-01"

payload = json.dumps(

  {

    "name": AI_SEARCH_INDEX_NAME,

    "fields": [

      # Unique identifier for each document

      {

        "name": "id",

        "type": "Edm.String",

        "key": True,

        "filterable": True,

      },

      # Text content of the document

      {

        "name": "content",

        "type": "Edm.String",

        "searchable": True,

        "retrievable": True,

      },

      # Vector embedding of the text content

      {
```

```
            "name": "contentVector",
            "type": "Collection(Edm.Single)",
            "searchable": True,
            "retrievable": True,
            "dimensions": EMBEDDING_LENGTH,
            "vectorSearchProfile": "vectorConfig",
        },
    ],
    "vectorSearch": {
        "algorithms": [{"name": "hnswConfig", "kind": "hnsw", "hnswParameters": {"metric": "cosine"}}],
        "profiles": [{"name": "vectorConfig", "algorithm": "hnswConfig"}],
    },
  }
)
headers = {"Content-Type": "application/json", "api-key": AI_SEARCH_API_KEY}

response = requests.request("PUT", url, headers=headers, data=payload)
if response.status_code == 201:
  print("Index created!")
elif response.status_code == 204:
  print("Index updated!")
else:
  print(f"HTTP request failed with status code {response.status_code}")
  print(f"HTTP response body: {response.text}")
```





3. The next step is to upload the chunks to the newly created Azure AI Search index. The Azure AI Search REST API supports up to 1000 "documents" per request. Note that in this case, each of our "documents" is in fact a chunk of the original file

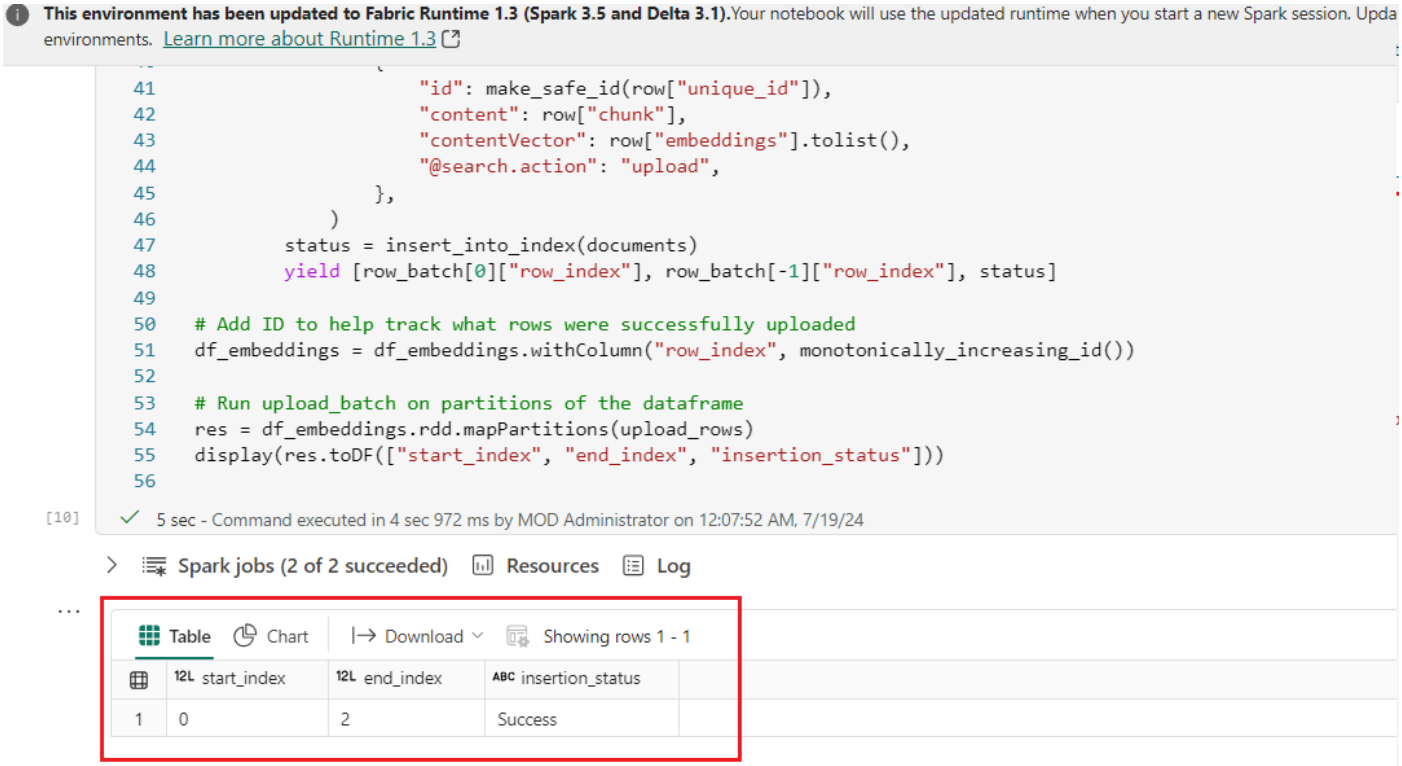
4. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

import re

```
from pyspark.sql.functions import monotonically_increasing_id


def insert_into_index(documents):
    """Uploads a list of 'documents' to Azure AI Search index."""
    url = f"https://{AI_SEARCH_NAME}.search.windows.net/indexes/{AI_SEARCH_INDEX_NAME}/docs/index?api-version=2023-11-01"
    payload = json.dumps({"value": documents})
    headers = {
        "Content-Type": "application/json",
        "api-key": AI_SEARCH_API_KEY,
    }
    response = requests.request("POST", url, headers=headers, data=payload)
    if response.status_code == 200 or response.status_code == 201:
        return "Success"
    else:
        return f"Failure: {response.text}"
def make_safe_id(row_id: str):
    """Strips disallowed characters from row id for use as Azure AI search document ID."""
    return re.sub("[^0-9a-zA-Z_-]", "_", row_id)


def upload_rows(rows):
    """Uploads the rows in a Spark dataframe to Azure AI Search.
    Limits uploads to 1000 rows at a time due to Azure AI Search API limits.
    """
    BATCH_SIZE = 1000
    rows = list(rows)
    for i in range(0, len(rows), BATCH_SIZE):
        row_batch = rows[i : i + BATCH_SIZE]
        documents = []
        for row in rows:
            documents.append(
                {
                    "id": make_safe_id(row["unique_id"]),
```

```
            "content": row["chunk"],
            "contentVector": row["embeddings"].tolist(),
            "@search.action": "upload",
        },
    )
    status = insert_into_index(documents)
    yield [row_batch[0]["row_index"], row_batch[-1]["row_index"], status]

# Add ID to help track what rows were successfully uploaded
df_embeddings = df_embeddings.withColumn("row_index", monotonically_increasing_id())

# Run upload_batch on partitions of the dataframe
res = df_embeddings.rdd.mapPartitions(upload_rows)
display(res.toDF(["start_index", "end_index", "insertion_status"]))
```



Exercise 4: Retrieving Relevant Documents and Answering Questions

After processing the document, we can proceed to pose a question. We will use SynapseML to convert the user's question into an embedding and then utilize cosine similarity to retrieve the top K document chunks that closely match the user's question.

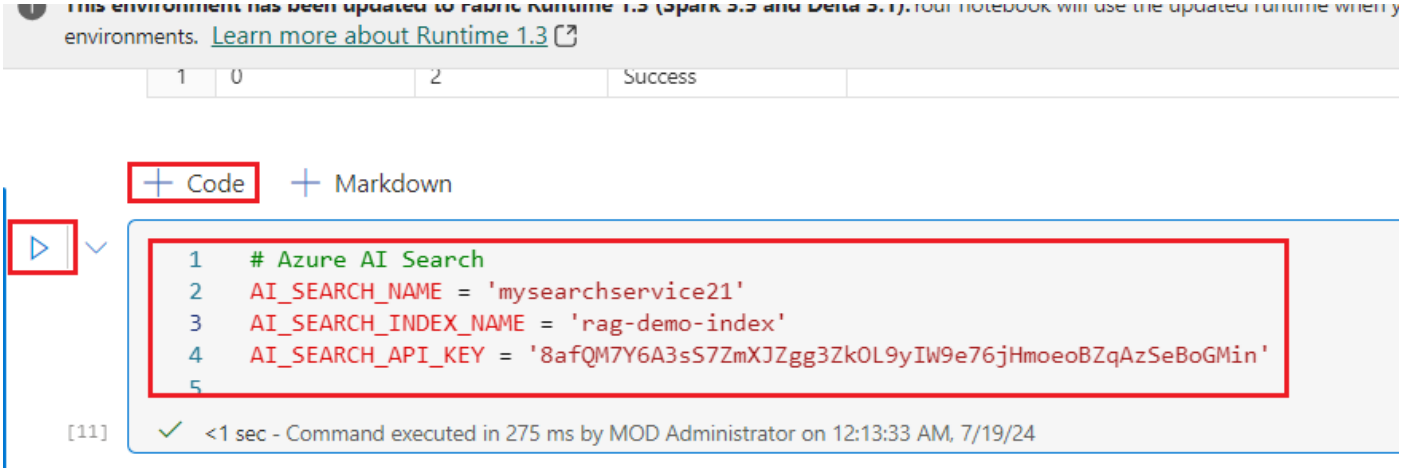
Task 1: Configure Environment & Azure API Keys

Create a new notebook in the Lakehouse and save it as rag_application. We'll use this notebook to build the RAG application.

1. Provide the credentials for access to Azure AI Search. You can copy the values from the from Azure Portal.(Exercise 1>Task 4)
2. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

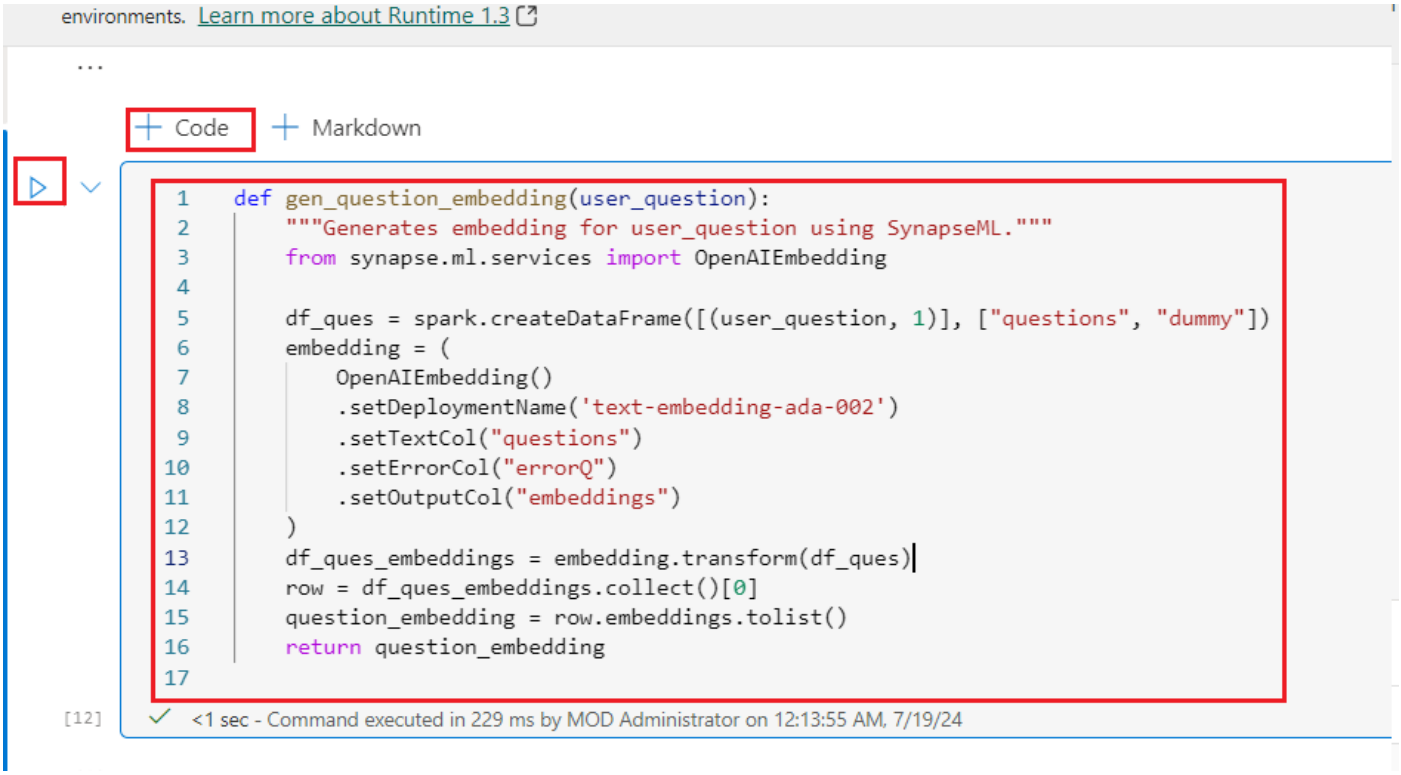
```
# Azure AI Search

AI_SEARCH_NAME = ''

AI_SEARCH_INDEX_NAME = 'rag-demo-index'

AI_SEARCH_API_KEY = ''
```



3. The following function takes a user's question as input and converts it into an embedding using the text-embedding-ada-002 model. This code assumes you're using the Pre-built AI Services in Microsoft Fabric
4. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
def gen_question_embedding(user_question):
    """Generates embedding for user_question using SynapseML."""
    from synapse.ml.services import OpenAIEmbedding
    df_ques = spark.createDataFrame([(user_question, 1)], ["questions", "dummy"])
    embedding = (
        OpenAIEmbedding()
        .setDeploymentName('text-embedding-ada-002')
        .setTextCol("questions")
        .setErrorCol("errorQ")
        .setOutputCol("embeddings")
    )
    df_ques_embeddings = embedding.transform(df_ques)
    row = df_ques_embeddings.collect()[0]
    question_embedding = row.embeddings.tolist()
    return question_embedding
```



Task 2: Retrieve Relevant Documents

1. The next step is to use the user question and its embedding to retrieve the top K most relevant document chunks from the search index. The following function retrieves the top K entries using hybrid search
2. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
import json

import requests


def retrieve_top_chunks(k, question, question_embedding):

    """Retrieve the top K entries from Azure AI Search using hybrid search."""

    url = f"https://{AI_SEARCH_NAME}.search.windows.net/indexes/{AI_SEARCH_INDEX_NAME}/docs/search?api-version=2023-11-01"

    payload = json.dumps({

        "search": question,

        "top": k,

        "vectorQueries": [

            {

                "vector": question_embedding,

                "k": k,

                "fields": "contentVector",

                "kind": "vector"
```

```
        }
    ]
})
headers = {
    "Content-Type": "application/json",
    "api-key": AI_SEARCH_API_KEY,
}
response = requests.request("POST", url, headers=headers, data=payload)
output = json.loads(response.text)
return output
```

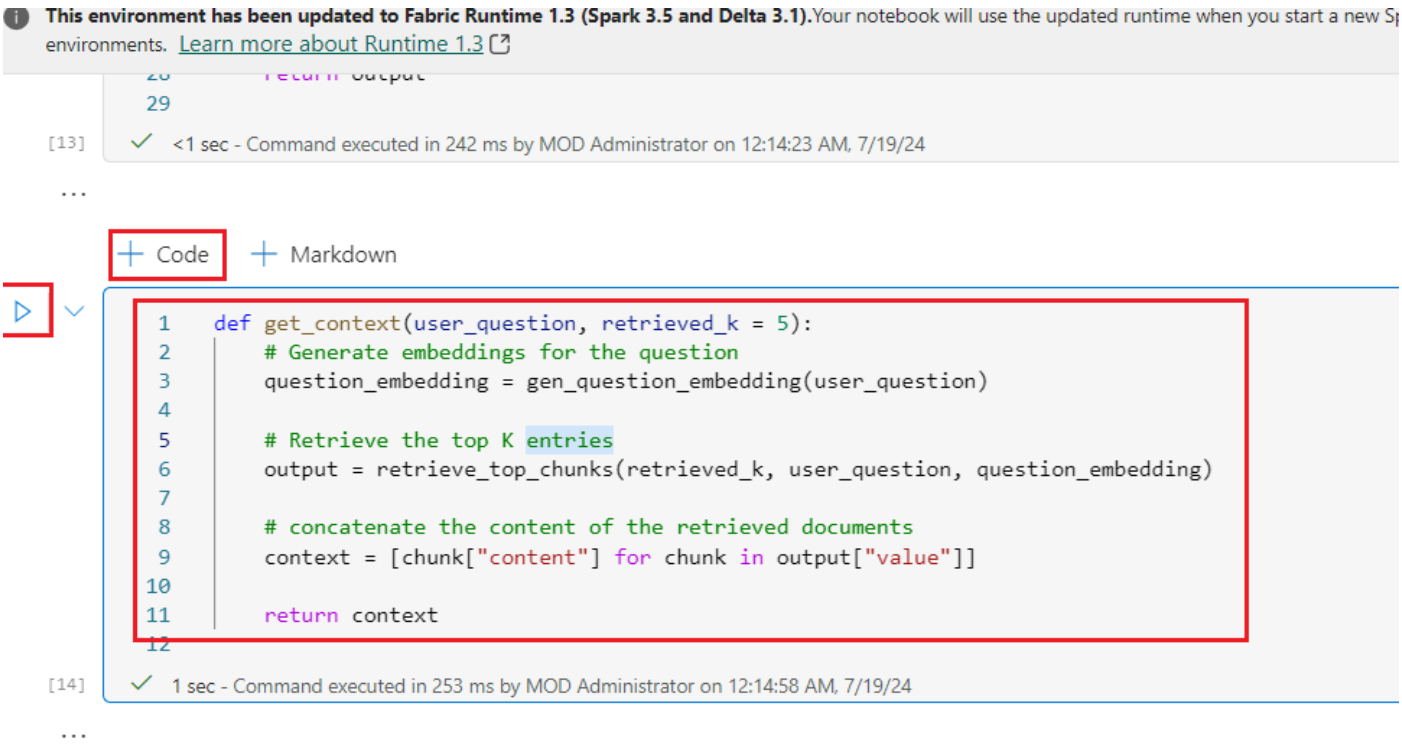
With those functions defined, we can define a function that takes a user's question, generates an embedding for the question, retrieves the top K document chunks, and concatenates the content of the retrieved documents to form the context for the user's question.

3. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
def get_context(user_question, retrieved_k = 5):
    # Generate embeddings for the question
    question_embedding = gen_question_embedding(user_question)
    # Retrieve the top K entries
    output = retrieve_top_chunks(retrieved_k, user_question, question_embedding)
    # concatenate the content of the retrieved documents
    context = [chunk["content"] for chunk in output["value"]]
```

return context



Task 3: Answering the User's Question

Finally, we can define a function that takes a user's question, retrieves the context for the question, and sends both the context and the question to a large language model to generate a response. For this demo, we'll use the gpt-35-turbo-16k, a model that is optimized for conversation.

1. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
from pyspark.sql import Row

from synapse.ml.services.openai import OpenAIChatCompletion



def make_message(role, content):
    return Row(role=role, content=content, name=role)


def get_response(user_question):
    context = get_context(user_question)
    # Write a prompt with context and user_question as variables
    prompt = f"""
    context: {context}
    Answer the question based on the context above.
    If the information to answer the question is not present in the given context then reply "I don't know".
```

```
"""
chat_df = spark.createDataFrame(
    [
        (
            [
                make_message(
                    "system", prompt
                ),
                make_message("user", user_question),
            ],
        ),
    ]
).toDF("messages")
chat_completion = (
    OpenAIChatCompletion()
    .setDeploymentName("gpt-35-turbo-16k") # deploymentName could be one of {gpt-35-turbo, gpt-35-turbo-16k}
    .setMessagesCol("messages")
    .setErrorCol("error")
    .setOutputCol("chat_completions")
)
result_df = chat_completion.transform(chat_df).select("chat_completions.choices.message.content")
result = []
for row in result_df.collect():
    content_string = ' '.join(row['content'])
    result.append(content_string)
# Join the list into a single string
result = ' '.join(result)
return result
```

2. Now, we can call that function with an example question to see the response:
3. Use the **+ Code** icon below the cell output to add a new code cell to the notebook, and enter the following code in it. Click on ▷ **Run cell** button and review the output

Copy

```
user_question = "how do i make a booking?"

response = get_response(user_question)

print(response)
```

This environment has been updated to Fabric Runtime 1.3 (Spark 3.5 and Delta 3.1). Your notebook will use the updated runtime when you start a new Spark session. Update in your workspace settings and custom environments. Learn more about Runtime 1.3

+ Code + Markdown

```
1  user_question = "how do i make a booking?"
2  response = get_response(user_question)
3  print(response)
4
```

[16] ✓ 8 sec - Command executed in 8 sec 18 ms by MOD Administrator on 12:15:56 AM, 7/19/24 PySpark (Py

Spark jobs (2 of 2 succeeded) Resources Log

```
To make a booking on Contoso Real Estate, follow these steps:

1. Search for Rentals:
   - Enter your destination, check-in and check-out dates, and the number of guests.
   - Apply filters such as price range, property type, and amenities to narrow down your options.
   - Browse through the listings to find the perfect place for your stay.

2. View Listing Details:
   - Click on a listing to view detailed information, including photos, property description, reviews, and host information.

3. Make a Booking:
   - Click the "Book Now" button on the listing page.
   - Review the booking details, including the total cost and house rules.
   - Confirm your booking by providing payment information.
   - Once the host accepts your booking, you'll receive a confirmation.

Please note that Contoso Real Estate handles the payment process securely, and you'll only be charged once your booking is confirmed. You can also
communicate with the host through our messaging system for any questions or special requests.
```