Lab 3 - Retrieval Augmented Generation (RAG) using PySpark on EMR

> **Important**
> Run with **JupyterLab**: We're going to use `lab-03-rag/Lab_3_RAG_on_SageMaker_Studio_using_EMR.ipynb` notebook for this section

Contents

Overview

In this section of the lab we're going to be learning how to build a Retrieval Augmented Generation (RAG) System using,

- Amazon EMR
- SageMaker + EMR Integration
- NLP Data Processing at Scale using EMR PySpark
- OpenSearch to store and retrieve vector embeddings
- Launching a StreamLit App on SageMaker Studio

We're going to be running `lab-03-rag/Lab_3_RAG_on_SageMaker_Studio_using_EMR.ipynb` for this lab.

Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a hybrid model that combines the generative power of a large language model (LLM) with the information retrieval capabilities of an external database, typically using an embedding model to encode the information. This setup allows the LLM to generate responses not only based on its pre-trained knowledge but also by referring to up-to-date, specific information fetched from the database. The embedding model helps to find the most relevant documents or data entries by mapping the query and documents into a shared embedding space, where the proximity of embeddings represents their relevance. This approach enhances the LLM's ability to provide detailed, accurate, and contextually relevant answers, especially for queries that require external knowledge or facts.

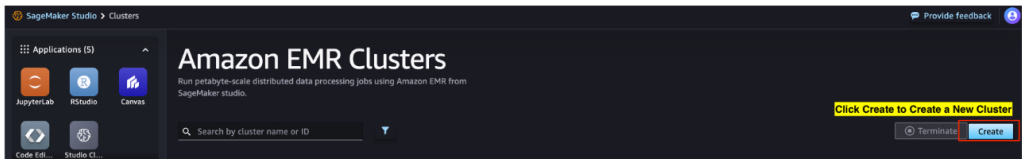
What is Amazon EMR?

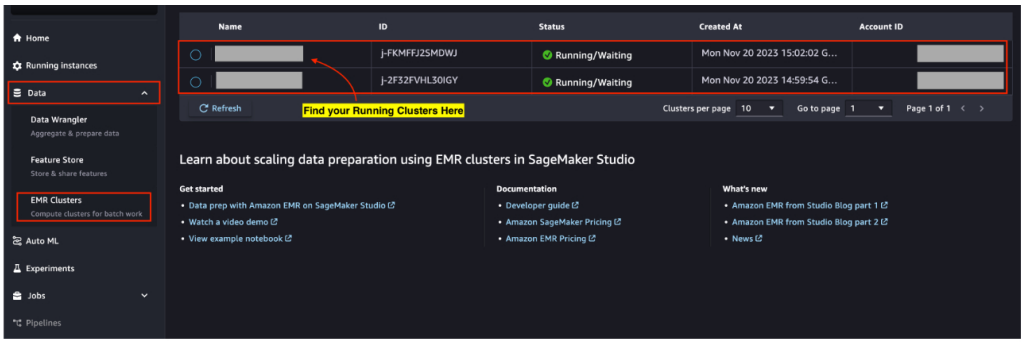
Amazon EMR (Elastic MapReduce) is a cloud service offered by Amazon Web Services (AWS) that provides a managed framework for processing, analyzing, and visualizing large amounts of data. It simplifies running big data frameworks such as Apache Hadoop and Apache Spark on AWS to process and analyze vast amounts of data. By using Amazon EMR, you can transform and move large amounts of data into and out of other AWS data storage services like Amazon S3 and Amazon DynamoDB.

Amazon EMR handles the provisioning, configuration, and tuning of the cloud infrastructure and big data frameworks, which allows data scientists, developers, and analysts to focus on their data without worrying about the underlying hardware or big data ecosystem. Users can run data processing jobs and interactive queries, and they can also use EMR for machine learning purposes. Its scalability means that it can expand or shrink the number of instances (virtual servers) to efficiently process data while controlling costs, making it a flexible option for big data processing tasks.

Amazon EMR and SageMaker Studio Integration

SageMaker Studio provides direct integration into Amazon EMR via "EMR Clusters". You are able to configure and provision clusters, terminate existing clusters and access application interfaces all from within Studio. You can access Cluster information from the Studio UI navigating to `Data` > `EMR Clusters`.

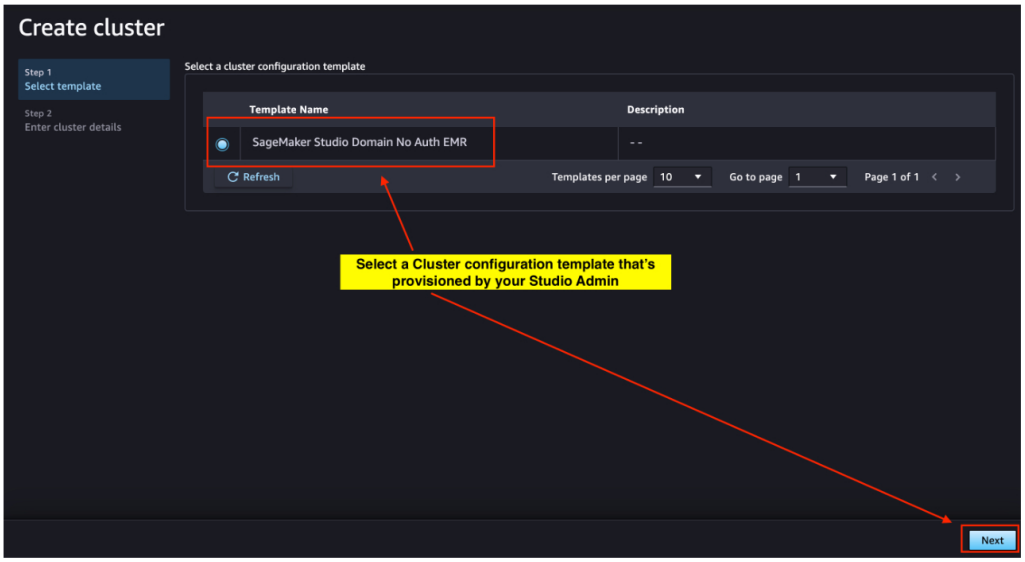


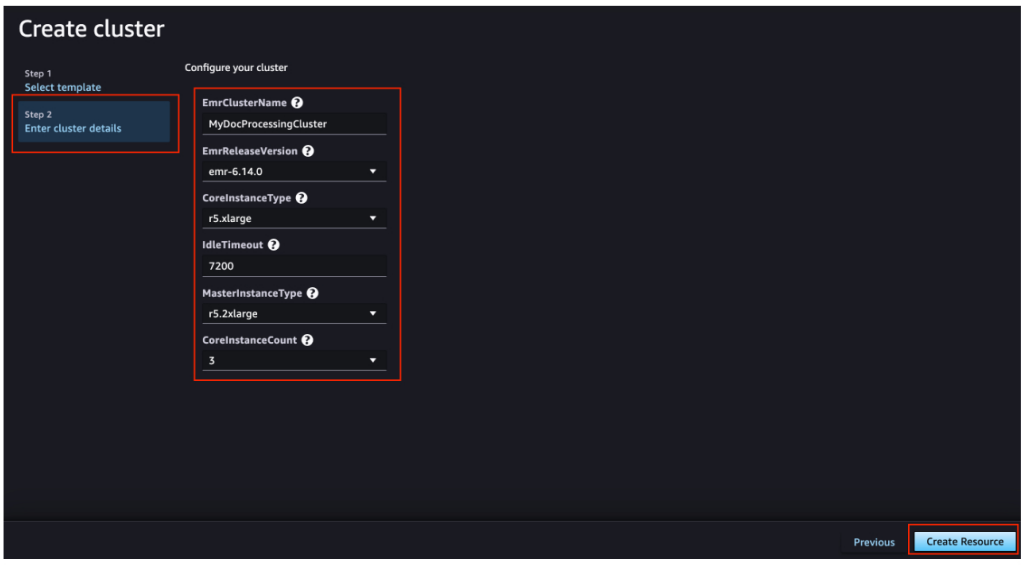


You should already see an EMR Cluster in `Running/Waiting` state. We're going to use this EMR Cluster for Data Processing.

However, let's create a new EMR Cluster, you can see how easy it can be for a user to spin up a new EMR Cluster right from Studio with just a few clicks.

Click `Create` to begin creating a new Cluster. Select a Service Catalog template that's provisioned by your SageMaker/EMR Admin. This template can be very flexible and can be used to surface self-service values (like Core Node count, Instance Types, Cluster Name, etc.).





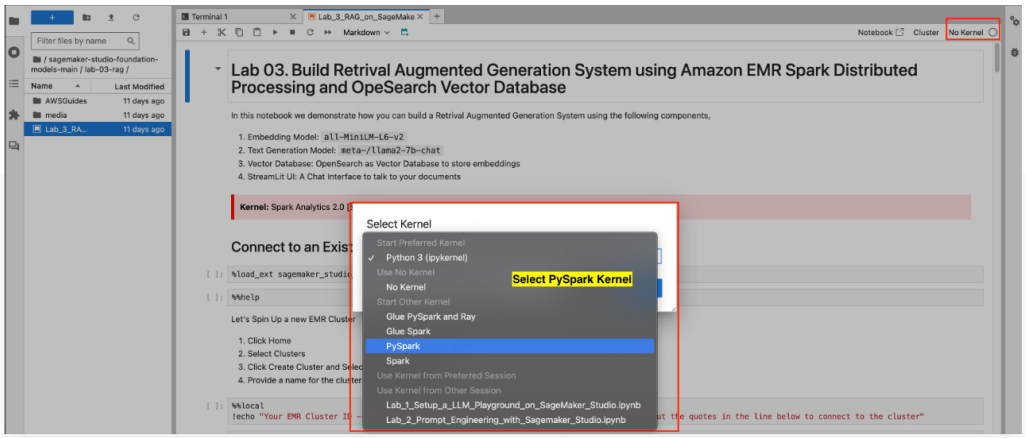
Why do we need a Cluster?

The primary purpose of RAG is update the knowledge of pre-trained LLMs without needing to fine-tune LLMs by simply referencing our model to an embedding vector DB (OpenSearch is our Vector DB in this lab). Depending on context, the amount of documents we need to process during RAG can range from several MB to hundreds of GBs. EMR can help simplify data processing at scale, with ease, using PySpark.

Connecting to an EMR Cluster

Its easy to list and connect existing EMR cluster from JupyterLab Space. Using the base SageMaker Distribution Image.

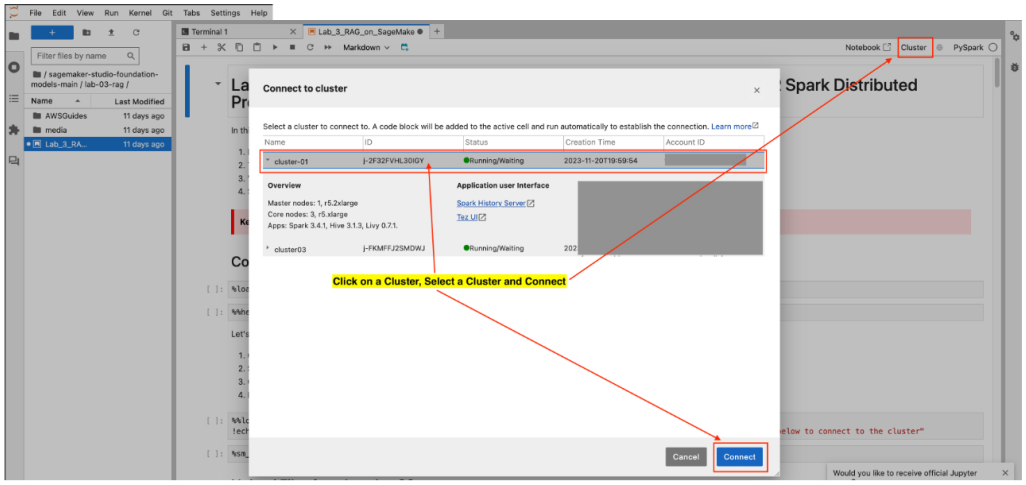
From your Space, open `lab-03-rag/Lab_3_RAG_on_SageMaker_Studio_using_EMR.ipynb`. Choose `PySpark` Kernel that's pre-provisioned.



Connect your Notebook to an EMR cluster to parallelize your data processing task. To connect your notebook to an EMR cluster, use Option 1 OR Option 2 from below.

Option 1: Connect to Cluster using UI

Using UI, we can connect our notebook to an EMR cluster by navigating to `Cluster` > Select a Running Cluster from dropdown > `Connect`



Option 2: Connect to Cluster using code

Using code, we can connect our notebook to an EMR cluster using `%sm_analytics` auto-magic command.

```
%%local
!echo "Your EMR Cluster ID ---> $(aws emr list-clusters | jq '.Clusters[0].Id')  Paste without the quotes in the line below to connect to the cluster"
```

```
%sm_analytics emr connect --verify-certificate False --cluster-id j-30VVYADZRJQ6Q --auth-type None --language python
```

Parallelize Read and Write Operations using EMR PySpark

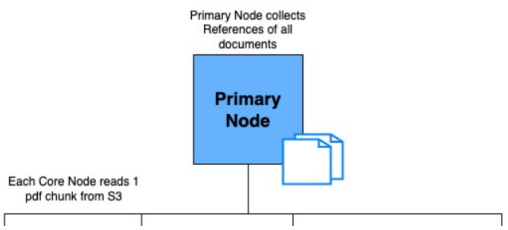
To fully leverage EMR capabilities, we are going to start off by parallelizing pdf read from s3 directly into memory and collect results in EMR Primary.

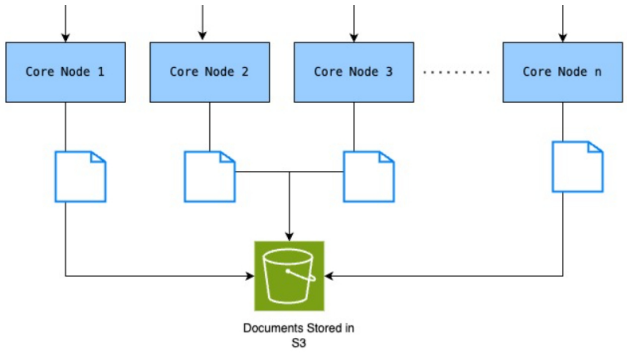
```
def load_pdf_from_s3_into_memory(row):
    """
    Load a PDF file from an S3 bucket directly into memory.
    """
    try:
        src_bucket_name, src_file_key = row
        s3 = boto3.client('s3')
        pdf_file = io.BytesIO()
        s3.download_fileobj(src_bucket_name, src_file_key, pdf_file)
        pdf_file.seek(0)
        pdf_reader = PdfReader(pdf_file)
        return (src_file_key, pdf_reader, len(pdf_reader.pages))

    except Exception as e:
        return (os.path.basename(src_file_key), str(e))
```

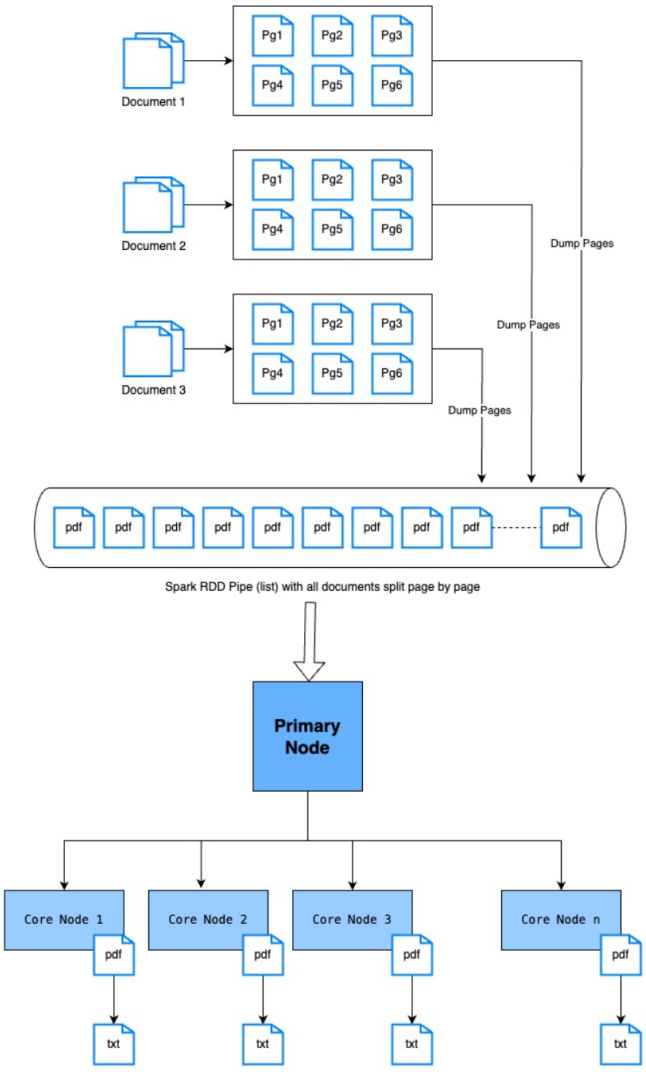
Let's concurrently load pdf files into memory using rdd map and collect the results back to our Primary Node

```
pdfs_in_memory = pdfs_rdd.map(load_pdf_from_s3_into_memory).collect()
```





We can further parallelize PDF extraction by splitting every document into individual page and extracting text at page level in a parallel fashion.



Large large language models have a maximum token limit for each input sequence, any text that exceeds this limit needs to be split into smaller chunks to ensure that the model can process it effectively.

The recursive element of the splitter refers to its ability to divide the text into smaller parts iteratively until all chunks are below the maximum token threshold. This ensures that no information is lost due to truncation. The chunks can then be individually processed by the RAG model, which retrieves relevant information from a large corpus of text based on these inputs. Once processed, the outputs from the chunks can be aggregated to form a complete response.

```
global_text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=750,
    chunk_overlap=10
)
docs = global_text_splitter.split_documents(documents_custom)
print(f"Total number of docs pre-split {len(documents_custom)} | after split {len(docs)}")
```

Invoke Embedding Model and Ingest Results into OpenSearch

Embedding Model

The purpose of an embedding model in a Retrieval Augmented Generation (RAG) system is to map high-dimensional data, like text, into a

lower-dimensional vector space in a way that preserves semantic meaning. These embeddings enable efficient similarity searches within a large corpus of documents. When a query is made, the embedding model computes the vector representation of the query and retrieves the most semantically related documents from the database. This is crucial for a RAG system because it allows the generative component of the model to focus on producing answers that are not just contextually relevant but also factually accurate, drawing from the most pertinent information in the retrieved documents. By using an embedding model, RAG can effectively combine the breadth of knowledge contained in external databases with the nuanced understanding and generation capabilities of a large language model, leading to more informed and precise outputs.

OpenSearch Vector DB

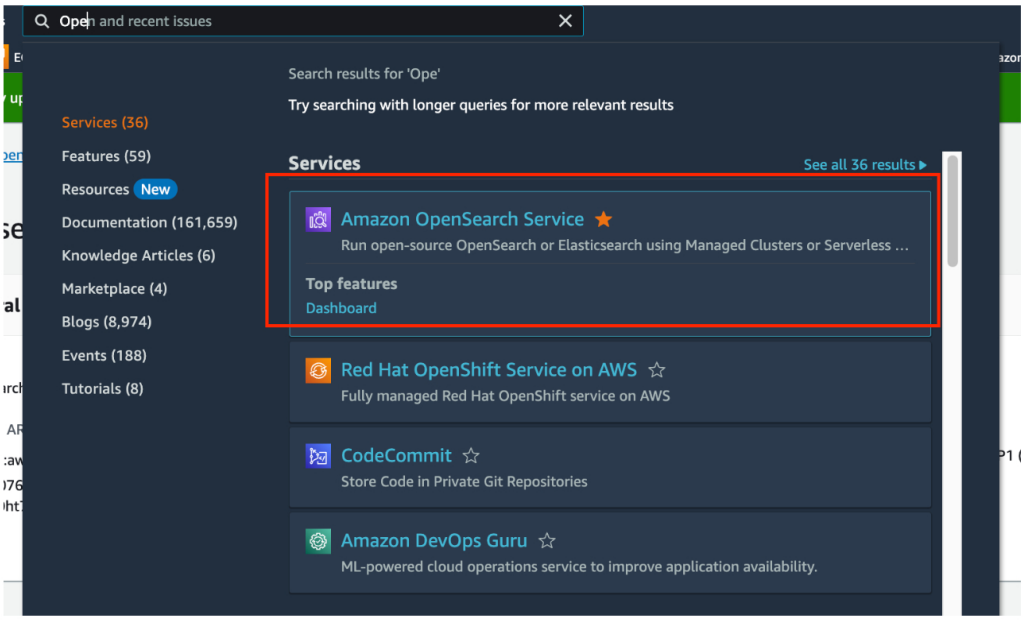
In a Retrieval Augmented Generation (RAG) system, the Vector Database (DB) serves as a repository for storing precomputed vector embeddings of a large collection of documents or knowledge sources. When a query is input into the RAG system, the Vector DB is used to perform a fast similarity search to retrieve the most relevant documents based on their vector proximity to the query. This retrieval step enriches the generative process of the language model with context-specific information, allowing the model to generate responses that are both contextually relevant and informationally rich, based on the content of the retrieved documents.

In this lab we're going to leverage Amazon OpenSearch Vector DB to store and retrieve embeddings during online inference query by a user.

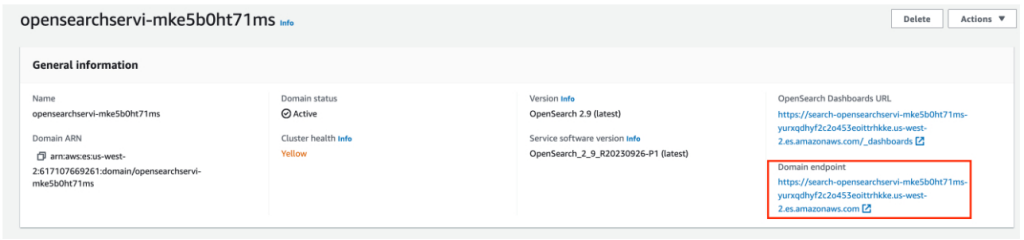
Using EMR to Ingest Embeddings into OpenSearch

We are going to parallelize the process of converting text chunks into embeddings using an Embedding Model and ingest these results into OpenSearch using the OpenSearch domain URL.

To manually determine the OpenSearch Domain URL to ingest your Embeddings,



Select the domain URL:



However, in this lab, we're going to auto-retrieve this endpoint URL using a mix of `%local` and `aws cli` commands,

```
%%local
!echo -n "https://$(aws es describe-elasticsearch-domain \
--domain-name "$(aws es list-domain-names --query 'DomainNames[*].DomainName' --output text)" \
--query 'DomainStatus.Endpoint' --output text)" > ../studio-local-ui/opesearchurl.txt
```

```
%%local
OPENSEARCH_DOMAIN_URL = open("../studio-local-ui/opesearchurl.txt", "r").read()
print(f"Your OpenSearch Domain URL --->", OPENSEARCH_DOMAIN_URL)
```

Run embedding ingestion processing using EMR and `langchain`'s built in `OpenSearchVectorSearch` API.

```
import time
from langchain.vectorstores import OpenSearchVectorSearch

start = time.time()
docsearch = OpenSearchVectorSearch.from_documents(
    docs,
    EmbeddingsGenerator,
    opensearch_url=OPENSEARCH_DOMAIN_URL,
    bulk_size=len(docs),
    http_auth=(user, pwd),
    index_name=INDEX_NAME_OSE,
    engine="faiss"
)
```

```
end = time.time()
print(f"Total Time for ingestion: {round(end - start, 2)} secs")
```

```
query = "What is a Amazon SageMaker?"
sample_responses = docsearch.similarity_search(
    query,
    k=5,
    space_type="cosineSimilarity",
    search_type="painless_scripting"
)
```

Quick Recap and Putting it All Together

Recap

1. We create a Spark Cluster to leverage PySpark for Distributed Data Processing at scale!
2. We pushed some raw data into S3 (in reality, this data can be housed anywhere RedShift, S3, RDS, Dynamo, Snowflake, etc..)
3. We Parallelized our document extraction from S3 using PySpark - our PySpark `Core` nodes were able to reach out to doc store (S3) read a file into memory for downstream processing
4. We then split our processing at Document - at a page level and further parallelize our pdf reading process using PySpark
5. We chunk our document corpus using `LangChain`'s `RecursiveCharacterTextSplitter`. We then convert our text into Embeddings using our BGE Embedding LLM Model and ingest these embeddings into OpenSearch index. - all using PySpark Parallel Processing technique
6. Now we use `Streamlit` to interact with text generation model and document embeddings with a UI

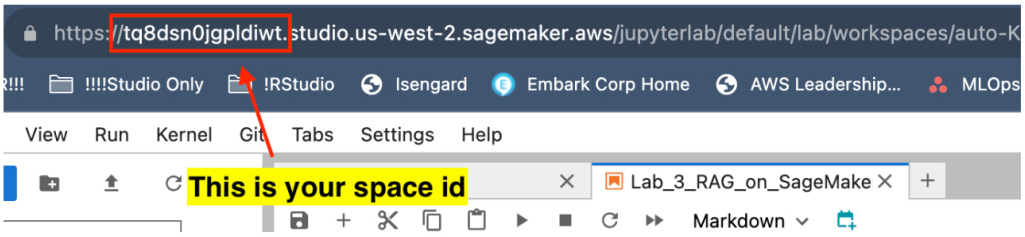
Launch StreamLit UI

Run this cell inside your notebook to run a bash command that launches a streamlit UI,

```
%%bash
cd ../studio-local-ui
streamlit run chat_app.py --server.runOnSave true --server.port
```

Navigate to the suggested url to access your StreamLit UI (please replace the `spaceid` with your actual space url name),

```
1  https://<spaceid>.studio.us-west-2.sagemaker.aws/jupyterlab/default/proxy/8502/
```



Navigate to the suggested URL, you should see a Streamlit app,