

# RISHIT BURMAN

## Intelligent Financial Document Engine - Project Documentation

---

### Overview

This project presents a robust **Intelligent Document Retrieval System** designed to handle the **extraction, semantic indexing, and question-answering over financial documents**. It leverages powerful AWS services including **Textract, DynamoDB, S3, SageMaker, and OpenSearch** for secure, scalable, and efficient processing of enterprise documents — while keeping data *private* (no use of public LLMs).

---

### Why This Project Matters

#### Business Impact

- **Automates the reading and searching of thousands of financial reports**, audit documents, quarterly filings, etc.
- Eliminates manual efforts by finance teams, saving **hours of time per document**.
- Ensures **accuracy** using semantic search powered by **embeddings**.
- Keeps enterprise data **completely private** and never leaves AWS.

#### Potential Use-Cases:

- Financial firms (for annual/quarterly report analysis)
- Hospitals and pharma (for report lookup & compliance)

- Enterprises (for internal document retrieval)
  - Law firms (for fast document clause retrieval)
- 

## Architecture Breakdown

### Part 1: Document Upload & Text Extraction

**Flow:**

1. User uploads a PDF/Scanned image to **S3 Bucket** (doc-engine-bucket-risbur).
2. A PUT event triggers a **Lambda function** named doc-upload-processor.
3. This Lambda uses **Amazon Textract** to extract structured text from the uploaded document.
4. The text and document metadata (like file name) are stored in **DynamoDB table** DocumentTextTable.

#### Why Amazon Textract?

- It supports scanned financial documents (images, tables, PDFs).
- Can extract text from complex layouts with high accuracy.
- Fully serverless and scalable.

The screenshot shows the AWS S3 console with the path: Amazon S3 > Buckets > doc-engine-bucket-risbur. The main content area displays a table for event notifications:

Name	Event types	Filters	Destination type	Destination
trigger-lambda-upload	Put	-	Lambda function	doc-upload-processor

Below the table, there are sections for "Amazon EventBridge" and "Transfer acceleration".

- Upload event from S3 triggering Lambda.

The screenshot shows the AWS CloudWatch Logs console with the path: CloudWatch > Log groups > /aws/lambda/doc-upload-processor > 2025/06/19/[LATEST]30b301ffe2a74de8a97d69f1e3b3711a. The main content area displays a table of log events:

Timestamp	Message
2025-06-19T14:15:14.773Z	INIT_START Runtime Version: python:3.10.v78 Runtime Version ARN: arn:aws:lambda:us-east-1:runtime:b556158cad85934b6c377a5efb9a60...
2025-06-19T14:15:15.064Z	START RequestId: 5c62ef22-f576-4bb1-b5e2-19e2389a043b Version: \$LATEST
2025-06-19T14:15:15.065Z	Event: {"Records": [{"eventVersion": "2.1", "eventSource": "aws:s3", "awsRegion": "us-east-1", "eventTime": "2025-06-19T14:15:13..."}}
2025-06-19T14:15:18.594Z	Extracted text: Financial Report - Q4 2024
2025-06-19T14:15:18.594Z	Company: GlobalShop Inc.
2025-06-19T14:15:18.594Z	Report Date: December 31, 2024
2025-06-19T14:15:18.594Z	Prepared By: Finance Department
2025-06-19T14:15:18.594Z	Summary:
2025-06-19T14:15:18.594Z	Q4 Revenue reached \$2 million, marking a 12% increase from Q3.
2025-06-19T14:15:18.594Z	Operating expenses stood at \$750,000.
2025-06-19T14:15:18.594Z	Net profit recorded: \$1.25 million.
2025-06-19T14:15:18.594Z	Top performing product: SmartGadget Pro

- Extracted Text from CloudWatch logs.

## Part 2: Embedding Generation & Semantic Indexing

### Tool: Jupyter Notebook (generate\_embeddings.ipynb)

1. Loads each document's text from **DynamoDB**.

The screenshot shows the AWS DynamoDB console. On the left, the navigation pane includes 'Dashboard', 'Tables', 'Explore items' (selected), 'PartiQL editor', 'Backups', 'Exports to S3', 'Imports from S3', 'Integrations' (New), 'Reserved capacity', and 'Settings'. Under 'Explore items', there are sections for 'Clusters', 'Subnet groups', 'Parameter groups', and 'Events'. The main area shows the 'DocumentTextTable' table. A search bar at the top says 'Find tables'. Below it, a 'Scan' button is selected. The table details show one item: 'Table - DocumentTextTable' with 'All attributes' projection. The item itself is labeled 'Completed - Items returned: 1'. The table data shows a single row with 'DocumentName (String)' as 'sample\_financial\_report.pdf' and 'ExtractedText' as 'Financial Report - Q4 2024 Company: GlobalShop Inc. Report Date: December 31...'. A status message at the bottom right says 'Scan started on June 19, 2025, 20:49:42'.

2. Uses a **SentenceTransformer MiniLM model** hosted on **SageMaker** to generate **text embeddings** (vector representations of document meaning).

The screenshot shows a Jupyter Notebook interface with a single open cell. The code in the cell is:

```
[10]: model = SentenceTransformer('all-MiniLM-L6-v2')

dynamodb = boto3.resource('dynamodb', region_name='us-east-1')
table = dynamodb.Table('DocumentTextTable')
response = table.get_item(Key={'DocumentName': 'sample_financial_report.pdf'})
text = response['Item']['ExtractedText']

embedding = model.encode(text)

print("Embedding vector shape:", embedding.shape)
print("Embedding preview:", embedding[:10])
```

The output of the code is:

```
Embedding vector shape: (384,)
Embedding preview: [-0.0393213  0.03358566  0.02696446 -0.00726199  0.00048609  0.00547366
 0.00602496  0.06151114 -0.01936604  0.07668573]
```

### 3. Embeddings + metadata are stored in OpenSearch (KNN Index) for fast semantic retrieval.

- Output of generated embeddings.

The screenshot shows a Jupyter Notebook interface with a single code cell. The code performs the following steps:

- Imports necessary modules: `os` and `openai`.
- Loads a saved model from a file named `model.pkl`.
- Generates embeddings for the query "What was the net profit in Q4?" using the loaded model.
- Creates a search body for an OpenSearch index named `documents-index` with a size of 1 and a KNN query.
- Searches the index with the provided body.
- Prints the retrieved document's source and text.

```
[27]: query = "What was the net profit in Q4?"
query_embedding = model.encode(query)
search_body = {
    "size": 1,
    "query": {
        "knn": {
            "embedding": {
                "vector": query_embedding.tolist(),
                "k": 1
            }
        }
    }
}
response = client.search(index=index_name, body=search_body)
top_hit = response['hits'][0]['_source']
print("Retrieved Document:")
print(top_hit['text'])
```

Output of the code:

```
Retrieved Document:
Financial Report - Q4 2024
Company: GlobalShop Inc.
Report Date: December 31, 2024
Prepared By: Finance Department
Summary:
Q4 Revenue reached $2 million, marking a 12% increase from Q3.
Operating expenses stood at $750,000.
Net profit recorded: $1.25 million.
Top performing product: SmartGadget Pro
Signed by: John Doe, Chief Financial Officer
```

- Indexing confirmation on OpenSearch dashboard.

The screenshot shows the Amazon OpenSearch Service console with the path: Amazon OpenSearch Service > Domains > doc-engine-domain > indices > documents-index.

The left sidebar shows navigation links for Serverless, Ingestion, and Integrations.

The main panel displays the following information for the `documents-index`:

- Index information:**

Name: documents-index	Document count: 1	Query total: 105
	Document size (byte): 17.64 KiB	Mapping type: properties
- Mapping structure:** Shows fields: doc, embedding, fileName, text.
- Field mappings (3):**

Field	Field type
doc.embedding	knn_vector
doc.fileName	keyword
doc.text	text

## OpenSearch Vector Database — How It Works

OpenSearch's KNN indexing allows you to:

- Store **high-dimensional embedding vectors** (from SageMaker).
- Perform **Approximate Nearest Neighbour Search** on these vectors.
- Retrieve **most semantically relevant document** based on the user's question.

### Example:

Query: "What is the net profit?"

OpenSearch returns the document that has the closest vector embedding to this question and pulls the matching line.

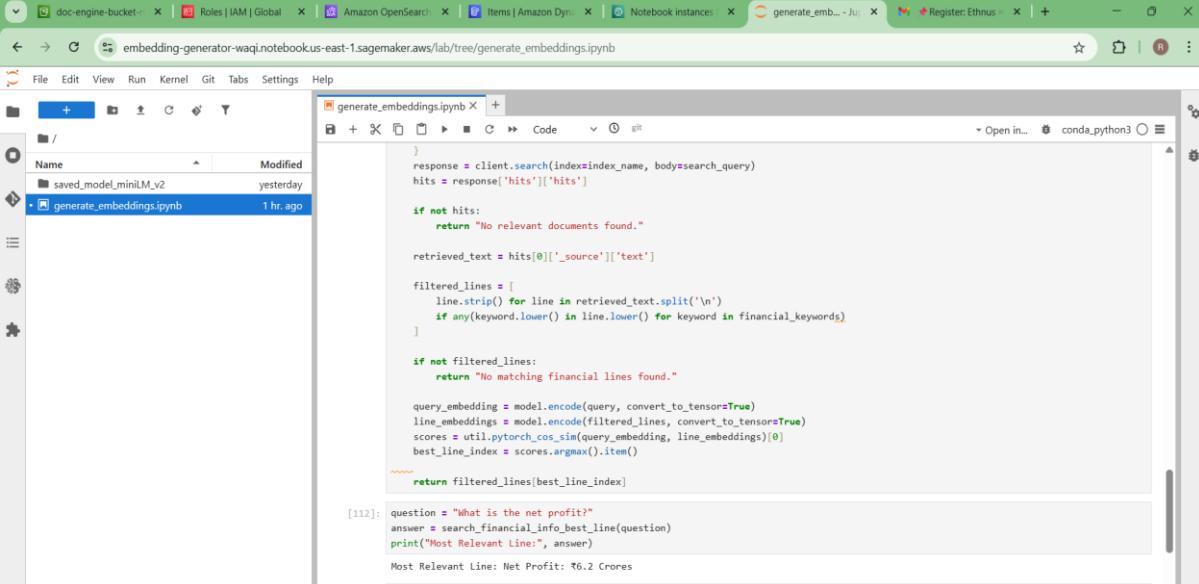
### Advantage over keyword search:

- Can handle **paraphrased queries** (e.g., "profit for the quarter" → "Net profit recorded: \$1.25 million").
- Robust even when documents use different wordings.
- OpenSearch index view.

The screenshot shows the OpenSearch UI with the 'Indexes' tab selected. The page title is 'Indexes (4)'. A sub-header states: 'Indexing is the method by which search engines organize data for fast retrieval. Before you can search data, you must index it.' Below this is a search bar labeled 'Find indexes' and a navigation bar with icons for back, forward, and refresh. The main content is a table with the following data:

Index	Document count	Size (byte)	Query total	Mapping type	Field mappings
kibana_1	1	5.21 KiB	52	dynamic, _meta, proper...	128
.opensearch-observability	0	208.00 B	0	dynamic, properties	11
.plugins-ml-config	1	3.94 KiB	1	_meta, properties	6
documents-index	1	17.64 KiB	105	properties	3

- Search query payload and matching result.



The screenshot shows a Jupyter Notebook interface with several tabs at the top: 'doc-engine-bucket...', 'Roles | IAM | Global', 'Amazon OpenSearch', 'Items | Amazon Dyn...', 'Notebook instance...', 'generate\_embeddings.ipynb', 'Register: Ethnus...', and '+'. The main area displays a code cell for 'generate\_embeddings.ipynb' containing Python code. The code performs a search on an index named 'index\_name' using a query defined in 'search\_query'. It retrieves hits, strips the retrieved text, filters for financial keywords, and then encodes the query and lines to calculate similarity scores. The most relevant line is identified and its content is printed. A command-line output at the bottom shows a question about net profit and the resulting answer from the search.

```

response = client.search(index=index_name, body=search_query)
hits = response['hits']['hits']

if not hits:
    return "No relevant documents found."

retrieved_text = hits[0]['_source']['text']

filtered_lines = [
    line.strip() for line in retrieved_text.split('\n')
    if any(keyword.lower() in line.lower() for keyword in financial_keywords)
]

if not filtered_lines:
    return "No matching financial lines found."

query_embedding = model.encode(query, convert_to_tensor=True)
line_embeddings = model.encode(filtered_lines, convert_to_tensor=True)
scores = util.pytorch_cos_sim(query_embedding, line_embeddings)[0]
best_line_index = scores.argmax().item()

return filtered_lines[best_line_index]

[112]: question = "What is the net profit?"
answer = search.financial_info_best_line(question)
print("Most Relevant Line:", answer)

Most Relevant Line: Net Profit: ₹6.2 Crores

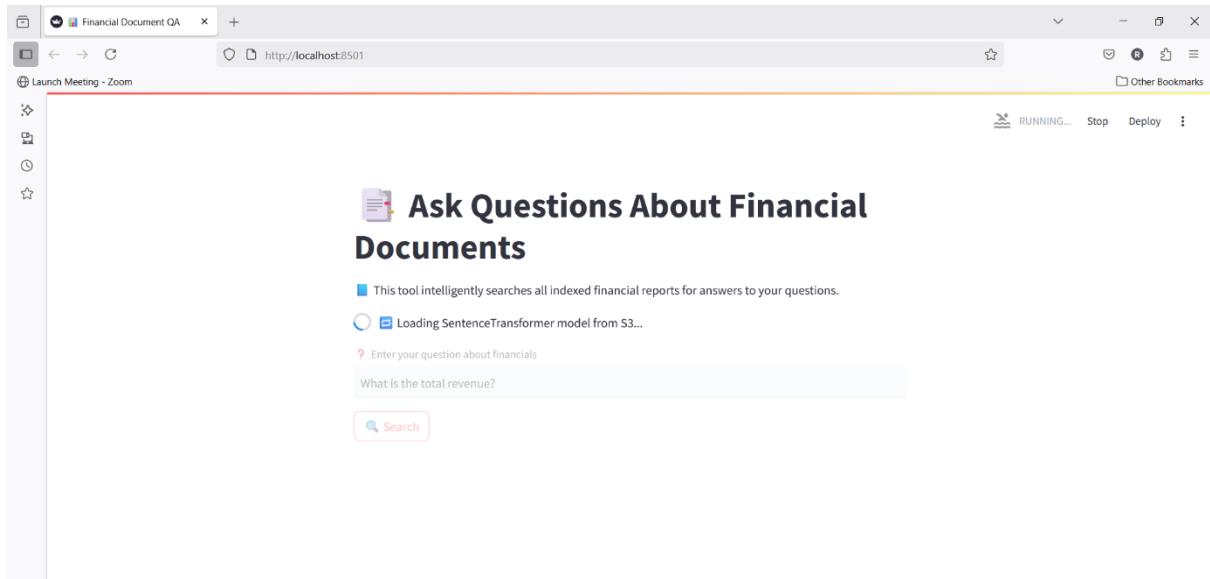
```

## Streamlit Frontend App

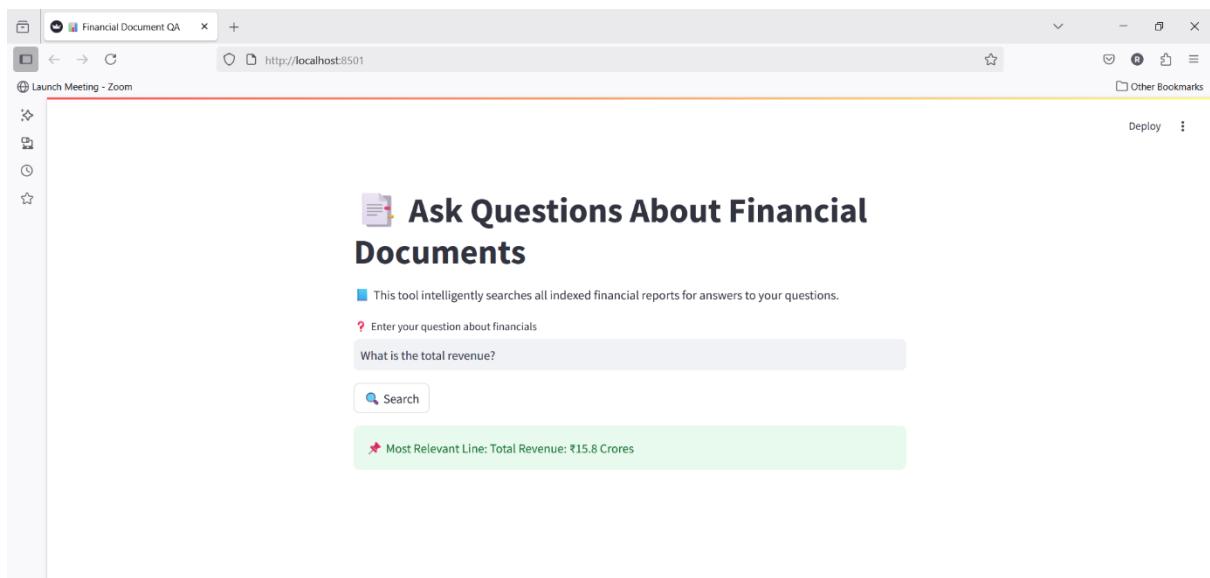
### Functionality:

- Web UI for users to **ask questions** about uploaded financial documents.
- Loads the model dynamically from **S3**.
- Searches **across all indexed documents** and returns the **most relevant answer**.
- No document selection needed — all searches are **aggregated**.

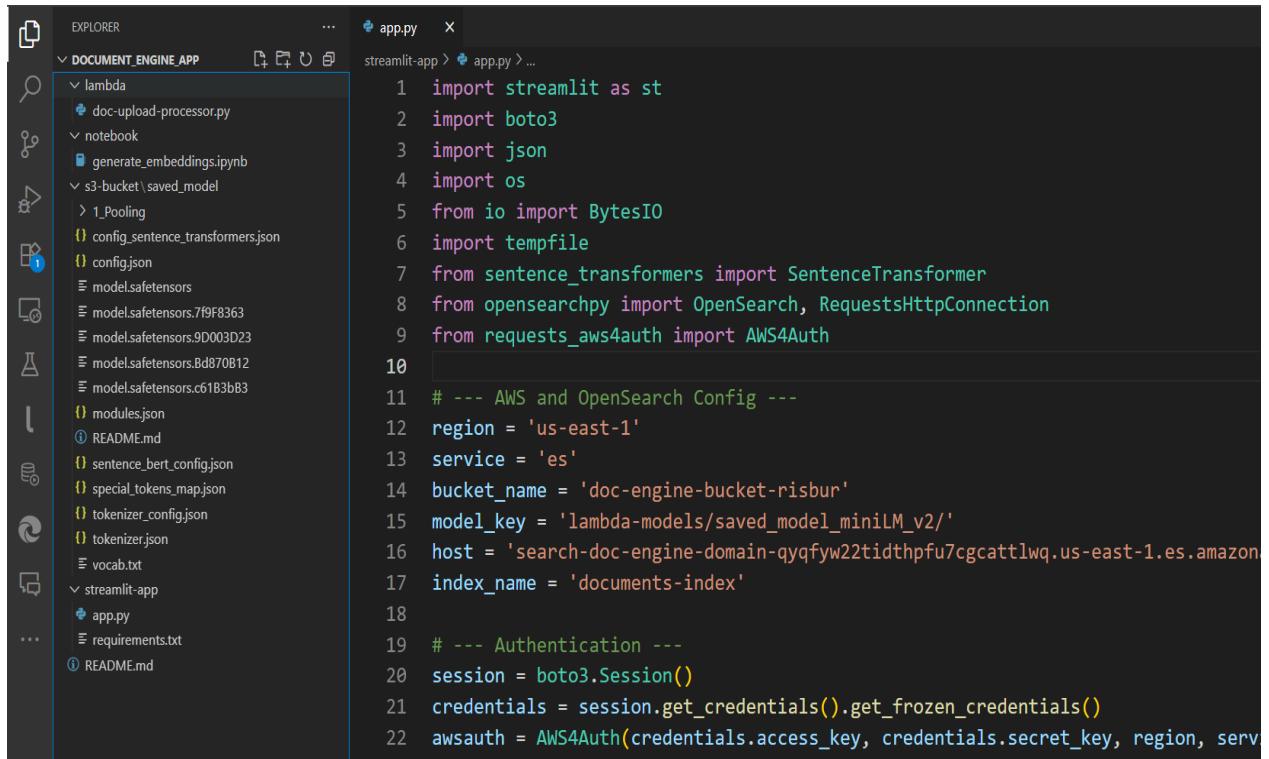
- Streamlit interface with user question.



- Output with the most relevant line highlighted.



## Project Structure



The screenshot shows a code editor interface with two panes. The left pane, titled 'EXPLORER', displays the project structure:

- DOCUMENT\_ENGINE\_APP
  - lambda
    - doc-upload-processor.py
  - notebook
    - generate\_embeddings.ipynb
  - s3-bucket\ saved\_model
    - 1\_Pooling
    - config\_sentence\_transformers.json
    - config.json
    - modelsafetensors
      - modelsafetensors.7f9F8363
      - modelsafetensors.9D003D23
      - modelsafetensors.Bd870B12
      - modelsafetensors.c61B3bB3
    - modules.json
    - README.md
    - sentence\_bert\_config.json
    - special\_tokens\_map.json
    - tokenizer\_config.json
    - tokenizer.json
    - vocab.txt
  - streamlit-app
    - app.py
    - requirements.txt
  - README.md

```
app.py  x
streamlit-app > app.py > ...

1 import streamlit as st
2 import boto3
3 import json
4 import os
5 from io import BytesIO
6 import tempfile
7 from sentence_transformers import SentenceTransformer
8 from opensearchpy import OpenSearch, RequestsHttpConnection
9 from requests_aws4auth import AWS4Auth
10
11 # --- AWS and OpenSearch Config ---
12 region = 'us-east-1'
13 service = 'es'
14 bucket_name = 'doc-engine-bucket-risbur'
15 model_key = 'lambda-models/saved_model_minilm_v2/'
16 host = 'search-doc-engine-domain-qyqfyw22tidthpfu7cgcatlwq.us-east-1.es.amazonaws.com'
17 index_name = 'documents-index'
18
19 # --- Authentication ---
20 session = boto3.Session()
21 credentials = session.get_credentials().get_frozen_credentials()
22 awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service)
```

## Technologies Used

Component	Technology
Text Extraction	AWS Textract
Storage	S3, DynamoDB
Embeddings	SentenceTransformer (MiniLM)
Model Hosting	SageMaker
Vector Index	OpenSearch KNN
App Frontend	Streamlit
Deployment	EC2 t2.micro (Free Tier)

Component	Technology
Authentication	Boto3 + SigV4 (requests-aws4auth)

---

## Benefits Recap

- **Privacy First:** No data goes to third-party LLMs.
  - **Fast Search:** Instant semantic answers from large documents.
  - **Contextual Understanding:** Not just keyword matching.
  - **Industry-Ready:** Applicable in finance, law, healthcare.
- 

## How to Run the Project

### 1. Upload Financial Docs to S3

S3 Bucket: doc-engine-bucket-risbur

Trigger Lambda: doc-upload-processor

Stores in: DocumentTextTable

### 2. Run generate\_embeddings.ipynb

Generates vector embeddings & stores in OpenSearch index.

### 3. Launch Streamlit UI

streamlit run app.py

---

- ✓ S3 document upload interface.

The screenshot shows the AWS S3 "Upload: status" page. At the top, there's a summary table with two rows: "Succeeded" (1 file, 1.2 KB) and "Failed" (0 files, 0 B). Below this, a table lists a single file: "sample\_financial\_report.pdf" (application/pdf, 1.2 KB, Succeeded). A note at the top says: "After you navigate away from this page, the following information is no longer available."

Summary	
Destination s3://doc-engine-bucket-risbur	Succeeded 1 file, 1.2 KB (100.00%)
	Failed 0 files, 0 B (0%)

Files and folders (1 total, 1.2 KB)						
Name	Folder	Type	Size	Status	Error	
sample_financial_report.pdf	-	application/pdf	1.2 KB	Succeeded	-	

- ✓ Lambda function deployment

The screenshot shows the AWS Lambda "Functions" page with a success message: "Successfully updated the function doc-upload-processor." On the left, the "Code source" tab is selected, showing the "lambda\_function.py" code in the "DOC-UPLOAD-PROCESSOR" folder. The code handles S3 events and uses Boto3 to interact with the Text Extractor service. On the right, there are "Info" and "Tutorials" tabs, along with a "Create a simple web app" tutorial section.

```

def lambda_handler(event, context):
    print("Event:", json.dumps(event))

    s3 = boto3.client('s3')
    textract = boto3.client('textract', region_name='us-east-1') # Specify region explicitly

    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = urllib.parse.unquote_plus(record['s3']['object']['key'])

        try:
            response = textract.detect_document_text(
                Document={'S3Object': {'Bucket': bucket, 'Name': key}})
        
```

- ✓ Lambda execution log showing Textract.

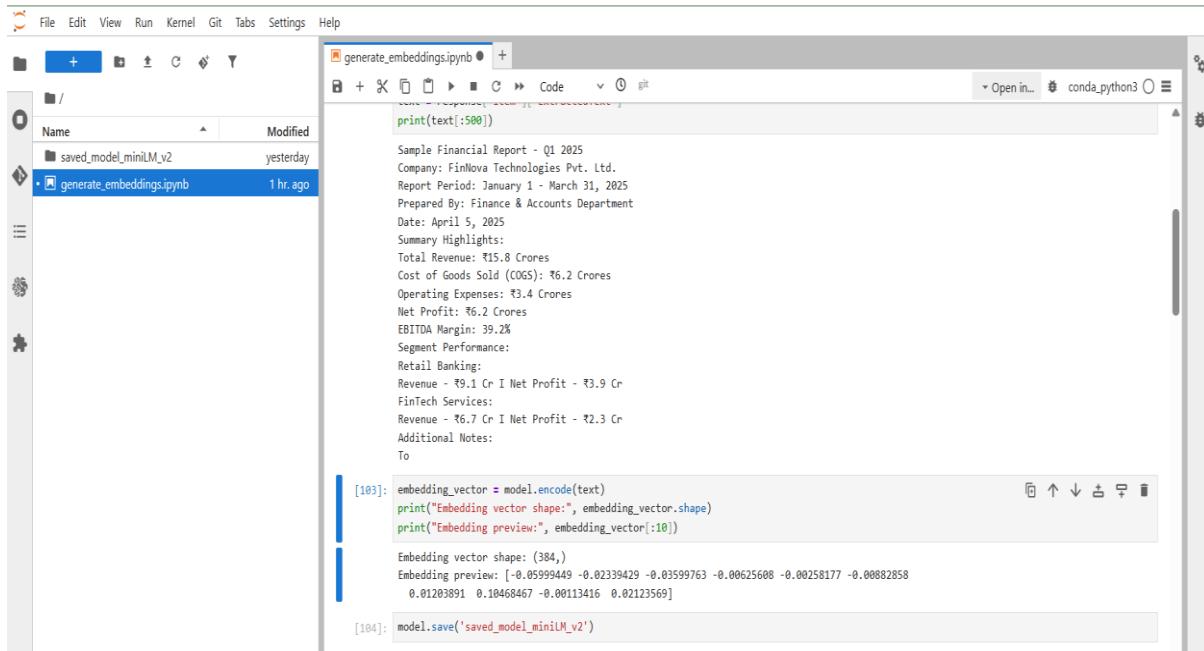
The screenshot shows the AWS CloudWatch Log Events interface. The left sidebar navigation includes 'CloudWatch' (selected), 'Favorites and recents', 'Dashboards', 'AI Operations' (Preview), 'Alarms' (0), 'Logs' (selected), 'Log groups' (selected), 'Metrics' (All metrics), and 'Explorer'. The main content area is titled 'Log events' and displays a table of log entries. The columns are 'Timestamp' and 'Message'. The first few log entries are:

Timestamp	Message
2025-06-19T14:15:14.773Z	INIT_START Runtime Version: python:3.10.v78 Runtime Version ARN: arn:aws:lambda:us-east-1::runtime:b556158cad85934b6c377a5eb9a60...
2025-06-19T14:15:15.064Z	START RequestId: 5c62ef22-f576-4bb1-b5e2-19e2389a043b Version: \$LATEST
2025-06-19T14:15:15.065Z	Event: {"Records": [{"eventVersion": "2.1", "eventSource": "aws:s3", "awsRegion": "us-east-1", "eventTime": "2025-06-19T14:15:13...", "s3": {"objectKey": "sample_financial_report.pdf", "bucket": "doc-upload-processor", "size": 123456789}}]}
2025-06-19T14:15:18.594Z	Extracted text: Financial Report - Q4 2024
2025-06-19T14:15:18.594Z	Company: GlobalShop Inc.
2025-06-19T14:15:18.594Z	Report Date: December 31, 2024
2025-06-19T14:15:18.594Z	Prepared By: Finance Department
2025-06-19T14:15:18.594Z	Summary:
2025-06-19T14:15:18.594Z	Q4 Revenue reached \$2 million, marking a 12% increase from Q3.
2025-06-19T14:15:18.594Z	Operating expenses stood at \$750,000.
2025-06-19T14:15:18.594Z	Net profit recorded: \$1.25 million.
2025-06-19T14:15:18.594Z	Top performing product: SmartGadget Pro

- ✓ DynamoDB console showing text records.

The screenshot shows the AWS DynamoDB Explore items interface. The left sidebar navigation includes 'DynamoDB' (selected), 'Tables', 'Explore items' (selected), 'PartiQL editor', 'Backups', 'Exports to S3', 'Imports from S3', 'Integrations' (New), 'Reserved capacity', and 'Settings'. The main content area shows a 'Scan' configuration panel with 'DocumentTextTable' selected as the table and 'All attributes' as the attribute projection. Below it is a message box stating 'Completed - Items returned: 1 - Items scanned: 1 - Efficiency: 100% - RCU consumed: 2'. At the bottom is a table titled 'Table: DocumentTextTable - Items returned (1)' showing one item with the key 'DocumentName (String)' set to 'sample\_financial\_report.pdf' and the value 'ExtractedText' set to 'Financial Report - Q4 2024 Company: GlobalShop Inc. Report Date: December 31...'. There are also 'Actions' and 'Create item' buttons.

- ✓ SageMaker embedding generation outputs.



The screenshot shows a Jupyter Notebook interface with a single open file named "generate\_embeddings.ipynb". The notebook displays Python code and its execution output. The code prints the first 500 characters of a sample financial report and then generates an embedding vector for the entire text. The resulting vector has a shape of (384,) and includes the first 10 elements of the preview. Finally, the model is saved to a file named "saved\_model\_minilm\_v2".

```

print(text[:500])

Sample Financial Report - Q1 2025
Company: FinNova Technologies Pvt. Ltd.
Report Period: January 1 - March 31, 2025
Prepared By: Finance & Accounts Department
Date: April 5, 2025
Summary Highlights:
Total Revenue: ₹15.8 Crores
Cost of Goods Sold (COGS): ₹6.2 Crores
Operating Expenses: ₹3.4 Crores
Net Profit: ₹6.2 Crores
EBITDA Margin: 39.2%
Segment Performance:
Retail Banking:
Revenue - ₹9.1 Cr | Net Profit - ₹3.9 Cr
FinTech Services:
Revenue - ₹6.7 Cr | Net Profit - ₹2.3 Cr
Additional Notes:
To

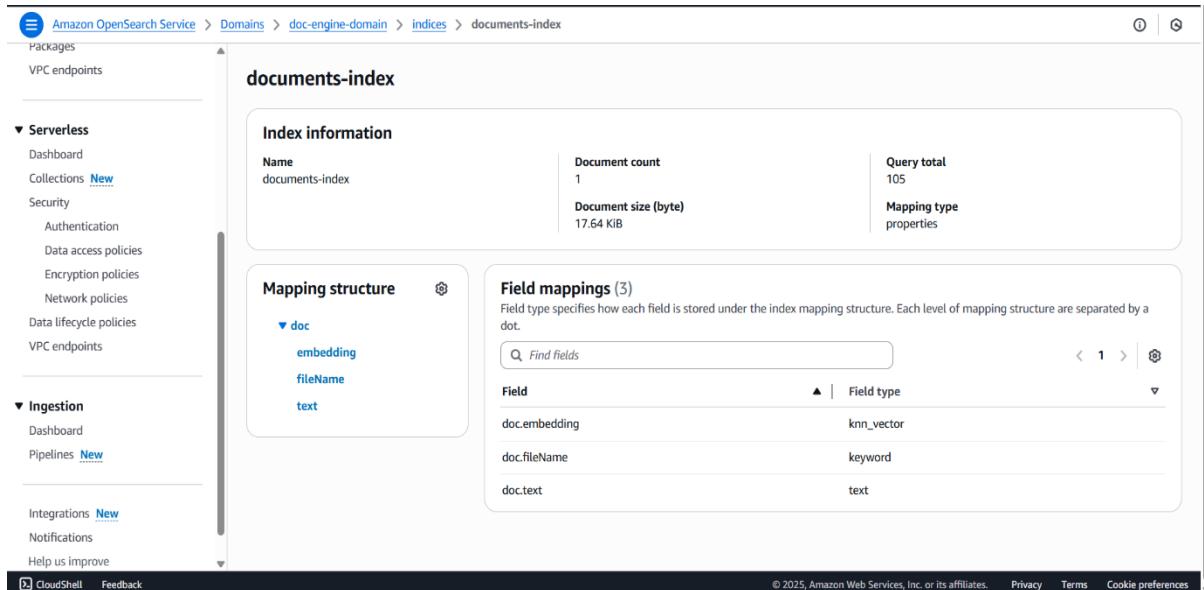
[103]: embedding_vector = model.encode(text)
print("Embedding vector shape:", embedding_vector.shape)
print("Embedding preview:", embedding_vector[:10])

Embedding vector shape: (384,)
Embedding preview: [-0.05999449 -0.02339429 -0.03599763 -0.00625608 -0.00258177 -0.00882858
 0.01203891 0.10468467 -0.00113416 0.02123569]

[104]: model.save('saved_model_minilm_v2')

```

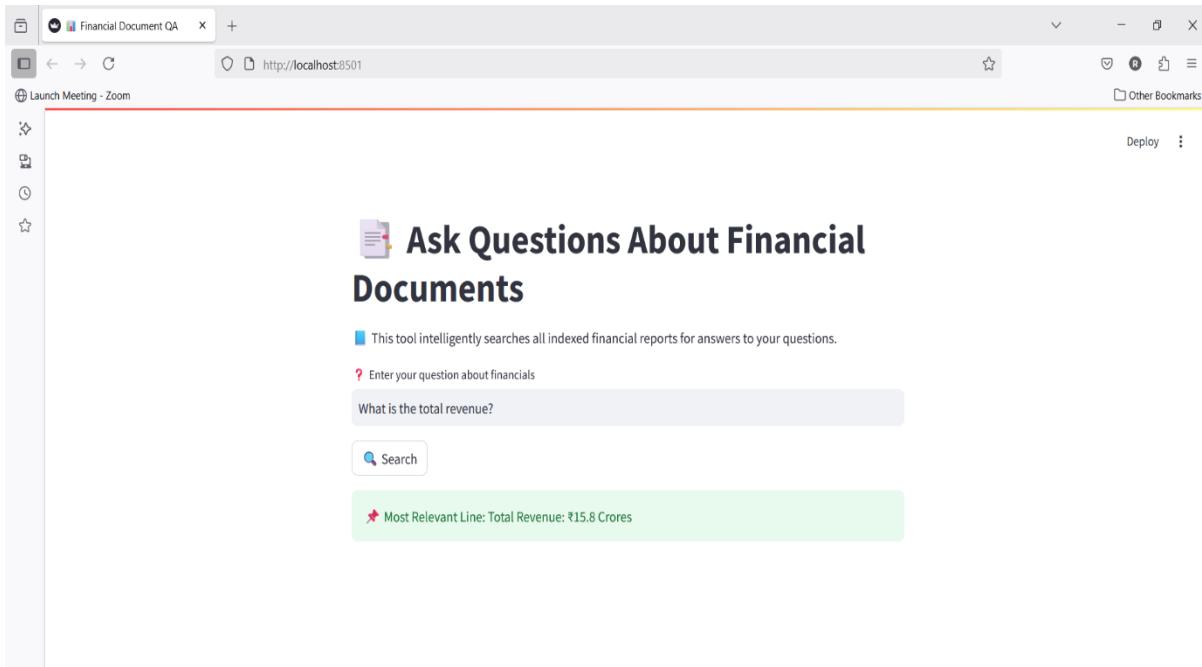
- ✓ OpenSearch dashboard (index + knn).



The screenshot shows the Amazon OpenSearch Service dashboard for the "documents-index" index. The left sidebar navigation includes "Serverless" (Dashboard, Collections, Security, Authentication, Data access policies, Encryption policies, Network policies, Data lifecycle policies, VPC endpoints), "Ingestion" (Dashboard, Pipelines), and "Integrations". The main panel displays "Index information" for the "documents-index" index, showing a document count of 1, a document size of 17.64 KiB, a query total of 105, and a mapping type of properties. Below this, the "Mapping structure" section lists fields: doc, embedding, fileName, and text. The "Field mappings (3)" section shows the field types: doc.embedding is knn\_vector, doc.fileName is keyword, and doc.text is text.

Field	Field type
doc.embedding	knn_vector
doc.fileName	keyword
doc.text	text

- ✓ Streamlit UI answering questions.



- ✓ IAM Roles configuration

Role name	Trusted entities	Last activity
<a href="#">AWSServiceRoleForAmazonOpenSearchService</a>	AWS Service: opensearchservice	12 minutes ago
<a href="#">AWSServiceRoleForSupport</a>	AWS Service: support	-
<a href="#">AWSServiceRoleForTrustedAdvisor</a>	AWS Service: trustedadvisor	-
<a href="#">EC2_StreamlitDocRole</a>	AWS Service: ec2	-
<a href="#">lambda-doc-engine-role</a>	AWS Service: lambda	8 hours ago
<a href="#">sagemaker-doc-engine-role</a>	AWS Service: sagemaker	18 minutes ago

## Hosting the Streamlit App on AWS EC2

To make the application accessible publicly and ensure scalability, I deployed the Streamlit-based Q&A interface on an **Amazon EC2 instance** using a **Linux AMI (Amazon Machine Image)** with the **Free Tier t2.micro** configuration.

### EC2 Configuration Details:

Configuration	Value
AMI	Amazon Linux 2
Instance Type	t2.micro (Free Tier eligible)
Storage	8 GB EBS (General Purpose SSD)
Security Group	Allowed inbound traffic on ports <b>8501</b> (Streamlit), <b>22</b> (SSH), and optionally <b>80</b> for HTTP
Key Pair	Generated and used for SSH access securely
Public IP Assignment	Enabled (auto-assigned)

---

## App Deployment Steps on EC2

Here's how I deployed the Streamlit application:

### 1. SSH into the EC2 instance:

```
ssh -i my-key.pem ec2-user@your-ec2-public-ip
```

### 2. Installed required system packages:

```
sudo yum update -y
```

```
sudo yum install python3 git -y
```

**3. Created a virtual environment:**

```
python3 -m venv venv
```

```
source venv/bin/activate
```

**4. Cloned the GitHub repo containing my Streamlit app:**

```
git clone https://github.com/rishit911/document_engine_proj.git
```

**5. Installed Python dependencies:**

```
pip install -r requirements.txt
```

**6. Configured AWS credentials using aws configure to authenticate with SageMaker, OpenSearch, and S3.**

**7. Ran the Streamlit application:**

```
streamlit run app.py --server.port 8501 --server.enableCORS false
```

**8. Kept the Streamlit app running persistently using nohup:**

```
nohup streamlit run app.py &
```

**9. Accessed the app via browser using:**

```
http://3.110.245.87:8501
```

## **Deployment Observations (EC2 Hosting Experience)**

**After developing and testing the complete system locally, I deployed the Streamlit app to an AWS EC2 instance for public access. This phase added a new layer of complexity and learning.**

### **EC2 Instance Configuration**

- Instance Type: t2.micro (AWS Free Tier)**
- AMI: Amazon Linux 2023 AMI**

- **Storage Volume:** Initially 8 GiB (default)
  - **Security Group:** Opened inbound rules for port 8501 (Streamlit) and 22 (SSH)
  - **Dependencies Installed:**
    - Python 3.10
    - streamlit, boto3, opensearch-py, sentence-transformers, requests-aws4auth
    - Model downloaded from S3 inside EC2 at runtime
- 

## Issue #1: Cold Start Latency

On each EC2 boot, downloading the SentenceTransformer model from S3, initializing the OpenSearch client with SigV4, and loading dependencies led to a cold start delay (30–50 seconds). This was acceptable for light usage, but for faster production-grade deployments, the following improvements are recommended:

- Cache model in EBS volume between reboots
  - Containerize and pre-load using Docker on boot
  - Use AWS Lambda with provisioned concurrency (if architecture is serverless)
- 

## Issue #2: Default Volume Too Small

When installing large packages like sentence-transformers, torch, and transformers, I encountered No space left on device errors due to the default 8 GiB volume limit.



Resolution:

- **Stopped EC2**
- **Went to Volumes > Actions > Modify Volume**
- **Increased size to 20 GiB**
- **Rebooted EC2 and confirmed new size with df -h**

## Personal Learnings & Project Reflections

### What I Learned

Working on this project was a deep dive into real-world **cloud engineering, MLOps, and AI automation**. It helped me:

- Understand **end-to-end data pipelines** from ingestion (S3) to processing (Lambda & Textract) to storage (DynamoDB) and retrieval (OpenSearch).
  - Gain hands-on experience with **semantic search** using vector databases and **SageMaker** model serving.
  - Learn to orchestrate multiple AWS services securely using **IAM roles, boto3, and SigV4 authentication**.
  - Improve my debugging and deployment skills — from local dev on Streamlit to hosting on **EC2 (free-tier t2.micro)**.
  - Structure and document a scalable, modular, and production-grade cloud-native application.
-

## Mistakes I Made (and Fixed)

- **✗ Initial Misconfigurations** in OpenSearch domain endpoint caused timeout errors — I learned how to use correct hostname syntax and adjusted connection retries.
- **✗** I tried to **load heavy ML models inside Lambda**, which exceeded size limits. I pivoted to **SageMaker inference endpoints** for scalability.
- **✗ Model saving/loading errors** (due to meta tensors and PyTorch 2.x issues) taught me the value of version compatibility and lazy loading models from S3 dynamically.
- **✗** Initially added dropdown document selection in Streamlit — later realized **aggregating search across all documents** was more useful and user-friendly.

---

## How This Project Prepares Me for a Cloud Engineer Role

- Showcases my **ability to integrate multiple AWS services** into a functional, cloud-native solution.
- Demonstrates my understanding of **event-driven architectures** using Lambda and Textract.
- Proves my skills in **deploying scalable apps on EC2**, including dependency handling, model loading, and real-time querying.
- Highlights **secure authentication practices** using AWS4Auth and boto3.Session.
- Combines **machine learning, data engineering, and cloud operations** in a unified solution.