PRJ-GAI-027: Retrieval-Augmented Generation (RAG) Pipeline

Certification: AWS Certified Generative AI Developer – Professional
Domain: Generative AI Application Development

---

1. Project Overview

This project implements a complete, end-to-end **Retrieval-Augmented Generation (RAG)** pipeline on AWS. Large Language Models (LLMs) are powerful, but their knowledge is limited to the data they were trained on, and they can sometimes "hallucinate" or invent facts. RAG solves this problem by grounding the LLM with up-to-date, external information. It retrieves relevant documents from a private knowledge base and provides them to the LLM as context when generating an answer, resulting in more accurate, factual, and trustworthy responses.

We will build a serverless RAG pipeline that can answer questions about a private collection of documents. We will use **Amazon Bedrock** to access foundation models for both **embedding** (turning text into vectors) and **generation**. The document vectors will be stored and indexed in **Amazon OpenSearch Service** for efficient similarity search. The entire process, from data ingestion to answer generation, will be orchestrated by AWS Lambda functions.

Key Objectives

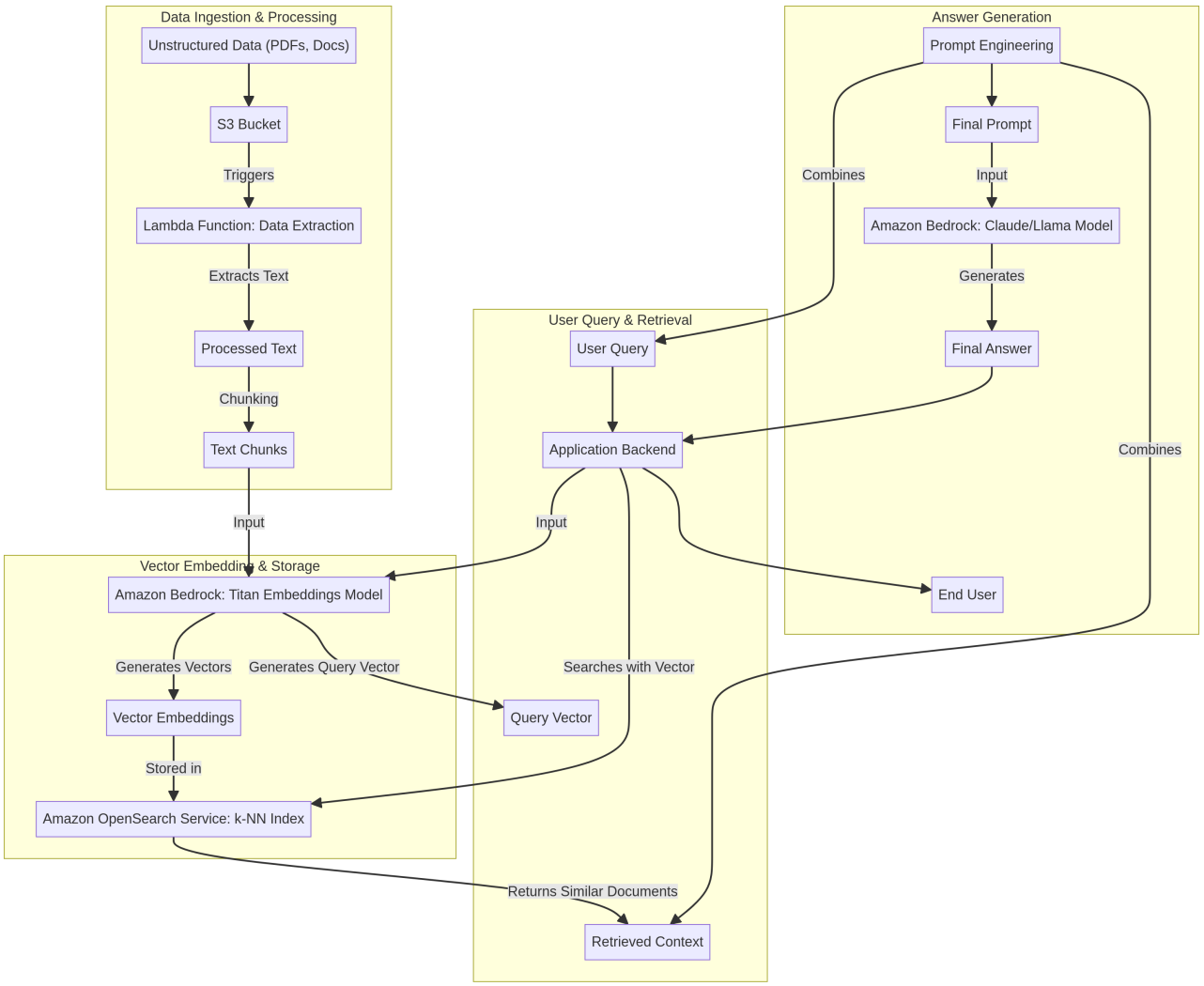
- Build a data ingestion pipeline that automatically processes and chunks documents uploaded to S3.
- Use the Amazon Titan Embeddings model via Amazon Bedrock to convert text chunks into vector embeddings.
- Store and index these vector embeddings in an Amazon OpenSearch Service k-NN (k-Nearest Neighbors) index.

- Develop a query pipeline that takes a user question, converts it to a vector, and retrieves the most relevant document chunks from OpenSearch.
- Use prompt engineering to combine the original question with the retrieved context.
- Send the final, augmented prompt to a powerful generative model (like Anthropic's Claude) via Amazon Bedrock to generate a final, context-aware answer.

---

2. Architecture

The architecture is split into two main pipelines: an asynchronous data ingestion pipeline and a synchronous query pipeline.



Data Ingestion Pipeline (Asynchronous):

1. **Upload:** A user uploads unstructured documents (PDFs, Word docs, etc.) to a designated S3 bucket.
2. **Extraction:** The S3 `PutObject` event triggers a Lambda function. This function uses libraries like `pdf-parse` to extract the raw text from the documents.
3. **Chunking:** The extracted text is split into smaller, semantically meaningful chunks (e.g., paragraphs). This is crucial for effective retrieval.
4. **Embedding:** Each text chunk is sent to the **Amazon Titan Embeddings** model via the Amazon Bedrock API. The model returns a high-dimensional vector embedding for each chunk.
5. **Storage:** The Lambda function stores each vector embedding, along with its corresponding text chunk, in an **Amazon OpenSearch Service** cluster. The data is stored in an index specifically configured for efficient k-NN similarity search.

Query Pipeline (Synchronous):

1. **User Query:** A user submits a question through an application frontend (e.g., a web app).
2. **Query Embedding:** The application backend (e.g., another Lambda function exposed via API Gateway) takes the user's question and sends it to the same **Amazon Titan Embeddings** model to get a query vector.
3. **Retrieval:** The backend uses this query vector to perform a k-NN search against the OpenSearch index. OpenSearch efficiently finds the `k` most similar document vectors (and their corresponding text chunks) from the knowledge base.
4. **Prompt Augmentation:** The backend performs **prompt engineering**. It constructs a new prompt that includes the retrieved text chunks as context, along with the original user question. A typical prompt template looks like this: > "You are a helpful assistant. Answer the following question based only on the context provided below. > > **Context:** > [Retrieved text chunk 1] > [Retrieved text chunk 2] > … > > **Question:** > [Original user question]"
5. **Generation:** This final, augmented prompt is sent to a powerful generative LLM (like **Anthropic's Claude** or **Meta's Llama 2**) via the Amazon Bedrock API.

6. **Final Answer:** The LLM generates an answer that is grounded in the provided context. The backend returns this answer to the user.

---

3. Prerequisites

- An AWS account with administrative permissions.
- Access to foundation models (Titan Embeddings, Claude) enabled in **Amazon Bedrock**.
- Node.js and npm installed for the Lambda functions.

---

4. Step-by-Step Implementation Guide

Step 4.1: Set Up the Vector Database

1. Go to the **Amazon OpenSearch Service console -> Create domain**.
2. **Domain name:** `rag-vector-db`
3. **Deployment type:** Development and testing
4. **Engine options:** Ensure the version is compatible with k-NN (most recent versions are).
5. **Network:** For simplicity, choose **Public access**, but for production, you would place it within a VPC. Configure the access policy to restrict access to your IP address or specific IAM roles.
6. Create the domain. This can take 15-20 minutes.

Step 4.2: Create the Data Ingestion Lambda

1. Create a Node.js Lambda function named `ingestion-lambda`.
2. Give it an IAM role with permissions for S3 (`GetObject`), Bedrock (`InvokeModel`), and OpenSearch (`es:ESHttpPost`).
3. **Configure Trigger:** Set the S3 bucket where you will upload documents as the trigger.

4. **`index.js` Code (Conceptual):**

```
const { S3Client, GetObjectCommand } = require("@aws-sdk/client-s3");
const { BedrockRuntimeClient, InvokeModelCommand } = require("@aws-
sdk/client-bedrock-runtime");
const { Client } = require("@opensearch-project/opensearch");
const pdf = require("pdf-parse");


exports.handler = async (event) => {
    const bucket = event.Records[0].s3.bucket.name;
    const key =
decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, "
"));


    // 1. Get document from S3 and extract text
    const s3Client = new S3Client({});
    const s3Object = await s3Client.send(new GetObjectCommand({
Bucket: bucket, Key: key }));
    const pdfData = await pdf(s3Object.Body);
    const text = pdfData.text;


    // 2. Chunk the text
    const chunks = text.split("\n\n"); // Simple chunking by paragraph


    // 3. Get embeddings for each chunk
    const bedrockClient = new BedrockRuntimeClient({ region: "us-east-
1" });
    const embeddings = [];
    for (const chunk of chunks) {
        const response = await bedrockClient.send(new
InvokeModelCommand({
            modelId: "amazon.titan-embed-text-v1",
            contentType: "application/json",
            body: JSON.stringify({ inputText: chunk }),
        }));
        const result =
JSON.parse(Buffer.from(response.body).toString());
        embeddings.push({ text: chunk, vector: result.embedding });
    }


    // 4. Store in OpenSearch
    const osClient = new Client({ node: "<YOUR_OPENSEARCH_ENDPOINT>"
```

```
});
    for (const item of embeddings) {
        await osClient.index({
            index: "knowledge-base",
            body: {
                text: item.text,
                embedding: item.vector,
            },
        });
    }
};
```

Note: You also need to create the `knowledge-base` index in OpenSearch with the correct mapping for the `embedding` field (type `knn_vector`).

Step 4.3: Create the Query Lambda

1. Create another Node.js Lambda function named `query-lambda`.
2. Give it permissions for Bedrock and OpenSearch.
3. Expose this Lambda via an **API Gateway** endpoint.
4. **`index.js` Code (Conceptual):**

```
// ... (similar client initializations)


exports.handler = async (event) => {
    const userQuery = JSON.parse(event.body).query;


    // 1. Get embedding for the user query
    const queryEmbedding = await getEmbedding(userQuery);


    // 2. Search OpenSearch for similar documents
    const searchResponse = await osClient.search({
        index: "knowledge-base",
        body: {
            size: 3, // Get top 3 results
            _source: ["text"],
            query: {
                knn: {
                    embedding: {
                        vector: queryEmbedding,
                        k: 3,
                    },
                },
            },
        },
    });
    const context = searchResponse.body.hits.hits.map(hit =>
hit._source.text).join("\n---\n");


    // 3. Construct the augmented prompt
    const augmentedPrompt = `Context: ${context}\n\nQuestion:
${userQuery}\n\nAnswer:`;


    // 4. Invoke the generative model
    const generationResponse = await bedrockClient.send(new
InvokeModelCommand({
        modelId: "anthropic.claude-v2",
        contentType: "application/json",
        body: JSON.stringify({ prompt: `\n\nHuman:
${augmentedPrompt}\n\nAssistant:`, max_tokens_to_sample: 500 }),
    }));
    const finalAnswer =
```

```
JSON.parse(Buffer.from(generationResponse.body).toString()).completion;


    return {
        statusCode: 200,
        body: JSON.stringify({ answer: finalAnswer }),
    };
};
```

---

5. How to Test

1. **Upload Documents:** Upload some PDF or text documents to your S3 bucket. This will trigger the ingestion Lambda and populate your OpenSearch index.
2. **Query the API:** Use a tool like `curl` or Postman to send a POST request to your API Gateway endpoint with a JSON body like `{"query": "What is the main topic of the document?"}`.
3. **Observe the Result:** The API should return a JSON response containing the LLM-generated answer, which should be based on the content of the documents you uploaded.

---

6. Cleanup

1. **Delete the API Gateway endpoint.**
2. **Delete the `ingestion-lambda` and `query-lambda` functions.**
3. **Delete the Amazon OpenSearch Service domain.**
4. **Empty and delete the S3 bucket.**
5. **Delete the IAM roles** created for the Lambda functions.

Business Context

The Problem

Organizations want to leverage generative AI for content creation, customer service, and automation but face challenges with data privacy, prompt injection attacks, and hallucinations. Lack of guardrails leads to inappropriate content generation and compliance risks.

The Solution

Secure generative AI application with content filtering, prompt engineering, and responsible AI controls. Implements Amazon Bedrock with guardrails, RAG (Retrieval Augmented Generation) for accuracy, and comprehensive logging for auditability. Protects sensitive data while enabling AI innovation.

Business Value

- **Productivity Gains:** Automates content creation, reducing manual effort by 70%
- **Customer Experience:** 24⁄7 AI-powered support improves satisfaction scores
- **Data Privacy:** Keeps sensitive data within AWS, no third-party AI exposure
- **Responsible AI:** Content filters prevent inappropriate or harmful outputs

Risk Mitigation

Prevents data leakage to third-party AI providers, blocks prompt injection attacks, filters inappropriate content, and ensures compliance with AI regulations.

GRC Mapping

Compliance Frameworks

- **NIST AI RMF:** Govern, Map, Measure, Manage
- **ISO/IEC 42001:** AI Management System

- **OWASP Top 10 for LLM:** Prompt injection, data leakage, model theft prevention
- **Responsible AI Framework:** Transparency, fairness, accountability

Security Controls Implemented

- Content filtering and moderation
- Prompt injection detection and prevention
- Data anonymization before AI processing
- Model access controls and API rate limiting
- Comprehensive logging of AI interactions

Audit Evidence

- AI interaction logs with prompts and responses
- Content filter activation records
- Data access and processing logs
- Model usage and cost tracking

Regulatory Alignment

- **AI Act (EU):** High-risk AI system requirements
- **GDPR:** Article 22 (Automated decision-making), Article 35 (DPIA)
- **CCPA:** Consumer data privacy in AI systems
- **SOC 2:** CC6.1 (Data access), CC7.2 (Monitoring)