

AWS RAG Pipeline Challenge

Healthcare Compliance Standards Query System

Challenge Overview

Build a Retrieval-Augmented Generation (RAG) pipeline using AWS services to query NIAHO (National Integrated Accreditation for Healthcare Organizations) standards. You will process a PDF document, chunk it intelligently, create vector embeddings, and implement a dual-mode query system using AWS Bedrock.

Time Estimate: 4-6 hours

Difficulty: Intermediate

Prerequisites: AWS account, basic Python knowledge, understanding of RAG concepts

Key Requirement: Your system must handle TWO types of queries:

1. General Questions - Conversational queries about standards (e.g., "What are the fire safety requirements?")
 2. Exact Standard Lookups - Requests for specific standard citations with verbatim text (e.g., "Show me standard 2.2.3.4")
-

What You'll Build

A serverless RAG pipeline that:

1. Ingests a NIAHO standards PDF document
2. Extracts and chunks text by logical sections (chapters/subsections)
3. Generates vector embeddings for each chunk
4. Stores embeddings in a searchable vector database
5. Intelligently detects query type and routes accordingly:
 - o Q&A Mode: Natural language questions → RAG-generated answers with citations
 - o Citation Mode: Specific standard requests → Exact verbatim text from standards

No user interface required - all functionality can be tested via AWS Console (Lambda Test Events) or CLI.

AWS Services Required (All Free Tier Compatible)

Service	Purpose	Free Tier Limit
S3	Store PDF, chunks, and vector embeddings	5 GB storage, 20,000 GET/2,000 PUT requests
Lambda	Processing functions	1M requests/month, 400,000 GB-seconds compute
Bedrock - Titan Embeddings	Generate vector embeddings	Pay-per-use (~\$0.0001/1K tokens)
Bedrock - Claude 3.5 Haiku	Query processing and generation	Pay-per-use (~\$0.001/1K tokens)

Estimated Total Cost: < \$1.50 for entire challenge completion

Note: All vector storage and retrieval happens in S3 using JSON files. No separate vector database required.

Architecture Requirements

S3 Bucket Structure:

```
|── raw/niaho_standards.pdf  
|── chunks/  
|   |── chunk_001.json  
|   |── chunk_002.json  
|   |── ...  
└── embeddings/  
    └── embeddings_index.json (all vectors)
```

Flow:

NIAHO PDF (S3/raw/)

↓

Lambda: PDF Processor → Extract & Chunk

↓

S3 (chunks/) → Store JSON chunks

↓

Lambda: Embedding Generator

↓

Bedrock Titan Embeddings

↓

S3 (embeddings/) → Store vectors + metadata

↓

Lambda: Query Handler

↓

Load embeddings from S3 → Cosine similarity search

↓

Bedrock Claude 3.5 Haiku (RAG)

↓

Response with citations

Technical Requirements

1. PDF Processing & Chunking (Lambda Function)

- **Extract text from provided NIAHO standards PDF**
- **Chunk by chapter (preserve semantic meaning)**
- **Extract chapter identifiers (e.g., "QM.1", "LS.2", "IC.3") from text**

- **Each chunk should include:**
 - **chunk_id: Unique identifier**
 - **text: Content (500-1500 tokens recommended)**
 - **metadata:**
 - **document: "NIAHO Standards" (or document name)**
 - **section: Section name (e.g., "Quality Management", "Life Safety", "Infection Control")**
 - **chapter: Chapter identifier (e.g., "QM.1", "LS.2")**
 - **embedding: Vector representation (1536 dimensions for Titan)**
- **Store chunks in S3 as JSON**
- **Deliverable: pdf_processor.py Lambda function**

2. Vector Embedding Generation (Lambda Function)

- **Use Amazon Titan Embeddings G1 - Text model**
- **Process all chunks from S3 chunks/ folder**
- **Generate embeddings for each chunk**
- **Store individual chunk JSONs with embeddings back to S3**
- **Create consolidated embeddings_index.json for quick loading**
- **Implement batch processing to stay within Lambda timeout (can process chunks in parallel or sequentially)**
- **Deliverable: embedding_generator.py Lambda function**

3. Vector Storage (S3)

- **Bucket structure:**
- **s3://your-bucket-name/**
- **| — raw/**
- **| | — niaho_standards.pdf**
- **| — chunks/**

- | |—chunk_001.json
- | |—chunk_002.json
- | \—...
- \—embeddings/
- \—embeddings_index.json
- **Each chunk JSON should contain:**

JSON

```
{
  "chunk_id": "001",
  "text": "Chapter content...",
  "metadata": {
    "document": "NIAHO Standards",
    "section": "Quality Management",
    "chapter": "QM.1"
  },
  "token_count": 850,
  "embedding": [0.123, -0.456, ...] // 1536 dimensions
}
```

- **embeddings_index.json aggregates all chunks for efficient loading**
- **Deliverable: S3 bucket structure screenshot and sample chunk JSON**

4. Query Handler (Lambda Function)

This is the core of your system and must intelligently handle both query types.

Query Type Detection:

- **Implement logic to detect if user is asking a question vs. requesting a specific chapter**
- **Patterns to detect citation mode:**

- "Show me chapter X"
- "What does chapter QM.1 say?"
- "Cite LS.2"
- "Give me the exact text for..."
- Contains chapter ID patterns (e.g., "QM.1", "LS.2", "IC.3")

Mode 1: Q&A Mode (General Questions)

- Generate query embedding using Titan
- Load embeddings from S3 (embeddings_index.json)
- Perform in-memory similarity search (cosine similarity)
- Retrieve top 3-5 most relevant chunks
- Pass chunks to Claude 3.5 Haiku with instruction to summarize/synthesize
- Return response with:
 - Natural language answer
 - Citations (document, section, chapter)
 - Confidence score

Mode 2: Citation Mode (Specific Chapters)

- Extract chapter identifier from query (regex/NLP)
- Search chunks by metadata.chapter field (exact match)
- If found, return EXACT verbatim text from the chunk (no AI generation)
- Must include:
 - Chapter ID (e.g., "QM.1")
 - Full verbatim text as it appears in document
 - Section name
 - Document name
 - Timestamp of retrieval
- If not found, use vector search as fallback + inform user

Important Scalability Note: This in-memory approach works for a single NIAHO document (~100-500 chunks, ~5-10MB embeddings). When scaling to multiple documents or larger datasets (1000+ chunks), you will need to migrate to a proper vector database like:

- AWS OpenSearch Serverless
- Amazon RDS with pgvector extension
- Pinecone, Weaviate, or similar vector DB

For this challenge, in-memory is acceptable and efficient given the scope.

Deliverable: query_handler.py Lambda function with dual-mode logic

Prompt Engineering Requirements

Your query handler must use different strategies for each mode:

Q&A Mode Prompt:

- Instructs the model to answer based on provided context
- Requires citation of specific chapters/sections
- Allows summarization and synthesis across multiple chunks
- Handles cases where information is not found in retrieved chunks
- Returns responses in consistent JSON format

Citation Mode Response:

- NO AI generation - return exact text only
- Must preserve original formatting, punctuation, and wording
- Include disclaimer: "Exact text from NIAHO standards document"
- Add metadata about source location

Example Output Formats:

Q&A Mode:

JSON

{

```
"query": "What are the requirements for quality improvement programs?",  
"query_type": "question",  
"answer": "NIAHO standards require healthcare organizations to maintain ongoing quality improvement programs that systematically monitor and evaluate the quality and safety of patient care. These programs must include data collection, analysis, and action plans for identified opportunities for improvement.",  
"citations": [  
  {  
    "chunk_id": "012",  
    "document": "NIAHO Standards",  
    "section": "Quality Management",  
    "chapter": "QM.1",  
    "relevance_score": 0.94  
  }  
,  
  {"confidence": "high"}  
]
```

Citation Mode:

JSON

```
{  
  "query": "Show me chapter QM.1",  
  "query_type": "citation",  
  "chapter": "QM.1",  
  "exact_text": "The organization's leaders create and maintain a culture of safety and quality throughout the hospital. Quality improvement programs shall be established to systematically monitor, analyze, and improve the quality of care and patient safety. These programs must include mechanisms for identifying opportunities for improvement."}
```

improvement, implementing changes, and measuring the effectiveness of those changes.",

```
"source": {  
    "document": "NIAHO Standards",  
    "section": "Quality Management",  
    "chapter": "QM.1",  
    "chunk_id": "012"  
},  
"disclaimer": "Exact text from NIAHO standards document - retrieved [timestamp]"  
}
```

Evaluation Criteria

Your submission will be evaluated on:

Technical Implementation (50%)

- PDF successfully processed and chunked logically
- Chapter IDs extracted and stored in metadata correctly
- Metadata structure follows document > section > chapter hierarchy
- Embeddings generated and stored correctly
- Query type detection works accurately (Q&A vs. Citation)
- Q&A mode: Vector similarity search returns relevant results
- Q&A mode: RAG pipeline produces accurate synthesized answers with citations
- Citation mode: Exact chapter text returned verbatim (no AI paraphrasing)
- Citation mode: Metadata search by chapter ID works correctly
- Error handling for chapters not found
- Edge cases addressed (ambiguous queries, partial chapter IDs)

Code Quality (20%)

- [] Clean, readable Python code
- [] Proper IAM roles and permissions (least privilege)
- [] Environment variables used for configuration
- [] Logging implemented (CloudWatch)
- [] Comments explaining complex logic
- [] Lambda memory configuration documented (recommend 1024MB-2048MB for embedding operations)

Important Note on Lambda Memory: Lambda has a maximum memory limit of 10GB. The in-memory vector search approach used in this challenge works because you're loading a single document's embeddings (~5-10MB). When you scale to multiple documents or thousands of chunks, the embeddings alone could exceed Lambda's memory capacity. At that point, you must migrate to a proper vector database (OpenSearch, pgvector, Pinecone, etc.) that can handle pagination and distributed search.

Cost Optimization (15%)

- [] Efficient chunking strategy (minimize redundant embeddings)
- [] Batch processing where appropriate
- [] Stays within \$5 budget for all test queries
- [] Lambda memory/timeout optimized

Documentation (15%)

- [] README with setup instructions
- [] Architecture diagram (can be hand-drawn/whiteboard photo)
- [] Test queries and expected results documented
- [] Challenges faced and solutions explained

Deliverables

Submit a GitHub repository containing:

1. Lambda Functions (3 Python files)

- **pdf_processor.py** - PDF ingestion and chunking
- **embedding_generator.py** - Vector embedding creation
- **query_handler.py** - RAG query processing

2. Infrastructure Code

- **cloudformation_template.yaml OR setup instructions**
- **IAM policy documents (S3 read/write, Bedrock invoke)**
- **S3 bucket structure and naming conventions**

3. Documentation

- **README.md - Setup and usage instructions**
- **ARCHITECTURE.md - System design explanation**
- **TEST_RESULTS.md - Sample queries and outputs**

4. Test Data

- **5 sample Q&A queries with outputs**
- **5 sample citation queries with exact text outputs**
- **Corresponding outputs demonstrating both modes working correctly**
- **Screenshot of CloudWatch logs showing successful execution**
- **Examples of edge case handling**

Test Queries (Examples)

Your system should be able to handle both query types:

Q&A Mode Queries:

1. "What are the requirements for quality improvement programs?"
2. "Describe the infection control requirements for surgical areas"
3. "How should hospitals handle medication errors?"
4. "What are the staff competency assessment requirements?"
5. "Explain the patient rights and responsibilities outlined in the standards"

Citation Mode Queries:

1. "Show me chapter QM.1"
2. "What does chapter LS.2 say exactly?"
3. "Give me the exact text for chapter IC.3"
4. "Cite chapter PE.1"
5. "I need the verbatim language from chapter MM.2"

Mixed/Edge Case Queries:

1. "What does the quality management chapter say and also show me the exact text"
 2. "Is there a chapter about hand hygiene? Show me the exact wording"
 3. "Chapters related to patient safety" (ambiguous - should trigger Q&A mode)
-

Submission Guidelines

1. **Code Repository:** Provide public GitHub repo link
2. **AWS Resources:** Include screenshots of deployed resources (Lambda functions, S3 bucket structure)
3. **Demo Video:** Optional 3-5 minute walkthrough showing query execution
4. **Time Log:** Approximate hours spent on each component

Deadline: [To be specified by recruiter]

Helpful Resources

- [AWS Bedrock Documentation](#)
 - [Titan Embeddings Guide](#)
 - [Lambda Best Practices](#)
 - [S3 Best Practices](#)
 - [Cosine Similarity in Python](#)
-

Bonus Challenges (Optional)

Impress us by implementing:

- **Hybrid Query Mode:** Detect when user wants both explanation AND exact text (e.g., "Explain quality management requirements and show me chapter QM.1")
 - **Partial Chapter Matching:** Handle queries like "show me all chapters in the Quality Management section"
 - **Multi-Chapter Citations:** When user asks for multiple chapters (e.g., "show me chapters QM.1 and QM.2")
 - **Semantic Chapter Search:** "Find chapters related to medication management" → returns list of relevant chapter IDs
 - **Streaming Responses:** Implement Bedrock response streaming for Q&A mode
 - **Cost Dashboard:** Calculate and display per-query cost by mode
 - **Scalability Plan:** Document how you would refactor this to handle 10+ documents and 5000+ chunks (would need to migrate from in-memory to proper vector database)
-

Support

If you encounter issues with:

- **AWS account setup or free tier limits**
- **Bedrock model access (requires requesting access in AWS Console)**
- **Technical questions about the challenge**

Contact: rwdostert@medlaunchconcepts.com

Good luck! We're excited to see your implementation.