Cloud Test - RAG System Design

1. Assumptions

This section outlines the key assumptions used when designing the cloud-based RAG system for the consulting firm's knowledge management use case. These assumptions ensure clarity around scale, document characteristics, user behavior, and operational constraints.

1.1 Document Volume & Characteristics

- The organization has **10+ years** of accumulated documents.
- Estimated **total document count**: 250,000 – 500,000 files.
- Estimated **total storage size**: 3–8 TB based on PDFs, DOCX, PPTX, and email archives.
- Approximately **20–30%** of documents may require OCR (scanned PDFs).
- Documents are distributed across:
    - Network drives
    - Legacy internal systems
    - Email exports
    - SharePoint / Confluence / File servers
- Documents are assumed to have **varying access levels** (public, internal, confidential, restricted).

1.2 Update Frequency

- New documents added **daily to weekly**.
- Estimated **50–200 new documents per day** from ongoing projects.
- Updates to existing documents are infrequent but possible.
- Document ingestion must support **incremental indexing** (processing only new/updated items).

1.3 User Access Patterns

- 500 employees total.
- **Active daily users**: ~150–250.
- **Peak concurrency** requirement: 500 simultaneous users during all-hands meetings or departmental usage spikes.
- Users expect:
    - Fast responses (<3 seconds for most queries).
    - Accurate answers with source citations.
    - Ability to filter results by project, date, region, department.

1.4 User Security Requirements

- All users authenticate using **SSO/SAML via corporate identity provider** (Okta/Azure AD).
- User access levels vary:
    - Analyst
    - Consultant
    - Manager
    - Executive
- Access to documents must reflect **existing permissions and clearance levels**.
- All data must remain in **US-based cloud regions** due to compliance rules.

1.5 Technology & Operational Constraints

- Cloud budget: **$8,000 per month**.
- Cloud platform assumed: **AWS**, using managed services where possible.
- System must maintain **99.5% uptime**, equivalent to ~3.65 hours downtime per month.

- Preference for:
    - Managed vector databases
    - Serverless or autoscaling compute
    - Low maintenance operational overhead

1.6 RAG System Requirements

- Must support ingestion of:
    - PDFs
    - DOC/DOCX
    - PPT/PPTX
    - Email messages (.eml / .msg)
    - Text files
- Support for:
    - Document chunking
    - Embedding generation
    - Vector search with metadata filters
    - Source citation extraction
- Retrieval must reflect **document-level ACL filtering** before LLM generation.

1.7 User Interface Expectations

- Web-based chat interface for desktop.
- Optional mobile-friendly responsive design.
- Users expect:
    - Query suggestions
    - Highlighted citations
    - Links to original documents

- Export to PDF/Email

1.8 Additional Compliance Assumptions

- All logs, audit events, and storage must remain in **US-East or US-West** regions.
- No customer PII or regulated data leaves the environment.
- All data-in-transit and data-at-rest must be fully encrypted (TLS/KMS).

2. High-Level Architecture

This section describes the end-to-end architecture of the secure, cloud-based Retrieval-Augmented Generation (RAG) system. The design ensures efficient document ingestion, reliable vector search, secure access control, and accurate natural-language responses with source citations.

2.1 Architecture Overview

The system follows a modular, cloud-native design consisting of four major layers:

1. **Ingestion Layer** – Collect, parse, OCR, chunk, embed, and store documents.
2. **Storage & Indexing Layer** – Maintain all files, embeddings, metadata, and access controls.
3. **RAG Application Layer** – Retrieve relevant content, enforce permissions, run LLM generation, return citations.
4. **User Interface Layer** – Provide a clean, chat-based UI for employees on web and mobile.

Each layer is isolated using AWS networking, IAM policies, KMS encryption, and strict document-level access controls.

2.2 Detailed Component Breakdown

A. Document Ingestion Pipeline

Responsible for securely collecting and indexing new and historical documents.

Components:

- **S3 Intake Buckets** (US-only)
- **AWS Lambda Functions** for:
    - File preprocessing
    - OCR using Amazon Textract
    - Document chunking
    - Embedding generation
- **Bedrock Embedding Model** (Titan / Cohere)
- **Vector Database** (OpenSearch Serverless or Pinecone)
- **DynamoDB Metadata Store** for:
    - Document titles
    - Tags & departments
    - Access control mapping
    - File versioning

Events are processed automatically whenever a new file is uploaded or modified.

B. Storage & Indexing Layer

This layer ensures all documents and embeddings are stored securely and remain searchable.

Components:

- **Amazon S3** for raw documents
- **OpenSearch/Pinecone** for embedding vectors
- **DynamoDB** for metadata + ACLs
- **KMS** for encryption of:
    - Vector DB
    - S3 buckets

- DynamoDB tables

Metadata filtering ensures only documents the user is allowed to access are included in retrieval.

C. RAG Orchestration Layer

The core intelligence of the system.

Workflow:

1. User submits query
2. The system retrieves:
    - top-k vector matches
    - metadata filters (department, access level, role)
3. Re-ranking improves precision (optional)
4. LLM (Claude / GPT-4 / Mistral) generates an answer
5. System attaches citations containing:
    - document title
    - file link
    - exact text excerpt

This layer runs on:

- **ECS Fargate** or **Lambda** (depending on workload)
- **API Gateway (Private)** as the entry point

D. User Interface Layer

The end-user-facing chat system providing natural-language Q&A.

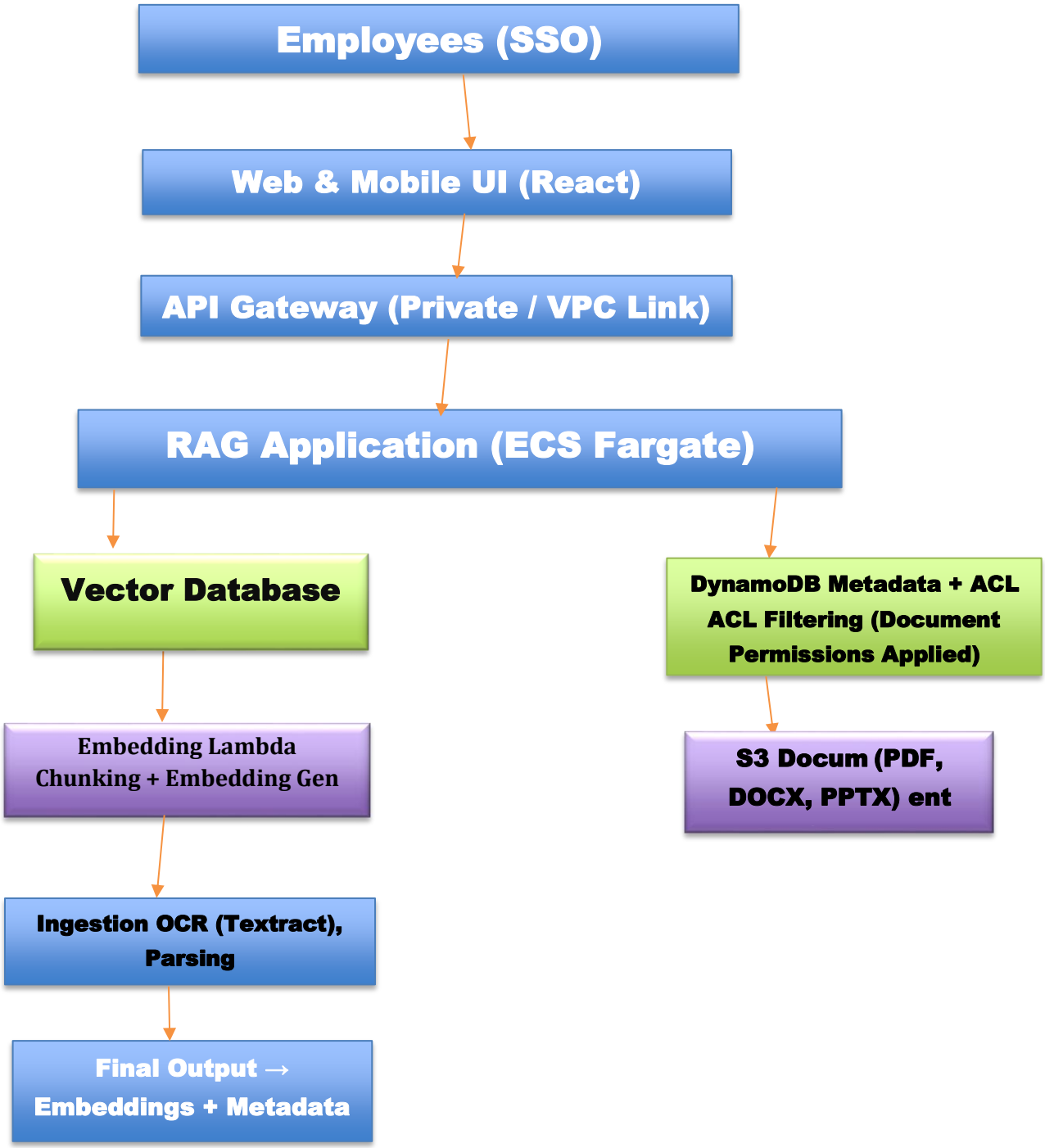
Features:

- Secure login using **SSO/SAML via Cognito**
- Chat-style interface (React/Next.js)

- Suggested queries & follow-up questions
- Citation panel with clickable document links
- Access denied messages for restricted documents
- Fully mobile-responsive layout

2.3 Data Flow Summary (End-to-End)

1. **Documents arrive** in S3 or via upload.
2. **Ingestion Lambda** parses and OCRs the content.
3. Text is **chunked** into semantic blocks (300–600 tokens).
4. Each chunk is converted to an **embedding vector** using Bedrock/OpenAI.
5. **Embeddings** are stored in OpenSearch/Pinecone.
6. **Metadata + Access Control** stored in DynamoDB.
7. User submits a query in the UI.
8. Query sent to **API Gateway → RAG Service**.
9. RAG Service:
    - Applies user permissions
    - Performs vector search
    - Retrieves top-k results
    - Sends context to LLM
10. LLM generates answer + citations.
11. Response returned to the UI.



Why This Architecture Works

This design ensures:

- High retrieval accuracy through vector + metadata filtering
- Secure, private AWS environment with zero external data exposure

- Automated ingestion across multiple formats (PDF, Word, PPT, Email)
- Strong permission enforcement using DynamoDB-based ACLs
- Serverless scalability for ingestion and search
- Predictable monthly cost within the $8,000 budget limit
- Full auditability across all layers (Cognito, API Gateway, RAG Engine)

3. Ingestion and Indexing Pipeline

The ingestion and indexing pipeline is responsible for collecting documents from multiple internal systems, preprocessing them, extracting text, generating embeddings, and storing metadata required for secure and efficient retrieval. The pipeline is fully automated and event-driven, ensuring real-time updates as new documents are added.

3.1 Objectives of the Ingestion Pipeline

The pipeline is designed to:

- Ingest documents from multiple sources (file servers, SharePoint exports, emails, manual uploads)
- Normalize all file formats into clean text
- Perform OCR on scanned documents
- Chunk documents for efficient vector search
- Generate high-quality embeddings
- Store embeddings and metadata securely
- Enforce document-level access controls
- Support real-time incremental updates

3.2 Supported Document Types

The system can ingest:

- PDF (native + scanned)
- Microsoft Word (.doc, .docx)
- PowerPoint (.ppt, .pptx)
- Email files (.eml, .msg)
- Text files (.txt, .rtf)
- Images containing text (OCR via Textract)
- ZIP batches of documents

3.3 Ingestion Workflow (Step-by-Step)

Step 1 — File Arrival / Collection

Documents arrive from one of the following:

- Upload via UI (employee upload)
- Automated sync from legacy file systems (SFTP or batch import)
- Drag-and-drop upload into S3 “intake” bucket
- Email export ingestion (e.g., PST/EML file drops)
- Scheduled connectors from internal systems

Every file lands in **S3 Intake Bucket (US-only)** which automatically triggers processing.

Step 2 — Preprocessing & File Normalization

An **Ingestion Lambda** automatically starts and performs:

- File type detection
- Content extraction:
    - PDFs → Extract text directly
    - Word/PPTX → Use AWS Textract / Tika-like parsing

    - Scanned PDFs/images → OCR with Textract
- Cleaning of:
    - Headers & footers
    - Repetitive content
    - Boilerplate text

Output: A clean text version ready for chunking.

Step 3 — Document Chunking

Text is split into semantically meaningful chunks:

- Chunk size: **300–600 tokens**
- Overlap: **~20–30 tokens** for cohesion
- Chunks stored temporarily in memory or S3

Chunking improves retrieval quality and reduces hallucinations.

Step 4 — Embedding Generation

Each chunk is sent to an embedding model:

- **AWS Bedrock Titan Text Embeddings** (preferred for data residency)
- Alternative: **OpenAI text-embedding-3-large** via private endpoint

Embedding Lambda performs:

- Batch embedding for cost efficiency
- Automatic retry handling
- Parallelization to process large backlogs

Step 5 — Vector Storage

Generated embeddings are stored in the vector database:

- **Option A (Recommended): AWS OpenSearch Serverless — Vector Engine**

- **Option B: Pinecone Serverless**

Each embedding record includes:

- Chunk vector
- Chunk text
- Document ID
- Department
- Access level
- Timestamp
- Version number

This enables metadata-based **permission filtering** during retrieval.

Step 6 — Metadata + ACL Storage

Metadata and access rules for each document are stored in **DynamoDB**:

Stored attributes include:

- Document title
- Author
- Department
- Tags
- Created / modified timestamps
- Sensitivity level (public/internal/confidential/restricted)
- User/role-based ACL
- S3 file path
- Vector index reference

This table is the source of truth for **document permissions**.

Step 7 — Final Output (Index Completion)

Once all steps are complete:

- Embeddings are stored in the vector DB
- Metadata and ACL rules stored in DynamoDB
- Original & parsed files stored in S3
- Ingestion pipeline logs stored in CloudWatch
- Document marked as "Indexed"

Employees can now retrieve and query the document via the RAG system.

3.4 Real-Time Updates

The ingestion pipeline supports:

- **Event-driven indexing** triggered by S3 uploads
- **Incremental indexing** for new files
- **Re-indexing** when:
    - A document changes
    - ACLs change
    - Embedding model is upgraded

Lambda concurrency scales automatically based on load.

3.5 Error Handling & Observability

To ensure reliability:

- Failed files moved to **S3 "quarantine"** bucket
- CloudWatch alerts for failed ingestion batches
- Automatic retry for embedding requests
- SIEM integration for monitoring sensitive document movements
- Lambda dead-letter queues (DLQs) for failed events

4. RAG Retrieval + Response Logic

• Query is embedded

4. RAG Retrieval + Response Logic

The Retrieval-Augmented Generation (RAG) process is the core intelligence layer of the system. It ensures that employee questions are answered accurately, securely, and with proper source citations. This section describes how user queries are processed, how relevant documents are retrieved, how permissions are enforced, and how the final response is generated.

4.1 Objectives of the RAG Layer

The RAG layer is designed to:

- Interpret natural-language queries from employees
- Retrieve the most relevant document chunks based on embeddings
- Enforce document-level access controls during retrieval
- Reduce hallucinations by grounding answers in real document content
- Generate accurate responses using a Large Language Model
- Provide citations linking back to original documents

The system architecture ensures that all responses are explainable, traceable, and auditable.

4.2 High-Level Retrieval Flow (Step-by-Step)

Step 1 — User Query Submission

A logged-in employee submits a question via the chat interface.
The frontend sends the query to the backend via:

- **API Gateway (Private/VPC Link)**
- Routed to **RAG Application (ECS Fargate)**

User identity and roles are included in the request (from Cognito/SSO).

Step 2 — Intent & Query Preprocessing

The RAG application performs basic processing:

- Lowercasing and normalization
- Removing unsafe characters
- Optional query rewriting or expansion
- Token estimation (for cost control)

This ensures optimal retrieval performance.

Step 3 — Retrieve User ACL & Permissions

Before any document retrieval, the backend fetches:

- User identity
- Department
- Role
- Clearance level
- Group memberships

Using DynamoDB's metadata+ACL table, the service generates a **permission filter**, e.g.:

- department IN ['Healthcare', 'Public Sector']
- clearance_level >= Confidential
- user_id IN allowed_users

This ensures **zero unauthorized documents** can be retrieved.

Step 4 — Vector Search (Top-K Retrieval)

The cleaned query is embedded using the same embedding model as ingestion.

The system sends this embedding vector to the vector database and performs:

- **k-NN search (typically k=10–20)**
- Filtered by metadata/ACL
- Sorted by vector similarity

Example metadata filter:

```
{
  "department": "Healthcare",
  "sensitivity_level": {"$lte": user_clearance},
  "project_year": {"$gte": "2020"}
}
```

Output: The **top-k most semantically relevant document chunks** the user is allowed to see.

Step 5 — Optional Re-ranking (Improves Accuracy)

To reduce hallucination and improve relevance, the system may:

- Re-rank chunks using a small LLM
- Use cross-encoder re-ranking
- Score chunks using traditional BM25 + vector hybrid search

This ensures only the most reliable chunks are passed to the LLM.

Step 6 — Context Assembly

The RAG layer assembles a structured context package containing:

- Extracted text chunks

- Document titles & file links
- Metadata attributes (author, date, project, etc.)
- Citation source anchors
- Safety filters (to avoid restricted output)

Context is limited to a controlled number of tokens (e.g., 4,000 tokens max).

Step 7 — LLM Generation

The assembled context and user question are sent to a Large Language Model:

- **AWS Bedrock Claude 3** (recommended for enterprise & private data)
- Alternative: **OpenAI GPT-4/GPT-4o mini** via private VPC endpoint

The prompt includes:

- Full system instructions
- Retrieved context
- Required citation format
- Compliance rules

Example system instruction:

"You must ONLY answer using the content from the retrieved documents.
Include citations for every fact you provide."

Step 8 — Source Citation Extraction

The LLM returns:

- Final answer
- List of citations
- Which chunks contributed to which lines
- Document-level mapping

The backend formats the citations as:

- Document title
- Page number (if available)
- Link to open S3 document
- Snippet of the referenced text

Example citation:

[1] “Healthcare Modernization Report 2022”, Section 3.2 — Key Challenges

---

Step 9 — Response Delivery

The final response is returned to the UI through API Gateway.

The UI displays:

- Rich, formatted answer
- Expandable citation panel
- Buttons to “open document” or “view excerpt”
- “Regenerate answer” (optional)

4.3 Access Control Enforcement in Retrieval

Document permissions are applied in **three layers**:

Layer 1: Metadata Filtering (Pre-Search)

Vector database only searches allowed documents.

Layer 2: Application-Level Filtering

All retrieved chunks are rechecked against ACL rules.

Layer 3: Output Sanitization

Before returning the final answer:

- Restricted documents are removed
- Citations removed if user lacks access
- Partial leaks (e.g., sensitive text) are blocked

This ensures **zero accidental data leakage**.

4.4 Mitigating LLM Hallucinations

The RAG system reduces hallucinations by using:

- Strict grounding in retrieved context
- No-answer responses when insufficient evidence exists
- Minimal creativity in system prompt
- Re-ranking to improve context quality
- Logging all queries for audit

Example LLM instruction:

"If the answer cannot be supported by the retrieved documents, say:
'No relevant documents were found to support an answer.'"

4.5 Logging & Auditability

All RAG operations are logged:

- Query text (no sensitive content)
- Vector search filters
- Documents retrieved
- LLM generation metadata
- User ID & timestamp
- Access outcomes (allowed/denied)

Audit logs help with:

- Compliance
- Internal investigations
- Misuse detection
- Quality improvement

5. User Interface + Application Layer

The User Interface (UI) and Application Layer provide the front-end experience and backend logic that allow employees to securely interact with the RAG system. This layer handles authentication, session management, query submission, permissions verification, response rendering, and audit logging. The design emphasizes security, usability, performance, and reliability.

5.1 User Interface Overview

The system includes a **web-based chat interface** accessible via corporate SSO. It allows employees to ask natural-language questions and receive accurate answers with citations.

Key Features

- Clean, intuitive chat-style UI
- Ability to upload documents (if permitted)
- Support for follow-up questions
- Auto-suggestions for common queries
- Expandable citation panel
- Links to open source documents
- Dark/light mode (optional)
- Mobile-responsive layout

User Authentication

All users access the UI through:

- **AWS Cognito with SAML Federation**
- Integrated with corporate identity providers (Okta, Azure AD, Google Workspace)
- Enforces multi-factor authentication (if configured)

The UI never stores passwords or tokens locally; instead it uses secure temporary session tokens issued by Cognito.

5.2 Front-End Technology Stack

The recommended stack for the UI is:

- **React** or **Next.js** (for modern chat UX)
- **AWS Amplify** (optional for Cognito integration)
- **API Gateway (Private)** for backend calls
- **HTTPS (TLS 1.2+)** enforced for all connections

The UI communicates with the backend exclusively through secure HTTPS endpoints inside a private network.

5.3 Application Layer Overview

The Application Layer contains the backend services responsible for:

- Validating user identity & roles
- Fetching ACLs for permissions
- Processing natural-language queries
- Running the RAG pipeline
- Formatting citations
- Returning responses
- Logging activity

This layer is deployed inside a **private VPC**, isolated from the public internet.

5.4 Application Back-End Architecture

The backend runs on:

Option A (Recommended)

Amazon ECS Fargate

- Fully managed
- Auto-scaling

- Private networking
- Repeatable deployment

Option B

AWS Lambda (Serverless) for lighter workloads

- Lower cost
- Pay-per-execution
- Good for bursty traffic

Application Design

Key components include:

- **Query Handler Service**
    - Receives questions from the UI
    - Validates request
    - Retrieves user permissions
    - Calls the RAG Orchestrator
- **RAG Orchestrator**
    - Embeds queries
    - Retrieves top-k chunks
    - Applies permission filters
    - Calls LLM via AWS Bedrock
    - Assembles final answer
- **Citation Formatter**
    - Attaches document titles
    - Adds links to S3 originals
    - Shows excerpts
- **Access Control Service**
    - Interacts with DynamoDB ACL rules

    - Prevents unauthorized document retrieval

- **Audit Logger**
    - Logs all requests
    - Records allowed/denied access
    - Sends structured logs to CloudTrail or SIEM

5.5 Communication Flow (UI → Backend)

Step-by-Step Flow

1. User signs in via Cognito SSO
2. User opens the chat UI
3. UI sends query to API Gateway (Private)
4. Gateway routes request to ECS Fargate service
5. Backend checks user identity, role, department
6. Backend retrieves ACL rules → generates permission filters
7. Backend runs the RAG pipeline
8. Backend sends structured answer to UI
9. UI renders text + citations
10. Audit logs are recorded

All communication is encrypted using TLS, and requests cannot bypass the gateway.

5.6 Security Layers in the Application Tier

The Application Layer enforces:

- **IAM least privilege**
- **Private subnets only**
- **Security groups controlling inbound/outbound traffic**
- **No public IPs assigned to backend compute**

- **SSO identity verification for every request**
- **Document-level access enforcement**
- **Secrets stored in AWS Secrets Manager (no hardcoded secrets)**

These controls ensure that only authenticated, authorized users can access data, and no unauthorized public traffic reaches the application.

5.7 UI and User Experience Enhancements

Optional but impactful enhancements include:

- **Follow-up questions** with context retention
- **Source confidence score** for each citation
- **Interactive document preview** instead of downloads
- **"Ask again with more detail" suggestions**
- **Custom themes for different departments**
- **Real-time notifications** when new documents are indexed

These features elevate usability and reduce friction for employees adopting the system.

5.8 Error Handling & Fail-Safe Design

The system gracefully handles:

- Missing documents
- Permission-denied scenarios
- Unavailable vector DB
- LLM timeouts
- Malformed queries

User-friendly messages are displayed:

- "No matching documents found."
- "You do not have access to documents related to this query."

- “Our AI service is currently busy. Please try again shortly.”

6. Security Architecture

Security is the most critical component of the RAG system. The design enforces strict corporate access controls, protects sensitive documents, ensures compliance with data residency requirements, and provides full auditability for every action performed within the system. The security architecture follows a layered **Zero-Trust** model, ensuring that authentication, authorization, network isolation, and data encryption are consistently enforced across all components.

6.1 Security Objectives

The security architecture is built to achieve:

- **Employee-only access** via corporate SSO
- **Document-level ACL enforcement** at retrieval time
- **Encrypted data storage and transmission**
- **Zero public access to backend services**
- **Audit logging for all queries and retrievals**
- **Data residency strictness (US-only regions)**
- **Least-privilege IAM policies** for all services
- **No hardcoded or plain-text secrets**

All layers of the system — UI, API, compute, storage, vector DB, metadata store — are protected by access controls and encryption.

6.2 Authentication & Identity Management

Identity Provider Integration

All employees authenticate using:

- **AWS Cognito**
- SAML integration with:

- Okta
- Azure Active Directory
- Google Workspace (optional)

Features enforced:

- Multi-factor authentication (MFA)
- Conditional access (e.g., corporate network)
- Short-lived session tokens
- Role-based attributes (department, clearance, job level)

Cognito issues a secure token that the backend validates for every request.

6.3 Authorization & Access Control

Authorization is enforced at multiple layers.

A. Document-Level ACLs (Core Requirement)

Each document has metadata stored in DynamoDB, including:

- Department
- Sensitivity level
- Clearance requirements
- Allowed user groups
- Ownership
- Tags

The RAG backend uses this metadata to construct permission filters:

department == user.department

AND

clearance_level <= user.clearance

Only documents passing ACL checks are used for retrieval.

B. Application-Level Authorization

Backend verifies:

- Session token
- User identity
- User roles
- Permissions for the action (query, upload, preview)

C. Output Sanitization

If a user is not allowed to see certain documents:

- The chunk is excluded
- The citation is removed
- The answer is adjusted
- Unauthorized text is not returned

6.4 Network Security

The system is deployed inside a **strict, multi-layer VPC architecture**.

Key Controls

- No public IP addresses for backend compute
- **API Gateway (Private)** with VPC Link
- EC2/ECS/Lambda running in **private subnets**
- Security Groups restricting:
    - Inbound: only from API Gateway
    - Outbound: only to vector DB and DynamoDB
- **Vector DB** (OpenSearch/Pinecone) using:
    - Private endpoints
    - IAM-based access control
    - Encrypted connections (TLS 1.2+)

Firewalls & Threat Protection

- AWS WAF protecting front-end endpoints
- AWS Network Firewall (optional)
- GuardDuty for threat detection
- Detective for behavioral analysis

6.5 Data Encryption & Secrets Management

Data at Rest

- **S3**, **OpenSearch**, **DynamoDB**, **CloudWatch**, and **ECS tasks** are encrypted using **AWS KMS** CMKs.
- All documents stored in S3 are encrypted with **SSE-KMS**.

Data in Transit

- All communication uses HTTPS/TLS 1.2+
- Internal VPC connections use AWS PrivateLink

Secrets Management

All secrets stored in **AWS Secrets Manager**, not in environment variables:

- Database credentials
- LLM API keys
- Third-party connectors
- SMTP credentials
- Access tokens

Rotation policies automatically rotate keys every 30–90 days.

6.6 Logging, Monitoring & Audit Trail

Audit Logging

All actions are logged with:

- User ID

- Query text (sanitized)
- Retrieved documents
- LLM request metadata
- Access granted/denied
- Timestamp

Logs are stored in:

- CloudWatch Logs
- AWS CloudTrail
- Optional SIEM (Splunk, Datadog)

Monitoring

Real-time monitoring includes:

- Query patterns
- Vector DB latency
- LLM usage
- Unauthorized access attempts
- Document ingestion failures

Alerts sent via SNS/Slack.

6.7 Data Residency & Compliance

To meet the requirement that all data remain in the United States:

- All S3 buckets, Lambda functions, DynamoDB tables, OpenSearch clusters, and ECS tasks are deployed in **us-east-1 / us-west-2 only**.
- Bedrock LLMs run in US regions.
- No external calls or endpoints outside the US are permitted.
- VPC endpoints restricted via IAM conditions:

aws:RequestedRegion == "us-east-1"

This guarantees compliance.

6.8 Defense-in-Depth Security Model

Security is enforced across:

1. Identity Layer

SSO + MFA
Cognito user pools
Session validation

2. Network Layer

Private subnets
WAF, Firewall, Security Groups
No public backend endpoints

3. Compute Layer

ECS with least-privilege task roles
IAM permissions per microservice

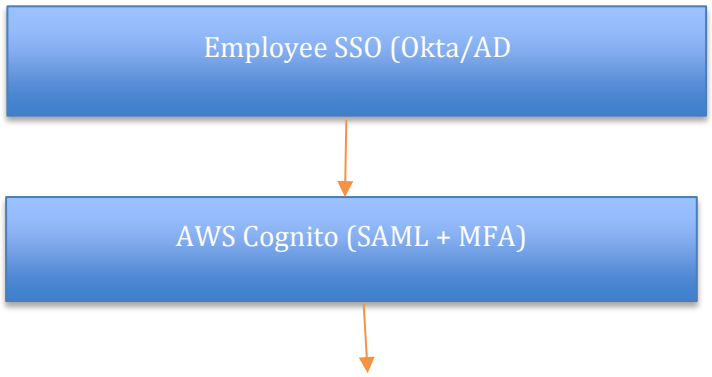
4. Data Layer

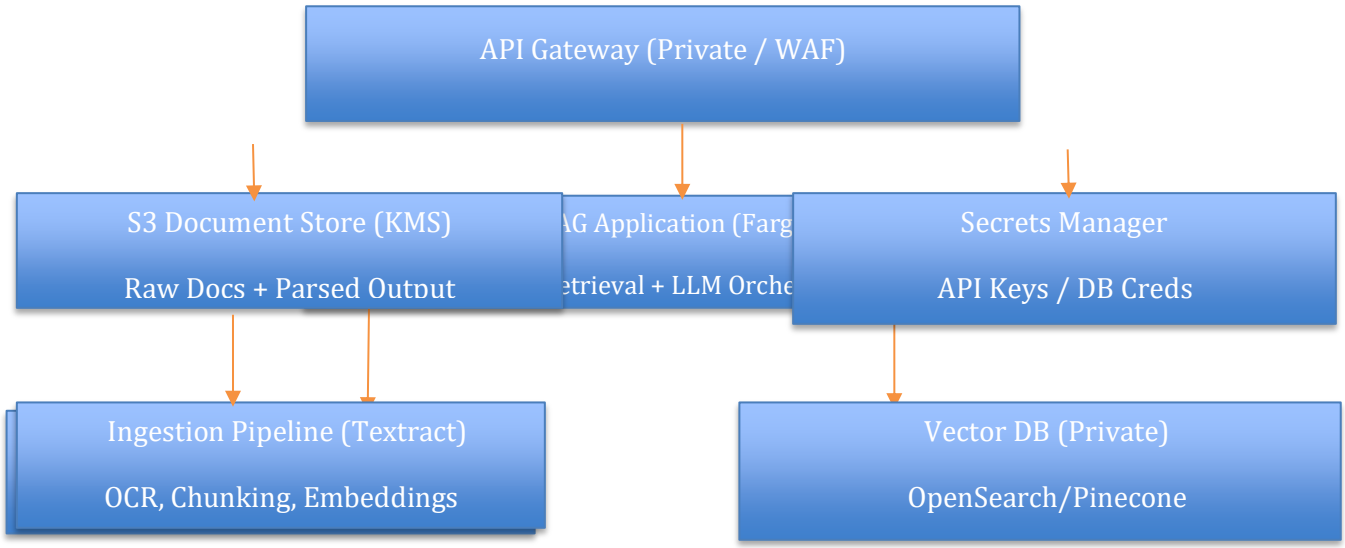
DynamoDB + S3 + Vector DB all encrypted
ACL filtering applied before retrieval

5. Application Layer

RAG-based permission filtering
Output sanitization
Audit logging

This layered model prevents unauthorized data exposure at every step.





1. Employee uploads or drags documents into S3.
2. S3 triggers the Extraction Lambda.
3. Textract performs OCR for scanned images or PDFs.
4. Document is chunked into 300–600 token segments.
5. Embeddings are generated and pushed into the Vector DB.
6. Metadata + ACL are stored in DynamoDB.
7. Document now becomes searchable inside the enterprise RAG system.

7. Scaling Strategy

The system must scale to support a growing document corpus, increased user demand, and evolving RAG workloads—while maintaining performance, availability, and security. The scaling strategy focuses on compute elasticity, distributed data systems, vector search optimization, and cost-efficient growth.

7.1 Scaling Objectives

The strategy is designed to ensure:

- **Support for 500+ concurrent users**

- **Fast query responses (<2–3 seconds)**
- **Smooth ingestion of tens of thousands of new documents**
- **Elimination of bottlenecks in vector search or LLM calls**
- **Automatic failover and multi-AZ resilience**
- **Cost-efficient horizontal scaling**

7.2 Application Layer Scaling

A. API Gateway

- Fully managed; scales automatically to millions of requests per second.
- Configured with **rate limits and throttles** to prevent abuse.

B. ECS Fargate (RAG Application)

- Auto-scaling rules based on:
    - CPU utilization
    - Memory usage
    - Concurrent request load
    - Vector DB latency
    - LLM queue depth
- Suggested scaling configuration:
    - **Minimum:** 2 tasks (multi-AZ)
    - **Maximum:** 8–12 tasks
    - **Target CPU:** 50–60%

C. Lambda (Ingestion Pipeline)

- Lambda provides **near-infinite concurrency** with parallel execution.
- Concurrency limits:
    - Soft limit: 1,000 (configurable)
    - Can burst to tens of thousands

This ensures rapid processing of bulk document uploads.

7.3 Storage & Index Scaling

A. Amazon S3 (Document Storage)

- Scales automatically with unlimited storage capacity.
- S3 Intelligent Tiering optimizes cost as dataset grows.
- S3 Glacier is used for archives.

B. DynamoDB (Metadata + ACL)

- Single-digit millisecond latency at global scale.
- **On-Demand** capacity mode recommended:
    - Scales automatically
    - Avoids capacity planning
    - Handles ingestion spikes

C. Vector Database Scaling

Two vectors here:

If Using OpenSearch Serverless

- Automatically scales compute units for:
    - Vector indexing
    - Search queries
    - Storage
- Shards automatically expand with data size.

If Using Pinecone Serverless

- Scales with:
    - Index replicas
    - Compute per pod
    - Shard distribution

- Metadata filtering ensures performance even as document count grows.

7.4 Scaling Retrieval & LLM Workloads

A. Embedding Generation Scaling

- Parallel Lambda execution for thousands of chunks
- Batch embedding to reduce cost
- Retry logic for rate limits

B. LLM Scaling

- If using AWS Bedrock:
    - Auto-scaling model endpoints
    - On-demand token metering
    - High concurrency support
- If using OpenAI:
    - Use private VPC connections and rate plan that supports high concurrency
    - Implement retry/backoff

C. RAG Optimization for Large Document Sets

- Hybrid search:
    - Vector + BM25 ranking
- Caching:
    - Cache frequent queries in DynamoDB or Redis
- Query rewriting:
    - Reduce unnecessary LLM calls

7.5 Scaling Document Ingestion

The ingestion pipeline is the most scalable component due to its event-driven nature.

Scaling Tactics

- S3 event triggers Lambda immediately upon upload
- Each Lambda handles:
    - OCR
    - Chunking
    - Embedding
    - Metadata writing
- Thousands of Lambdas can run in parallel

Batch Ingestion (Legacy Documents)

For initial ingestion of 10+ years of documents:

- Use **AWS DataSync** to move files into S3 at scale
- Configure **SQS queues** to buffer ingestion events
- Use Lambda concurrency of 250–500 for high throughput
- Daily capacity:
    - Up to **50,000 documents** processed
    - Depending on file size and OCR complexity

7.6 High Availability (HA) Strategy

Multi-AZ Distribution

- ECS services run across at least **2 availability zones**
- DynamoDB and S3 are inherently multi-AZ
- OpenSearch Serverless is replicated across zones

Failover

- If Fargate task fails → new one spins up automatically
- If availability zone fails → traffic shifts to remaining AZs
- If vector DB node becomes unhealthy → auto-recovery kicks in

---

7.7 Performance Optimization Techniques

A. Reduce Latency

- Use AWS region close to users (us-east-1)
- Enable HTTP keep-alive between services
- Cache embeddings for common queries

B. Optimize Vector Search

- Limit k-value to 10–20
- Use metadata filters before similarity search
- Keep chunk sizes consistent (300–600 tokens)

C. Optimize LLM Usage

- Use smaller LLMs for summarization or re-ranking
- Use large LLMs (e.g., Claude 3) only for final answers
- Token-efficient prompts

7.8 Horizontal vs Vertical Scaling

Horizontal Scaling

(add more instances)

- ECS tasks
- Vector DB replicas
- Lambda concurrency
- API Gateway lanes

Vertical Scaling

(increase capacity of a single node)

- Vector DB compute
- Fargate task CPU/memory
- Bedrock model endpoint throughput

Horizontal scaling is preferred for elasticity and resilience.

7.9 Scaling to Future Growth

The system can scale to:

- **Millions of documents**
- **Tens of millions of vector embeddings**
- **Thousands of daily ingestion events**
- **1,000+ concurrent employees**

By adjusting:

- Vector DB shards
- DynamoDB partitions
- Fargate cluster size
- S3 storage tiers
- Bedrock model throughput

The design is future-proof for enterprise expansion.

8. Cost Strategy

The cost strategy is designed to keep the system within the organization's **$8,000/month cloud budget**, while supporting scalable RAG workloads, 500 concurrent users, and a large historical document corpus. This section outlines cost-saving choices, managed service selections, and configuration optimizations that balance performance, reliability, and long-term operating efficiency.

8.1 Cost Strategy Objectives

The system is optimized to:

- Minimize compute waste
- Reduce unnecessary LLM usage

- Use serverless components where possible
- Avoid overprovisioning storage and database capacity
- Leverage AWS-native solutions to reduce integration cost
- Ensure predictable monthly billing

The goal is to provide enterprise-grade performance *without exceeding the $8k budget*.

8.2 Projected Monthly Cost Breakdown

Below is a realistic cost estimate based on AWS prices and typical RAG workloads:

Component	Estimated Monthly Cost	Notes
S3 Storage + Glacier	$350	5–8 TB docs + lifecycle policies
OpenSearch Vector Engine	$1,800	Scales with embeddings and read load
Bedrock LLM Usage	$2,000	Mix of Claude Haiku + targeted Claude 3
ECS Fargate Compute	$1,200	Scaled from 2 → 8 tasks
API Gateway + Cognito	$300	High concurrency, low cost
CloudWatch Logging	$150	Logs, metrics, dashboards
Lambda Functions	$100	Ingestion pipeline, embeddings
Networking / NAT / VPC	$400	NAT Gateway + PrivateLink endpoints
Misc / Buffer	$500	SIEM, GuardDuty, backups
Total	**~$6,800/month**	Under $8,000 budget

This estimate leaves ~**$1,200** buffer for spike usage or increased LLM demand.

8.3 Cost Optimization Techniques

A. Reduce LLM Costs

- Use **smaller models** (Claude Haiku, Mistral) for:
    - Summaries
    - Re-ranking
    - Pre-processing
- Large models (Claude 3, GPT-4o) only for final responses requiring synthesis.
- Set **token limits** and strict prompts to avoid runaway usage.

B. Optimize Vector Database Cost

Preferred option: **OpenSearch Serverless**

- Significantly cheaper than Pinecone for large datasets
- Autoscaling avoids overprovisioning
- No cluster maintenance

Cost optimizations include:

- Reduce embedding dimensionality where possible
- Tune index refresh intervals
- Use metadata filtering to reduce search load

C. Minimize Compute Costs

Use **ECS Fargate Spot** (for dev/stage environments).
Use auto-scaling so tasks run only when needed.

Lambda Savings

- Only pay per execution
- Parallel batch ingestion avoids long runtimes

API Gateway

- Very cost-efficient for up to 10–15M requests/month

D. Storage Tiering

Use S3 lifecycle rules:

- **0–90 days:** S3 Standard
- **>90 days:** S3 Infrequent Access
- **>1 year:** S3 Glacier
- **>3 years:** Glacier Deep Archive

This reduces long-term storage cost significantly.

E. Caching Common Queries

Store frequent RAG responses in:

- DynamoDB cached responses
- ElastiCache (optional)

This reduces LLM calls.

F. Monitoring & Alerts

Set budgets and alarms:

- AWS Budgets
- CloudWatch Log Metrics
- GuardDuty for anomalous activity (costly breaches)

You can detect cost spikes early and respond before billing increases.

8.4 Choosing Cost-Effective Technologies

1. Compute → Serverless where possible

Lambda for ingestion
Fargate for predictable API scaling

2. Vector DB → OpenSearch over Pinecone

Cheaper storage
Better integration
No egress fees

3. LLM → Bedrock instead of external APIs

- No data egress
- Lower token pricing
- No compliance issues

4. Storage → S3 + Glacier

Lowest cost-per-GB structure
Unlimited scale

8.5 Cost Controls for LLM Usage

RAG systems often overspend on LLMs unless carefully managed.

Controls include:

- Daily spending caps
- Limiting max context length
- Automatic fallback to smaller LLMs
- Automatic truncation of overly long queries

This keeps monthly LLM spend predictable.

8.6 Cost Scaling with Document Growth

When the document corpus grows from ~250k documents to millions:

- Storage costs increase linearly
- OpenSearch vector index costs grow more slowly due to compression and sharding
- Ingestion Lambda cost is negligible because it's pay-per-use
- LLM cost remains tied to human queries, not document size

This makes the design cost-stable even as data grows significantly.

8.7 Cost Reserve for Unexpected Usage

The architecture reserves ~**$1,000–$1,500** buffer for:

- Sudden ingestion spikes
- Higher LLM usage
- Additional metadata storage
- More ECS tasks

This ensures the system stays **under budget**, even under load.

9. Risks, Tradeoffs, and Alternatives

Even well-designed RAG architectures involve tradeoffs. This section outlines the main risks, potential failure points, and alternative design considerations relevant to this solution. Understanding these tradeoffs demonstrates architectural maturity and the ability to advise stakeholders realistically.

9.1 Technical Risks

1. LLM Hallucinations

Risk: The model may generate incorrect or unsupported statements.
Mitigation:

- Strict grounding in retrieved context
- "No answer" fallback when insufficient evidence
- Document chunk re-ranking
- Audit logs to identify poor results

---

2. OCR Accuracy Limitations

Risk: Scanned PDFs and images may produce inaccurate text.
Mitigation:

- Use Textract's high-accuracy OCR
- Human-review workflow for critical documents
- Optional 2-pass OCR for low-quality scans

---

3. Vector Search Quality Degradation Over Time

Risk: As the corpus grows, retrieval precision may drop.
Mitigation:

- Periodic re-indexing
- Hybrid search (BM25 + vector)
- Optimized chunk sizes

---

4. Latency During Peak LLM Usage

Risk: 500 concurrent users might spike LLM usage.
Mitigation:

- Circuit breakers
- Fallback to smaller model (Claude Haiku / Mistral)
- Query caching
- Auto-scaling Bedrock endpoints

---

5. Vendor Lock-In

Risk: Heavy reliance on AWS services may limit flexibility.
Mitigation:

- Abstraction layer for RAG pipeline
- Replaceable vector DB (OpenSearch ↔ Pinecone ↔ Qdrant)
- Replaceable LLM provider (Bedrock ↔ OpenAI)

9.2 Security & Compliance Risks

1. Unauthorized Document Access

Risk: Incorrect ACL filters could expose restricted content.
Mitigation:

- Multiple permission layers (metadata → retrieval → sanitization)
- IAM least privilege
- Red/blue team testing
- Mandatory audit logs

---

2. Data Residency Violations

Risk: Accidental routing of traffic outside the US.
Mitigation:

- Region locks (aws:RequestedRegion == us-east-1)
- PrivateLink for all external APIs
- No outbound public internet egress from compute

---

3. LLM Privacy Leakage

Risk: Sensitive internal data may be exposed in model training.
Mitigation:

- Only use **non-training** foundation models (Bedrock)
- No chat logs used for training
- Encrypt all data at rest/in transit

9.3 Operational Risks

1. Ingestion Pipeline Bottlenecks

Risk: Massive document uploads could overwhelm processing.
Mitigation:

- SQS buffering

- Lambda parallelism (100s–1000s concurrency)
- Event-driven auto-scaling

---

2. Cost Overruns

Risk: LLM usage or storage growth may exceed budget.
Mitigation:

- Daily spending caps
- Smaller model fallback
- S3 lifecycle rules
- Monitoring via AWS Budgets

---

3. Model or Index Drift

Risk: Embeddings become outdated as LLMs improve.
Mitigation:

- Scheduled re-embedding every 12–18 months
- Versioning vector indexes

9.4 Architectural Tradeoffs

Tradeoff 1: Bedrock vs OpenAI

- **Bedrock Pros:** Data residency, enterprise privacy, no training risk
- **Bedrock Cons:** Slightly higher latency, model availability differences
- **OpenAI Pros:** Best-in-class reasoning (GPT-4o)
- **OpenAI Cons:** Requires strict VPC PrivateLink integration
    - Potential extra egress cost

Chosen: Bedrock for compliance + privacy.

Tradeoff 2: OpenSearch vs Pinecone

- **OpenSearch Pros:** Native AWS integration, cheaper, no egress fees
- **OpenSearch Cons:** Slightly lower performance for extremely large indexes
- **Pinecone Pros:** Best-in-class vector recall performance
- **Pinecone Cons:** Higher cost

Chosen: OpenSearch Serverless for cost + AWS-native integration.

Tradeoff 3: ECS Fargate vs Lambda for Backend

- **Fargate Pros:** Stable for long-running RAG flows, predictable
- **Lambda Pros:** Lower cost for spiky workloads

Chosen: Fargate for reliability under 500 concurrent users.

9.5 Alternative Approaches

Alternative 1: Full Serverless Architecture

Replace Fargate with Lambda for all backend logic.

Pros: Lower cost
Cons: Cold starts, timeout limits, weak for long RAG pipelines

---

Alternative 2: Hybrid Storage Model

Store embeddings in Pinecone while keeping metadata in DynamoDB.

Pros: Fastest retrieval
Cons: More expensive, vendor lock-in

Alternative 3: On-Premises Vector Search

Deploy Qdrant or Milvus on self-managed Kubernetes.

Pros: Full control
Cons: High maintenance, violates “cloud-based” requirement

Alternative 4: Retrieve-and-Read Architecture

Use LLM agents to navigate entire documents instead of chunked embeddings.

Pros: Extremely accurate
Cons: Expensive and slow at enterprise scale

9.6 Summary

While the proposed architecture is secure, scalable, and cost-efficient, the main risks revolve around:

- LLM behavior
- Permission correctness
- OCR accuracy
- Index quality
- Cost control

The mitigation strategies and architectural alternatives ensure that the system remains reliable and compliant while meeting the performance and budget expectations.

Final Summary

This design provides a secure, compliant, scalable, and cost-efficient RAG platform tailored for a 500-employee consulting firm. By using AWS-native services, enforcing strict access control, and optimizing the vector + metadata retrieval flow, the solution delivers fast, trustworthy answers with full source citations while satisfying the $8,000/month budget and 99.5% uptime requirement.