Enterprise Hybrid-Search RAG on AWS — Implementation Guide & Medium Blog Draft

> A step-by-step, DevOps-ready implementation plan + Medium blog draft describing how to build a production-grade Hybrid-Search Retrieval-Augmented Generation (RAG) system on AWS. This document is structured so you can follow along, implement, and copy the blog content directly.

Table of Contents

---

1 — Executive summary

This guide helps you build a **Hybrid-Search RAG** system on AWS: **OpenSearch (BM25 + vector)**, **SageMaker** for embeddings & rerankers, **EKS** for retrieval services, **Textract/Comprehend** for extraction & PII redaction, and **Aurora/Postgres** for metadata & feedback. The document contains runnable templates and a longform Medium article draft.

2 — Project goals & success metrics

- *Goal:* Production-grade RAG answering enterprise queries with high precision & recall while preserving security.
- *Primary metrics:* Recall@5 $\geq$ 0.8, Precision@5 $\geq$ 0.8, Latency $\leq$ 800ms (retrieval + rerank), Cost per query $\leq$ target.
- *Operational metrics:* ingestion throughput, embedding latency, OpenSearch health, endpoint error rates.

3 — Architecture (AWS-native)

High-level components: - VPC with private subnets - EKS for microservices (retriever, preprocessor, context builder) - OpenSearch Domain (BM25 + vector indexes) - SageMaker endpoints for embedding model & cross-encoder reranker - Textract + Comprehend for extraction & PII detection - Aurora PostgreSQL for metadata & feedback store - API Gateway + Lambda for public entry points and lightweight preprocessing - CodePipeline/CodeBuild (CI), Terraform (IaC), Helm (K8s) for deployment

See diagram in your workspace (user-provided).

---

4 — Prerequisites & accounts

- AWS account with permissions for EKS, SageMaker, OpenSearch, Textract, Comprehend, S3, Aurora, IAM, KMS
- A dev workstation with: Terraform, AWS CLI configured, kubectl, Helm, Docker, Git
- GitHub (or CodeCommit) repo for code and CI
- Budget estimates and approvals for SageMaker endpoints and OpenSearch

---

5 — Phase-by-phase implementation (with code snippets)

Phase 0 — Repo & IaC skeleton

Repo layout (recommended)

```
ragg-aws/
├─ infra/                # Terraform modules
│  ├─ vpc/
│  ├─ eks/
│  ├─ opensearch/
```

```
│   ├─ sagemaker/
│   └─ rds/
├─ charts/               # Helm charts for k8s services
│   ├─ retriever/
│   ├─ reranker/
│   └─ generator-adapter/
├─ services/
│   ├─ retriever/
│   ├─ embed-service/
│   ├─ reranker/
│   └─ ingestion/
├─ ci/                   # GitHub Actions or CodeBuild scripts
└─ docs/
```

Create a Terraform `backend` (S3 + DynamoDB locking) and gitignore for local state.

Phase 1 — VPC, EKS, IAM, KMS (Terraform)

Below is a high-level Terraform module skeleton for VPC and EKS (abbreviated). Fill with your naming, region, and tagging conventions.

`infra/vpc/main.tf` **(skeleton)**

```
provider "aws" { region = var.region }
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"
  name = var.name
  cidr = var.cidr
  azs  = slice(data.aws_availability_zones.available.names, 0, 3)
  public_subnets  = var.public_subnets
  private_subnets = var.private_subnets
  tags = var.tags
}
```

`infra/eks/main.tf` **(skeleton)**

```
module "eks" {
  source = "terraform-aws-modules/eks/aws"
  cluster_name = var.cluster_name
  cluster_version = "1.27"
  subnets = module.vpc.private_subnets
  node_groups = {
```

```
    default = { desired_capacity = 2, instance_type = "m5.large" }
  }
  tags = var.tags
}
```

Notes & best practices - Use IRSA (IAM Roles for Service Accounts) for fine-grained permissions from EKS to call AWS SDK. - Provision KMS keys and store sensitive config in Secrets Manager.

Phase 2 — OpenSearch & Aurora (Terraform)

OpenSearch domain (skeleton)

```
resource "aws_opensearch_domain" "ragg" {
  domain_name = "ragg-search-${var.env}"
  engine_version = "OpenSearch_2.7"
  cluster_config { instance_type = "r5.large.search" }
  ebs_options { ebs_enabled = true; volume_size = 100 }
  advanced_options = { "rest.action.multi.allow_explicit_index" = "true" }
  encrypt_at_rest { enabled = true; kms_key_id = aws_kms_key.opensearch.arn }
  node_to_node_encryption { enabled = true }
}
```

Aurora PostgreSQL (skeleton)

```
module "aurora" {
  source = "terraform-aws-modules/rds-aurora/aws"
  engine = "aurora-postgresql"
  name = "ragg-meta-${var.env}"
  cluster_size = 1
  instance_type = "db.serverless"
}
```

Mapping notes - Configure an OpenSearch index with both `text` fields and `knn_vector` fields for hybrid queries. - Keep chunk_id as a keyword field to cross-reference between OpenSearch and Aurora.

Phase 3 — Ingest pipeline (Glue/MWAA, Textract, Comprehend)

Approach 1. Use Airflow (MWAA) to orchestrate: fetch docs from S3/SharePoint → Textract → clean → Comprehend PII → chunk → send to embed service. 2. Each chunk is saved to S3 as JSON with metadata and enqueued for embedding.

Airflow DAG (simplified)

```
from airflow import DAG
from airflow.providers.amazon.aws.operators.textract import
TextractStartDocumentTextDetectionOperator

with DAG('ragg_ingest', schedule_interval='@daily') as dag:
    start_tex = TextractStartDocumentTextDetectionOperator(...)
    # subsequent tasks: poll textract, call comprehend, chunker lambda, push to
s3
```

Textract notes - For tabular extraction, use Textract AnalyzeDocument API. - Post-process tables into CSV or structured JSON before embedding.

Phase 4 — Embedding service (SageMaker) and indexing

Options - Host a domain-adapted SBERT on SageMaker (recommended) or use vendor-private embeddings. - Use a SageMaker real-time endpoint for embed API: `/embed`.

Embed step (pseudo)

```
# batch chunk -> call sagemaker endpoint
chunks = load_chunks()
vectors = sagemaker_runtime.invoke_endpoint(EndpointName='embed-endpoint',
Body=json.dumps(chunks))
# upsert into OpenSearch: id, metadata, text, vector
```

Index template (OpenSearch)

```
{
  "mappings": {
    "properties": {
      "chunk_id": {"type":"keyword"},
      "text": {"type":"text"},
      "department": {"type":"keyword"},
      "ingested_at": {"type":"date"},
      "vec": {"type": "knn_vector", "dimension": 768}
    }
  }
}
```

Phase 5 — Hybrid Retriever (EKS service) and fusion logic

Service responsibilities - Accept user query + metadata filters - Optionally rewrite query (Lambda small LLM or rules) - Run BM25 against OpenSearch text index (size = 50) - Run vector search (knn) against OpenSearch vector index (top_k = 200) - Normalize scores and fuse (alpha weight default 0.6) - Deduplicate & return candidate pool (e.g., 200) to reranker

Alpha fusion pseudo-code

```
# assume bm25_hits and vec_hits dictionaries keyed by chunk_id
for id in union_ids:
    bm = normalize(bm25.get(id, 0))
    sv = normalize(vec.get(id, 0))
    hybrid_score = alpha * sv + (1-alpha) * bm
```

Deployment - Containerized microservice (FastAPI or Flask) deployed to EKS. Use HPA and metrics server.

Phase 6 — Reranker (SageMaker endpoint)

- Train a cross-encoder (query + passage → relevance score) using in-domain labeled pairs.
- Host as SageMaker endpoint; batch score top-200 candidates and return top-5.

Reranking flow 1. Retriever returns 200 candidates. 2. Reranker batches candidates (size 16–32) to endpoint. 3. Scores are sorted, top-k selected.

Phase 7 — Generator adapter & verifier (LLM)

- Generator: either Bedrock (if available) or a managed LLM on SageMaker (Llama variants / custom). If using closed-source gpt-5-mini, call vendor API from generator adapter with necessary data handling and redaction.
- Verifier: small LLM or heuristic that checks claims against top-k contexts (LLM-as-judge).

Prompt strategy - Provide numbered citations. If claim unsupported, instruct to say "insufficient info".

Phase 8 — CI/CD, Helm, containers

- Build images: embed-service, retriever, reranker, generator-adapter.
- Push images to ECR; use GitHub Actions or CodeBuild to build and push.
- Helm charts for each service with values for env, resources, HPA.

Sample GitHub Action (build & push)

```
name: Build and Push
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: aws-actions/amazon-ecr-login@v1
      - run: docker build -t ${{ env.IMAGE }} .
      - run: docker push ${{ env.IMAGE }}
```

Phase 9 — Monitoring, logging, metrics

- CloudWatch for logs + metrics
- Prometheus/Grafana for custom ML metrics (recall@k, embedding latency)
- OpenSearch slowlogs for query perf
- Set alarms for recall drop, high error rate, endpoint throttling

---

Phase 10 — Testing, evaluation, rollout

- Unit tests for chunking, embedding, merge logic
- Integration tests with local OpenSearch (Docker) + FAISS for dev
- Golden dataset for Recall@k and MRR
- Canary rollout: expose to limited users and compare metrics

---

6 — Operational concerns (security, compliance, cost)

- PII: Comprehend detection + redaction prior to external API calls
- Encryption: KMS for S3, EBS, RDS, OpenSearch
- Access: use IRSA for EKS pods to talk to AWS services; minimize IAM privileges
- Cost: use spot instances for batch tasks; auto-scaling for endpoints; OpenSearch cold storage

---

7 — Retraining & continuous improvement loop

- Keep a feedback DB (Aurora) storing queries, retrieved chunk_ids, user feedback
- Periodic job to sample false negatives/positives and add to training set

- Retrain embedding bi-encoder with hard negatives; retrain reranker
- Deploy updated endpoints with blue/green or canary strategies

---

8 — Medium blog draft (Longform)

> Title: Building a Production-Ready Hybrid-Search RAG System on AWS — Step-by-Step Implementation

Intro Explain the problem (LLM hallucinations, context windows), introduce RAG and hybrid search, and why AWS is suited for enterprise.

Sections 1. Executive summary & what you will build 2. Architecture overview (diagram + explanation) 3. Data ingestion pipeline (Textract + Comprehend + chunking) 4. Building private embeddings (SageMaker) and why domain adaptation matters 5. Hybrid retrieval design: BM25 + vector, fusion logic 6. Reranking & generation: cross-encoder reranker + LLM prompt engineering 7. Putting it in production: EKS, CI/CD, and monitoring 8. Security & compliance considerations 9. Measuring success: metrics and evaluation 10. Next steps: memory, Reflexion loops, GraphRAG

Conclusion & TL;DR Include short checklists and links to code examples.

Call-to-action Encourage readers to try the repo and star the project.

---

9 — Appendix: code snippets & templates

Helm chart skeleton (charts/retriever/Chart.yaml)

```
apiVersion: v2
name: retriever
version: 0.1.0
```

values.yaml (skeleton)

```
replicaCount: 2
image:
  repository: <ECR_REPO>/retriever
  tag: latest
resources:
  limits:
    cpu: 1000m
    memory: 1Gi
```

Airflow DAG sample — see Phase 3 snippet above.

Next steps I have prepared for you

1. Terraform + Helm starter templates (infra skeleton) included in this doc's appendix.
2. Complete Medium blog draft (full article + images/diagram suggestions) included below this section.
3. Ready-to-run local Docker Compose for dev testing (FAISS + ES + simple retriever) I can produce on request.

End of document.