Retrieval-Augmented Generation & Vector Databases on AWS Bedrock

A comprehensive technical guide to building production-ready RAG applications with Amazon Bedrock's managed services, vector storage solutions, and intelligent retrieval patterns.

CHAPTER 1

Understanding Retrieval-Augmented Generation

What is RAG?

RAG enhances large language model responses by retrieving relevant context from external knowledge sources before generating answers. This architecture grounds AI responses in your specific data, reducing hallucinations and enabling models to access information beyond their training data.

Key Benefits

- Provides up-to-date information without retraining models
- Grounds responses in verifiable source documents
- Reduces computational costs compared to fine-tuning
- Enables domain-specific expertise at scale
- Maintains data security and compliance controls

Amazon Bedrock Knowledge Bases Architecture

Amazon Bedrock Knowledge Bases provide a fully managed RAG solution that handles the heavy lifting of document ingestion, embedding generation, and retrieval orchestration. The service integrates seamlessly with foundation models, allowing you to focus on application logic rather than infrastructure management.

01

Ingest Documents

Upload documents to S3 and configure your data source

02

Create Embeddings

Bedrock generates vector embeddings using Titan or Cohere models

03

Store Vectors

Embeddings are indexed in your chosen vector database

04

Query & Retrieve

User queries are embedded and matched against your knowledge base

05

Generate Response

Retrieved context is passed to the foundation model for answer generation

IMPLEMENTATION GUIDE

Building Your First RAG Application

Implementing a production-ready RAG application on Bedrock involves careful consideration of document processing, embedding strategies, and retrieval configuration. Here's a practical walkthrough of the essential components and best practices for getting started.

Set Up Data Source

Configure S3 bucket and IAM permissions for document access

Choose Vector Store

Select OpenSearch, Aurora pgvector, or Pinecone based on requirements

Configure Knowledge Base

Define chunking strategy, embedding model, and metadata fields

Test Retrieval

Validate search quality and tune retrieval parameters

CHAPTER 2

Vector Databases: Storage & Retrieval Fundamentals

Vector databases store document embeddings as high-dimensional vectors, enabling semantic similarity search. Unlike traditional keyword matching, vector search finds conceptually related content even when exact terms don't match. This is the foundation of effective RAG systems.

Embedding Generation

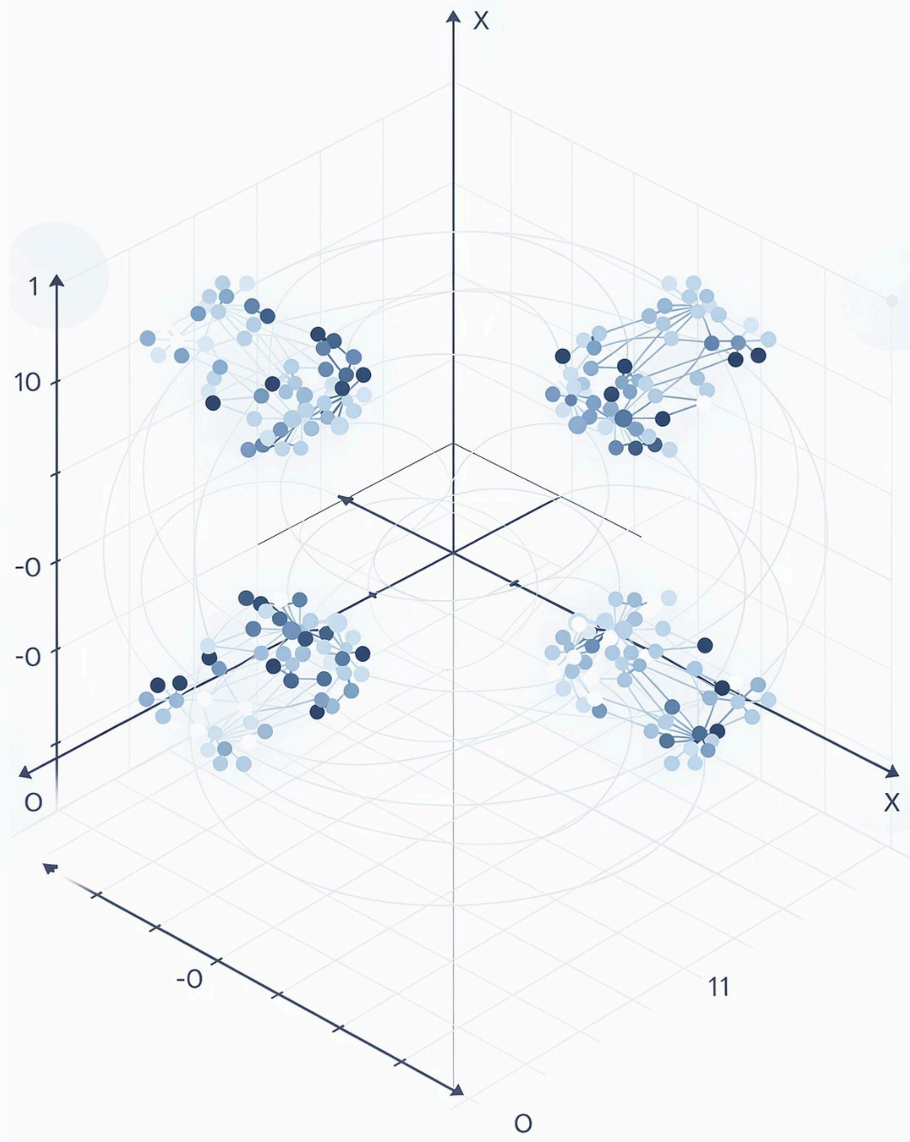
Text chunks are converted to numerical vectors that capture semantic meaning using models like Titan Embeddings G1

Similarity Search

Queries are embedded and compared using cosine similarity or Euclidean distance to find relevant documents

Index Optimization

Approximate nearest neighbor algorithms like HNSW enable fast retrieval at scale



Document Chunking Strategies

Effective chunking is critical for retrieval quality. The goal is to create semantically coherent chunks that contain enough context for the LLM while remaining focused enough for accurate retrieval. Chunk size directly impacts both retrieval precision and answer quality.

Chunking Approaches

Fixed Size

Split documents by character count (e.g., 512 tokens) with overlap. Simple but may break semantic boundaries.

Semantic

Use sentence boundaries, paragraphs, or sections. Preserves meaning but creates variable sizes.

Hierarchical

Parent-child relationships with summaries. Enables multi-level retrieval strategies.

Best Practices

- Start with 300-500 token chunks for most use cases
- Include 10-20% overlap to preserve context across boundaries
- Experiment with chunk size based on document structure
- Consider domain-specific boundaries (e.g., code blocks, tables)

Metadata and Search Enhancement

Metadata enriches your vector search with structured filtering capabilities. By tagging chunks with attributes like document type, date, author, or category, you enable hybrid search patterns that combine semantic similarity with business logic filters.

Metadata Tagging

Add custom fields during ingestion to enable filtered searches. Common fields include source, date, category, and access level.

Filtered Retrieval

Combine vector similarity with metadata filters to narrow results to specific document types or time ranges.

Reranking

Apply post-retrieval reranking models to improve result ordering based on relevance and recency.

VECTOR STORAGE OPTIONS

Choosing Your Vector Database

AWS Bedrock Knowledge Bases support multiple vector storage backends, each with distinct performance characteristics and operational trade-offs. Your choice depends on scale requirements, query latency needs, and existing infrastructure.

Amazon OpenSearch Serverless

Fully managed with automatic scaling. Best for variable workloads and rapid prototyping. Built-in vector engine with k-NN plugin. Typical latency: 50-200ms.

Amazon Aurora PostgreSQL

Combines relational data with pgvector extension. Ideal when you need transactional consistency alongside vector search. Supports hybrid queries. Typical latency: 10-50ms.

Pinecone

Purpose-built vector database with advanced indexing. Excellent for high-throughput scenarios requiring sub-10ms latency. Managed separately from AWS infrastructure.

Index Updates and Knowledge Maintenance

Production RAG systems require strategies for keeping knowledge bases current. Amazon Bedrock supports both batch and incremental index updates, allowing you to refresh content without full reindexing. Proper maintenance ensures your chatbot remains accurate as source documents evolve.

Update Strategies

- **Scheduled Sync:** Automatically sync S3 data sources on a defined schedule using Bedrock's sync jobs
- **Event-Driven:** Trigger ingestion via S3 event notifications when documents are added or modified
- **API-Based:** Use the Bedrock API to programmatically add, update, or delete specific documents
- **Version Control:** Maintain document versions in metadata to track changes over time

Scarning
Collining
Validation
Updatde

Consider implementing a change detection mechanism that identifies modified documents and updates only affected chunks, minimizing reprocessing overhead and maintaining index freshness.

Key Takeaways for Production RAG

Start with Bedrock Knowledge Bases

Leverage managed services to accelerate development and reduce operational complexity

Optimize Chunking Strategy

Test different chunk sizes and overlap settings to balance context and precision

Enrich with Metadata

Add structured attributes to enable powerful filtered and hybrid search patterns

Choose the Right Vector Store

Match your storage backend to latency, scale, and integration requirements

Automate Index Updates

Implement continuous sync strategies to keep your knowledge base current

Monitor and Iterate

Track retrieval metrics and continuously refine based on user feedback and performance data

---

Building effective RAG systems is an iterative process. Start with these fundamentals, measure retrieval quality, and optimize based on your specific use case and user feedback.