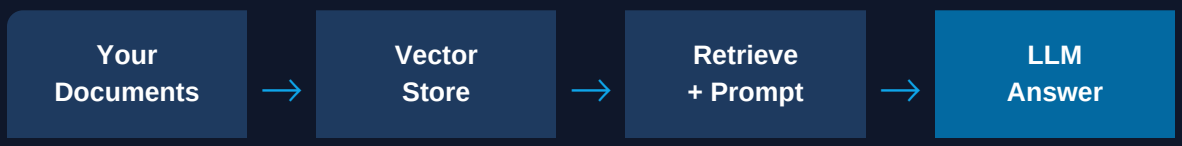
CLOUD AI DEVELOPER GUIDE SERIES

Guide #2

RAG on the Cloud

Retrieval-Augmented Generation with
AWS Bedrock · Azure AI Search · GCP Vertex AI Search



Mid-Level ~30 min read Tier 2 of 3

cloudai.dev · 2025 Edition

Table of Contents

01 What Is RAG?

The Problem RAG Solves

Foundation models are trained on data with a knowledge cutoff — they know nothing about your company's internal documents, your product's latest specs, last week's earnings report, or any private data. Without RAG, you'd have to either fine-tune a model (expensive, slow) or cram all your data into the context window (token-expensive, hits limits fast).

Retrieval-Augmented Generation (RAG) solves this by dynamically fetching only the most relevant chunks of your data at query time and injecting them into the prompt — giving the model the right context without bloating every request with your entire knowledge base.

RAG vs. Fine-Tuning vs. Prompt Stuffing

Approach	How It Works	Best For	Cost
RAG	Retrieve relevant chunks at query time	Dynamic, up-to-date knowledge	Low — pay per query
Fine-Tuning	Retrain model weights on your data	Style/format adaptation, domain tone	High — GPU hours + storage
Prompt Stuffing	Paste all docs into system prompt	Small, static knowledge bases	Medium — large context tokens
Hybrid RAG	RAG + fine-tuned retriever/model	High-accuracy production systems	Medium — balanced approach

The RAG Pipeline Explained

RAG has two distinct phases — **Ingestion** (done once or on a schedule) and **Retrieval** (done at query time).

INGESTION

1	**Ingest**	Load your documents (PDFs, HTML, JSON, databases)
2	**Chunk**	Split into smaller pieces (sentences, paragraphs, fixed tokens)
3	**Embed**	Convert each chunk to a vector using an embedding model
4	**Store**	Save vectors + metadata in a vector database

RETRIEVAL

5	**Query**	User asks a question → embed the question
6	**Retrieve**	Find top-K most similar chunks using cosine/dot-product similarity
7	**Augment**	Inject retrieved chunks into the LLM prompt
8	**Generate**	LLM produces a grounded answer citing your data

02 Core Concepts

Embeddings & Vector Databases

An **embedding** is a list of floating-point numbers (a vector) that captures the semantic meaning of a piece of text. Two pieces of text with similar meaning will have vectors that are close together in high-dimensional space — even if they use completely different words.

A **vector database** is optimised to store these vectors and run ultra-fast **approximate nearest neighbour (ANN)** searches across millions of them. Popular options include pgvector (PostgreSQL), Pinecone, Weaviate, Qdrant, Chroma, and each cloud platform's managed offering.

Embedding Model	Provider	Dimensions	Best For
amazon.titan-embed-text-v2	AWS	1,024	English text, Bedrock-native
text-embedding-3-large	Azure / OpenAI	3,072	High accuracy, multilingual
text-embedding-3-small	Azure / OpenAI	1,536	Cost-efficient, fast
textembedding-gecko@003	GCP Vertex AI	768	GCP-native, low latency
text-multilingual-embedding-002	GCP Vertex AI	768	100+ languages

Chunking Strategies

How you split your documents dramatically impacts retrieval quality. Too large = noisy context, too small = missing context. Here are the main strategies:

Strategy	Description	Typical Size	Use When
Fixed-size	Split by character count, no overlap	500–1000 chars	Quick start, uniform data
Overlapping	Fixed-size with % overlap between chunks	512 tokens, 10% overlap	Prevents context loss at boundaries
Sentence	Split on sentence boundaries	1–5 sentences	Q&A;, conversational data
Semantic	Split when topic changes (embedding-based)	Variable	Long, multi-topic documents
Hierarchical	Parent chunks + child chunks, retrieve child, return parent	Child: 256, Parent: 1024	Production RAG, highest quality

Similarity Search

When a user asks a question, the system embeds it and runs a similarity search to find the K most relevant chunks. There are three common distance metrics:

Metric	Formula	Best For
Cosine Similarity	dot(A,B) / (|A| * |B|)	Text similarity — most common choice
Dot Product	sum(A * B)	When vectors are normalised (same as cosine)
Euclidean (L2)	sqrt(sum((A-B)^2))	Image embeddings, dense numeric data

■ Start with Cosine Similarity

For text-based RAG, cosine similarity is almost always the right default. It measures the angle between vectors (not magnitude), making it robust to document length differences.

03 Platform RAG Services

AWS Bedrock Knowledge Bases

Bedrock Knowledge Bases is a fully managed RAG service — you point it at your data, it handles chunking, embedding, and vector storage automatically. No infrastructure to manage.

How it works: Connect a data source (S3, Confluence, SharePoint, Salesforce, web URLs). Bedrock chunks your documents, embeds them with Titan or Cohere embeddings, and stores them in your chosen vector store. At query time, call RetrieveAndGenerate to get a grounded answer.

- Supports OpenSearch Serverless, Aurora pgvector, Pinecone, Redis Enterprise, MongoDB Atlas as vector stores
- Built-in chunking: fixed-size, hierarchical, or semantic (auto)
- Hybrid retrieval: combines vector search + BM25 keyword search
- Source attribution — every answer includes citations back to S3 documents
- GuardRails integration for content filtering on retrieved context

Data Sources	Amazon S3, Confluence, SharePoint, Salesforce, web crawler, custom API connector
Vector Store	Amazon OpenSearch Serverless (default), Aurora pgvector, Pinecone, Redis, MongoDB Atlas

Azure OpenAI + Azure AI Search

Azure's RAG stack combines Azure OpenAI (for the LLM and embeddings) with Azure AI Search (formerly Cognitive Search) for the vector store and retrieval layer. It's more modular than Bedrock — each piece is a separate service you wire together.

How it works: Upload documents to Azure Blob Storage or use AI Search's built-in indexer to crawl data sources. The indexer runs a pipeline: document cracking, chunking, embedding, and indexing. At query time, use the 'on your data' API flag or the Retrieval SDK to retrieve and generate.

- Integrated vector + full-text + semantic hybrid search in one index
- Semantic ranking layer re-ranks results using a cross-encoder model
- On-your-data feature: one API call does full RAG without custom code
- Supports 100+ file types via document intelligence
- Azure Private Link support for zero-trust, network-isolated RAG

Data Sources	Azure Blob Storage, SharePoint, SQL, Cosmos DB, Salesforce, web crawler, OneLake
Vector Store	Azure AI Search (built-in vector fields) — no separate vector DB needed

GCP Vertex AI Search

Vertex AI Search (formerly Enterprise Search) is Google's managed RAG and search service. It combines Google's search technology with Gemini models and deep BigQuery integration.

How it works: Create a Data Store pointing at your documents (Cloud Storage, BigQuery, websites). Vertex AI automatically processes, chunks, and indexes them. At query time, use the Discovery Engine API to retrieve with summaries, or use the Grounding API to add Google Search grounding to Gemini.

- Follow-up questions supported natively — conversational search out of the box
- Grounding API: attach real-time Google Search results to any Gemini call
- Answer API: multi-turn RAG conversations with citation links
- Native BigQuery connector — query structured + unstructured data together
- Supports 40+ languages with multilingual embeddings

Data Sources	Google Cloud Storage, BigQuery, Cloud SQL, Spanner, AlloyDB, websites, Drive, Gmail (via CMEK)
Vector Store	Vertex AI Vector Search (Matching Engine) — Google's managed ANN service

04 Side-by-Side Comparison

RAG Feature Matrix

Feature	AWS Bedrock KB	Azure AI Search	GCP Vertex Search
Setup complexity	Low — fully managed	Medium — configure indexer	Low — fully managed
Chunking control	Fixed, semantic, hierarchical	Custom via skillset	Auto (limited manual)
Hybrid search	Yes (vector + BM25)	Yes (vector + BM25 + semantic re-rank)	Yes (vector + text)
Conversational RAG	Yes (multi-turn)	Yes (chat extension)	Yes (Answer API native)
Source citation	Yes — S3 file metadata	Yes — field-level	Yes — URL/document links
Custom embedding	Titan, Cohere choices	OpenAI ada/3-large	Gecko, multilingual
BYO vector store	Yes — 5 options	No — AI Search built-in	Yes — Matching Engine
Pricing model	Per query + storage	Per index unit + queries	Per query + doc storage
No-code option	Yes — console wizard	Yes — Azure portal	Yes — Cloud Console
BigQuery native	No	No	Yes

When To Use Each Platform

Choose...	When...
AWS Bedrock KB	Your data lives in S3, you want plug-and-play RAG with Bedrock models, and you need Guardrails or agent integration.
Azure AI Search	You're on Microsoft stack (SharePoint, M365, Azure SQL), need advanced semantic re-ranking, or want zero-trust private networking.
GCP Vertex Search	Your data is in BigQuery or GCS, you want Google Search grounding for real-time facts, or you need multilingual RAG at scale.

05 Build Your First RAG App

Each example below shows an end-to-end RAG call using each platform's Python SDK. For brevity, ingestion (uploading documents) is done via the cloud console — the code focuses on the **query + generate** retrieval path.

AWS Bedrock Knowledge Base — Query & Generate

Prerequisites: Knowledge Base created in Bedrock console · Knowledge Base ID noted · IAM role with bedrock:RetrieveAndGenerate permission

Install:

```
pip install boto3
```

Code:

```
import boto3

client = boto3.client("bedrock-agent-runtime", region_name="us-east-1")

KB_ID = "YOUR_KNOWLEDGE_BASE_ID"
MODEL_ARN =
"arn:aws:bedrock:us-east-1::foundation-model/anthropic.claude-3-haiku-20240307-v1:0"

response = client.retrieve_and_generate(
input={"text": "What is our refund policy for enterprise customers?"},
retrieveAndGenerateConfiguration={
"type": "KNOWLEDGE_BASE",
"knowledgeBaseConfiguration": {
"knowledgeBaseId": KB_ID,
"modelArn": MODEL_ARN,
"retrievalConfiguration": {
"vectorSearchConfiguration": {"numberOfResults": 5}
},
},
},
)

# LLM answer
print(response["output"]["text"])

# Source citations
for citation in response.get("citations", []):
for ref in citation.get("retrievedReferences", []):
print("Source:", ref["location"]["s3Location"]["uri"])
```

Azure OpenAI + AI Search — RAG Query

Prerequisites: Azure OpenAI resource · Azure AI Search index with documents · API keys for both services

Install:

```
pip install openai
```

Code:

```
import os
from openai import AzureOpenAI

client = AzureOpenAI(
api_key=os.environ["AZURE_OPENAI_KEY"],
api_version="2024-05-01-preview",
azure_endpoint=os.environ["AZURE_OPENAI_ENDPOINT"],
)

response = client.chat.completions.create(
model="gpt-4o",
messages=[
{"role": "user", "content": "What is our refund policy for enterprise customers?"}
],
extra_body={
"data_sources": [
{
"type": "azure_search",
"parameters": {
"endpoint": os.environ["AZURE_SEARCH_ENDPOINT"],
"index_name": "your-index-name",
"authentication": {
"type": "api_key",
"key": os.environ["AZURE_SEARCH_KEY"],
},
"query_type": "vector_semantic_hybrid",
"top_n_documents": 5,
},
}
]
},
)

message = response.choices[0].message
print(message.content)

# Source citations
if hasattr(message, "context") and message.context:
for citation in message.context.get("citations", []):
print("Source:", citation.get("url") or citation.get("title"))
```

GCP Vertex AI Search — Answer API

Prerequisites: GCP project · Vertex AI Search data store created · Discovery Engine API enabled · ADC configured

Install:

```
pip install google-cloud-discoveryengine
```

Code:

```
from google.cloud import discoveryengine_v1 as discoveryengine

PROJECT_ID = "your-gcp-project-id"
LOCATION = "global"
DATA_STORE = "your-data-store-id"

client = discoveryengine.ConversationalSearchServiceClient()

serving_config = (
f"projects/{PROJECT_ID}/locations/{LOCATION}"
f"/collections/default_collection"
f"/dataStores/{DATA_STORE}"
f"/servingConfigs/default_config"
)

request = discoveryengine.ConverseConversationRequest(
name=f"{serving_config}/conversations/-",
query=discoveryengine.TextInput(
input="What is our refund policy for enterprise customers?"
),
serving_config=serving_config,
summary_spec=discoveryengine.SearchRequest.ContentSearchSpec.SummarySpec(
summary_result_count=5,
include_citations=True,
),
)

response = client.converse_conversation(request=request)

print(response.reply.summary.summary_text)

for ref in response.reply.summary.summary_with_metadata.references:
print("Source:", ref.document.name)
```

■■ Keep API keys out of code

Use environment variables, AWS Secrets Manager, Azure Key Vault, or GCP Secret Manager. For AWS, prefer IAM roles. For GCP, use Application Default Credentials. For Azure, use Managed Identity. Never commit credentials to version control.

06

Production Checklist

Chunking Best Practices

Overlap your chunks	Use 10–20% overlap between consecutive chunks to avoid cutting context at boundaries. A sentence that starts at the end of chunk N often completes in chunk N+1.
Include metadata in chunks	Prepend the document title, section heading, or date to each chunk before embedding. This dramatically improves retrieval relevance.
Keep semantic units together	Avoid splitting bullet lists, code blocks, or table rows. Split at paragraph or section boundaries when possible.
Test chunk sizes empirically	Run your top 20 real user queries and check if retrieved chunks actually contain the answer. Tune chunk size and K (number of results) based on hit rate.

Evaluation & Quality Metrics

Metric	What It Measures	Target
Context Recall	Were the relevant chunks actually retrieved?	> 0.85
Context Precision	How many retrieved chunks were actually relevant?	> 0.80
Answer Faithfulness	Does the LLM answer stay grounded in retrieved context?	> 0.90
Answer Relevancy	Does the answer actually address the question?	> 0.85
Latency (P95)	End-to-end time from question to answer	< 3 seconds

■ Use RAGAS for evaluation

RAGAS (pip install ragas) is the leading open-source framework for evaluating RAG pipelines. It measures context recall, precision, answer faithfulness, and relevancy automatically using an LLM judge. Run it on a golden dataset of 50–100 question/answer pairs.

Cost Optimisation Tips

- Use a smaller/cheaper embedding model (e.g. text-embedding-3-small vs large) — quality difference is minimal for most use cases.

- Cache embedding results for frequently queried chunks. Re-embedding is expensive if your document set rarely changes.
- Lower K (number of retrieved chunks) if your answers are accurate — fewer chunks = shorter prompt = lower LLM cost.
- Use a fast, cheap model (Claude Haiku, GPT-4o mini, Gemini Flash) for the generation step in RAG — it's reading context, not reasoning from scratch.
- Implement a query router: classify the question first, and only run RAG for questions that need it.
- Set a max context length and truncate retrieved content — many chunks are partially relevant; only the top portion is needed.

■ Up Next: Guide #3 — Multi-Agent Orchestration

Learn how to build multi-agent systems using AWS Bedrock Agents, Azure AI Agent Service, and Vertex AI Agent Builder. Combine RAG, tool use, and agent handoffs into production pipelines.