🔍 RAG & Advanced RAG

Stage 4: Retrieval-Augmented Generation with Vector Search

Ground LLM responses in your actual documentation. Eliminate hallucinations, provide sources, and keep knowledge up-to-date without retraining. The gold standard for production Q&A systems.

🎯 What is RAG?

TL;DR
RAG (Retrieval-Augmented Generation) combines two steps: (1) Search your knowledge base to find relevant documents, (2) Inject those documents into the LLM prompt so it generates responses based on YOUR data, not just its training. Result: Accurate, sourced, up-to-date responses without hallucinations.

The Problem RAG Solves

❌ Without RAG (Stage 3)

- **Hallucinations:** LLM invents plausible-sounding answers
- **No sources:** Can't verify where information came from
- **Outdated knowledge:** Model training data is frozen in time
- **Limited context:** Can't access your company docs
- **Compliance risk:** Banking requires documented sources

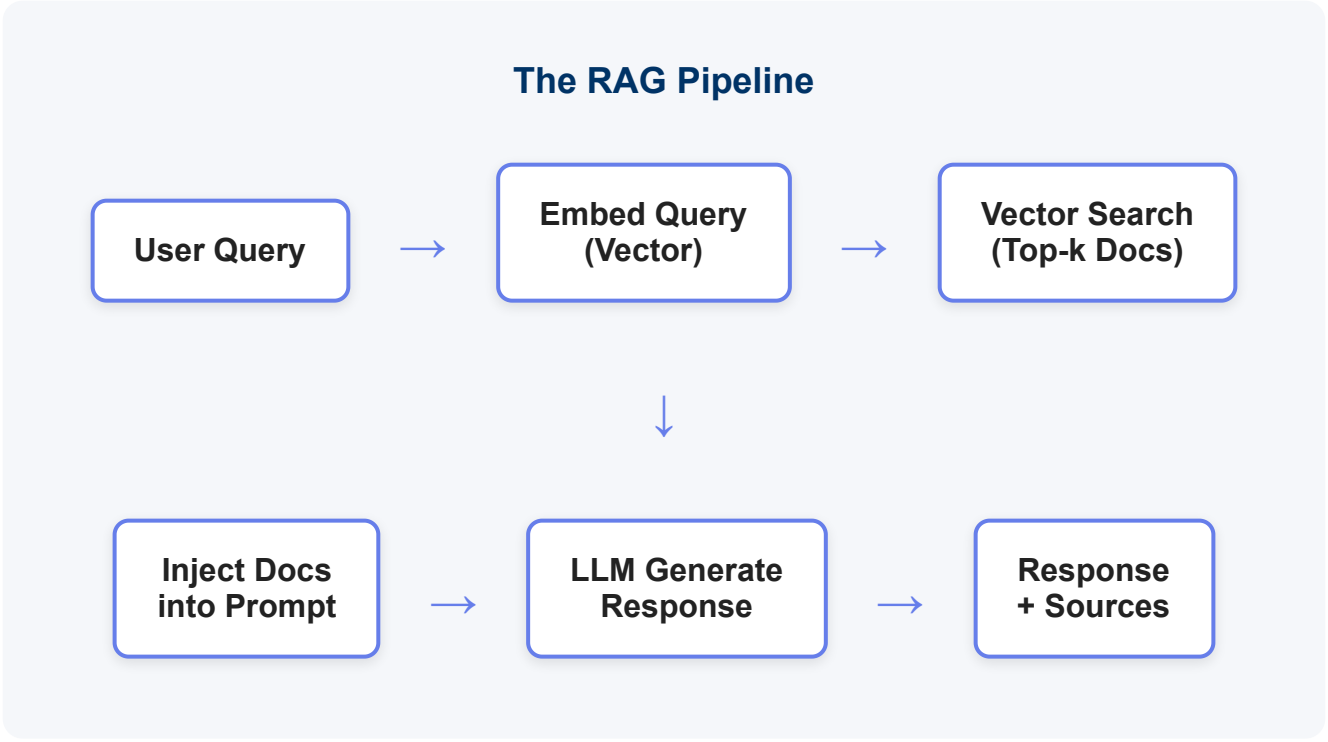
Example: "How do I fix schema errors?"
LLM: "Run this command..." ← Might be wrong!

✅ With RAG (Stage 4)

- **Grounded:** Responses based on retrieved documents
- **Cited sources:** Every claim includes source reference
- **Always current:** Search reflects latest documentation
- **Company knowledge:** Accesses your runbooks, SOPs
- **Auditable:** Full trail from query → docs → response

Example: "How do I fix schema errors?"
RAG: "Per SOP-451, run schema_sync..." ← Verified!

How RAG Works



🏗️ Basic RAG Implementation

Setup & Installation

```
# Install required packages pip install chromadb sentence-transformers
google-generativeai --break-system-packages # For production, also
consider: # - Pinecone (managed vector DB) # - Weaviate (open-source
vector DB) # - pgvector (PostgreSQL extension)
```

Step 1: Index Your Knowledge Base

```
import chromadb from chromadb.utils import embedding_functions from typing
import List, Dict # Initialize ChromaDB (local vector database) client =
chromadb.PersistentClient(path="./chroma_db") # Use Gemini's embedding
model gemini_ef = embedding_functions.GoogleGenerativeAiEmbeddingFunction(
api_key="YOUR_API_KEY", model_name="models/embedding-001" ) # Create or
get collection collection = client.get_or_create_collection(
name="support_docs", embedding_function=gemini_ef, metadata=
{"description": "IT support runbooks and SOPs"} ) # Sample documents (your
runbooks, SOPs, wikis) documents = [ { "id": "sop-451", "text": """Schema
Validation Error Resolution - SOP-451 When a pipeline fails with schema
validation error: 1. Identify the affected table from error message 2.
Navigate to Data Platform Dashboard > Schema Management 3. Click 'Run
Schema Sync Job' for the affected table 4. Wait for sync to complete
(typically 2-5 minutes) 5. Restart the failed pipeline job 6. Verify job
completes successfully IMPORTANT: Never manually alter schema without
approval. Last Updated: 2025-01-15""", "metadata": { "category":
"pipeline", "sop_number": "451", "last_updated": "2025-01-15" } }, { "id":
"sop-234", "text": """Pipeline Timeout Configuration - SOP-234 To adjust
pipeline timeout settings: 1. Access pipeline configuration in Airflow UI
2. Locate 'execution_timeout' parameter 3. Current default is 1800 seconds
(30 minutes) 4. Recommended: Set to 2x average runtime 5. Maximum allowed:
4 hours (14400 seconds) 6. Apply changes and test with small batch first
Timeout values by pipeline type: - ETL pipelines: 1-2 hours - Data quality
checks: 30 minutes - Report generation: 15 minutes Last Updated: 2025-01-
10""", "metadata": { "category": "pipeline", "sop_number": "234",
"last_updated": "2025-01-10" } }, { "id": "sop-789", "text": """Database
Access Request Process - SOP-789 Standard access request workflow: 1. User
submits ticket with justification 2. Verify manager approval is attached
3. Determine appropriate role: - viewer: Read-only access (most common) -
editor: Read + write to specific schemas - admin: Full access (requires VP
approval) 4. Grant access via IAM console 5. Access expires after 90 days
(automatic) 6. Send confirmation email to user Emergency access (audit,
compliance): - Requires approval from compliance officer - Maximum
```

```
duration: 24 hours - Must enable additional audit logging - Notify
security team immediately Last Updated: 2025-01-12""", "metadata": {
"category": "access", "sop_number": "789", "last_updated": "2025-01-12" }
} ] # Add documents to collection (embeddings created automatically)
collection.add( ids=[doc["id"] for doc in documents], documents=
[doc["text"] for doc in documents], metadatas=[doc["metadata"] for doc in
documents] ) print(f"✓ Indexed {len(documents)} documents")
```

Step 2: Retrieve Relevant Documents

```
def retrieve_context(query: str, top_k: int = 3) -> List[Dict]: """ Search
vector database for relevant documents. Args: query: User's
question/ticket top_k: Number of documents to retrieve Returns: List of
relevant documents with metadata """ # Query the collection (embedding
done automatically) results = collection.query( query_texts=[query],
n_results=top_k, include=["documents", "metadatas", "distances"] ) #
Format results retrieved_docs = [] for i in range(len(results["ids"][0])):
retrieved_docs.append({ "id": results["ids"][0][i], "text":
results["documents"][0][i], "metadata": results["metadatas"][0][i],
"similarity": 1 - results["distances"][0][i] # Convert distance to
similarity }) return retrieved_docs # Example retrieval query = "How do I
fix a schema mismatch error in pipeline?" docs = retrieve_context(query,
top_k=2) for doc in docs: print(f"Found: {doc['id']} (similarity:
{doc['similarity']:.2f})")
```

Step 3: Generate Response with Context

```
import google.generativeai as genai
genai.configure(api_key="YOUR_API_KEY") model =
genai.GenerativeModel('gemini-1.5-pro') def answer_with_rag(query: str) ->
Dict: """ Answer query using RAG pipeline. Args: query: User's question
Returns: Dict with answer, sources, and confidence """ # Step 1: Retrieve
relevant documents retrieved_docs = retrieve_context(query, top_k=3) #
Step 2: Format context from retrieved docs context = "\n\n---\n\n".join([
f"Document: {doc['id']}\n{doc['text']}" for doc in retrieved_docs ]) #
Step 3: Create grounded prompt prompt = f"""You are an IT support
assistant. Answer the question using ONLY the provided documentation.
CRITICAL RULES: 1. Base your answer ONLY on the provided documents 2. Cite
the SOP number for each claim 3. If the documents don't contain the
answer, say "I don't have information about this in the available
documentation" 4. Never invent or assume information DOCUMENTATION:
```

```
{context} QUESTION: {query} ANSWER (with SOP citations):""" # Step 4:
Generate response response = model.generate_content(prompt) # Step 5:
Return answer with sources return { "answer": response.text, "sources": [
{ "id": doc["id"], "sop": doc["metadata"].get("sop_number"), "similarity":
doc["similarity"] } for doc in retrieved_docs ], "retrieved_docs":
retrieved_docs } # Example usage query = "How do I handle a schema
validation error?" result = answer_with_rag(query) print("Answer:",
result["answer"]) print("\nSources:") for source in result["sources"]:
print(f" - SOP-{source['sop']} (relevance: {source['similarity']:.1%})")
```

⚡ Advanced RAG Techniques

1. Chunking Strategies

The Problem: Documents are often too long to embed effectively. Need to break them into chunks.

Fixed-Size Chunking

Method: Split by character count (e.g., 500 chars)

Pros: Simple, consistent size

Cons: Can split mid-sentence

```
def fixed_chunk(text, size=500): return [text[i:i+size] for i in range(0,
len(text), size)]
```

Semantic Chunking

Method: Split by paragraphs/sections

Pros: Preserves meaning

Cons: Variable sizes

```
def semantic_chunk(text): # Split by double newlines return [p.strip() for p in text.split('\n\n') if p.strip()]
```

Sliding Window

Method: Overlapping chunks

Pros: No context lost at boundaries

Cons: More storage needed

```
def sliding_window(text, size=500, overlap=100): chunks = [] for i in range(0, len(text), size-overlap): chunks.append(text[i:i+size]) return chunks
```

Recommendation for Production: Use semantic chunking for SOPs (split by procedure steps) with 100-character overlap to maintain context.

2. Hybrid Search (Keyword + Vector)

Concept: Combine traditional keyword search (BM25) with vector similarity for better retrieval.

```
from rank_bm25 import BM25Okapi from typing import List, Tuple def hybrid_search(query: str, documents: List[str], embeddings, top_k: int = 5) -> List[Tuple[int, float]]: """ Combine BM25 keyword search with vector similarity. Returns: List of (doc_index, combined_score) tuples """ # 1. BM25 keyword search tokenized_docs = [doc.lower().split() for doc in documents] bm25 = BM25Okapi(tokenized_docs) bm25_scores = bm25.get_scores(query.lower().split()) # 2. Vector similarity search query_embedding = embed_query(query) vector_scores = [ cosine_similarity(query_embedding, doc_emb) for doc_emb in embeddings ] # 3. Combine scores (weighted average) alpha = 0.5 # Weight: 0=only BM25, 1=only vector combined_scores = [ alpha * vector_scores[i] + (1-alpha) * bm25_scores[i] for i in range(len(documents)) ] # 4. Get top-k results
```

```
ranked = sorted( enumerate(combined_scores), key=lambda x: x[1], reverse=True )[:top_k] return ranked
```

When to Use: Hybrid search catches both exact keyword matches (like SOP numbers) and semantic similarity. Improves retrieval quality by 15-25%.

3. Query Rewriting

Concept: Improve user queries before searching to get better results.

```
def rewrite_query(original_query: str) -> List[str]: """ Generate multiple versions of query for better retrieval. """ prompt = f"""Given this support ticket query, generate 3 alternative phrasings that would help find relevant documentation: Original: "{original_query}" Generate variations that: 1. Use technical terminology 2. Rephrase as a question 3. Add context/keywords Return as JSON array of strings.""" response = model.generate_content(prompt) variations = json.loads(response.text) return [original_query] + variations # Example original = "pipeline broke" variations = rewrite_query(original) # Results: # 1. "pipeline broke" # 2. "pipeline execution failure" # 3. "What causes pipeline jobs to fail?" # 4. "pipeline error troubleshooting data platform" # Search with all variations and combine results all_results = [] for var in variations: all_results.extend(retrieve_context(var, top_k=2)) # Deduplicate and re-rank unique_results = deduplicate(all_results)
```

4. Re-ranking Retrieved Documents

Concept: Initial retrieval is fast but imprecise. Use a more powerful model to re-rank top results.

```
def rerank_documents(query: str, documents: List[Dict], top_k: int = 3) -> List[Dict]: """ Re-rank retrieved documents using cross-encoder for better relevance. """ # Use Gemini to score relevance scored_docs = [] for doc in documents: prompt = f"""Rate how relevant this document is to the question on a scale of 0-10. Respond with ONLY a number. Question: {query} Document: {doc['text'][:500]} Relevance score (0-10):""" response = model.generate_content(prompt) try: score = float(response.text.strip())
```

```
except: score = 5.0 # Default if parsing fails doc['rerank_score'] = score
scored_docs.append(doc) # Sort by score and return top-k
scored_docs.sort(key=lambda x: x['rerank_score'], reverse=True) return
scored_docs[:top_k]
```

5. Metadata Filtering

```
# Search with metadata constraints results = collection.query(
query_texts=["schema error solution"], n_results=5, where={ "category":
"pipeline", # Only pipeline docs "last_updated": {"$gte": "2024-01-01"} #
Recent docs only } ) # This is critical for banking - only use approved,
current SOPs!
```

📊 Performance Optimization

Metrics to Track

Metric	Target	How to Measure
Retrieval Precision	>80%	% of retrieved docs that are relevant
Retrieval Recall	>90%	% of relevant docs that were retrieved
End-to-End Latency	<5 seconds	Query → Response time
Answer Quality	>4/5	Human ratings on sample queries
Source Accuracy	100%	Citations match actual source content
Hallucination Rate	<5%	Responses not supported by retrieved docs

Cost Optimization

> **RAG Cost Breakdown (500 tickets/month):**
>
> • **Embedding costs:** $0.50/month
> (500 queries × $0.001/1K tokens)

- **Vector DB:** $10-50/month
  (ChromaDB free, managed services $10-50)

- **LLM generation:** $2,500/month
  (500 queries × 3 docs × 500 tokens × $0.003/1K)

Total: ~$3,000/month

Optimization Strategies:

- Use Gemini Flash instead of Pro (-60% cost)
- Cache frequently used docs
- Limit top-k to 2-3 docs (not 5+)
- Batch similar queries

🏭 Production Considerations

Production RAG System

1 Document Ingestion Pipeline

- Watch for updates to SOPs (file system monitoring)
- Chunk documents using semantic splitting
- Generate embeddings (batch process)
- Update vector DB (upsert changed docs only)
- Track document versions and timestamps

2 Query Processing

- Sanitize and validate user query
- Optionally rewrite query for better retrieval
- Apply metadata filters (category, date, approval status)
- Retrieve top-k candidates (k=10-20)
- Re-rank to top-3 most relevant

3 Response Generation

- Construct grounded prompt with retrieved context
- Add safety constraints (only use provided docs)

- Generate response with LLM
- Extract and format source citations
- Validate response quality (hallucination check)

4 Monitoring & Logging

- Log query, retrieved docs, response
- Track latency at each stage
- Collect user feedback (thumbs up/down)
- Alert on high latency or error rates
- Audit trail for compliance

Banking-Specific Requirements

Compliance Checklist:

- ☐ All source documents approved and version-controlled
- ☐ Audit log of every query and response
- ☐ Data retention policy implemented (90 days)
- ☐ PII detection and redaction in place
- ☐ Access controls on vector database
- ☐ Regular accuracy audits by compliance team
- ☐ Disaster recovery and backup procedures

🆚 RAG vs Fine-Tuning

Aspect	RAG	Fine-Tuning
Best For	Knowledge that changes frequently	Consistent behavior/tone/format
Setup Time	3-4 weeks	6-8 weeks
Update Speed	Instant (just update docs)	Slow (requires retraining)
Explainability	High (shows sources)	Low (baked into model)

Cost (500 tix/mo)	$3K-6K	$1K-2K (lower per-query)
Hallucination Risk	Low (~5%)	Very Low (~3%)

Best Practice: Use RAG + Fine-Tuning together!
• Fine-tune for consistent tone and formatting
• RAG for up-to-date factual information
• Result: Professional responses with fresh, grounded content

---

🔗 Next Steps

Continue Learning

- Stage 5: Fine-Tuning
- Stage 6: Agentic AI
- Back to Master Guide

Try It Yourself

- Install ChromaDB
- Index your team's docs
- Test retrieval quality
- Build Q&A prototype

Resources

- ChromaDB Docs
- Gemini API
- Code Examples
