

RAG (Retrieval-Augmented Generation)

Build LLM applications that answer questions from your own data — not just training data.

Last reviewed 28 Sep 2026 ·

Prerequisites: [Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) · [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/)

Related: [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) · [LangChain & LlamaIndex](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

Challenge: LLMs are trained on public data up to a cutoff date. They have no knowledge of internal systems, data dictionaries, runbooks, or events after training.

Solution: Retrieval-augmented generation retrieves relevant documents from an organization's own knowledge base and includes them in the prompt, so the model answers from that content rather than from its training data alone.

Without RAG:

User: "What columns does our orders table have?"

LLM: "I don't have information about your specific database schema."

(or worse: makes something up)

With RAG:

User: "What columns does our orders table have?"

Retriever: [finds the data dictionary entry for the orders table]

LLM: "Based on your data dictionary: order_id (VARCHAR),

```
customer_id (INT), amount (DECIMAL), status (VARCHAR)..."
```

The LLM is now a search interface over your own documents.

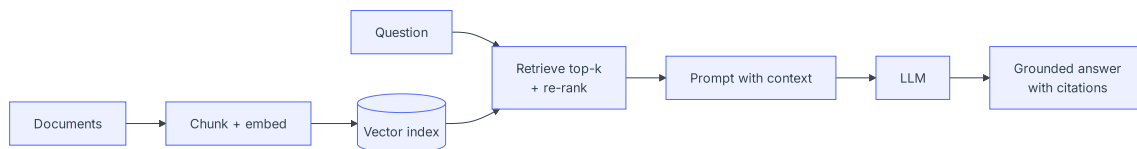
The two phases of RAG:

1. Indexing (offline, run once):

Your docs → Split into chunks → Convert to vectors → Store in vector DB

2. Querying (online, per question):

Question → Convert to vector → Find similar chunks → Build prompt → LLM → Answer



On this page

Basic - [What Is RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#what-is-rag) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#what-is-rag) - [The RAG Pipeline](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#the-rag-pipeline) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#the-rag-pipeline) - [Minimal Working Example](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#minimal-working-example) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#minimal-working-example)

Intermediate - [Chunking Strategy](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#chunking-strategy) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#chunking-strategy) - [Metadata Filtering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#metadata-filtering) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#metadata-filtering) - [Retrieval Tuning](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#retrieval-tuning) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#retrieval-tuning) - [The Generation Step](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#the-generation-step) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#the-generation-step)

Advanced - [Hybrid Search](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#hybrid-search) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#hybrid-search) - [Re-ranking](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#re-ranking) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#re-ranking) - [Advanced RAG Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#advanced-rag-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#advanced-rag-patterns) - [Evaluating RAG Quality](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#evaluating-rag-quality) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#evaluating-rag-quality)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/#interview-questions)

What Is RAG

RAG = Retrieval-Augmented Generation. Instead of asking an LLM to answer from its training data, you: 1. Find relevant documents from your own knowledge base 2. Include those documents in the prompt 3. Ask the LLM to answer based only on those documents

Without RAG:

User: "What is our orders table schema?"

LLM: [hallucinates or says "I don't know"]

With RAG:

User: "What is our orders table schema?"

Retriever: [finds the data dictionary entry for orders]

LLM: "Based on your data dictionary: the orders table has columns order_id (VARCHAR), customer_id (INT), amount (DECIMAL), status (VARCHAR), created_at (TIMESTAMP)"

When to use RAG vs fine-tuning:

	RAG	Fine-tuning
Data changes frequently	Yes	No — requires retraining
Need source citations	Yes	No
Factual accuracy required	Yes	Sometimes
Change model behavior/style	No	Yes
Cost	Low (per query)	High (one-time training)
Start with	Always	After RAG isn't enough

The RAG Pipeline

```
INDEXING (offline, run once or incrementally)
```

```
Documents → Chunk → Embed → Store in vector DB
```

```
QUERYING (online, per user question)
```

```
Question → Embed → Retrieve top-K → Build prompt  
→ LLM → Answer
```

Components: 1. **Document loader** — read PDFs, markdown, HTML, databases 2. **Chunker** — split documents into 256–512 token chunks 3. **Embedder** — turn chunks into vectors 4. **Vector store** — index vectors for fast similarity search 5. **Retriever** — given a query, find the top-K most relevant chunks 6. **Generator** — LLM that answers using retrieved context

Minimal Working Example

```
import numpy as np
from openai import OpenAI
import anthropic
```

```
openai_client = OpenAI()
claude_client = anthropic.Anthropic()
```

```
# — 1. Documents
```

```
documents = [
    {"id": "orders_schema", "text": "The orders table
has: order_id (VARCHAR PK), customer_id (INT FK), amount
(DECIMAL 10,2), status (VARCHAR:
placed/shipped/delivered/cancelled), created_at
(TIMESTAMP)."},
```

```

    {"id": "customers_schema", "text": "The customers table
has: id (INT PK), name (VARCHAR), email (VARCHAR UNIQUE),
region (VARCHAR), created_at (TIMESTAMP)."},
    {"id": "pipeline_sla",      "text": "The orders pipeline
runs at 2am UTC daily. SLA: data must be available by 6am
UTC. Alert: data-oncall@example.com."},
    {"id": "data_freshness",    "text": "All tables in the
gold layer are updated daily. The silver layer updates
every 6 hours. The bronze layer is near-real-time via
Kafka."},
    {"id": "access_policy",     "text": "Gold layer tables
require the analyst role. Bronze and silver require the
engineer role. PII columns are masked for non-PII roles."},
]

```

— 2. Embed documents

```

def embed(texts: list[str]) -> np.ndarray:
    response = openai_client.embeddings.create(
        input=texts,
        model="text-embedding-3-small"
    )
    vecs = np.array([r.embedding for r in response.data])
    return vecs / np.linalg.norm(vecs, axis=1,
keepdims=True) # normalize

doc_texts = [d["text"] for d in documents]
doc_vecs = embed(doc_texts)

```

— 3. Retrieve

```

def retrieve(query: str, k: int = 3) -> list[dict]:
    q_vec = embed([query])[0]
    scores = doc_vecs @ q_vec
    top_k = np.argsort(scores)[::-1][:k]
    return [{"text": documents[i]["text"], "score":
float(scores[i])} for i in top_k]

```

— 4. Generate

```
def answer(question: str) -> str:
    context_docs = retrieve(question)
    context = "\n\n".join(f"[{i+1}] {d['text']}" for i, d
in enumerate(context_docs))

    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=512,
        system="""Answer questions using ONLY the provided
context.
If the answer isn't in the context, say "I don't have that
information."
Be concise. Cite the context number like [1] when you use
it.""",
        messages=[{
            "role": "user",
            "content": f"Context:\n{context}\n\nQuestion:
{question}"
        }]
    )
    return next(b.text for b in response.content if b.type
== "text")
```

— 5. Ask

```
print(answer("What columns does the orders table have?"))
# "The orders table has: order_id (VARCHAR PK), customer_id
(INT FK), amount (DECIMAL 10,2),
# status (VARCHAR: placed/shipped/delivered/cancelled),
created_at (TIMESTAMP). [1]"
```

```
print(answer("When will today's gold layer data be
ready?"))
# "Gold layer tables are updated daily. The orders pipeline
runs at 2am UTC and data
# must be available by 6am UTC [1][2]."
```

```
print(answer("What is the revenue for Q1?"))
# "I don't have that information."
```

Chunking Strategy

How you split documents is one of the biggest levers for RAG quality.

```
import re
from dataclasses import dataclass

@dataclass
class Chunk:
    text: str
    doc_id: str
    doc_title: str
    chunk_idx: int
    metadata: dict

def chunk_document(doc_id: str, title: str, text: str,
                  max_tokens: int = 400, overlap_tokens:
int = 50) -> list[Chunk]:
    """
    1. Split on section boundaries (## headings) first
    2. If a section is still too long, split by tokens with
    overlap
    """
    # Split on markdown headings
    sections = re.split(r'\n(?:#{1,3}\s)', text)
    chunks = []

    for section in sections:
        if not section.strip():
            continue
        words = section.split()

        if len(words) <= max_tokens:
            chunks.append(Chunk(
                text=section.strip(),
                doc_id=doc_id,
                doc_title=title,
                chunk_idx=len(chunks),
                metadata={"section": section.split("\n")
[0].strip("# ")})
            ))
        else:
```

```

# Sliding window
start = 0
while start < len(words):
    end = min(start + max_tokens, len(words))
    chunk_text = " ".join(words[start:end])
    chunks.append(Chunk(
        text=chunk_text,
        doc_id=doc_id,
        doc_title=title,
        chunk_idx=len(chunks),
        metadata={"section":
section.split("\n")[0].strip("# "), "is_split": True}
    ))
    start += max_tokens - overlap_tokens

return chunks

# Add metadata to each chunk text before embedding
def format_chunk_for_embedding(chunk: Chunk) -> str:
    """Prepend metadata — helps retrieval by giving the
    model more signal."""
    return f"Document: {chunk.doc_title}\nSection:
{chunk.metadata.get('section', '')}\n\n{chunk.text}"

```

Metadata Filtering

Filter by metadata before or during vector search — much faster than post-filtering.

```

# Example: only retrieve docs from a specific team or
updated after a date

```

```

@dataclass
class IndexedChunk:
    chunk: Chunk
    embedding: np.ndarray

class FilterableIndex:
    def __init__(self):
        self.items: list[IndexedChunk] = []

```

```

def add(self, chunk: Chunk, embedding: np.ndarray):
    self.items.append(IndexedChunk(chunk=chunk,
embedding=embedding))

def search(self, query_vec: np.ndarray, k: int = 5,
           filters: dict = None) -> list[dict]:
    candidates = self.items

    # Apply pre-filters
    if filters:
        for key, value in filters.items():
            candidates = [
                item for item in candidates
                if item.chunk.metadata.get(key) ==
value
            ]

    if not candidates:
        return []

    vecs = np.array([c.embedding for c in
candidates])
    vecs = vecs / np.linalg.norm(vecs, axis=1,
keepdims=True)
    scores = vecs @ query_vec
    top_k = np.argsort(scores)[::-1][:k]

    return [{
        "text": candidates[i].chunk.text,
        "doc_id": candidates[i].chunk.doc_id,
        "score": float(scores[i]),
        "metadata": candidates[i].chunk.metadata,
    } for i in top_k]

# Search only docs tagged as "data_dictionary"
results = index.search(query_vec, k=5, filters={"type":
"data_dictionary"})

```

Retrieval Tuning

```

# How many chunks to retrieve?
# Too few: miss relevant info
# Too many: dilute context, increase cost, confuse the LLM
# Start with k=3-5, increase if answers are incomplete

# Score threshold – don't include irrelevant chunks. The
# score scale depends on the embedding model,
# so 0.7 is only a placeholder: pick the threshold from
# labelled queries of your own
def retrieve_with_threshold(query: str, k: int = 5,
min_score: float = 0.7) -> list[dict]:
    results = retrieve(query, k=k)
    return [r for r in results if r["score"] >= min_score]

# Contextual compression – summarize each chunk before
# including
# Useful when chunks are long and only a part is relevant
def compress_chunk(query: str, chunk: str) -> str:
    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=200,
        messages=[{
            "role": "user",
            "content": f"Extract only the parts of this
text relevant to the question.\n\nQuestion:
{query}\n\nText: {chunk}\n\nRelevant excerpt:"
        }]
    )
    return next(b.text for b in response.content if b.type
== "text")

# Parent-child chunking
# Index small chunks (for precision), but retrieve larger
# parent chunks (for context)
# Child: 128 tokens → used for matching
# Parent: 512 tokens → used in prompt

```

The Generation Step

```

def generate_answer(question: str, context_chunks:
list[dict],
                    model: str = "claude-sonnet-5-5") ->
dict:
    if not context_chunks:
        return {"answer": "I don't have relevant
information to answer this question.", "sources": []}

    # Build context block
    context_parts = []
    for i, chunk in enumerate(context_chunks):
        source = chunk.get("doc_id", f"Source {i+1}")
        context_parts.append(f"[{i+1}]
({source})\n{chunk['text']}")
    context = "\n\n---\n\n".join(context_parts)

    system = """You are a helpful data engineering
assistant.

Rules:
- Answer using ONLY the provided context
- If the answer isn't in the context, say exactly: "I don't
have that information in my knowledge base."
- Cite sources using [1], [2] etc.
- Be concise and direct
- Do not make up table names, column names, or pipeline
details"""

    response = claude_client.messages.create(
        model=model,
        max_tokens=1024,
        system=system,
        messages=[{
            "role": "user",
            "content": f"Context:\n\n{context}\n\n---
\n\nQuestion: {question}"
        }]
    )

    return {
        "answer": next(b.text for b in response.content if
b.type == "text"),

```

```

        "sources": [c.get("doc_id") for c in
context_chunks],
        "tokens": response.usage.input_tokens +
response.usage.output_tokens,
    }

```

Hybrid Search

Combine vector search (semantic) with keyword search (exact match) — better than either alone.

```

# BM25 keyword search
# pip install rank-bm25
from rank_bm25 import BM25Okapi

def build_bm25_index(texts: list[str]) -> BM25Okapi:
    tokenized = [t.lower().split() for t in texts]
    return BM25Okapi(tokenized)

def hybrid_search(query: str, doc_texts: list[str],
doc_vecs: np.ndarray,
                  bm25: BM25Okapi, k: int = 5, alpha: float
= 0.5) -> list[int]:
    """
    alpha=0: pure keyword
    alpha=1: pure vector
    alpha=0.5: equal blend
    """
    # Vector scores
    q_vec = embed([query])[0]
    vec_scores = doc_vecs @ q_vec

    # BM25 scores (normalized)
    bm25_scores =
np.array(bm25.get_scores(query.lower().split()))
    if bm25_scores.max() > 0:
        bm25_scores = bm25_scores / bm25_scores.max()

    # Combine
    combined = alpha * vec_scores + (1 - alpha) *

```

```
bm25_scores
    return np.argsort(combined)[::-1][:k].tolist()
```

Re-ranking

After retrieval, use a cross-encoder to re-score and re-order chunks. More expensive but much more accurate.

```
# pip install sentence-transformers
from sentence_transformers import CrossEncoder

reranker = CrossEncoder("cross-encoder/ms-marco-MiniLM-L6-v2")

def rerank(query: str, chunks: list[dict], top_n: int = 3)
-> list[dict]:
    """Re-rank retrieved chunks using a cross-encoder."""
    pairs = [(query, chunk["text"]) for chunk in chunks]
    scores = reranker.predict(pairs)
    ranked = sorted(zip(scores, chunks), key=lambda pair:
pair[0], reverse=True) # key avoids comparing dicts on
ties
    return [chunk for _, chunk in ranked[:top_n]]

# Full pipeline: retrieve more (k=10), re-rank, use top 3
def retrieve_and_rerank(query: str, k_retrieve: int = 10,
k_final: int = 3) -> list[dict]:
    initial = retrieve(query, k=k_retrieve)
    return rerank(query, initial, top_n=k_final)
```

Advanced RAG Patterns

Hypothetical Document Embeddings (HyDE)

Generate a hypothetical answer, embed it, use that vector for retrieval. Works well when queries are vague.

```
def hyde_retrieve(query: str, k: int = 5) -> list[dict]:
    # Step 1: generate a hypothetical answer
    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=200,
        messages=[{"role": "user", "content": f"Write a
short paragraph that would be a good answer to: {query}"}]
    )
    hypothetical = next(b.text for b in response.content if
b.type == "text")

    # Step 2: embed the hypothetical answer (not the query)
    hyp_vec = embed([hypothetical])[0]
    scores = doc_vecs @ hyp_vec
    top_k = np.argsort(scores)[::-1][:k]
    return [{"text": doc_texts[i], "score":
float(scores[i])} for i in top_k]
```

Multi-query retrieval

Generate multiple phrasings of the query, retrieve for each, deduplicate.

```
def multi_query_retrieve(question: str, k: int = 5) ->
list[dict]:
    # Generate 3 alternative phrasings
    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=256,
        messages=[{
            "role": "user",
            "content": f"Generate 3 different ways to
phrase this question. Return as a JSON array.\n\nQuestion:
{question}"
        }]
    )
    import json
    queries = json.loads(next(b.text for b in
response.content if b.type == "text"))
    queries.append(question) # include original

    # Retrieve for each query, deduplicate
```

```

seen = set()
all_results = []
for q in queries:
    for r in retrieve(q, k=k):
        key = r["text"][:100]
        if key not in seen:
            seen.add(key)
            all_results.append(r)

# Sort by score
return sorted(all_results, key=lambda x: x["score"],
reverse=True)[:k]

```

RAG with conversation history

```

def chat_with_rag(messages: list[dict],
knowledge_base_vecs, knowledge_base_docs) -> str:
    """
    Multi-turn RAG: use the latest user message for
    retrieval,
    but include full conversation history in the generation
    prompt.
    """
    latest_question = next(m["content"] for m in
reversed(messages) if m["role"] == "user")
    context_chunks = retrieve(latest_question, k=4)
    context_text = "\n\n".join(c["text"] for c in
context_chunks)

    # Inject context into system prompt
    system = f"""You are a data engineering assistant.
Answer questions based on the context below AND the
conversation history.
If unsure, say so.

CONTEXT:
{context_text}"""

    response = claude_client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=1024,

```

```

        system=system,
        messages=messages
    )
    return next(b.text for b in response.content if b.type
== "text")

```

Evaluating RAG Quality

```

# RAGAS-style metrics (pip install ragas)

# 1. Answer Faithfulness – does the answer stick to the
context?
def eval_faithfulness(question: str, answer: str, context:
str) -> float:
    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=64,
        extra_body={"temperature": 0}, # Haiku 4.5
accepts it; the Python SDK has no temperature parameter
        messages=[{
            "role": "user",
            "content": f"""On a scale of 0.0 to 1.0, how
faithful is this answer to the context?
1.0 = every claim in the answer is supported by the context
0.0 = the answer contradicts or ignores the context

Context: {context}
Answer: {answer}

Score (just the number):"""
        }]
    )
    try:
        return float(next(b.text for b in response.content
if b.type == "text").strip())
    except ValueError:
        return 0.0

# 2. Answer Relevance – does it actually answer the
question?

```

```

def eval_relevance(question: str, answer: str) -> float:
    response = claude_client.messages.create(
        model="claude-haiku-4-5-20251001",
        max_tokens=64,
        extra_body={"temperature": 0},    # Haiku 4.5
    accepts it; the Python SDK has no temperature parameter
    messages=[{
        "role": "user",
        "content": f"Does this answer address the
question? Score 0.0-1.0.\n\nQuestion: {question}\nAnswer:
{answer}\n\nScore:"
    }]
    )
    try:
        return float(next(b.text for b in response.content
if b.type == "text").strip())
    except ValueError:
        return 0.0

# 3. Retrieval precision – are retrieved chunks actually
relevant?
def eval_retrieval(question: str, chunks: list[str]) ->
float:
    relevant = 0
    for chunk in chunks:
        resp = claude_client.messages.create(
            model="claude-haiku-4-5-20251001",
            max_tokens=8,
            extra_body={"temperature": 0},
            messages=[{"role": "user", "content": f"Is this
chunk relevant to '{question}'? Answer YES or
NO.\n\n{chunk}"}]
        )
        if "YES" in next(b.text for b in resp.content if
b.type == "text").upper():
            relevant += 1
    return relevant / len(chunks)

# Run a full eval suite
test_cases = [
    {"question": "What columns does the orders table
have?", "expected_keywords": ["order_id", "amount",

```

```

    "status"]}],
    {"question": "When does the gold layer update?",
    "expected_keywords": ["daily"]}],
    {"question": "Who do I contact if the pipeline is
late?", "expected_keywords": ["alert", "email"]}],
]

for tc in test_cases:
    chunks = retrieve(tc["question"])
    answer_text = generate_answer(tc["question"], chunks)
    ["answer"]
    faithfulness = eval_faithfulness(tc["question"],
answer_text, "\n".join(c["text"] for c in chunks))
    relevance = eval_relevance(tc["question"],
answer_text)
    has_keywords = all(kw.lower() in answer_text.lower()
for kw in tc["expected_keywords"])
    print(f"Q: {tc['question'][:50]}")
    print(f" Faithfulness: {faithfulness:.2f} Relevance:
{relevance:.2f} Keywords: {has_keywords}")

```

Common Pitfalls

Pitfall	Symptom	Fix
Judging RAG by reading a few answers	Looks great in the demo, wrong in production	A labelled question set; measure retrieval (recall@k) and generation (faithfulness) separately
Blaming the LLM for retrieval failures	Prompt tweaks don't help	Check first whether the right chunk was retrieved at all — most RAG failures are retrieval failures
Naive fixed-size chunking	Answers split across chunks; tables and code broken apart	Structure-aware chunking; include the title/section path in each chunk; tune size on your eval set
Vector search only	Misses exact terms: error codes, table names, IDs	Hybrid search (<u>BM25</u> + vectors), then re-rank

Pitfall	Symptom	Fix
Stuffing 20+ chunks "to be safe"	Higher cost, slower, and the model misses the key passage	Retrieve more, re-rank, and pass the top few
No metadata filters	Answers mix environments, old doc versions, or other tenants' data	Filter by source, date, version, and access permissions at query time
Ignoring document permissions	Users see content they're not allowed to see	Enforce ACLs in retrieval (filter on the user's groups) — never rely on the prompt
Stale index	Answers cite outdated runbooks	Incremental re-indexing on change; store <code>updated_at</code> ; delete vectors when documents are deleted
No "I don't know" path	Confident hallucinations when nothing relevant is retrieved	Relevance threshold plus an explicit instruction to say the answer isn't in the context
Treating retrieved text as trusted	Prompt injection from documents	Delimit context as data; don't let retrieved text trigger tools without checks

Cheat Sheet

The RAG pipeline in one view

INDEX (offline) (online)	QUERY
load docs → clean → chunk → embed → upsert (rewrite) → embed → retrieve top-k + metadata (source, date, ACL) filter → re-rank → top-n chunks	question → → →
prompt with citations → answer → log + eval	

Knob	Typical starting point	Tune by
Chunk size / overlap	300–800 tokens / 10–20%	Recall@k on your eval questions

Knob	Typical starting point	Tune by
Retrieve k	20–50 candidates	Recall of the right chunk
Final n after re-ranking	3–8 chunks	Answer quality vs. tokens
Hybrid weighting	Reciprocal Rank Fusion (RRF)	Queries with exact terms vs. paraphrases
Similarity threshold	Calibrate on labelled data	Rate of "no answer" vs. hallucination

Metrics

Stage	Metric	Question it answers
Retrieval	Recall@k / hit rate	Is the right chunk in the top k?
Retrieval	MRR / nDCG	How high is it ranked?
Generation	Faithfulness / groundedness	Is every claim supported by the context?
Generation	Answer relevance	Does it actually answer the question?
End to end	Correctness vs. a reference answer	Is it right?

When RAG is the wrong tool: questions about aggregates ("total revenue last month") → text-to-SQL against the warehouse · stable style or format changes → prompting or fine-tuning · a small corpus that fits in context → put it all in the prompt and cache it

Interview Questions

Q: What is RAG and what problem does it solve? A: RAG (Retrieval-Augmented Generation) solves the limitation that LLMs only know what they were trained on. Before generating an answer, a retriever finds relevant documents from your own knowledge base and includes them in the prompt. The LLM then answers based on that retrieved context, not just training data. This enables factual, up-to-date, citable answers over private data without retraining the model.

Q: What is chunking and why does the chunk size matter? A: Chunking splits large documents into smaller pieces before embedding. Too large: the embedding captures too much meaning, retrieval is imprecise. Too small: each chunk lacks context, the answer

may be incomplete. Typical sweet spot: 256–512 tokens with 10–20% overlap between adjacent chunks to preserve context at boundaries. Semantic splits (by paragraph or section) usually beat fixed-size splits.

Q: What is hybrid search and when is it better than pure vector search? A: Hybrid search combines vector search (semantic similarity) with BM25 keyword search. Pure vector search can miss exact term matches (a product code like "SKU-4872" has no semantic neighbors). Pure keyword search misses paraphrases ("how many orders" won't match "order count"). Hybrid combines both scores (e.g., 50/50 blend or Reciprocal Rank Fusion) and outperforms either alone in practice — most production RAG systems use it.

Q: What is re-ranking and when would you use it? A: After initial retrieval (fast, ANN search), re-ranking uses a more expensive cross-encoder model to re-score each (query, chunk) pair holistically. The cross-encoder sees both query and document together — more accurate than comparing independent embeddings. Use it when retrieval precision matters more than latency, or when you're retrieving 10-20 candidates and need to select the top 3. Adds ~200-500ms latency but significantly improves relevance.

Q: How do you evaluate a RAG pipeline? A: Four metrics: (1) Faithfulness — does the answer only use information from retrieved context? (2) Answer relevance — does it actually answer the question? (3) Context precision — how many retrieved chunks were actually useful? (4) Context recall — did retrieval find all the relevant information? Use LLM-as-judge for automated evaluation, and maintain a golden test set of question-answer pairs to catch regressions when you change chunking, retrieval, or the generation prompt.

Further Reading

- [Anthropic: Contextual Retrieval](https://www.anthropic.com/engineering/contextual-retrieval) (https://www.anthropic.com/engineering/contextual-retrieval) — adding context to chunks before embedding, with benchmark results
- [RAGAS documentation](https://docs.ragas.io/) (https://docs.ragas.io/) — RAG evaluation metrics
- [LlamaIndex](https://developers.llamaindex.ai/python/framework/) (https://developers.llamaindex.ai/python/framework/) and [LangChain retrieval docs](https://docs.langchain.com/oss/python/langchain/retrieval) (https://docs.langchain.com/oss/python/langchain/retrieval)
- *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks* — Lewis et al., 2020 (the original RAG paper)
- [Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) • [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) • [Eval & Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/)

Previous: [Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/) · **Next:** [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026