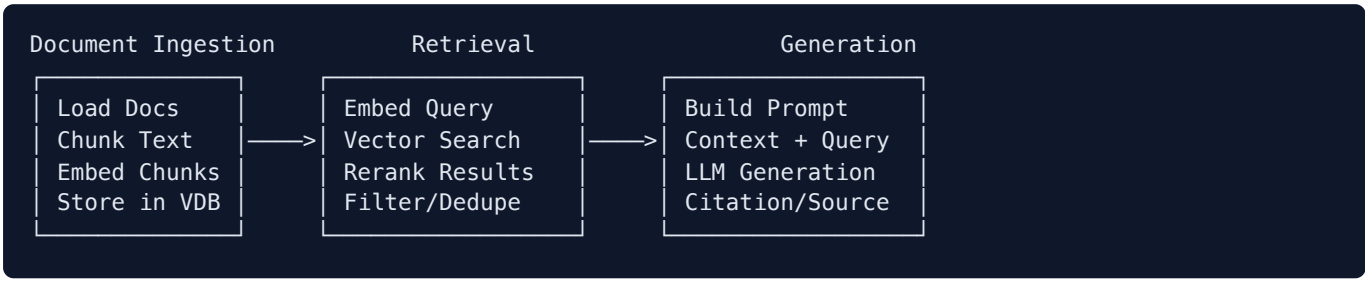
RAG Pipeline Overview

RAG is the most practical way to give an LLM access to private or current data without fine-tuning. Every production RAG system follows the same three-phase pattern -- get this pipeline right and the rest is optimization.



Chunking Strategies

Chunking is where most RAG pipelines silently fail -- chunks that are too small lose context, chunks that are too large dilute relevance. The right strategy depends entirely on your document type.

Strategy	Chunk Size	Overlap	Best For
Fixed-size	256-512 tokens	10-20%	General purpose, simple
Sentence-based	3-5 sentences	1 sentence	Narrative text
Paragraph-based	1-3 paragraphs	1 paragraph	Structured docs
Recursive character	500-1000 chars	100-200 chars	Mixed content
Semantic	Varies	Concept boundary	Research papers
Document-specific	By section/page	Headers as context	Technical docs
Agentic (LLM-based)	Varies	Determined by model	Complex/messy content

Chunk Size Guidelines

Content Type	Recommended Size	Reasoning
FAQ / Q&A	100-200 tokens	Short, self-contained
Technical docs	300-500 tokens	Enough context per concept
Legal contracts	500-1000 tokens	Clauses need full context
Code	Function/class level	Logical units
Chat logs	Per conversation turn	Natural boundaries

Chunking Code Example (LangChain)

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=50,
    length_function=len,
    separators=["\n\n", "\n", ". ", " ", ""]
)
chunks = splitter.split_text(document_text)
```

Embedding Models Comparison

Your choice of embedding model determines the ceiling on retrieval quality -- no amount of reranking or prompt tuning can fix a bad embedding. Match the model to your data type, language, and latency budget.

Model	Dimensions	Max Tokens	MTEB Score	Cost	Notes
OpenAI text-embedding-3-large	3072	8191	~64.6	$0.13/1M tokens	Dimension reduction supported
OpenAI text-embedding-3-small	1536	8191	~62.3	$0.02/1M tokens	Good price/performance
Cohere embed-v3	1024	512	~64.5	$0.10/1M tokens	Multi-lingual
Voyage-3	1024	32000	~67.2	$0.06/1M tokens	Long context, code-aware
BGE-large-en-v1.5	1024	512	~64.2	Free (self-host)	Open source
E5-mistral-7b	4096	32768	~66.6	Free (self-host)	Best open source
GTE-Qwen2-7B	3584	131072	~67.2	Free (self-host)	Very long context
Google text-embedding-004	768	2048	~66.3	$0.025/1M chars	Vertex AI

Embedding Best Practices

- Normalize embeddings before cosine similarity
- Use the same model for indexing and querying
- Prefix queries with "query: " and docs with "passage: " for asymmetric models
- Batch embedding requests (100-1000 items per call)
- Cache embeddings; do not re-embed unchanged content

Vector Database Comparison

The vector database you choose dictates your scaling ceiling, operational complexity, and whether you can do hybrid search natively. Pick based on your scale and team capacity, not hype.

Database	Type	Max Vectors	Filtering	Hybrid Search	Managed?
Pinecone	Cloud-native	1B+	Metadata	Yes	Yes
Weaviate	Self-host/Cloud	Billions	GraphQL	Yes	Both
Qdrant	Self-host/Cloud	Billions	Payload	Yes	Both
ChromaDB	Embedded	Millions	Metadata	No	No
pgvector	PostgreSQL ext	Millions	SQL	Yes (via SQL)	Both
Milvus	Distributed	Billions	Expressions	Yes	Both
FAISS	In-memory lib	Millions	No (manual)	No	No
Elasticsearch	Distributed	Billions	Full query DSL	Yes	Both

When to Use Which

Scenario	Recommended
Prototype / POC	ChromaDB, FAISS
Already using Postgres	pgvector
Production, managed	Pinecone, Weaviate Cloud
High scale, self-managed	Milvus, Qdrant
Need full-text + vector	Elasticsearch, Weaviate

Retrieval Strategies

Dense retrieval alone misses exact keyword matches; sparse retrieval alone misses semantic meaning. Understanding when to use each strategy -- and when to combine them -- is the difference between a good RAG system and a great one.

Strategy	Description	Pros	Cons
Dense retrieval	Embedding similarity	Semantic understanding	Misses keywords
Sparse retrieval (BM25)	Keyword/TF-IDF	Exact match, fast	No semantic understanding
Hybrid	Dense + Sparse fusion	Best of both	More complex
HyDE	LLM generates hypothetical doc, embed that	Better query representation	Extra LLM call

Strategy	Description	Pros	Cons
Multi-query	LLM generates N query variants	Better recall	N x retrieval cost
Parent-child	Retrieve child, return parent	Precise match, full context	More indexing work
Contextual retrieval	Prepend document context to chunks	Better relevance	Higher indexing cost

Hybrid Search Fusion

```
# Reciprocal Rank Fusion (RRF)
def rrf_score(ranks, k=60):
    return sum(1.0 / (k + r) for r in ranks)

# Example: combine dense and sparse results
dense_results = vector_search(query, top_k=20)
sparse_results = bm25_search(query, top_k=20)

scores = {}
for rank, doc in enumerate(dense_results):
    scores[doc.id] = scores.get(doc.id, 0) + 1/(60 + rank)
for rank, doc in enumerate(sparse_results):
    scores[doc.id] = scores.get(doc.id, 0) + 1/(60 + rank)

final = sorted(scores.items(), key=lambda x: x[1], reverse=True)
```

Retrieval Metrics

You cannot improve what you do not measure. These metrics tell you whether your retrieval is actually finding the right documents and whether the LLM is using them faithfully.

Metric	Formula	What It Measures
Recall@k	relevant_in_top_k / total_relevant	Coverage of relevant docs
Precision@k	relevant_in_top_k / k	Accuracy of top results
MRR	1/rank_of_first_relevant	Position of first hit
NDCG@k	DCG@k / IDCG@k	Ranking quality
Hit Rate	queries_with_hit / total_queries	Basic retrieval success
Faithfulness	claims_supported / total_claims	LLM output grounded in context
Answer Relevancy	Semantic sim(answer, question)	Output addresses the question

Reranking

Initial vector search returns "good enough" candidates, but reranking transforms them into precisely ordered results. Adding a reranker is often the single highest-ROI improvement you can make to an existing RAG

pipeline.

Reranker	Type	Latency	Quality
Cohere Rerank v3	API	~200ms	High
Jina Reranker v2	API/Self-host	~150ms	High
BGE Reranker v2-m3	Self-host	~100ms	Good
Cross-encoder (ms-marco)	Self-host	~50ms	Good
LLM-based (GPT/Claude)	API	~1-3s	Highest
ColBERT	Self-host	~30ms	Good

Reranking Pattern

```
# Retrieve more, rerank to fewer
candidates = vector_search(query, top_k=50)  # over-fetch
reranked = reranker.rank(query, candidates, top_k=5)  # narrow down
context = "\n\n".join([doc.text for doc in reranked])
```

Prompt Construction for RAG

The prompt is where retrieval meets generation -- a poorly structured RAG prompt produces hallucinations even when the right documents are retrieved. Always enforce citation and include a fallback for missing information.

```
Use the following context to answer the question.
If the answer is not in the context, say "I don't have enough information."
Always cite the source document for each claim.

Context:
{chunk_1}
Source: {source_1}

{chunk_2}
Source: {source_2}

Question: {user_query}
Answer:
```

Common Pitfalls

Every RAG failure mode here has been encountered in real production systems. When your RAG answers are wrong, irrelevant, or missing, start debugging from this list.

Pitfall	Problem	Fix
Chunks too small	Lost context	Increase size or use parent-child

Pitfall	Problem	Fix
Chunks too large	Diluted relevance	Decrease size, improve chunking
No overlap	Split concepts	Add 10-20% overlap
Wrong embedding model	Poor retrieval	Benchmark on your data
No metadata filtering	Irrelevant results	Add source, date, category metadata
No reranking	Mediocre top-k	Add cross-encoder or Cohere reranker
Ignoring freshness	Stale answers	Track document timestamps, re-index
Single retrieval strategy	Missed results	Use hybrid (dense + sparse)
No evaluation	Unknown quality	Measure recall, precision, faithfulness
Embedding query = doc embedding	Sub-optimal for asymmetric search	Use query prefixes for asymmetric models

Quick Sizing Reference

Use this table to estimate infrastructure requirements before you start building. Getting the storage and database choice wrong early means a painful migration later.

Documents	Embedding Dim	Approx Vector Storage	Recommended DB
< 10K	1536	~60 MB	ChromaDB, FAISS
10K-1M	1536	~6 GB	pgvector, Qdrant
1M-100M	1024	~400 GB	Pinecone, Milvus
100M+	768-1024	~4+ TB	Milvus cluster, Elasticsearch