RAG Complete Guide

Master Cheatsheet 2025

Retrieval Augmented Generation | From Basics to Advanced

■ What is RAG?

RAG (Retrieval Augmented Generation) = Retrieve relevant context from external data → Augment the prompt → Generate accurate response

Problem with LLMs	How RAG Solves It
Knowledge cutoff date	Access real-time/updated documents
Hallucinations	Ground responses in actual data
No private data access	Query your own documents/databases
Generic responses	Provide domain-specific context
Token limit constraints	Retrieve only relevant chunks
No source attribution	Cite specific documents/pages

■■ RAG Architecture - The Pipeline

Stage	Step	Description	Tools
1. Ingestion	Load	Load documents from sources	PyPDF, Unstructured, Web loaders
2. Ingestion	Split	Chunk documents into pieces	RecursiveCharacterTextSplitter
3. Ingestion	Embed	Convert chunks to vectors	OpenAI, HuggingFace embeddings
4. Ingestion	Store	Save vectors in database	Chroma, Pinecone, FAISS
5. Retrieval	Query	Convert question to vector	Same embedding model
6. Retrieval	Search	Find similar vectors	Similarity search, MMR
7. Generation	Augment	Add context to prompt	Prompt template
8. Generation	Generate	LLM generates response	GPT-4, Claude, Llama

■ Step 1: Document Loading

Document Type	Loader	Code Example
PDF	PyPDFLoader	PyPDFLoader("doc.pdf").load()
PDF (better)	UnstructuredPDFLoader	UnstructuredPDFLoader("doc.pdf").load()
Word (.docx)	Docx2txtLoader	Docx2txtLoader("doc.docx").load()
Text	TextLoader	TextLoader("file.txt").load()
CSV	CSVLoader	CSVLoader("data.csv").load()
JSON	JSONLoader	JSONLoader("data.json", jq_schema=".[]").load()

HTML	UnstructuredHTMLLoader	UnstructuredHTMLLoader("page.html").load()
Markdown	UnstructuredMarkdownLoader	UnstructuredMarkdownLoader("doc.md").load()
Web Page	WebBaseLoader	WebBaseLoader("https://...").load()
Multiple URLs	WebBaseLoader	WebBaseLoader(["url1", "url2"]).load()
YouTube	YoutubeLoader	YoutubeLoader.from_youtube_url(url, add_video_info=True).load()
Wikipedia	WikipediaLoader	WikipediaLoader(query="AI", load_max_docs=3).load()
Arxiv	ArxivLoader	ArxivLoader(query="RAG", load_max_docs=5).load()
Directory	DirectoryLoader	DirectoryLoader("./docs", glob="**/*.pdf").load()
Notion	NotionDBLoader	NotionDBLoader(token, database_id).load()
Google Drive	GoogleDriveLoader	GoogleDriveLoader(folder_id="...").load()
Confluence	ConfluenceLoader	ConfluenceLoader(url, username, api_key).load()
Slack	SlackDirectoryLoader	SlackDirectoryLoader("./slack_export").load()

```
# Multiple document types
from langchain_community.document_loaders import (
PyPDFLoader, WebBaseLoader, DirectoryLoader, TextLoader
)

# Load single PDF
pdf_docs = PyPDFLoader("report.pdf").load()

# Load all PDFs from directory
all_pdfs = DirectoryLoader("./documents", glob="**/*.pdf", loader_cls=PyPDFLoader).load()

# Load web pages
web_docs = WebBaseLoader(["https://docs.langchain.com"]).load()

# Combine all documents
all_docs = pdf_docs + all_pdfs + web_docs
```

✂■ Step 2: Chunking Strategies

■ *Golden Rule: chunk_size=500-1000 tokens, chunk_overlap=10-20% of chunk_size*

Splitter	Best For	Key Parameters
RecursiveCharacterTextSplitter	General purpose (RECOMMENDED)	chunk_size, chunk_overlap, separators
CharacterTextSplitter	Simple splitting	separator, chunk_size
TokenTextSplitter	Token-accurate splits	chunk_size, chunk_overlap
SentenceTransformersTokenTextSplitter	Sentence-transformer models	chunk_overlap, tokens_per_chunk
MarkdownHeaderTextSplitter	Markdown documents	headers_to_split_on
HTMLHeaderTextSplitter	HTML documents	headers_to_split_on
RecursiveCharacterTextSplitter.from_language	Source code	language (Python, JS, etc.)
SemanticChunker	Meaning-based splits	breakpoint_threshold_type

■■ Chunk Size Guidelines

Use Case	Chunk Size	Overlap	Why
Q&A / Chatbots	500-1000	50-100	Focused, relevant answers
Summarization	1000-2000	100-200	More context per chunk
Code Analysis	1000-1500	100-150	Keep functions intact
Legal / Medical	300-500	50	Precise, detailed retrieval
General Search	500-800	80-100	Balanced approach

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Standard splitter (RECOMMENDED)
splitter = RecursiveCharacterTextSplitter(
chunk_size=1000,
chunk_overlap=200,
length_function=len,
separators=["\n\n", "\n", " ", ""] # Try these in order
)
chunks = splitter.split_documents(docs)

# For code
from langchain.text_splitter import Language
code_splitter = RecursiveCharacterTextSplitter.from_language(
language=Language.PYTHON, chunk_size=1000, chunk_overlap=100
)

# Semantic chunking (meaning-based)
from langchain_experimental.text_splitter import SemanticChunker
semantic_splitter = SemanticChunker(embeddings, breakpoint_threshold_type="percentile")
```

■ Step 3: Embeddings

Provider	Model	Dimensions	Best For
OpenAI	text-embedding-3-small	1536	Cost-effective, good quality
OpenAI	text-embedding-3-large	3072	Best quality, higher cost
OpenAI	text-embedding-ada-002	1536	Legacy, still popular
Cohere	embed-english-v3.0	1024	English focus, great quality
Cohere	embed-multilingual-v3.0	1024	Multi-language support
Voyage AI	voyage-large-2	1536	High quality, RAG optimized
Google	text-embedding-004	768	Free tier available
HuggingFace	all-MiniLM-L6-v2	384	Free, fast, local
HuggingFace	all-mpnet-base-v2	768	Free, better quality
HuggingFace	bge-large-en-v1.5	1024	Top open-source
HuggingFace	e5-large-v2	1024	Microsoft, excellent
Ollama	nomic-embed-text	768	Free, local, good
Ollama	mxbai-embed-large	1024	Free, local, better

```
# OpenAI (best for production)
from langchain_openai import OpenAIEmbeddings
embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

# HuggingFace (free, local)
from langchain_huggingface import HuggingFaceEmbeddings
embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

# Ollama (free, local)
from langchain_ollama import OllamaEmbeddings
embeddings = OllamaEmbeddings(model="nomic-embed-text")

# Test embedding
vector = embeddings.embed_query("Hello world")
print(f"Dimensions: {len(vector)}") # e.g., 1536
```

■■ Step 4: Vector Stores

Vector DB	Type	Best For	Key Feature
Chroma	Local/Embedded	Prototyping, small projects	Easy setup, persistent
FAISS	Local/Memory	Fast similarity search	Facebook AI, very fast
Pinecone	Cloud Managed	Production, scale	Fully managed, metadata
Weaviate	Self-host/Cloud	Hybrid search	GraphQL, multi-modal
Qdrant	Self-host/Cloud	Production, filtering	Fast, advanced filters
Milvus	Self-host/Cloud	Large scale	Billion+ vectors
PGVector	PostgreSQL ext	SQL + vectors	Use existing Postgres
Redis	In-memory	Real-time, caching	Ultra-fast retrieval
Supabase	Cloud	Full-stack apps	Postgres + pgvector
MongoDB Atlas	Cloud	Existing MongoDB users	Document + vector
Elasticsearch	Self-host/Cloud	Full-text + vector	Hybrid search
LanceDB	Embedded	Serverless, embedded	No server needed

```
# Chroma (easiest to start)
from langchain_chroma import Chroma
vectorstore = Chroma.from_documents(chunks, embeddings, persist_directory="./chroma_db")

# FAISS (fast, in-memory)
from langchain_community.vectorstores import FAISS
vectorstore = FAISS.from_documents(chunks, embeddings)
vectorstore.save_local("faiss_index") # Save to disk
vectorstore = FAISS.load_local("faiss_index", embeddings) # Load

# Pinecone (production)
from langchain_pinecone import PineconeVectorStore
vectorstore = PineconeVectorStore.from_documents(chunks, embeddings, index_name="my-index")

# Add more documents later
vectorstore.add_documents(new_chunks)
```

■ Step 5: Retrieval Strategies

Strategy	Description	When to Use

Similarity Search	Find k most similar vectors	Default, general purpose
MMR (Max Marginal Relevance)	Balance relevance & diversity	Avoid redundant results
Similarity Score Threshold	Only return above score	Quality filtering
Multi-Query Retriever	Generate multiple queries from one	Complex questions
Contextual Compression	Compress retrieved docs	Reduce token usage
Self-Query Retriever	Extract filters from query	Metadata filtering
Parent Document Retriever	Retrieve parent of matched chunk	Need more context
Ensemble Retriever	Combine multiple retrievers	Best of multiple methods
Hybrid Search	Vector + keyword (BM25)	When keywords matter
Re-ranking	Re-order results with cross-encoder	Improve precision

```
# Basic retriever
retriever = vectorstore.as_retriever(search_kwargs={"k": 4})

# MMR - diverse results
retriever = vectorstore.as_retriever(
search_type="mmr", search_kwargs={"k": 4, "fetch_k": 10, "lambda_mult": 0.5}
)

# Score threshold
retriever = vectorstore.as_retriever(
search_type="similarity_score_threshold", search_kwargs={"score_threshold": 0.7}
)

# Multi-query retriever (generates variations)
from langchain.retrievers.multi_query import MultiQueryRetriever
retriever = MultiQueryRetriever.from_llm(retriever=base_retriever, llm=llm)

# Hybrid search with BM25
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
bm25 = BM25Retriever.from_documents(docs)
ensemble = EnsembleRetriever(retrievers=[bm25, vectorstore_retriever], weights=[0.4, 0.6])
```

■■ Complete RAG Chain

```
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_chroma import Chroma
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# 1. Setup components
embeddings = OpenAIEmbeddings(model='text-embedding-3-small')
vectorstore = Chroma(persist_directory='./db', embedding_function=embeddings)
retriever = vectorstore.as_retriever(search_kwargs={'k': 4})
llm = ChatOpenAI(model='gpt-4o-mini', temperature=0)

# 2. RAG prompt template
template = '''Answer based on context. If unsure, say I dont know.
Context: {context}
Question: {question}'''
prompt = ChatPromptTemplate.from_template(template)

# 3. Build chain
def format_docs(docs):
return '\n\n'.join(doc.page_content for doc in docs)

rag_chain = (
{'context': retriever | format_docs, 'question': RunnablePassthrough()}
| prompt | llm | StrOutputParser()
)

# 4. Query
answer = rag_chain.invoke('What is RAG?')
```

■ Advanced RAG Techniques

Technique	Description	When to Use
Query Rewriting	LLM rewrites user query for better retrieval	Vague or complex queries
HyDE	Generate hypothetical answer, embed that	Short/unclear queries
Step-Back Prompting	Ask broader question first	Specific technical questions
Query Decomposition	Break complex query into sub-queries	Multi-part questions
Contextual Embedding	Add document context to chunks before embedding	Better semantic search
Re-ranking	Use cross-encoder to re-order results	Improve precision
RAPTOR	Hierarchical summarization + clustering	Large document collections
CRAG	Corrective RAG - verify and correct retrieval	High accuracy needs
Self-RAG	LLM decides when to retrieve	Dynamic retrieval needs
GraphRAG	Knowledge graph + vector search	Connected information
Agentic RAG	Agent decides retrieval strategy	Complex workflows

```
# Re-ranking with Cohere
from langchain.retrievers import ContextualCompressionRetriever
from langchain_cohere import CohereRerank

reranker = CohereRerank(model="rerank-english-v3.0", top_n=3)
compression_retriever = ContextualCompressionRetriever(
base_compressor=reranker, base_retriever=retriever
)

# Re-ranking with cross-encoder (free)
from langchain_community.cross_encoders import HuggingFaceCrossEncoder
from langchain.retrievers.document_compressors import CrossEncoderReranker

cross_encoder = HuggingFaceCrossEncoder(model_name="cross-encoder/ms-marco-MiniLM-L-6-v2")
reranker = CrossEncoderReranker(model=cross_encoder, top_n=3)
```

■ RAG Evaluation Metrics

Metric	What it Measures	Tool/Method
Context Relevance	Are retrieved docs relevant to query?	RAGAS, LLM-as-judge
Context Precision	Ratio of relevant chunks retrieved	Manual, RAGAS
Context Recall	Were all relevant chunks retrieved?	RAGAS, ground truth
Faithfulness	Is answer grounded in context?	RAGAS, LLM-as-judge
Answer Relevance	Does answer address the question?	RAGAS, human eval
Answer Correctness	Is the answer factually correct?	Ground truth comparison
Hallucination Rate	Info in answer not in context?	LLM-as-judge
Latency	End-to-end response time	Timing, LangSmith
Token Usage	Tokens used per query	API tracking

```
# Evaluate with RAGAS
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy, context_precision

# Prepare dataset
dataset = Dataset.from_dict({
"question": ["What is RAG?"],
"answer": ["RAG is Retrieval Augmented Generation..."],
"contexts": [["RAG combines retrieval with generation..."]],
"ground_truth": ["RAG is a technique that..."]
})

# Evaluate
results = evaluate(dataset, metrics=[faithfulness, answer_relevancy, context_precision])
print(results)
```

■ Best Practices & Tips

Category	Best Practice
Chunking	Keep chunk_overlap 10-20% of chunk_size to maintain context
Chunking	Add metadata (source, page, date) to chunks for filtering
Chunking	Test different chunk sizes - smaller for Q&A, larger for summaries

Embedding	Use same embedding model for indexing and querying
Embedding	Consider domain-specific embedding models for specialized data
Retrieval	Start with k=4-6, adjust based on results
Retrieval	Use MMR to avoid redundant similar chunks
Retrieval	Implement hybrid search (vector + BM25) for best results
Prompt	Include "If you don't know, say so" to reduce hallucinations
Prompt	Ask LLM to cite sources from the provided context
Production	Cache embeddings and common queries
Production	Monitor with LangSmith for debugging
Production	Implement feedback loop to improve retrieval

■ Quick Reference - When to Use What

Scenario	Recommended Setup
Prototype / Learning	Chroma + OpenAI embeddings + RecursiveCharacterTextSplitter
Production (managed)	Pinecone + OpenAI text-embedding-3-small + Re-ranking
Budget-friendly	FAISS + HuggingFace embeddings + Ollama LLM
Private/Local	Chroma + Ollama embeddings + Llama 3
Large scale (millions)	Milvus/Qdrant + Optimized embeddings + Sharding
Existing Postgres	PGVector + Same app database
Real-time updates	Redis + Streaming ingestion
Complex queries	Multi-query + Hybrid search + Re-ranking

Created by Manoj | The AI Dude Tamil ■

YouTube: The AI Dude Tamil | Master AI, Automation & Prompt Engineering

■ RAG = Your data + LLM power = Amazing results!