RAG

Retrieval-Augmented Generation

Complete Developer Cheat Sheet

WHAT IS RAG?

Grounding LLMs with external knowledge

RAG (Retrieval-Augmented Generation) combines a **retrieval system** with a **language model** to answer questions using up-to-date or proprietary knowledge without retraining. The model only sees retrieved context — not all stored documents.

1 QUERY	2 RETRIEVE	3 AUGMENT	4 GENERATE
User sends a question	Vector search finds top-k chunks	Chunks injected into prompt	LLM produces grounded answer

WHY RAG?

- ✦ No retraining needed for new data
- ✦ Reduces hallucinations with cited sources
- ✦ Works with private/internal knowledge bases
- ✦ Cheaper than fine-tuning at scale
- ✦ Answers are traceable and auditable
- ✦ Knowledge can be updated in real-time

WHEN TO USE RAG?

- ✦ Q&A; over large document corpora
- ✦ Customer support with product knowledge
- ✦ Code search & developer assistants
- ✦ Legal / medical document analysis
- ✦ News summarization with live sources
- ✦ Enterprise search over internal wikis

RAG vs FINE-TUNING vs PROMPT ENGINEERING

	RAG	Fine-Tuning	Prompt Engineering
Knowledge source	External DB / docs	Baked into weights	Context window only
Update cost	Low (re-index)	High (retrain)	None
Handles private data	✓ Yes	✓ Yes (risky)	✓ Yes (limited)
Hallucination risk	Low	Medium	High
Latency	Medium (retrieval)	Low	Low
Best for	Dynamic knowledge	Style / behavior	Task formatting

RAG ARCHITECTURE DEEP DIVE

Ingestion Pipeline + Query Pipeline

INGESTION PIPELINE

```
# Full ingestion pipeline
documents = load_documents(source)      # PDFs, web, DB, APIs
chunks    = chunk_documents(documents)  # Split into passages
cleaned   = clean_and_filter(chunks)    # Remove noise/PII
embeddings= embed(chunks)               # text → vector
index.add(embeddings, metadata=chunks)  # Store in vector DB
```

Stage	Description	Tools / Libraries
Load	Read raw documents from any source	LangChain loaders, Unstructured, PyPDF2
Chunk	Split text into overlapping passages	RecursiveTextSplitter, TokenSplitter
Clean	Remove HTML, fix encoding, filter noise	BeautifulSoup, regex, ftfy
Embed	Convert text to dense vectors	OpenAI, Cohere, sentence-transformers
Index	Store vectors + metadata in vector DB	Pinecone, Weaviate, Chroma, FAISS

QUERY PIPELINE

```
# Query pipeline
def rag_query(user_question):
    query_vec = embed(user_question)              # Embed question
    results   = index.search(query_vec, top_k=5)  # Retrieve chunks
    context   = format_context(results)           # Build context string
    prompt    = build_prompt(context, user_question)
    answer    = llm.generate(prompt)              # Call LLM
    return answer, results                        # Return with sources
```

RETRIEVAL STRATEGIES

Strategy	How It Works	When to Use
Dense (ANN)	Approximate nearest neighbor in embedding space	Default — semantic understanding
Sparse (BM25)	Keyword matching via TF-IDF weighting	Exact term matching required
Hybrid	Linear combo of dense + sparse scores	Best overall recall
MMR	Maximal Marginal Relevance — diversifies results	Avoid redundant passages
Multi-query	Rephrase query N ways, union results	Ambiguous or complex queries
HyDE	Generate hypothetical doc, embed it, retrieve	Narrow queries with sparse hits
Parent-child	Retrieve small chunks, return parent doc	Need broader context around matches
Self-querying	LLM generates metadata filters automatically	Structured / filtered corpora

CHUNKING STRATEGIES

How you split text drastically affects retrieval quality

FIXED-SIZE CHUNKING

Split by character/token count with overlap. Simple and predictable.

```
from langchain.text_splitter import
    RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=512,
    chunk_overlap=64,
    separators=[
        "\n\n", "\n", ". ", " ", ""
    ]
)
chunks = splitter.split_documents(docs)
```

SEMANTIC CHUNKING

Split on semantic similarity boundaries. Better coherence.

```
from langchain_experimental
    .text_splitter import
    SemanticChunker
from langchain_openai import
    OpenAIEmbeddings

chunker = SemanticChunker(
    OpenAIEmbeddings(),
    breakpoint_type="percentile",
    breakpoint_threshold=0.85,
)
chunks = chunker.split_documents(docs)
```

DOCUMENT-STRUCTURE CHUNKING

Split at natural structural boundaries: paragraphs, headers, sentences.

```
# Markdown header splitting
from langchain.text_splitter import
    MarkdownHeaderTextSplitter

headers = [
    ("#", "h1"),
    ("##", "h2"),
    ("###", "h3"),
]
splitter = MarkdownHeaderTextSplitter(
    headers_to_split_on=headers
)
chunks = splitter.split_text(md_text)
```

CHUNK SIZE GUIDELINES

Use Case	Chunk Size
Q&A over facts	128–256 tokens
Summarization	512–1024 tokens
Code search	Function-level
Legal / medical	Paragraph-level
Conversational	256–512 tokens

CHUNK OVERLAP — RULE OF THUMB

Use **10–15% overlap** relative to chunk size to preserve context at boundaries. Example: 512-token chunk → 50–75 token overlap. Higher overlap improves recall but increases storage and embedding cost. For code, overlap on full function boundaries instead.

METADATA ENRICHMENT

```
# Always attach metadata to chunks for post-retrieval filtering
chunk_with_meta = {
    "text": chunk.page_content,
    "metadata": {
        "source":      "docs/product_manual_v3.pdf",
        "page":        12,
        "section":     "Installation Guide",
        "doc_type":    "manual",
        "created_at":  "2024-01-15",
        "version":     "3.0",
        "language":    "en",
    }
}
```

EMBEDDINGS

Converting text to vectors for semantic search

POPULAR EMBEDDING MODELS

Model	Dims	Notes
text-embedding-3-small	1536	OpenAI, fast & che
text-embedding-3-large	3072	OpenAI, highest q
embed-english-v3.0	1024	Cohere, multilingu
all-MiniLM-L6-v2	384	OSS, CPU-friendly
bge-large-en-v1.5	1024	OSS, strong perf
nomic-embed-text	768	OSS, long context
voyage-2	1024	Voyage, top bench

SIMILARITY METRICS

Metric	Formula	Best For
Cosine	dot(a,b)/|a||b|	Normalized vecs
Dot Product	sum(a*b)	Scaled embeddings
Euclidean	sqrt(sum((a-b)^2))	Absolute distances
Inner Product	dot(a,b)	OpenAI embeddings

BATCHING TIP

Always embed in batches of 256–2048 texts. Cache embeddings — re-embedding same text wastes API cost.

VECTOR DATABASES

DB	Type	Scale	Highlights	Best For
Pinecone	Managed	Billions	Serverless, sub-ms latency	Production / no-infra
Weaviate	OSS + Cloud	Billions	GraphQL, hybrid, modules	Complex schema needs
Qdrant	OSS + Cloud	Billions	Rust, payload filters	High-throughput
Chroma	OSS (local)	Millions	Developer-friendly	Prototyping / local
FAISS	OSS library	Millions	Fastest in-process ANN	Batch / offline use
pgvector	Postgres ext	Millions	SQL + vectors together	Existing Postgres stack
Redis VSS	OSS + Cloud	Millions	Real-time, in-memory	Low-latency apps
Milvus	OSS + Cloud	Billions	Distributed, cloud-native	Enterprise scale

```
# Chroma (local prototyping)          # Pinecone (production)
import chromadb                        import pinecone
client = chromadb.Client()             pc = pinecone.Pinecone(api_key=KEY)
col = client.create_collection("rag")  idx = pc.Index("rag-index")
col.add(                               idx.upsert([
  documents=texts,                       (id, vec, meta)
  embeddings=vecs,                       for id,vec,meta in data
  ids=ids,                             ])
  metadatas=metas,                     results = idx.query(
)                                        vector=query_vec, top_k=5,
results = col.query(                     filter={'lang': 'en'},
  query_embeddings=[q_vec], n_results=5) include_metadata=True)
```

PROMPT ENGINEERING FOR RAG

Structuring context for maximum accuracy

BASIC RAG PROMPT TEMPLATE

```
SYSTEM_PROMPT = """
You are a helpful assistant. Answer the user's question
using ONLY the provided context. If the answer is not in
the context, say 'I don't have enough information.'
Always cite the source document when possible.
"""

def build_rag_prompt(context_chunks, question):
    context = "\n\n".join([
        f"[Source {i+1}: {c['source']}]\n{c['text']}",
        for i, c in enumerate(context_chunks)
    ])
    return f"""
Context:
{context}

Question: {question}

Answer (cite sources by [Source N]):
"""
```

ADVANCED PROMPTING PATTERNS

Pattern	Description
Chain-of-Thought	Add 'Reason step by step before answerir
Citation forcing	Require [Doc N] inline citations in answer
Grounding check	'Only use facts from context, flag gaps'
Structured output	Request JSON with answer + sources + c
Fallback instruction	'Say I don't know if answer not in context'
Role + domain	Add domain expert persona to system pro

CONTEXT WINDOW BUDGETING

Total context = System prompt + Retrieved chunks + Query + Output reserve

Component	Tokens
System prompt	200–500
Retrieved context	1000–4000
Conversation history	500–1000
User query	50–200
Output reserve	500–2000
Total budget	4096–8192+

Prioritize: newest > most similar > diverse sources

QUERY TRANSFORMATION TECHNIQUES

```python
# Multi-query: generate variants for broader retrieval
MULTI_QUERY_PROMPT = 'Generate 3 different phrasings of this question: {q}'
variations = llm.generate(MULTI_QUERY_PROMPT.format(q=user_query))
all_results = union([retrieve(v) for v in variations])

# HyDE: hypothetical document embeddings
HYDE_PROMPT = 'Write a passage that would answer: {q}'
hypothetical_doc = llm.generate(HYDE_PROMPT.format(q=user_query))
results = retrieve(hypothetical_doc)  # embed hypothetical, search real docs

# Step-back prompting: abstract the query first
STEPBACK = 'What general concept does this question relate to? {q}'
abstract_query = llm.generate(STEPBACK.format(q=user_query))
results = retrieve(abstract_query + ' ' + user_query)
```

RERANKING & POST-RETRIEVAL PROCESSING

WHY RERANK?

Initial retrieval optimizes for recall. Reranking re-scores top-k results for precision using a cross-encoder (reads query+doc together).

RERANKING MODELS

Model	Notes
Cohere Rerank v3	Best accuracy, API
cross-encoder/ms-marco	OSS, HuggingFace
bge-reranker-large	OSS, top performer
Flashrank (local)	Fast, CPU-friendly

RERANKING PIPELINE

```
import cohere
co = cohere.Client(API_KEY)

# 1. Retrieve broad (top-20)
candidates = retrieve(
    query, top_k=20
)

# 2. Rerank to top-5
reranked = co.rerank(
    query=query,
    documents=candidates,
    top_n=5,
    model="rerank-english-v3.0"
)

# 3. Use reranked results
context = reranked.results
```

RECIPROCAL RANK FUSION (RRF)

```
# RRF: Combine multiple ranked lists without score normalization
def reciprocal_rank_fusion(ranked_lists, k=60):
    scores = defaultdict(float)
    for ranked in ranked_lists:
        for rank, doc_id in enumerate(ranked):
            scores[doc_id] += 1.0 / (k + rank + 1)
    return sorted(scores, key=scores.get, reverse=True)

# Merge dense + sparse + reranked results
final = reciprocal_rank_fusion([dense_ids, sparse_ids, reranked_ids])
```

CONTEXT COMPRESSION

```
# LLMLingua: compress retrieved context to fit more in context window
from llmlingua import PromptCompressor

compressor = PromptCompressor()
compressed = compressor.compress_prompt(
    context_chunks,
    instruction=system_prompt,
    question=user_query,
    target_token=1000,       # compress to 1000 tokens
    ratio=0.5,               # keep 50% of tokens
)

# LangChain contextual compression
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
compressor = LLMChainExtractor.from_llm(llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor, base_retriever=retriever
)
```

RAG FRAMEWORKS & TOOLING

Production-ready stacks

LANGCHAIN RAG

```
from langchain.chains import
    RetrievalQA
from langchain_openai import
    ChatOpenAI, OpenAIEmbeddings
from langchain_community
    .vectorstores import Chroma

# Setup
embedder = OpenAIEmbeddings()
vectordb = Chroma.from_documents(
    documents=chunks,
    embedding=embedder,
)
retriever = vectordb.as_retriever(
    search_type="mmr",
    search_kwargs={'k': 5},
)

# Chain
qa = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model='gpt-4o'),
    retriever=retriever,
    chain_type="stuff",
    return_source_documents=True,
)
result = qa.invoke(user_query)
```

LLAMAINDEX RAG

```
from llama_index.core import (
    VectorStoreIndex,
    SimpleDirectoryReader,
    Settings,
)
from llama_index.llms.openai
    import OpenAI
from llama_index.embeddings.openai
    import OpenAIEmbedding

# Config
Settings.llm = OpenAI('gpt-4o')
Settings.embed_model = \
    OpenAIEmbedding()

# Index
docs = SimpleDirectoryReader(
    './docs').load_data()
index = VectorStoreIndex
    .from_documents(docs)

# Query
engine = index.as_query_engine(
    similarity_top_k=5
)
response = engine.query(q)
```

HAYSTACK & RAGAS

HAYSTACK PIPELINE

```
from haystack import Pipeline
from haystack.components.retrievers
    import InMemoryEmbeddingRetriever
from haystack.components.builders
    import PromptBuilder

pipe = Pipeline()
pipe.add_component('retriever',
    InMemoryEmbeddingRetriever(
        document_store=store))
pipe.add_component('prompt',
    PromptBuilder(template=tmpl))
pipe.add_component('llm', generator)
pipe.connect('retriever', 'prompt')
pipe.connect('prompt', 'llm')
result = pipe.run(query=question)
```

RAGAS EVALUATION

```
from ragas import evaluate
from ragas.metrics import (
    faithfulness,
    answer_relevancy,
    context_precision,
    context_recall,
)

dataset = Dataset.from_dict({
    "question": questions,
    "answer": answers,
    "contexts": retrieved_ctx,
    "ground_truth": gt_answers,
})

result = evaluate(
    dataset,
    metrics=[faithfulness,
        answer_relevancy,
        context_precision]
)
print(result.to_pandas())
```

EVALUATION METRICS

Measure what matters at every stage

RETRIEVAL METRICS

Metric	Formula	Interpretation
Precision@K	Relevant in top-K / K	How many retrieved docs are relevant
Recall@K	Relevant in top-K / Total relevant	How many relevant docs were found
MRR	Mean(1/rank of first relevant)	How high is first relevant result ranked
NDCG@K	Normalized Discounted Cumulative Gain	Ranked quality of results
Hit Rate	Queries with ≥1 relevant in top-K / N	% queries answered correctly

GENERATION METRICS

Metric	Measures	Tool
Faithfulness	Answer grounded in context (no hallucination)	RAGAS, TruEra
Answer Relevancy	Answer addresses the question	RAGAS
Context Precision	Retrieved docs are relevant to question	RAGAS
Context Recall	Context covers ground truth answer	RAGAS
Correctness	Semantic similarity to reference answer	ROUGE, BERTScore
Groundedness	Claims traceable to sources	Azure AI Eval

END-TO-END EVALUATION PIPELINE

```
# Automated RAG evaluation with RAGAS + custom metrics
from ragas import evaluate
from ragas.metrics import faithfulness, answer_relevancy,
    context_precision, context_recall, answer_correctness

def evaluate_rag_system(rag_pipeline, test_dataset):
    results = []
    for sample in test_dataset:
        answer, contexts = rag_pipeline(sample['question'])
        results.append({
            "question":     sample["question"],
            "answer":       answer,
            "contexts":     [c.text for c in contexts],
            "ground_truth": sample["ground_truth"],
        })
    scores = evaluate(
        Dataset.from_list(results),
        metrics=[faithfulness, answer_relevancy,
                 context_precision, context_recall]
    )
    return scores  # DataFrame with per-sample scores
```

ADVANCED RAG PATTERNS

Beyond naive RAG

Pattern	Description	When to Use
Corrective RAG (CRAG)	Evaluate retrieved docs, web-search if irrelevant	Low-confidence retrieval domains
Self-RAG	LLM decides when to retrieve, critiques its output	Reduce unnecessary retrieval
Adaptive RAG	Route queries to different retrieval strategies	Mixed query complexity
Modular RAG	Plug-and-play components (re-rank, compress, rewrite)	Complex enterprise pipelines
Agentic RAG	LLM agent iterates: retrieve → reflect → re-retrieve	Multi-hop reasoning tasks
Graph RAG	Build KG from docs, traverse for multi-hop answers	Relational / entity-heavy data
Multi-modal RAG	Retrieve text + images + tables together	Rich document corpora
Long-context RAG	Use full documents in 128k+ context window	When chunking loses coherence

CACHING STRATEGIES

```
import hashlib, redis
cache = redis.Redis()

def cached_rag(query):
    key = hashlib.md5(
        query.encode()).hexdigest()
    cached = cache.get(key)
    if cached:
        return json.loads(cached)
    result = rag_pipeline(query)
    cache.setex(key, 3600,  # 1hr TTL
        json.dumps(result))
    return result

# Semantic cache: cache by similarity
# Use GPTCache or Momento
```

PERFORMANCE OPTIMIZATION

Optimization	Impact
Batch embed documents	10x faster ingestion
Quantize vectors (int8)	4x smaller index
HNSW index (vs flat)	1000x faster search
Async retrieval	Parallel chunk lookup
Semantic cache	50-80% cache hit rate
Streaming responses	Perceived 3x faster
Prompt caching	Cache system prompt

TROUBLESHOOTING GUIDE

Common RAG failures and fixes

Problem	Symptom	Fix
Poor retrieval	Irrelevant chunks returned	Try hybrid search; tune chunk size; improve embeddings; add m
Hallucination	Answer not in retrieved context	Strengthen system prompt grounding; add faithfulness check; us
Incomplete answers	Misses info that exists in docs	Increase top-k; use multi-query; check chunking splits context
Context overflow	Exceeds context window	Reduce top-k; add reranking; use context compression (LLMLin
Slow latency	Response takes >5s	Cache queries; async retrieval; smaller embedding model; ANN
Stale answers	Old data returned	Add TTL to index; re-embed changed docs; use timestamp meta
Cross-doc reasoning	Can't connect info across docs	Try Graph RAG; increase top-k; add agentic retrieval loop
Noisy chunks	Irrelevant text in passages	Better chunking strategy; metadata-based pre-filtering; MMR ret

QUICK REFERENCE — COMPLETE RAG STACK

```python
# ███ FULL RAG PIPELINE (production-grade) ██████████████████████████████
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_community.vectorstores import Pinecone
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import CohereRerank
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser

# 1. Embeddings + Vector store
embeddings = OpenAIEmbeddings(model='text-embedding-3-large')
vectorstore = Pinecone(index_name='rag-prod', embedding=embeddings)

# 2. Hybrid retriever + reranker
base_retriever = vectorstore.as_retriever(search_kwargs={'k': 20})
reranker = CohereRerank(top_n=5, model='rerank-english-v3.0')
retriever = ContextualCompressionRetriever(
    base_compressor=reranker, base_retriever=base_retriever)

# 3. Prompt + LLM chain
prompt = ChatPromptTemplate.from_messages([
    ("system", SYSTEM_PROMPT),
    ("human", "Context:\n{context}\n\nQuestion: {question}"),
])
llm = ChatOpenAI(model='gpt-4o', temperature=0, streaming=True)

# 4. LCEL chain
rag_chain = (
    {"context": retriever, "question": RunnablePassthrough()}
    | prompt | llm | StrOutputParser()
)
for chunk in rag_chain.stream(user_query):  # streaming
    print(chunk, end='', flush=True)
```

RAG Cheat Sheet • Key Resources: LangChain docs • LlamaIndex docs • RAGAS evaluation • Pinecone learning • Weaviate academy