RAG (Retrieval-Augmented Generation) — Interview & Revision Notes (Corrected Layout)

1. Definition

Retrieval-Augmented Generation (RAG) combines information retrieval from external knowledge sources with generation by a Large Language Model (LLM). It retrieves relevant documents or data (often via a vector database) and supplies them as context to an LLM to produce more accurate, grounded responses.

2. Why RAG is important

- LLMs have limited context window and training cutoff; RAG provides up-to-date or private knowledge.
- Reduces hallucination by grounding outputs in retrieved facts.
- More cost-effective than frequent fine-tuning for knowledge updates.

3. Core Components

Retriever: Finds candidate documents using embeddings + similarity search (vector DB).
Vector Database: Stores document embeddings and metadata (FAISS, Pinecone, Milvus, Weaviate, Qdrant, Chroma).
Embedding Model: Generates vectors for text (OpenAI, SBERT, Instructor, etc.).
Generator (LLM): Uses retrieved context + prompt to generate final answer (GPT, LLaMA, Claude).
Orchestrator / Chain: Connects components (LangChain, LlamaIndex, Haystack, Semantic Kernel).

4. RAG Workflow (step-by-step)

1. User query → embed the query using the same embeddings model used for documents.
2. Search the vector DB for top-k nearest neighbor document chunks (semantic similarity).
3. Optionally re-rank results (cross-encoder or dense + sparse hybrid).
4. Assemble prompt: include retrieved documents (or summaries) + user instruction.
5. Pass assembled prompt to LLM to generate a grounded response.
6. (Optional) Post-process: cite sources, verify facts, or run consistency checks.

5. Types / Patterns of RAG

- Naive RAG: Single retrieval step before generation.
- Iterative RAG: Retrieval and generation alternate; generator's output informs next retrieval.
- Hierarchical RAG: Multi-level retrieval (document → section → paragraph).
- Retrieval + Reader: Retriever finds candidates; Reader (LLM) reads and synthesizes.

6. Implementation Checklist & Best Practices

- Chunking: split documents into coherent chunks (200–800 tokens) with overlap to preserve context.
- Embedding consistency: use same model for document & query embeddings.
- Hybrid search: combine sparse (BM25) + dense (ANN) retrieval for better recall.

- Re-ranking: use a cross-encoder or a smaller model to re-score initial candidates.
- Prompt engineering: limit included tokens; use summaries or metadata to stay within LLM context window.
- Caching: cache recent retrievals and LLM responses to reduce latency and cost.
- Source attribution: include source ids and passages for traceability.

7. Code Examples (short, interview-friendly)

LangChain + FAISS (Python):

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatOpenAI

# Create embeddings and vectorstore
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_texts(["doc1 text", "doc2 text"], embeddings)

# Build retriever and QA chain
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})
llm = ChatOpenAI(model="gpt-3.5-turbo")
qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)
print(qa_chain.run("Explain RAG in simple terms"))
```

Pinecone (conceptual example):

```
# Pinecone query example (conceptual)
import pinecone
from openai import OpenAI

pinecone.init(api_key="PINECONE_KEY", environment="env")
index = pinecone.Index("my-index")

# Embed query then query index
query_embedding = embed_model.embed("what is RAG?")
results = index.query(vector=query_embedding, top_k=3, include_metadata=True)
```

8. Evaluation Metrics & Monitoring

- Top-k Recall / Precision: how many relevant docs are retrieved in top-k.
- Mean Reciprocal Rank (MRR): average ranking quality for queries.
- Latency: retrieval + generation time (SLA considerations).
- Faithfulness / Hallucination Rate: percentage of model outputs supported by sources.
- Cost per query: embedding + retrieval + LLM generation costs.

9. Advantages & Limitations

Advantages:

- Grounded answers with external knowledge.
- Easier to update knowledge (add docs) than retraining.
- Supports private and proprietary data without sharing it with LLM provider.

Limitations:

- Retrieval quality strongly affects final answer.
- Latency and cost overhead vs plain LLM query.
- Context window limits may require aggressive summarization.
- Requires engineering: chunking, index tuning, and monitoring.

10. Interview Q&A; (quick responses)

What is RAG?
Retrieval-Augmented Generation: retrieval of relevant docs to ground LLM generation.

RAG vs Fine-tuning?
RAG is dynamic and cheaper to update; fine-tuning bakes knowledge into model weights but is costly.

How to improve retrieval?
Use better embeddings, hybrid retrieval, re-ranking, and careful chunking.

Common re-ranker?
Cross-encoders (e.g., monoBERT) or small sequence-pair models.

Common embedding models?
all-MiniLM, OpenAI ada/text-embedding-3-small, Instructor, SBERT variants.

One-line summary:

RAG = Retrieval (vector DB) + Generation (LLM) $\rightarrow$ grounded, updatable, and more factual responses.