🗓️ MONTH 5 — WEEK 18 (Advanced RAG + Vector Databases)

✅DAY 1 — “Production-Style RAG” Mindset + Better Chunking + Metadata (Beginner Deep Notes + Code)

Welcome to Week 18.

Week 17 taught you **basic RAG**. Now we upgrade it into a **production-style RAG pipeline**.

🎯 **Day 1 Goal (Very Clear):**

By the end of today, you will understand and implement:

- What “production-style RAG” means
- Why basic RAG fails in real use
- **Advanced chunking strategies** (fixed-size vs semantic-style)
- **Chunk overlap tuning** (how to pick values)
- Adding **metadata** (source, page number, chunk id)
- Building a retrieval output that can show **citations**

✅ Day 2 will add: **ChromaDB + better retrieval strategies**

✅ Day 3 will add: **Re-ranking**

✅ Day 4–5 will add: **RAG evaluation** and comparison with “no RAG”

1️⃣What Does “Production-Style RAG” Mean?

✅ Definition (Easy)

A **production-style RAG** system means:

> A RAG system that gives *reliable answers*, shows sources, works on large document sets, and fails safely.

In real applications (companies, colleges), users will ask:

- unclear questions
- long questions
- tricky questions
- questions that are not in docs

So production RAG must handle all that.

✅ Basic RAG vs Production RAG

Week 17 basic RAG:

- Chunk → embed → FAISS → top-k → prompt → answer

Production RAG adds:

✅ Better chunking

✅ Metadata + citations

✅ Query improvement

✅ Re-ranking

✅ Evaluation (accuracy + faithfulness)

✅ Safety responses (“not found”)

✅ Logging and debugging

🧠 Real-life analogy

Basic RAG = student answers after reading 1 page

Production RAG = student:

- finds the correct chapter
- checks multiple sections
- quotes the book page number
- says “not in book” if missing

🔹 Quick Notes

- Production RAG = reliability + citations + evaluation

- Not just “it works once”, but “works consistently”

2️⃣ Advanced Chunking Strategies (The #1 Upgrade)

If chunking is poor, retrieval fails.

If retrieval fails, LLM will hallucinate or answer incorrectly.

So chunking is the biggest upgrade.

2.1 Fixed-size chunking (what you did before)

✅ Definition

Fixed-size chunking means:

> Break text into equal-size chunks by words/tokens/characters.

✅ Good:

- simple
- works when documents are messy

❌ Bad:

- chunk may mix topics
- not aligned with headings

2.2 Semantic-style chunking (meaning-based)

✅ Definition

Semantic chunking means:

> Split text based on meaning boundaries like headings, paragraphs, sections.

✅ Good:

- each chunk contains one topic

- retrieval is cleaner
- citations become meaningful (“Section 2.3”)

❌ Bad:

- needs structured docs
- harder if PDF text is messy

🔹 **Quick Notes**

- Fixed-size = robust but noisy
- Semantic = clean but depends on document structure

3️⃣ Hands-On: Implement Better Chunking + Metadata

We will build a chunking system that outputs:

✅ chunk text

✅ chunk_id

✅ source name

✅ page number (if PDF)

✅ start/end position (optional)

This metadata later enables citations.

4️⃣ Step-by-step Code (Beginner Friendly)

4.1 Install required libs

pip install pypdf sentence-transformers faiss-cpu numpy

4.2 Import libraries

```
import numpy as np
import re
import faiss

from pypdf import PdfReader
from sentence_transformers import SentenceTransformer
```

Explanation

- numpy → handling vectors
- re → cleaning text
- faiss → vector store
- PdfReader → read PDFs
- SentenceTransformer → embeddings model

5️⃣ Load PDF with Page Numbers (Metadata)

We will load each page separately so we can store:

✅ page number → later used as citation

5.1 PDF Loader that returns page-wise text

```
def load_pdf_pages(pdf_path):
    reader = PdfReader(pdf_path)
    pages = []

    for i, page in enumerate(reader.pages):
        text = page.extract_text()
        if text:
            pages.append({
                "page_number": i + 1,   # human-friendly pages start at 1
                "text": text
            })

    return pages
```

Line-by-line explanation

- PdfReader(pdf_path) opens PDF

- enumerate(reader.pages) gives:
    - i = 0,1,2…
    - page = page object
- extract_text() gets text
- store:
    - page number
    - text
- return list of pages

Test it

```
pages = load_pdf_pages("your_document.pdf")
print("Total pages extracted:", len(pages))
print("Sample page:", pages[0]["page_number"])
print(pages[0]["text"][:500])
```

🔹 Quick Notes

- Page-level loading is essential for citations
- If extraction returns None → PDF might be scanned

---

6 Clean Text (Light Cleaning)

```
def clean_text(text):
    text = text.replace("\u00a0", " ")
    text = re.sub(r"\s+", " ", text)
    return text.strip()
```

Explanation

- removes weird spaces
- compresses multiple spaces/newlines into one space

---

7 Create Chunks WITH METADATA

We will do **fixed-size chunking per page**, and attach metadata.

7.1 Chunking function that returns dict chunks

```
def chunk_page_text(page_text, chunk_size=350, overlap=80):
    words = page_text.split()
    chunks = []
    start = 0

    while start < len(words):
        end = start + chunk_size
        chunk = " ".join(words[start:end])
        chunks.append(chunk)
        start = end - overlap

    return chunks
```

This is fixed-size chunking with overlap.

7.2 Convert pages → chunks with metadata

```
def build_chunks_with_metadata(pages, source_name):
    all_chunks = []
    chunk_id = 0

    for p in pages:
        page_num = p["page_number"]
        text = clean_text(p["text"])

        page_chunks = chunk_page_text(text)

        for c in page_chunks:
            all_chunks.append({
                "chunk_id": chunk_id,
                "source": source_name,
                "page": page_num,
                "text": c
            })
            chunk_id += 1

    return all_chunks
```

Line-by-line explanation

- all_chunks stores all chunk dicts
- chunk_id increments uniquely
- for each page:

        - get page number
        - clean text
        - chunk it
- for each chunk:
    - store metadata:
        - chunk_id
        - source file name
        - page number
        - chunk text

Test it

chunks = build_chunks_with_metadata(pages, source_name="your_document.pdf")
print("Total chunks:", len(chunks))
print(chunks[0])

Expected:


```
{
 "chunk_id": 0,
 "source": "your_document.pdf",
 "page": 1,
 "text": "...."
}
```

🔹 **Quick Notes**

- Metadata enables citations later
- chunk_id helps debugging retrieval

8️⃣ Create Embeddings for Chunks (Using only chunk[“text”])

8.1 Load embedding model

embed_model = SentenceTransformer("all-MiniLM-L6-v2")

8.2 Embed all chunk texts

texts = [c["text"] for c in chunks]

```
embeddings = embed_model.encode(texts, convert_to_numpy=True)
print(embeddings.shape)
```

Explanation

- extract only texts list
- encode creates vectors
- shape = (num_chunks, embedding_dim)

9️⃣ Store Embeddings in FAISS

9.1 Create index

```
dimension = embeddings.shape[1]
index = faiss.IndexFlatL2(dimension)
index.add(embeddings)

print("Stored vectors:", index.ntotal)
```

🔹 **Quick Notes**

- FAISS stores vectors; you keep metadata list separately

🔟 Retrieval Function Returning Chunks + Metadata (Citations Ready)

```
def retrieve_with_metadata(query, top_k=5):
    q_vec = embed_model.encode([query], convert_to_numpy=True)
    distances, indices = index.search(q_vec, top_k)

    results = []
    for rank, idx in enumerate(indices[0]):
        chunk_info = chunks[int(idx)]
        results.append({
            "rank": rank + 1,
            "distance": float(distances[0][rank]),
            "chunk_id": chunk_info["chunk_id"],
            "source": chunk_info["source"],
            "page": chunk_info["page"],
```

```
            "text": chunk_info["text"]
        })
    return results
```

Test retrieval

```
results = retrieve_with_metadata("What is the final exam marks?", top_k=3)

for r in results:
    print("\nRank:", r["rank"])
    print("Source:", r["source"], "Page:", r["page"], "Chunk:", r["chunk_id"])
    print("Distance:", r["distance"])
    print(r["text"][:300])
```

What you achieved

✅ retrieval result now includes:

- exact source file
- page number
- chunk id
  So you can show citations like:

> “Answer based on your_document.pdf page 3”

1️⃣1️⃣ How to Build a “Citation-Friendly Context”

Instead of plain context, we add labels:

```
def build_cited_context(retrieval_results):
    context = ""
    for r in retrieval_results:
        context += f"[Source: {r['source']} | Page: {r['page']} | Chunk: {r['chunk_id']}]\n"
        context += r["text"] + "\n\n"
    return context
```

This makes the LLM see citations and you can display them too.

🔹 Quick Notes

- Citations build trust

- Debugging becomes easy (you know where answer came from)

✅ DAY 1 SUMMARY (Week 18)

Today you upgraded from basic RAG to production-style RAG fundamentals:

✔ Learned what production RAG means

✔ Understood fixed vs semantic chunking

✔ Implemented page-wise PDF loading

✔ Built chunking with overlap per page

✔ Added metadata (source, page, chunk_id)

✔ Stored embeddings in FAISS

✔ Retrieved top-k chunks with citations-ready metadata

🧾 Day 1 Quick Revision Sheet

- Production RAG = reliability + citations + evaluation
- Chunking quality controls retrieval quality
- Add metadata: source + page + chunk id
- Retrieval results should carry metadata for citations
- Rebuild embeddings if chunking changes

✅ Practice Questions (Day 1)

1. What is “production-style RAG” and why is it needed?
2. Difference between fixed-size chunking and semantic chunking.
3. Why is page-wise PDF loading important?
4. What metadata should be stored in vector store retrieval results?
5. Why must you rebuild embeddings after changing chunk size?
6. What does top_k control in retrieval?

DAY 2 — Vector Databases in Practice: ChromaDB + Metadata-Aware Retrieval (Very Detailed Beginner Notes)

Today we move **one big step closer to real-world GenAI systems**.

In **Week 17 + Week 18 Day 1**, you used **FAISS** in a simple way:

- vectors stored separately
- metadata handled manually

Today, you will learn how **real production systems** do this using a **vector database** that stores:

- embeddings
- text
- metadata
- all together

🎯 **Day 2 Goal (Very Clear):**

By the end of today, you will:

- Understand what a **vector database** really is
- Understand **FAISS vs ChromaDB**
- Learn why metadata is critical in real RAG systems
- Use **ChromaDB** to store:
    - embeddings
    - chunk text
    - metadata (source, page, chunk_id)
- Perform **metadata-aware retrieval**
- Build a cleaner, safer retrieval layer for RAG

⚠️ We will **NOT generate answers today**.

Today is about **retrieval quality & data management**.

1️⃣ What Is a Vector Database? (Proper Understanding)

✅ Definition (Very Easy)

A **vector database** is:

> A database designed to store embeddings **along with metadata**, and quickly retrieve the most similar ones.

Unlike FAISS alone, a vector DB:

- stores vectors
- stores text
- stores metadata
- supports filtering
- supports persistence (save to disk)

🧠 Analogy

Think of:

- **FAISS** = fast calculator (just math)
- **ChromaDB** = smart library system
  (books + labels + shelves + search rules)

🔹 Quick Notes

- Vector DB ≠ normal SQL database
- Designed for similarity search
- Core of modern RAG systems

2️⃣ FAISS vs ChromaDB (Conceptual Comparison)

🔸 FAISS

- Library for similarity search
- Stores vectors only
- Metadata handled manually
- Fast and lightweight
- Good for learning & research

🔸 ChromaDB

- Full vector database
- Stores vectors + documents + metadata together
- Supports filtering by metadata
- Persists data on disk
- Better for production-style RAG

Comparison Table (Simple)

Feature	FAISS	ChromaDB
Stores vectors	✅	✅
Stores text	❌ (manual)	✅
Stores metadata	❌ (manual)	✅
Filtering	❌	✅
Persistence	❌	✅
Beginner friendly	Medium	High

🔑 Key Rule

Use FAISS to **learn RAG**
Use ChromaDB to **build RAG systems**

3️⃣ Why Metadata Is CRITICAL in Advanced RAG

✅ Metadata = information

about

the chunk

Examples:

- source file name
- page number
- section name
- chunk id
- document type

Without metadata:

- You don’t know **where the answer came from**
- You can’t show citations
- You can’t filter by document
- Debugging becomes painful

With metadata:

- “Answer based on syllabus.pdf, page 3”
- “Answer only from HR_policy.pdf”
- “Ignore appendix sections”

🔹 Quick Notes

- Metadata = trust + control
- Required for production RAG

4️⃣ Installing & Importing ChromaDB

4.1 Install ChromaDB

pip install chromadb sentence-transformers

4.2 Import required libraries

import chromadb
from sentence_transformers import SentenceTransformer

Explanation

- chromadb → vector database
- SentenceTransformer → embeddings model

5️⃣ Initialize ChromaDB Client (Beginner Friendly)

5.1 Create client

client = chromadb.Client()

Explanation

- Creates an in-memory ChromaDB instance
- Data will live as long as the Python session runs
- Later, we'll add persistence (Week 18 Day 4)

🔹 Quick Notes

- Client = database manager
- Multiple collections can exist

6️⃣ Create a Collection (Like a Table)

✅ What is a collection?

A **collection** is like:

A table that stores embeddings + documents + metadata

6.1 Create a collection

```
collection = client.create_collection(
    name="rag_documents"
)
```

Explanation

- name uniquely identifies this collection
- You can have multiple collections for:
    - different projects
    - different document sets

7️⃣ Load Embedding Model (Same as Before)

```
embed_model = SentenceTransformer("all-MiniLM-L6-v2")
```

Explanation

- Fast
- Optimized for semantic similarity
- Standard for RAG

8️⃣ Prepare Data for ChromaDB

Assume from **Week 18 Day 1**, you already have:

```
chunks = [
  {
    "chunk_id": 0,
    "source": "syllabus.pdf",
    "page": 1,
    "text": "Attendance must be at least 75%..."
  },
  ...
]
```

8.1 Extract texts, metadata, and IDs

```
documents = [c["text"] for c in chunks]

metadatas = [
    {
        "source": c["source"],
        "page": c["page"],
        "chunk_id": c["chunk_id"]
    }
    for c in chunks
]

ids = [str(c["chunk_id"]) for c in chunks]
```

Line-by-line explanation

- documents → actual chunk text
- metadatas → dictionary per chunk
- ids → unique string ID (required by Chroma)

🔹 Important Rule

- IDs **must be strings**
- IDs must be **unique**

---

9 Add Data to ChromaDB Collection

```
collection.add(
    documents=documents,
    metadatas=metadatas,
    ids=ids
)
```

Explanation

- Chroma automatically:
    - embeds documents
    - stores vectors
    - stores metadata
    - links everything together

⚠️ You no longer manage embeddings manually here.

🔹 **Quick Notes**

- Chroma handles embedding internally
- Cleaner and safer than FAISS-only approach

🔟 Querying ChromaDB (Basic Retrieval)

10.1 Perform a similarity search

```
query = "What is the attendance requirement?"

results = collection.query(
    query_texts=[query],
    n_results=3
)
```

What does

results

contain?

```
results.keys()
```

Outputs:

- documents
- metadatas
- ids
- distances

10.2 Inspect results

```
for i in range(len(results["documents"][0])):
    print("\nResult", i+1)
    print("Text:", results["documents"][0][i][:200])
    print("Metadata:", results["metadatas"][0][i])
    print("Distance:", results["distances"][0][i])
```

Explanation

- documents[0][i] → retrieved chunk text
- metadatas[0][i] → source, page, chunk_id
- distances → similarity score

🔹 Quick Notes

- Lower distance = better match (default)
- Everything is aligned by index

1️⃣1️⃣ Metadata-Aware Retrieval (Major Upgrade)

Now we filter retrieval by metadata.

11.1 Example: Retrieve only from a specific source

```
results = collection.query(
    query_texts=["What is the final exam marks?"],
    n_results=3,
    where={"source": "syllabus.pdf"}
)
```

Explanation

- where applies metadata filter
- Only chunks from syllabus.pdf are considered

11.2 Example: Filter by page number

```
results = collection.query(
    query_texts=["attendance"],
    n_results=5,
    where={"page": 2}
```

)

Why this is powerful

- Multi-document RAG
- Control which documents can answer
- Avoid irrelevant sources

🔹 Quick Notes

- Metadata filtering = control + safety
- Essential in enterprise systems

1️⃣2️⃣ Compare FAISS vs Chroma Retrieval (Understanding)

FAISS style:

- retrieve vector indices
- manually map to text + metadata

Chroma style:

- retrieve text + metadata directly
- cleaner
- fewer bugs

🔑 Professional Insight

FAISS is a **tool**
ChromaDB is a **system**

1️⃣3️⃣ Preparing Retrieval Output for RAG Prompt

We now format retrieval results for later generation.

```
def build_context_from_chroma(results):
    context = ""
    docs = results["documents"][0]
    metas = results["metadatas"][0]

    for doc, meta in zip(docs, metas):
        context += f"[Source: {meta['source']} | Page: {meta['page']}]\n"
        context += doc + "\n\n"

    return context
```

Explanation

- loops through retrieved docs
- attaches source & page
- builds citation-ready context

1️⃣4️⃣ Common Beginner Mistakes (Read Carefully)

❌ Forgetting to store metadata

❌ Using non-unique IDs

❌ Mixing embeddings from different chunking strategies

❌ Not filtering when multiple sources exist

❌ Assuming top-1 is always enough

✅ DAY 2 SUMMARY (Week 18)

Today you learned:

✔ What a vector database really is

✔ Why ChromaDB is better for production RAG

✔ How to store documents + embeddings + metadata together

✔ How to query with metadata filters

✔ How to prepare citation-friendly retrieval results

✔ Difference between FAISS and ChromaDB in real systems

DAY 3 — Re-ranking: Query → Retrieve → Re-rank → (Better Context for Answers)

(Very detailed beginner notes + code explained line-by-line. No practice questions.)

Today you'll learn the **single most powerful upgrade** in RAG quality:

> **Re-ranking** = after retrieval, we re-check the top results with a smarter model so the final context is more accurate.

This is how production RAG systems stop returning "kind of related" chunks and start returning "exact answer" chunks.

1️⃣ Why Basic Retrieval is Not Enough

✅ What you currently do (basic retrieval)

- Convert query to embedding
- Find nearest chunks in vector DB (FAISS/Chroma)
- Take top-k chunks
- Use them as context

This is good, but it often fails in real documents.

❌ What goes wrong with pure embedding retrieval?

Problem A:

“Similar topic” but not “exact answer”

Embedding search is good at meaning similarity, but it might retrieve:

- general chunk about exams
- instead of the chunk with the exact marks breakdown

Problem B:

Long chunks cause noise

A chunk can mention “exam”, “marks”, “grading”, but the specific answer line may be buried.

Problem C:

Query wording mismatch

User may phrase differently:

- “How much is final exam worth?”
- Document says: “Final exam carries 60 marks”
  Embeddings might retrieve correct region, but not always best.

🧠 Analogy (very simple)

Imagine you search in a library:

- **Retriever (embeddings)** = librarian who brings 10 books related to your topic
- **Re-ranker** = teacher who opens those books and selects the 3 pages that actually answer your exact question

Retriever is fast. Re-ranker is precise.

2️⃣ What is Re-ranking?

✅ Definition (easy)

Re-ranking means:

> Take the top results from vector search (say top 20), then score them again with a more accurate model, and reorder them so the best chunks come first.

So your final top-k context becomes higher quality.

3️⃣ Retriever vs Re-ranker (Clear Comparison)

✅ Retriever (bi-encoder / embeddings)

- Encodes query and chunk **separately**
- Compares vectors
- Very fast
- Best for searching large data

✅ Re-ranker (cross-encoder)

- Reads **query + chunk together** at the same time
- Gives a relevance score
- Slower, but much more accurate
- Best for final selection (top 3–5 chunks)

4️⃣ Types of Re-ranking (Beginner view)

✅ 4.1 Rule-based re-ranking (simple)

- Add keyword matching bonus
- Useful but limited

✅ 4.2 ML re-ranking (recommended)

Use a **CrossEncoder** model (transformer) trained to score query–passage relevance.

We will use this.

5️⃣ Step-by-step: Implement Re-ranking (Hands-on)

We'll build this pipeline:

✅ **Query → Chroma retrieve top 20 → CrossEncoder re-rank → final top 5**

5.1 Install required libraries

pip install chromadb sentence-transformers

(You already installed in Day 2, but just in case.)

5.2 Imports

import chromadb
from sentence_transformers import SentenceTransformer, CrossEncoder

Explanation

- SentenceTransformer is for embedding retrieval (if needed)
- CrossEncoder is the re-ranking model (query + chunk pair scoring)

6️⃣ Setup: Chroma Collection (Same as Day 2)

If you already have your Day 2 setup, you'll have:

- client
- collection
- chunks stored with metadata (source, page, chunk_id)

Example:

client = chromadb.Client()
collection = client.create_collection(name="rag_documents")

And you did:

collection.add(documents=..., metadatas=..., ids=...)

If you already have it, you can continue.

7️⃣ Load the Re-ranker Model

✅ Which re-ranker should beginners use?

A popular one is:

- cross-encoder/ms-marco-MiniLM-L-6-v2

It's small and good for ranking passages.

Code

reranker = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")

Explanation

- Loads a transformer trained for ranking
- Input: (query, chunk_text)
- Output: relevance score (higher = more relevant)

🔹 Quick Notes

- Retriever finds "candidate chunks"
- Re-ranker decides which candidates are best

8️⃣ Step 1: Retrieve Candidates from Chroma

We'll retrieve more than we need (like 20), because re-ranking will reduce it.

```
def retrieve_candidates(query, n_candidates=20, where=None):
    results = collection.query(
        query_texts=[query],
        n_results=n_candidates,
        where=where  # optional metadata filter
    )
    return results
```

Line-by-line explanation

- query_texts=[query] → Chroma expects a list

- n_results=n_candidates → pull top 20 initially
- where=... → optional filter by source/page/etc.

9️⃣ Step 2: Convert Retrieval Output into a List of Candidate Items

Chroma returns nested lists like:

- results["documents"][0]
- results["metadatas"][0]
- results["ids"][0]

We convert it into a clean list of dicts:

```
def parse_chroma_results(results):
    docs = results["documents"][0]
    metas = results["metadatas"][0]
    ids = results["ids"][0]

    candidates = []
    for doc, meta, _id in zip(docs, metas, ids):
        candidates.append({
            "id": _id,
            "text": doc,
            "meta": meta
        })
    return candidates
```

Explanation

- zip(docs, metas, ids) keeps alignment
- Each candidate has:
    - text
    - metadata (source/page/chunk_id)
    - id

🔟 Step 3: Re-rank Candidates using CrossEncoder

✅ Key idea

CrossEncoder needs pairs:

- (query, chunk_text)

It outputs a score per pair.

```
def rerank(query, candidates, top_k=5):
    pairs = [(query, c["text"]) for c in candidates]
    scores = reranker.predict(pairs)

    # attach scores
    for c, s in zip(candidates, scores):
        c["rerank_score"] = float(s)

    # sort by rerank_score (descending)
    candidates = sorted(candidates, key=lambda x: x["rerank_score"], reverse=True)

    return candidates[:top_k]
```

Line-by-line explanation

pairs = [(query, c["text"]) for c in candidates]

- Creates list like:
    - (“What is attendance?”, “Attendance must be 75%…”)
    - (“What is attendance?”, “Exam marks are…”)

scores = reranker.predict(pairs)

- Returns numeric scores
- Higher score = more relevant

Attach score:

- adds rerank_score to each candidate

Sort descending:

- best scored chunk becomes #1

Return top_k:

- final best chunks (like 5)

🔹 **Quick Notes**

- We retrieve top 20 fast
- Then re-rank precisely
- Final top 5 is much better than raw top 5

1️⃣1️⃣ Full Pipeline Function: Retrieve → Re-rank

```
def retrieve_then_rerank(query, n_candidates=20, top_k=5, where=None):
    chroma_results = retrieve_candidates(query, n_candidates=n_candidates, where=where)
    candidates = parse_chroma_results(chroma_results)
    best = rerank(query, candidates, top_k=top_k)
    return best
```

Explanation

- Step 1: get candidates from Chroma
- Step 2: parse into clean list
- Step 3: rerank and return best

1️⃣2️⃣ Test Re-ranking (See the Difference)

```
query = "How many marks is the final exam worth?"

best_chunks = retrieve_then_rerank(query, n_candidates=20, top_k=5)

for i, c in enumerate(best_chunks, start=1):
    meta = c["meta"]
    print(f"\nRank {i}")
    print("Score:", c["rerank_score"])
    print("Source:", meta.get("source"), "| Page:", meta.get("page"), "| Chunk:",
meta.get("chunk_id"))
    print(c["text"][:300])
```

✅ Now you should see:

- the chunk containing “final exam is worth 60 marks” ranked higher.

1️⃣3️⃣ Build Citation-Friendly Context from Re-ranked Chunks

Now that re-ranked chunks are best, we format them for the LLM:

```
def build_context_from_reranked(best_chunks):
    context = ""
    for c in best_chunks:
        meta = c["meta"]
        context += f"[Source: {meta.get('source')} | Page: {meta.get('page')} | Chunk:
{meta.get('chunk_id')}]\n"
        context += c["text"] + "\n\n"
    return context
```

Explanation

- Adds citations
- Builds clean context
- This context is now “high precision”

1️⃣4️⃣ Where Re-ranking Fits in the Final RAG System

Your Week 18 “production-style” pipeline becomes:

```
User Question
  ↓
Chroma retrieve top 20 (fast, broad)
  ↓
CrossEncoder rerank to top 5 (slow, accurate)
  ↓
Build cited context
  ↓
LLM answers using context only
```

This is exactly what many real RAG stacks do.

1️⃣5️⃣ Performance Tips (Very Important in Real Systems)

Re-ranking is slower than embeddings. So:

✅ Retrieve 20–50 candidates

✅ Re-rank only those

✅ Use top 3–5 for context

Practical recommended values (beginner default)

- n_candidates = 20
- top_k = 5

If docs are huge:

- n_candidates = 50 (but slower)

1️⃣6️⃣ Common Beginner Mistakes (and fixes)

❌ Mistake 1: Re-ranking everything

If you re-rank 10,000 chunks, it will be extremely slow.

✅ Fix: re-rank only top candidates (20–50)

❌ Mistake 2: Not using metadata

Then you cannot cite sources or debug.

✅ Fix: store metadata in Chroma and keep it throughout.

❌ Mistake 3: Using too many chunks in context

Even if reranked, too many chunks can confuse the LLM.

✅ Fix: use 3–5 best chunks only.

✅ DAY 3 SUMMARY (Week 18)

Today you upgraded retrieval quality using **re-ranking**:

- Understood why embedding retrieval alone can be noisy
- Learned what re-ranking means and why it improves accuracy
- Implemented a **CrossEncoder re-ranker**
- Built a clean pipeline: **retrieve candidates → rerank → keep best**
- Prepared citation-friendly context from reranked results
- Learned performance best practices for production RAG

DAY 4 — Make RAG "Production-Like": Persistence, Better Metadata, Logging & Debugging (Very Detailed + Code)

(Beginner-friendly, step-by-step, no practice questions.)

So far in Week 18:

- **Day 1:** Better chunking + metadata (source/page/chunk_id)
- **Day 2:** ChromaDB storing docs + metadata + retrieval with filters
- **Day 3:** Re-ranking (retrieve top-N → rerank → keep best)

Today is about turning your system into something you can actually **reuse and trust**.

🎯 **Day 4 Goal (Very Clear):**

By the end of today you will:

1. Make your vector database **persistent** (saved to disk)
2. Improve metadata beyond page numbers (like **section titles**)
3. Add **logging** (store what was retrieved and why)
4. Add a clean **debugging workflow**
5. Create a small **evaluation dataset** (questions + expected answers) for Day 5

We still focus mostly on retrieval layer and system engineering.
Day 5 is where we do evaluation + compare "with RAG vs without RAG".

1️⃣ What Does “Persistence” Mean?

✅ Definition (easy)

Persistence means:

> Saving your vector database (embeddings + docs + metadata) to disk so it stays available even after you close Python.

Without persistence:

- every time you run your notebook, you must re-embed everything
- slow and annoying
- not realistic

With persistence:

- you embed once
- reuse forever (until docs change)

🧠 Analogy

- Non-persistent DB = writing notes on a whiteboard (gone later)
- Persistent DB = saving notes in a notebook (always there)

2️⃣ Persistent ChromaDB Setup (Step-by-step)

Chroma supports persistence using a folder on disk.

2.1 Install libraries (if needed)

pip install chromadb sentence-transformers pypdf numpy

2.2 Create a persistent client

```
import chromadb

client = chromadb.PersistentClient(path="./chroma_store")
```

Line-by-line explanation

- PersistentClient(...) creates a Chroma DB that saves data in a folder
- path="./chroma_store" means:
    - create a folder named chroma_store
    - store DB files there

✅ Now even if you restart notebook, DB is still there.

2.3 Create or load a collection safely

When running multiple times, you don't want "already exists" errors.

```
collection = client.get_or_create_collection(name="rag_documents")
```

Explanation

- If collection exists → loads it
- If not → creates it

🔹 Quick Notes

- PersistentClient = real "database-like" behavior
- get_or_create_collection avoids repeated creation errors

3️⃣ When to Rebuild the Database (Very Important Rule)

You MUST rebuild embeddings + vector store when:

- you changed chunking strategy
- you changed chunk size/overlap
- you added/removed documents
- you used a different embedding model

✅ Keep a “version” in your project folder:

- chunk_size=350_overlap=80_model=MiniLM

4️⃣ Better Metadata: Add “Section Title” (Beyond Page Number)

Page numbers are helpful, but real citations often need:

- section name
- heading
- topic label

Even if PDFs are messy, we can add section-like metadata using simple heuristics.

4.1 What is a “section title”?

✅ Definition

A section title is:

> A label that tells what this chunk is about.

Example:

- “Course Policy”
- “Assessment”
- “Week Schedule”

This helps:

- filtering
- better citations
- understanding retrieval results quickly

4.2 Simple heuristic: detect headings

Many documents have lines like:

- “Course Policy:”

- “Assessment:”
- “Unit 3 — CNNs”

We can detect headings by rules like:

- ends with “:”
- all caps
- short line length

Code: Find possible heading lines

```
import re

def detect_heading(line):
    line = line.strip()
    if len(line) == 0:
        return False

    # Rule 1: ends with colon
    if line.endswith(":") and len(line) < 60:
        return True

    # Rule 2: looks like ALL CAPS heading
    if line.isupper() and len(line) < 60:
        return True

    # Rule 3: "Unit 3", "Section 2.1" style
    if re.match(r"^(Unit|Section|Chapter)\s+\d+", line):
        return True

    return False
```

Explanation

- strips spaces
- checks patterns
- returns True if it is likely a heading

4.3 Assign section title while chunking page text

We will chunk page text but also track the last heading seen.

Code: Chunk with section metadata

```
def chunk_with_sections(page_text, chunk_size=350, overlap=80):
```

```
    lines = page_text.split("\n")
    current_section = "Unknown"

    # Build one big cleaned text but track headings
    cleaned_lines = []
    for line in lines:
        if detect_heading(line):
            current_section = line.strip().replace(":", "")
        cleaned_lines.append(line.strip())

    full_text = " ".join([l for l in cleaned_lines if l])
    words = full_text.split()

    chunks = []
    start = 0
    while start < len(words):
        end = start + chunk_size
        chunk = " ".join(words[start:end])

        chunks.append({
            "text": chunk,
            "section": current_section
        })

        start = end - overlap

    return chunks
```

Line-by-line explanation

- split page into lines
- if line is heading → update current_section
- join lines into one text
- chunk it by words
- each chunk stores:
    - chunk text
    - section name

⚠️ This is heuristic (not perfect), but good for beginner production-like behavior.

5️⃣ Build Final “Chunks with Metadata” (source, page, section, chunk_id)

Assume we have pages from PDF like Day 1.

Load pages (from Week 18 Day 1)

```
from pypdf import PdfReader

def load_pdf_pages(pdf_path):
    reader = PdfReader(pdf_path)
    pages = []
    for i, page in enumerate(reader.pages):
        text = page.extract_text()
        if text:
            pages.append({"page_number": i+1, "text": text})
    return pages
```

Build chunks with metadata

```
def build_chunks(pages, source_name):
    all_chunks = []
    chunk_id = 0

    for p in pages:
        page_num = p["page_number"]
        page_text = p["text"]

        page_chunks = chunk_with_sections(page_text)

        for ch in page_chunks:
            all_chunks.append({
                "id": str(chunk_id),          # Chroma requires string ids
                "text": ch["text"],
                "metadata": {
                    "source": source_name,
                    "page": page_num,
                    "section": ch["section"],
                    "chunk_id": chunk_id
                }
            })
            chunk_id += 1

    return all_chunks
```

Explanation

- each chunk has:
    - unique string id
    - text
    - metadata dict

- metadata now includes section label too

6️⃣ Add Data into Persistent ChromaDB

6.1 Embed model choice (Important)

Even when Chroma auto-embeds, production systems often control embedding.

But to keep it beginner-friendly, we will let Chroma embed internally OR you can embed externally.

For simplicity, we'll store documents and let Chroma handle embeddings.

6.2 Add chunks in batches (important for large docs)

Adding everything at once can be heavy. Batch insertion is safer.

```
def add_to_chroma(collection, chunk_items, batch_size=100):
    for i in range(0, len(chunk_items), batch_size):
        batch = chunk_items[i:i+batch_size]

        collection.add(
            documents=[b["text"] for b in batch],
            metadatas=[b["metadata"] for b in batch],
            ids=[b["id"] for b in batch],
        )
```

Explanation

- loops in chunks of 100
- adds to Chroma
- safe for memory

Use it

```
pages = load_pdf_pages("your_document.pdf")
chunk_items = build_chunks(pages, source_name="your_document.pdf")
add_to_chroma(collection, chunk_items)
```

✅ Now the database is saved inside ./chroma_store.

7️⃣ Logging: Make Your System Debuggable (Production Habit)

If your RAG gives a wrong answer, you must know:

- which chunks were retrieved
- what their metadata was
- their re-rank scores (if used)

That's logging.

7.1 What is logging?

✅ Definition

Logging means:

> saving important internal steps of your pipeline so you can debug later.

Examples to log:

- query text
- retrieved chunks
- distances
- metadata
- reranker scores
- final chosen context

7.2 Simple logging to a JSON file

```
import json
from datetime import datetime

def log_event(data, log_path="rag_logs.jsonl"):
    data["timestamp"] = datetime.now().isoformat()

    with open(log_path, "a", encoding="utf-8") as f:
        f.write(json.dumps(data, ensure_ascii=False) + "\n")
```

Explanation

- jsonl = JSON lines (one JSON per line)
- easy to append
- easy to read later

---

8 Retrieval Function with Filters + Logging

Now we build a retrieval function that:

- retrieves top-k
- supports metadata filter
- logs results

```
def chroma_retrieve(query, top_k=5, where=None, log=True):
    results = collection.query(
        query_texts=[query],
        n_results=top_k,
        where=where
    )

    docs = results["documents"][0]
    metas = results["metadatas"][0]
    dists = results["distances"][0]
    ids = results["ids"][0]

    retrieved = []
    for doc, meta, dist, _id in zip(docs, metas, dists, ids):
        retrieved.append({
            "id": _id,
            "distance": float(dist),
            "metadata": meta,
            "text_preview": doc[:250]
        })

    if log:
        log_event({
            "type": "retrieval",
            "query": query,
            "top_k": top_k,
            "where": where,
            "results": retrieved
```

```
    })

  return results
```

Explanation (key points)

- where allows filtering (source/page/section)
- we log:
    - query
    - filters
    - results preview
- preview keeps log light (not full chunk text)

9️⃣ Debugging Workflow (What to do when RAG fails)

This is the exact sequence professionals use.

✅ Step 1: Check retrieval quality (without LLM)

- If retrieved chunks don't contain answer → retrieval problem

✅ Step 2: Try filter by source or section

- wrong source might be dominating

✅ Step 3: Increase top_k

- answer may be in top 10 but not top 5

✅ Step 4: Re-chunk or tune overlap

- answer line might be split incorrectly

✅ Step 5: Add re-ranking (Day 3)

- if retrieval gives "topic-related" chunks, reranking selects "answer-rich" chunks

---

🔟 Create Evaluation Set Starter (For Day 5)

To evaluate, we need a small set of Q&A.

10.1 What is an evaluation set?

✅ Definition

Evaluation set means:

> a list of questions where you already know what the correct answer should be (from the document).

Even 10 questions is enough for Week 18.

10.2 Create an evaluation list

```
eval_set = [
    {
        "question": "What is the attendance requirement?",
        "expected": "75%"
    },
    {
        "question": "How many marks is the final exam worth?",
        "expected": "60"
    }
]
```

Explanation

- expected can be:
    - exact phrase
    - key number
    - short keyword
      This helps check if retrieved chunks contain the answer.

10.3 Quick check: does retrieval contain expected keyword?

```
def check_retrieval_contains_answer(question, expected, top_k=5):
```

```
    results = collection.query(query_texts=[question], n_results=top_k)
    docs = results["documents"][0]

    combined = " ".join(docs).lower()
    return expected.lower() in combined
```

This is a simple retrieval evaluation technique.

✅ DAY 4 SUMMARY (Week 18)

Today you made your RAG system far more production-like:

✅ **Persistence**: ChromaDB saved to disk

✅ **Better metadata**: added section labels + source/page/chunk_id

✅ **Batch insertion**: safe for big docs

✅ **Logging**: saves retrieval info for debugging

✅ **Debug workflow**: clear steps to diagnose failures

✅ **Evaluation starter set**: questions + expected keywords ready for Day 5

DAY 5 — Evaluating RAG: Accuracy, Relevance, Faithfulness & "With RAG vs Without RAG" (Very Detailed, Beginner-Friendly)

(No practice questions included, as requested.)

Today you **finish Week 18** by learning how professionals **measure whether RAG is actually helping**.

Up to now, you built:

- Better chunking + metadata
- Vector DB (ChromaDB) with persistence
- Re-ranking for precision
- Logging & debugging

🎯 **Day 5 Goal (Very Clear):**

By the end of today, you will be able to:

1. Understand **what "RAG evaluation" really means**
2. Define and measure:
    - **Relevance**
    - **Faithfulness (groundedness)**
    - **Answer accuracy**
    - **Hallucination rate**
3. Compare:
    - **LLM without RAG**
    - **LLM with RAG**
4. Build:
    - An **evaluation table**
    - A **short evaluation report** (deliverable)

This is what turns a demo into a **credible ML system**.

1️⃣ Why We Must Evaluate RAG (Very Important)

❌ Common beginner mistake

> "The answer looks correct, so RAG is working."

This is **not enough**.

LLMs can:

- sound confident
- be partially correct
- hallucinate details

Evaluation answers:

- Is the answer **relevant**?
- Is it **supported by documents**?
- Is RAG **better than no RAG**?

🧠 Analogy

A student answers a question:

- Sounds fluent ❌
- But quotes the wrong book ❌

Evaluation checks:

- Did they use the right book?
- Did they use the right page?
- Did they answer the right question?

2️⃣ What Does "RAG Evaluation" Mean?

✅ Definition (easy)

RAG evaluation means:

> Checking whether answers are correct, grounded in retrieved documents, and better than answers without retrieval.

Evaluation is NOT just accuracy.

It has multiple dimensions.

3️⃣ Core Evaluation Metrics (Beginner-Friendly)

We'll use **4 simple but powerful metrics**.

3.1 Relevance

✅ Definition

Relevance means:

> Does the answer actually address the user's question?

Example:

- Question: "What is the attendance requirement?"

- Answer: “The final exam is 60 marks”

❌ Not relevant

How we check relevance (simple)

- Manually inspect answer
- Or check if key terms from question appear in answer

3.2 Faithfulness (Groundedness)

✅ Definition (very important)

Faithfulness means:

> Is the answer fully supported by the retrieved context?

If the answer contains info **not present** in retrieved chunks → ❌ unfaithful (hallucination).

Example

Context says:

> “Attendance must be at least 75%”

Answer says:

> “Attendance must be 75% and medical leave is allowed”

❌ “medical leave” is hallucinated → not faithful

🔑 Key Rule

> Faithfulness is the MOST important RAG metric.

3.3 Answer Accuracy

✅ Definition

Accuracy means:

> Is the answer factually correct according to the document?

Example:

- Document: “Final exam is worth 60 marks”
- Answer: “Final exam is worth 50 marks”

❌ inaccurate

Difference from relevance

- Relevant ≠ accurate
- An answer can be relevant but wrong

3.4 Hallucination Rate

✅ Definition

Hallucination rate means:

> How often the model invents information not found in the document.

Lower is better.

4️⃣ Evaluation Setup (Step-by-step)

We’ll evaluate **two systems**:

1. ❌ **LLM without RAG**
2. ✅ **LLM with RAG**

Using the same questions.

5️⃣ Prepare Evaluation Questions (Ground Truth)

You already created a small evaluation set on Day 4.

We'll reuse and expand it slightly.

```
eval_set = [
    {
        "question": "What is the attendance requirement?",
        "expected_keywords": ["75"]
    },
    {
        "question": "How many marks is the final exam worth?",
        "expected_keywords": ["60"]
    },
    {
        "question": "What are the components of assessment?",
        "expected_keywords": ["exam", "assignment", "internal"]
    }
]
```

Explanation

- expected_keywords are **signals**, not full answers
- We use them to check accuracy & relevance

---

6 Baseline: Answering WITHOUT RAG

This simulates a user asking the LLM **without providing documents**.

6.1 Simple LLM-only function

```
from transformers import pipeline

llm = pipeline(
    "text2text-generation",
    model="google/flan-t5-base",
    max_new_tokens=150
)

def answer_without_rag(question):
    response = llm(question)[0]["generated_text"]
    return response
```

Explanation

- LLM only sees the question

- Uses its training knowledge
- Likely to hallucinate or guess

7️⃣ RAG Answering Function (With Retrieval)

We reuse what you built earlier:

- retrieve (Chroma)
- optional re-rank
- build context
- strict prompt

7.1 Build strict RAG prompt

```
def answer_with_rag(question, top_k=5):
    results = collection.query(
        query_texts=[question],
        n_results=top_k
    )

    docs = results["documents"][0]
    metas = results["metadatas"][0]

    context = ""
    for doc, meta in zip(docs, metas):
        context += f"[Source: {meta['source']} | Page: {meta['page']}]\n"
        context += doc + "\n\n"

    prompt = f"""
You are a helpful assistant.
Answer the question ONLY using the context below.
If the answer is not present, say "I don't know."

Context:
{context}

Question:
{question}

Answer:
```

"""

```
    response = llm(prompt)[0]["generated_text"]
    return response, context
```

Explanation

- Retrieval provides evidence
- Prompt restricts hallucination
- Context allows faithfulness checking

8️⃣ Automatic Checks (Beginner-Friendly Evaluation Logic)

We now build **simple automated checks**.

8.1 Check relevance + accuracy using keywords

```
def contains_expected(answer, expected_keywords):
    answer_lower = answer.lower()
    return all(k.lower() in answer_lower for k in expected_keywords)
```

Explanation

- Converts answer to lowercase
- Checks if expected keywords appear

⚠️ This is simple but effective for learning.

8.2 Check faithfulness (context containment)

```
def is_faithful(answer, context):
    answer_words = answer.lower().split()
    context_text = context.lower()

    unsupported = [
        w for w in answer_words
        if w.isalpha() and w not in context_text
    ]
```

```
    # If too many unsupported words → likely hallucination
    return len(unsupported) < 10
```

Explanation

- Very rough heuristic
- If answer introduces many words not in context → suspicious
- Real systems use advanced faithfulness models, but this is good for Week 18

9️⃣ Run Evaluation: With RAG vs Without RAG

Now we evaluate both systems side-by-side.

9.1 Evaluation loop

```
evaluation_results = []

for item in eval_set:
    q = item["question"]
    expected = item["expected_keywords"]

    no_rag_answer = answer_without_rag(q)
    rag_answer, rag_context = answer_with_rag(q)

    evaluation_results.append({
        "question": q,
        "no_rag_answer": no_rag_answer,
        "rag_answer": rag_answer,
        "no_rag_correct": contains_expected(no_rag_answer, expected),
        "rag_correct": contains_expected(rag_answer, expected),
        "rag_faithful": is_faithful(rag_answer, rag_context)
    })
```

What this produces

For each question:

- answer without RAG

- answer with RAG
- correctness comparison
- faithfulness check

🔟 Build the Evaluation Table (Deliverable)

```
import pandas as pd

df = pd.DataFrame(evaluation_results)
df
```

Expected columns:

- question
- no_rag_answer
- rag_answer
- no_rag_correct
- rag_correct
- rag_faithful

Typical observation

- ❌ Without RAG: confident but wrong
- ✅ With RAG: correct and grounded
- Faithfulness improves drastically

1️⃣1️⃣ Calculate Metrics (Simple Summary)

```
summary = {
    "No-RAG Accuracy": df["no_rag_correct"].mean(),
    "RAG Accuracy": df["rag_correct"].mean(),
    "RAG Faithfulness": df["rag_faithful"].mean()
}

summary
```

Interpretation

- Values closer to 1.0 are better
- You should see:
    - RAG accuracy > No-RAG accuracy
    - High faithfulness for RAG

1️⃣2️⃣ Hallucination Rate (Simple Estimation)

```
hallucination_rate = 1 - df["rag_faithful"].mean()
hallucination_rate
```

Lower hallucination rate = better RAG system.

1️⃣3️⃣ Interpreting Results (How to Think Like an Engineer)

Case 1: RAG accuracy low

- Retrieval failed
- Chunking needs improvement
- Re-ranking may help

Case 2: RAG correct but not faithful

- Prompt too weak
- Too much context
- Not strict enough “ONLY use context”

Case 3: No-RAG accuracy high

- Question is generic
- RAG may not be needed for such questions

1️⃣4️⃣ Write the Short Evaluation Report (Deliverable)

Your **Week 18 report (1–2 pages)** should include:

📄 Section 1: Setup

- Documents used
- Chunking strategy
- Vector DB (ChromaDB)
- Re-ranking used or not

📄 Section 2: Evaluation Method

- Number of questions
- Metrics used:
  - relevance
  - accuracy
  - faithfulness
- Why these metrics matter

📄 Section 3: Results Table

- With RAG vs Without RAG
- Observations

📄 Section 4: Key Findings

Example:

- RAG improves factual accuracy
- Hallucination rate reduced
- Retrieval quality directly impacts answer quality

📄 Section 5: Limitations

- Small evaluation set
- Simple faithfulness heuristic
- No human evaluation

1️⃣5️⃣ What You Have Achieved by End of Week 18

🎉 **This is a big milestone.**

You can now:

- Build **production-style RAG**
- Store embeddings persistently
- Retrieve with metadata
- Re-rank for accuracy
- Log and debug
- Evaluate RAG quality
- Prove that RAG works (not just claim it)

This puts you **ahead of most beginners**.

🧠 WEEK 18 — PRACTICE QUESTIONS

Advanced RAG + Vector Databases

📅 DAY 1 — Production-Style RAG, Advanced Chunking & Metadata

🔹 Very Short Answer

1. What does “production-style RAG” mean?
2. Why does basic RAG fail in real-world systems?
3. What is metadata in RAG?
4. Name two examples of metadata stored with chunks.
5. Why are page numbers useful in RAG systems?

🔹 Short Answer

6. Explain fixed-size chunking and one of its drawbacks.
7. Explain semantic chunking and when it works best.
8. Why must embeddings be regenerated after changing chunk size or overlap?
9. How does metadata help in debugging RAG systems?
10. Why are citations important in enterprise RAG systems?

🔹 Conceptual / Thinking Questions

11. Why is chunking considered the most important factor for RAG quality?
12. What problems occur if chunks contain multiple unrelated topics?
13. How does adding section-level metadata improve trust in answers?

📅

DAY 2 — Vector Databases & ChromaDB

🔹 Very Short Answer

1. What is a vector database?
2. How is a vector database different from a traditional SQL database?
3. What does ChromaDB store apart from embeddings?
4. What is a collection in ChromaDB?
5. Why must IDs in ChromaDB be unique strings?

🔹 Short Answer

6. Compare FAISS and ChromaDB in terms of metadata handling.
7. Why is metadata-based filtering important in multi-document RAG?
8. Explain how ChromaDB simplifies retrieval compared to FAISS.
9. What happens if metadata is not stored with embeddings?
10. Give one real-world example where metadata filtering is necessary.

🔹 Applied Questions

11. Why might you restrict retrieval to a specific source document?
12. How does metadata filtering reduce hallucinations?
13. Why is ChromaDB more suitable for production systems than FAISS alone?

📅

DAY 3 — Re-ranking (Retrieve → Re-rank → Answer)

🔹 Very Short Answer

1. What is re-ranking in RAG?
2. Why is embedding-based retrieval sometimes insufficient?
3. What is a cross-encoder?
4. Is re-ranking faster or slower than embedding retrieval?
5. What does a re-ranking score represent?

🔹 Short Answer

6. Explain the difference between a bi-encoder and a cross-encoder.
7. Why do we retrieve top-20 chunks and re-rank only a few?
8. How does re-ranking improve answer accuracy?
9. What happens if re-ranking is applied to the entire database?
10. Why should only top 3–5 re-ranked chunks be sent to the LLM?

🔹 System Design Questions

11. Draw or explain the full pipeline: Query → Retrieve → Re-rank → Answer.
12. How does re-ranking reduce “topic-related but answer-poor” chunks?
13. In what situations does re-ranking provide the biggest improvement?

📅

DAY 4 — Persistence, Logging & Debugging

🔹 Very Short Answer

1. What does persistence mean in vector databases?
2. Why is persistence required for production RAG systems?
3. What is logging in the context of RAG?

4. Name two things that should be logged in a RAG system.
5. What is an evaluation dataset?

🔹 Short Answer

6. Why should embeddings not be recomputed every time the system runs?
7. How does persistent ChromaDB improve system reliability?
8. Why is section-level metadata useful beyond page numbers?
9. Explain the purpose of batch insertion into a vector database.
10. Why is logging retrieval results important for debugging?

🔹 Debugging / Scenario Questions

11. The RAG system gives a wrong answer. What is the first thing you should check?
12. Retrieved chunks do not contain the answer even though it exists. What could be wrong?
13. How does logging help differentiate retrieval errors from generation errors?

📅

DAY 5 — RAG Evaluation & Comparison

🔹 Very Short Answer

1. What is RAG evaluation?
2. What is relevance in RAG evaluation?
3. What is faithfulness (groundedness)?
4. What is hallucination rate?
5. Why must RAG be compared with “no RAG”?

🔹 Short Answer

6. Explain the difference between relevance and accuracy.
7. Why is faithfulness more important than fluency?
8. How does RAG reduce hallucination compared to no-RAG systems?
9. Why can an answer be relevant but not faithful?
10. Why is keyword-based evaluation acceptable for beginner-level evaluation?

🔹 Long / Analytical Questions

11. Explain the full RAG evaluation process used in Week 18.
12. Compare LLM performance **with RAG vs without RAG** using evaluation metrics.
13. What conclusions can be drawn if RAG accuracy is high but faithfulness is low?
14. List limitations of the Week-18 RAG evaluation approach.
15. How would you improve evaluation for a production-grade RAG system?
