📝 Lecture Notes: Production RAG Data Pipeline

Core Philosophy: "The Engineer's Choice"

In a production environment (and interviews), there is no "best" tool. There are only **trade-offs**. You must justify your choice based on Data Type, Speed, Cost, and Accuracy.

---

1. Data Ingestion (The "Hidden" Bottleneck)

Before you can embed data, you must accurately extract it. The method depends entirely on the input format.

Scenario	Recommended Tool	Why? (Interview Rationale)
Simple Text PDFs	**PyMuPDF** (fitz)	**Speed.** It is ~10-15x faster than complex parsers. Use this for standard digital documents.
Scanned Docs / Images	**Tesseract** (OCR)	**Necessity.** PyMuPDF sees images as blocks. Tesseract extracts text via Optical Character Recognition. Essential for invoices or old archives.
Documents with Tables	**Dockling**	**Structure Preservation.** Most parsers destroy tables. Dockling preserves rows/columns and exports to Markdown/JSON, which is critical for the LLM to understand tabular data.
Websites (Crawling)	**FireCrawl**	Converts websites directly into LLM-ready **Markdown**, handling navigation automatically.

Complex Scraping	**Puppeteer**	For dynamic JS-heavy sites or when you need precise control (e.g., "Only scrape text inside `<div class='content'>`").

---

2. Chunking Strategies (The System Design Core)

How you split text determines retrieval quality. If you split a sentence in half, semantic meaning is lost.

A. Fixed-Size Chunking

- **Method:** Split text every N tokens (e.g., 500 characters) with some overlap.
- **Pros:** Computationally cheap, easy to implement.
- **Cons:** Breaks context mid-sentence. Loses semantic meaning.
- **Best For:** Massive datasets (millions of web pages) where speed > precision.

B. Structural Chunking

- **Method:** Split based on document headers (Section 1.1, Section 1.2).
- **Pros:** Preserves the author's intended logical flow.
- **Cons: High Variance.** One section might be 100 words, another 5,000 words (too big for context window).
- **Best For:** Highly structured docs (Legal Contracts, Financial Reports, Textbooks).

C. Recursive Chunking (⭐⭐ The "Gold Standard")

- **Method:** A hybrid approach.
    1. Try to split by **Section** (Structure).
    2. If the chunk is too big (> Max Size), split by **Paragraph**.
    3. If still too big, split by **Sentence**.
- **Pros:** "Best of both worlds." Preserves structure while guaranteeing consistent chunk sizes.
- **Best For:** General-purpose RAG applications. **(This is your default interview answer)**.

D. Semantic Chunking

- **Method:**
    1. Calculate cosine similarity between sequential sentences.
    2. If Similarity > Threshold (e.g., 0.8), keep them in the same chunk.
    3. If Similarity drops (topic change), start a new chunk.
- **Pros:** Guarantees that a chunk contains **one coherent idea**.
- **Cons:** Computationally expensive (requires running embeddings on every sentence).
- **Best For:** Unstructured text with "drifting" topics (e.g., **Transcripts**, Podcasts, Meeting Notes).

E. LLM-Based Chunking (Agentic Chunking)

- **Method:** Ask an LLM: *"Read this text and split it where the topic changes."*
- **Pros:** Highest quality. Handles nuance better than math.
- **Cons: Extremely Slow & Expensive.** Not scalable for large corpora.
- **Best For:** High-value, small datasets where accuracy is paramount.

---

3. Evaluation & Metrics

How do you know your chunking strategy is working?

1. **Chunk Size Variance:** Plot the distribution of chunk sizes.
    - *Too much variance* (Structural) = Inconsistent retrieval performance.
    - *Too small* (Semantic) = Loss of broader context.
2. **Visual Inspection:** Manually check if chunks cut off mid-sentence or mid-table.

Interview "Cheatsheet" Answers

- **"How do you handle tables in RAG?"** *-> "I would use **Dockling** to parse the PDF into Markdown, ensuring the table structure is preserved as text, rather than using a naive parser that reads it column-by-column."*
- **"Which chunking strategy should we start with?"** *-> "I would start with **Recursive Character Chunking** because it balances preserving document structure with strict token limits. If we find that retrieval is missing context in transcripts, I would switch to **Semantic Chunking**."*
- **"Our RAG is slow. Why?"** *-> "If we are using **LLM-based chunking** or **Semantic Chunking** on a massive dataset, the ingestion pipeline will bottleneck. We might need to switch to Fixed or Recursive strategies for bulk processing."*

📚 Part 2: Embedding, Retrieval, and Production Deployment

1. Implementation of Chunking (Code Phase)

- **Strategy Applied:** Fixed-size chunking (grouping **10 sentences** per chunk).
- **Rationale:**
    - Average sentences per page: ~10.
    - Average tokens per page: ~287.
    - **Constraint:** The embedding model (`all-mpnet-base-v2`) has a limit of **384 tokens**.
- **Cleanup:** Removed chunks with <30 tokens (usually footnotes/headers) to reduce noise.

---

2. Embeddings: The "Engineer's Choice"

- **Definition:** Converting text chunks into vector representations (lists of numbers) that preserve semantic meaning.
- **Model Used:** `all-mpnet-base-v2` (Sentence Transformer).
- **Performance Benchmarks (1680 chunks):**
    1. **CPU:** ~4 min 32 sec.
    2. **GPU (CUDA):** ~26 sec.
    3. **Batched GPU:** ~23 sec (Fastest).
- **Critical Trade-offs to Consider:**
    1. **Input Size:** Can the model handle long contexts? (OpenAI/Gemini are better for long texts vs. Sentence Transformers).
    2. **Output Dimension:** Larger vectors (e.g., 1536 dim) capture more nuance but increase storage/compute costs.
    3. **Open vs. Closed Source:** OpenAI is SOTA (State of the Art) but costs money. Local models (HuggingFace) ensure privacy.

---

3. Vector Storage: The "Tea Kettle Principle"

Where should we store these embeddings?

Tiered Decision Framework

1. **Small Data (<100k vectors):** Use `torch.tensor` or `NumPy` arrays. No DB needed.
2. **Medium Data:** Use **FAISS** (Facebook AI Similarity Search). It's a library, not a database.
3. **Production:** Vector Databases.
    - **Dedicated DBs:** Pinecone, Milvus, Weaviate, Chroma.
    - **The Recommended Approach: Postgres + pgvector (Supabase).**

Why Postgres (Supabase)?

- **The Tea Kettle Principle:** "If you have a hard problem, reduce it to a problem you have already solved."
- **Rationale:** Most companies already use SQL/RDBMS for user data (logins, profiles). Using **pgvector** allows you to store **Vectors + User Data** in the *same* database.
- **Benefit:** You can use SQL joins, filters, and standard backups. No need to manage a separate Pinecone instance and sync metadata manually.

Indexing Strategies (Under the hood)

To make search fast, you cannot scan 1 million vectors sequentially.

1. **IVF_Flat (Clustering):** Like K-Means. Groups vectors into clusters. You only search the relevant cluster.
2. **HNSW (Hierarchical Navigable Small Worlds):** A multi-layered graph approach. Faster and more accurate but slower to build index.

---

4. Retrieval & Generation

Retrieval

- **Metric: Cosine Similarity** (focuses on angle/meaning) is preferred over Dot Product (focuses on magnitude).
- **Process:** Query $\rightarrow$ Vector $\rightarrow$ Find top k chunks with highest cosine score.

Generation (The LLM)

- **Model:** `Gemma-2b-it` (Google's open source model).
- **Optimization:** Used **Quantization** (via `bitsandbytes`) to run the model on a free T4 GPU (Google Colab).
    - *Quantization:* Compressing model weights (e.g., 16-bit $\rightarrow$ 4-bit) to reduce memory usage with minimal accuracy loss.
- **Augmentation:** The prompt is dynamically constructed:
  Base Prompt + "Use the following context items to answer the query" + [Retrieved Chunks]

---

5. Evaluation (The Missing Piece)

How do we know the RAG system is actually working?

- **Library: Ragas** (Rag Assessment).
- **Key Metrics:**
    1. **Context Precision:** Are the retrieved chunks relevant to the ground truth?
    2. **Context Recall:** Did we retrieve *all* necessary information?
    3. **Faithfulness:** Is the answer derived *only* from the context (no hallucinations)?

4. **Answer Relevancy:** Does the answer directly address the user's prompt?
- **Golden Rule:** You **must** create a manual "Ground Truth" dataset (Question + Correct Answer + Correct Context) to run these evaluations effectively.

---

6. Production Deployment (Web App Workflow)

The workshop concluded by moving from a Notebook to a full-stack application.

Tech Stack

- **Database:** Supabase (Postgres with `pgvector` enabled).
- **Backend:** Next.js (TypeScript).
- **Frontend Generation: Lovable.dev** (AI tool to generate UI).

Steps to Build:

1. **Database Setup (Supabase):**
   - Enable `vector` extension.
   - Create a `match_documents` SQL function to perform cosine similarity search.
2. **Ingestion Script (`ingest.py`):**
   - Parses PDF $\rightarrow$ Chunks $\rightarrow$ Embeds (OpenAI) $\rightarrow$ Uploads to Supabase.
3. **Backend API (`route.ts`):**
   - Receives User Query.
   - Embeds Query.
   - Calls Supabase `match_documents` RPC.
   - Sends Context + Query to OpenAI for final answer.
4. **UI Generation (Lovable):**
   - Upload the code files (`ingest.py`, `route.ts`) to Lovable.
   - Prompt: *"Make a nutritional chatbot with an Arc Browser theme, clickable citations, and sound effects."*
   - Result: A deployed, sleek web app with citations.

Summary Checklist for Production RAG

- [ ] **Chunking:** Is fixed size enough, or do we need semantic?
- [ ] **Embedding:** Are we using a model with enough context window?
- [ ] **Storage:** Postgres (pgvector) for unified data handling.
- [ ] **Indexing:** HNSW for speed, IVF for simplicity.
- [ ] **Eval:** Setup Ragas with a manual ground truth set.