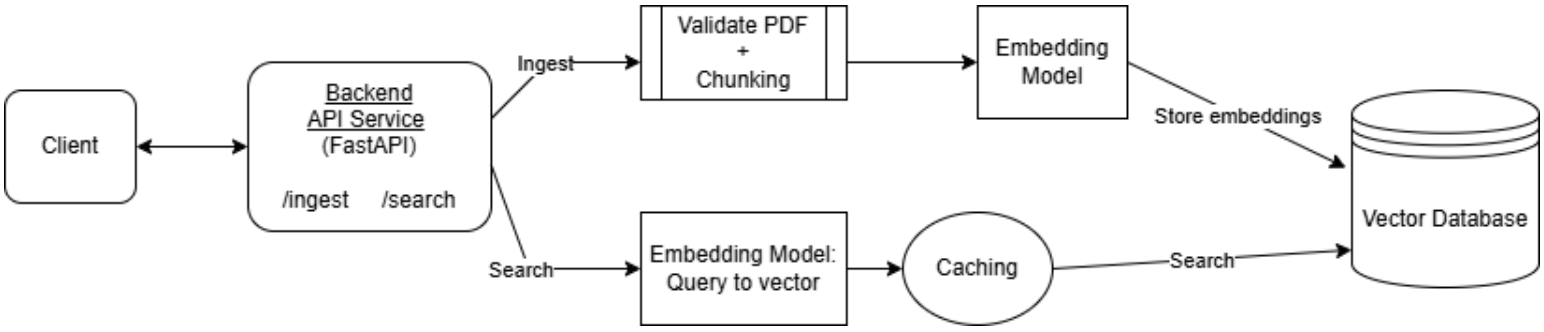
Brief Design Summary

1.System requirements:

- Ingest and chunk PDFs
- Generate embeddings for each chunk
- Store embeddings and metadata in a vector database to perform search query
- Provide `/ingest/` and `/search/` endpoints
- Containerize with Docker for easy deployment.

2.System Architecture:



Technologies Used: & Why

- Backend framework → FastAPI (Lightweight, async)
- PDF extraction → pdfplumber
- Embeddings → sentence-transformers "all-MiniLM-L6-v2"
- Vector DB → Chroma

3. Data Flows:

a. Ingestion Flow (PDF → Vector DB)

- User uploads PDF(s) via `POST /ingest/`.
- Backend (API Service) validates files and handles concurrent requests.
- PDF Processing extracts text , cleans and splits it into manageable chunks.
- Each chunk is converted into an embedding using the embedding model.
- Embeddings and metadata (filename, chunk ID, etc.) are stored in the vector database.
- API responds with a success message.

b. Search Flow

- User sends a query via `POST /search/`.
- Backend converts the query into an embedding using the same model.
- Optional caching layer checks for repeated queries.
- Vector database is queried for the top-k (3) semantically similar chunks.
- API returns results with document names, similarity scores, and content.

4.Bottlenecks & Trade-offs:

- **PDF Extraction:** Used pdfplumber instead of LangChain → **faster and lighter for Docker**, but slightly less accurate on scanned or complex PDFs.

- **Chunking & Search:** Sentence-aware chunking → **better semantic relevance**, but keeping it simple with ChromaDB means **search isn't fully precise**.

- **Embedding Generation:** Lightweight model (all-MiniLM-L6-v2) → **fast and efficient**, but slightly lower semantic accuracy than larger models.

- **Concurrency:** Async ingestion improves throughput, but system resources are limited; a semaphore prevents overload.