Chapter 1: Executive Summary & Architecture

IPO Intelligence Platform - A Production-Grade Document Intelligence System

---

Table of Contents for This Chapter

---

1. Introduction

1.1 What is the IPO Intelligence Platform?

The IPO Intelligence Platform is a sophisticated document intelligence system designed specifically for analyzing Initial Public Offering (IPO) documents. These documents, known as Draft Red Herring Prospectuses (DRHPs) in India, are typically 400-800 pages of dense legal, financial, and regulatory information that investors and analysts need to understand before making investment decisions.

The platform allows users to: - **Upload** PDF documents (DRHPs, quarterly reports, annual reports) - **Ask natural language questions** about the content - **Receive accurate, sourced answers** with references to specific pages - **Explore entity relationships** like company structures, promoter holdings, and management teams - **Look up definitions** for technical terms used in these documents

1.2 Why Was This Built?

The motivation for this project comes from real-world challenges:

The Information Overload Problem

When a company files for an IPO, they release a DRHP that contains: - Complete company history and business description - Financial statements for the past 3-5 years - Details of all subsidiaries, joint ventures, and related entities - Risk factors (often 50+ pages of risks) - Information about

promoters, directors, and key management - Legal proceedings and litigations - Use of proceeds from the IPO - Selling shareholders and their stakes - Offer structure and pricing details

An analyst reviewing this document manually might spend 8-16 hours just reading it, let alone extracting and analyzing the relevant information.

The Current Solutions and Their Limitations

1. Manual Reading - Time-consuming: 8-16 hours per document - Error-prone: Easy to miss important details - Not scalable: Can't compare multiple IPOs efficiently

2. Basic Document Search (Ctrl+F) - Only finds exact keyword matches - Misses semantic similarity ("revenue growth" vs "increase in income") - No understanding of document structure for context

3. General-Purpose Chatbots (ChatGPT, etc.) - Not trained on specific documents - Hallucinate facts that sound plausible but are wrong - Can't provide source citations - Privacy concerns with uploading sensitive documents

Our Solution

The IPO Intelligence Platform addresses all these limitations by: 1. Keeping documents local (privacy) 2. Grounding answers in actual document content (accuracy) 3. Providing source citations (verifiability) 4. Understanding semantic meaning (intelligence) 5. Extracting structured data (knowledge graph)

---

2. The Problem We're Solving

2.1 IPO Documents: A Unique Challenge

IPO documents present a unique set of challenges that make them difficult to work with:

2.1.1 Document Characteristics

Characteristic	Challenge It Creates
Length (400-800 pages)	Too long to read completely
Dense information	High cognitive load
Legal language	Complex sentence structures
Defined terms	"ESOP-2014", "Company", "Restated Financial Statements"
Tables and figures	Financial data in tabular format
Cross-references	"As described in 'Risk Factors' section..."
Temporal information	Multiple time periods (FY2021, Q1FY22, etc.)

2.1.2 Types of Questions Users Ask

Based on our analysis, questions about IPO documents fall into several categories:

Category 1: Factual Lookup - "What is the CIN of the company?" - "Who is the CEO?" - "What is the face value of equity shares?" - "What is the registered office address?"

These questions have a single, definitive answer that exists somewhere in the document.

Category 2: Definition Queries - "What does ESOP-2014 mean?" - "What is CCC in this context?" - "What does 'Restated Consolidated Financial Statements' refer to?"

IPO documents have extensive definition sections. Users need to understand these terms.

Category 3: Relationship Queries - "List all subsidiaries of the company" - "Who are the promoters?" - "What is the ownership structure?" - "Who are the selling shareholders in this offer?"

These require understanding entity relationships.

Category 4: Summarization Queries - "Summarize the key risk factors" - "What is the business model?" - "Explain the strengths of the company"

These require synthesizing information from multiple sections.

Category 5: Comparative/Analytical Queries - "How has revenue changed over the last 3 years?" - "Compare EBITDA margins across periods" - "What is the trend in customer acquisition?"

These require extracting multiple data points and analyzing them.

Category 6: Timeline Queries - "When was the company incorporated?" - "When did the company change its name?" - "What are the key milestones?"

These require understanding temporal information.

2.2 Why Traditional Search Fails

2.2.1 Keyword Search Limitations

Consider the question: "Who is responsible for running the company?"

A keyword search would look for "responsible" and "running" and "company" - but the answer might be in a sentence like:

> "Mr. Yashish Dahiya serves as the Chairman, Executive Director and Chief Executive Officer."

There's no keyword match, yet this clearly answers the question.

2.2.2 The Semantic Gap

The semantic gap is the disconnect between: - **What the user means** (intent) - **What words they use** (query) - **What words are in the document** (content)

Examples of semantic gaps:

User Query	Document Text	Gap
“Company profits”	“Net income after tax”	Different terminology
“Who owns the company?”	“Shareholding pattern of promoters”	Conceptual mapping
“IPO size”	“Aggregate issue size of ₹5,710 crore”	Terminology
“Growth”	“Increase of 42% year-on-year”	Abstract vs. specific

2.2.3 Why LLMs Alone Aren’t Enough

Large Language Models (LLMs) are incredibly powerful, but they have fundamental limitations:

Context Window Limits - Even with 128K tokens, a 600-page document doesn’t fit - Truncating = losing information - Processing everything = slow and expensive

Hallucination Problem - LLMs confidently generate plausible-sounding but incorrect information - They blend training data with the prompt in unpredictable ways - Critical for financial documents where accuracy matters

No Source Attribution - LLMs generate fluent text but can’t point to specific sources - Users can’t verify the answer - Regulatory/compliance issues

Training Data Cutoff - LLMs don’t know about recent IPOs - They can’t access your private documents

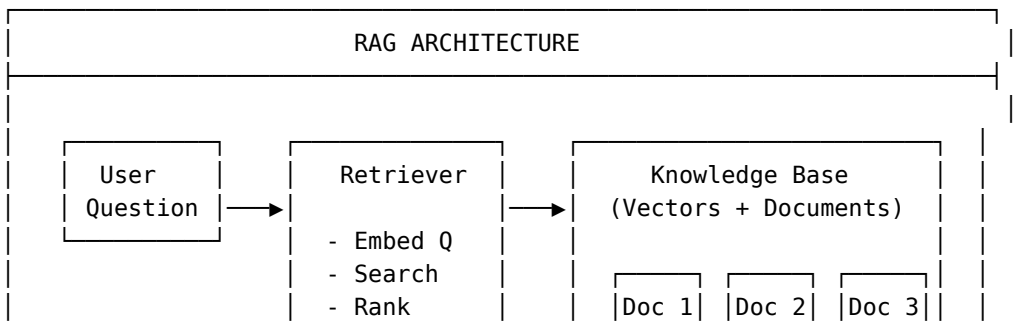
---

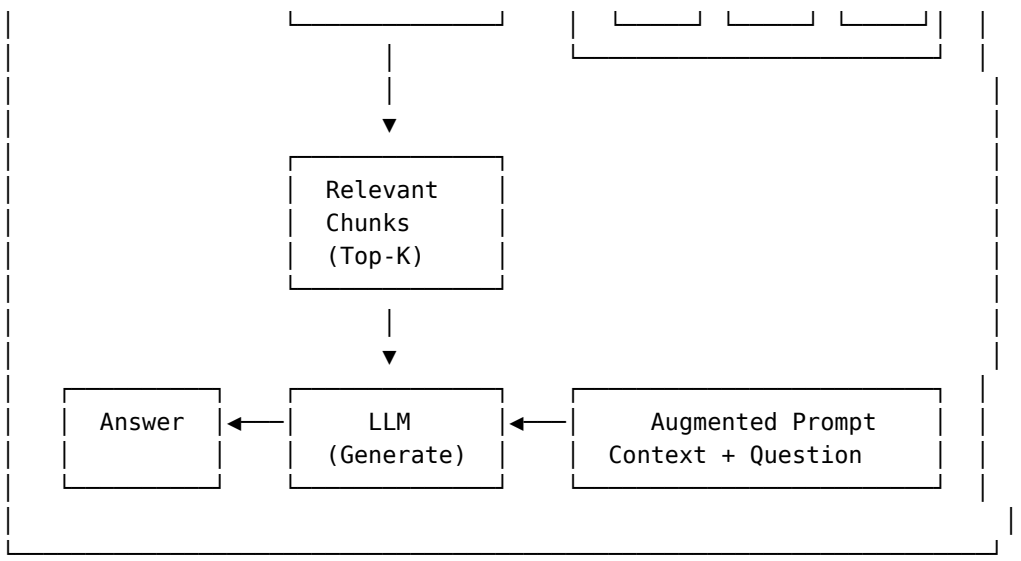
3. What is RAG?

3.1 Retrieval Augmented Generation Explained

RAG (Retrieval Augmented Generation) is an architecture that combines: 1. **Retrieval**: Finding relevant information from a knowledge base 2. **Augmentation**: Adding this information to the LLM’s context 3. **Generation**: Having the LLM generate an answer based on the context

3.1.1 The RAG Process





3.1.2 Why RAG Works

RAG solves the core LLM problems:

LLM Problem	RAG Solution
Context limits	Only retrieve relevant chunks, not entire document
Hallucination	Ground answers in actual document text
No sources	Chunks include metadata (page, section)
Stale knowledge	Works with any document, including new ones
Privacy	Documents stay local, no upload to cloud LLMs

3.1.3 Components of a RAG System

1. Document Processor - Ingests raw documents (PDFs, text, etc.) - Extracts text content - Splits into manageable chunks - Preserves metadata

2. Embedding Model - Converts text into dense vectors (embeddings) - Captures semantic meaning - Enables similarity comparison

3. Vector Store - Stores embeddings with metadata - Enables fast similarity search - Scales to millions of vectors

4. Retriever - Takes a query - Finds most relevant chunks - Returns ranked results

5. Generator (LLM) - Takes context + question - Generates fluent answer - Follows instructions/ formatting

3.2 Evolution of RAG

3.2.1 First Generation: Basic RAG

The simplest RAG approach: 1. Chunk documents uniformly 2. Embed all chunks 3. For each question, embed and find similar chunks 4. Concatenate chunks and ask LLM

Limitations: - Chunks might split mid-sentence - No understanding of document structure - All chunks treated equally - No verification of answers

3.2.2 Second Generation: Advanced RAG

Improvements include: - Better chunking (respecting paragraphs, sections) - Metadata filtering (only search relevant sections) - Reranking (use a more powerful model to reorder results) - Query expansion (rewrite query for better matches)

3.2.3 Third Generation: Hybrid/Agentic RAG

Current state of the art: - Multiple retrieval methods (vector + keyword + structured) - Knowledge graphs for relationships - Multi-step reasoning - Tool use (calculators, databases, APIs) - Self-verification of answers

Our system is a Third Generation RAG, combining: - Vector search for semantic similarity - Knowledge graph for structured data - Entity extraction for relationships - Definition lookup for terminology

---

4. Why Hybrid RAG?

4.1 The Limitations of Vector-Only RAG

While vector search is powerful, it has inherent limitations:

4.1.1 Scattered Information Problem

Consider: "List all subsidiaries of the company"

In a typical DRHP, subsidiaries might be mentioned in: - Page 45: "Our Subsidiaries include Policybazaar..." - Page 120: "...through our subsidiary Paisabazaar..." - Page 230: "The following table shows our group structure..." - Page 450: "PB Health Insurance, a subsidiary..."

Vector search might find some of these, but: - The information is scattered across the document - Each chunk only has partial information - No single chunk has the complete answer

4.1.2 Exact Fact Lookup Problem

Consider: "What is the CIN of the company?"

The answer is exactly one line in the document: > "CIN: U74999DL2008PLC178155"

Vector search finds similar chunks, but: - "Similar" doesn't mean "contains the exact fact" - Might return paragraphs discussing the company without the CIN - Low recall for point-fact queries

4.1.3 Relationship Query Problem

Consider: "Who is the CEO of which company?"

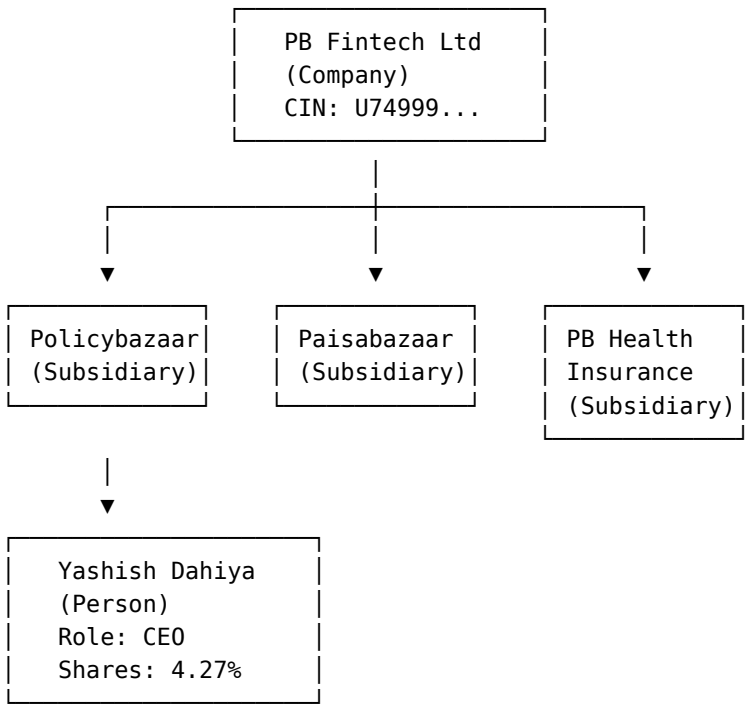
This requires understanding: - Entity: Yashish Dahiya (Person) - Relationship: CEO_of - Entity: PB Fintech Limited (Company)

Vector search treats this as a bag of words, missing the structured relationship.

4.2 Enter the Knowledge Graph

A Knowledge Graph (KG) stores information as: - **Entities**: Things with identity (people, companies, products) - **Attributes**: Properties of entities (name, CIN, date) - **Relationships**: Connections between entities (CEO_of, subsidiary_of)

4.2.1 KG Structure

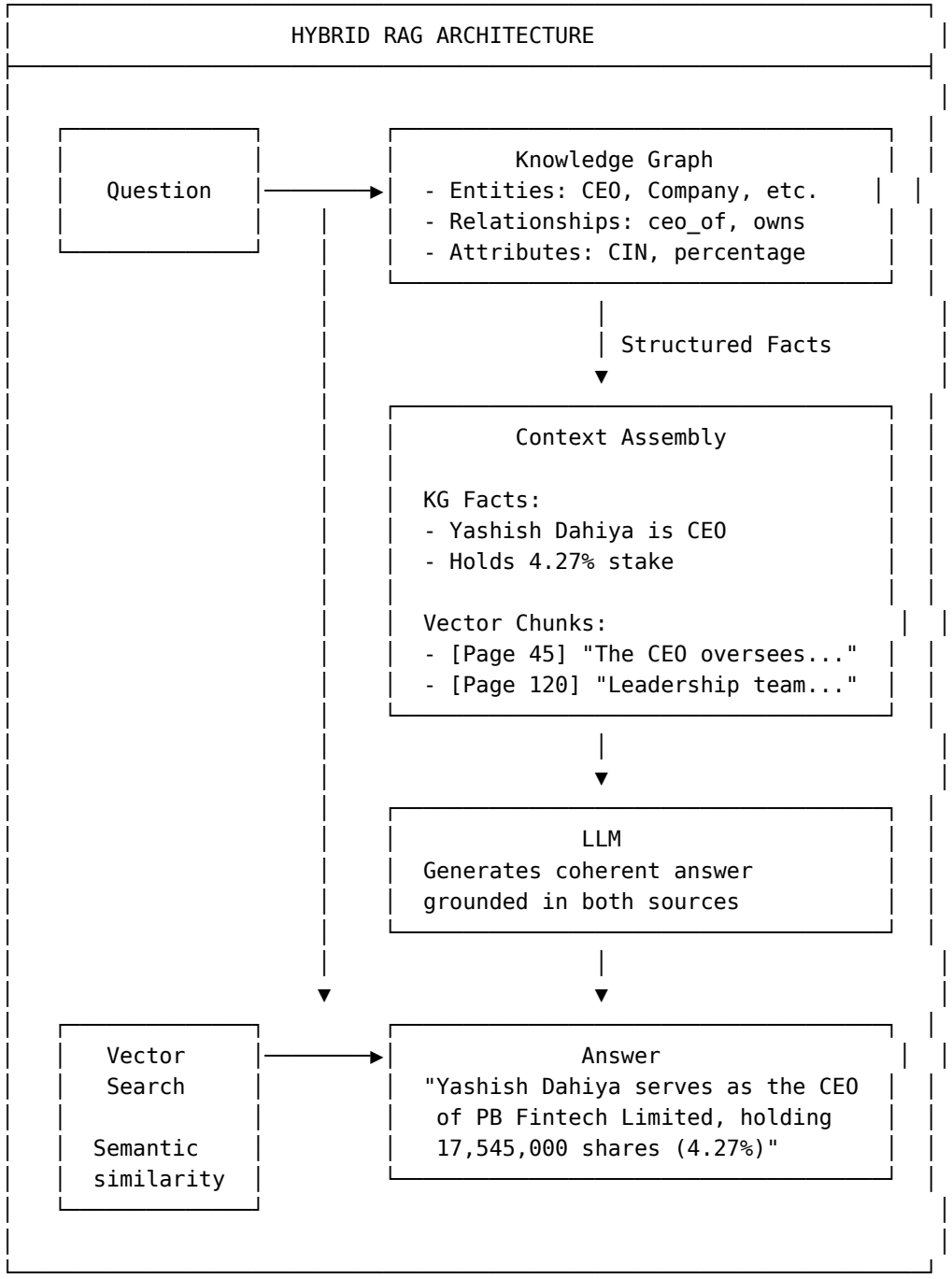


4.2.2 What KG Enables

Query Type	Vector RAG	KG RAG
"List subsidiaries"	Multiple scattered chunks	Direct graph traversal
"Who is CEO?"	Similar chunks about leadership	Entity lookup
"CIN number?"	Might miss exact line	Attribute lookup
"Promoter holdings?"	Text about promoters	Structured data query

4.2.3 The Hybrid Advantage

By combining both:



4.3 When to Use Which

Our system intelligently routes queries:

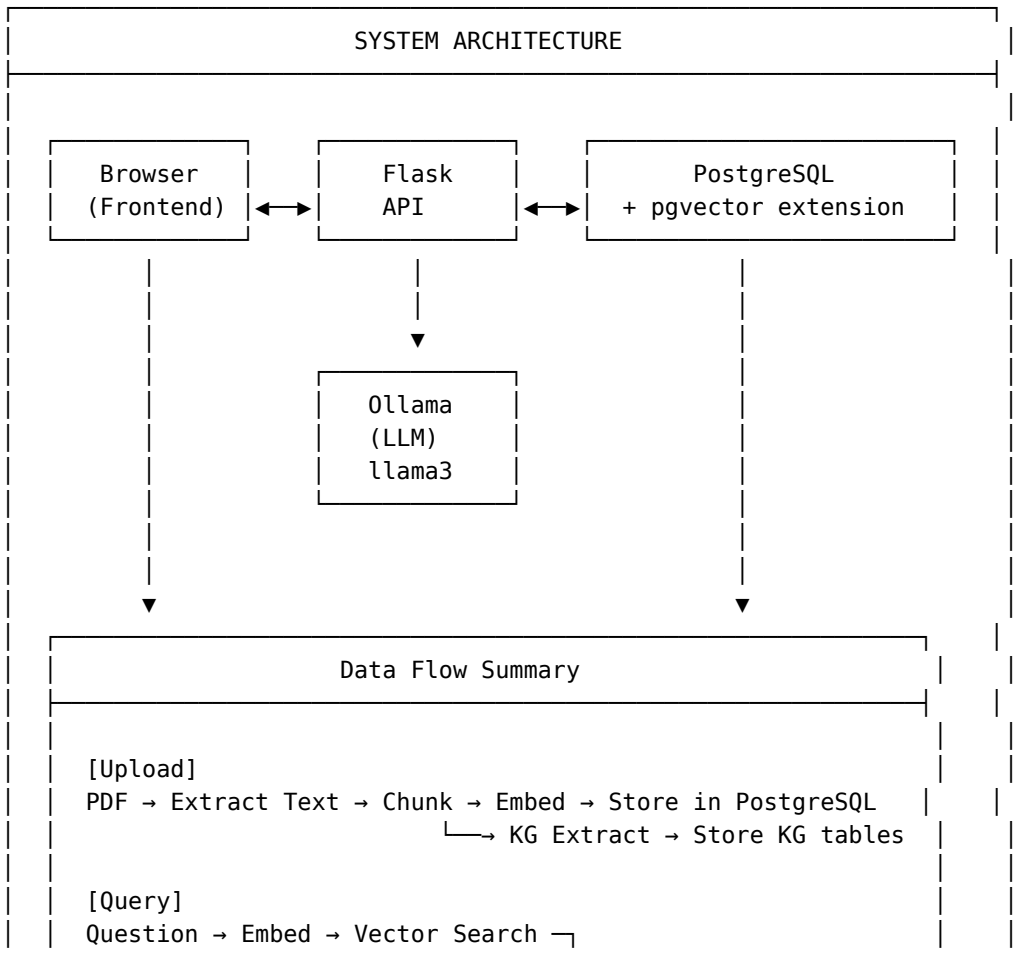
Question Pattern	Best Approach	Why
“Who is X?”	KG first	Direct entity lookup
“List all Y”	KG first	Relationship traversal
“What does X mean?”	Definition lookup	Exact match in glossary
“Explain the business”	Vector first	Need narrative context
“Summarize risks”	Vector first	Need full paragraphs
“Revenue in FY2021?”	KG first	Structured financial data
“Compare X and Y”	Hybrid	Need both data and context

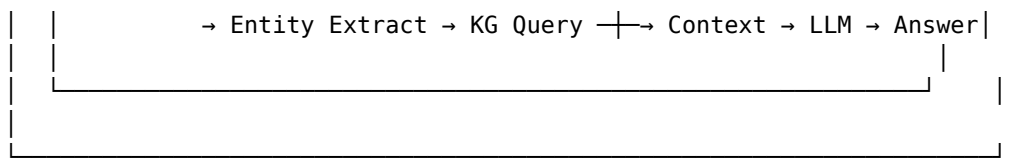
---

5. System Architecture Overview

5.1 High-Level Components

The system consists of five major components:





5.1.1 Component 1: Frontend (Browser)

Purpose: User interface for document management and Q&A

Technologies: - HTML5 for structure - CSS3 with CSS variables for theming - Vanilla JavaScript (no frameworks) - Streaming response handling

Key Features: - Document selector dropdown - Chat-style Q&A interface - Thinking/reasoning display - Source attribution - Real-time streaming output

5.1.2 Component 2: Flask API

Purpose: REST API backend, orchestrates all operations

Technologies: - Flask web framework - Flask-CORS for cross-origin requests - Streaming responses with generators

Endpoints: | Endpoint | Method | Purpose | |———-|———|———| | `/` | GET | Serve frontend | | `/api/documents` | GET | List documents | | `/api/upload` | POST | Upload new PDF | | `/api/ask` | POST | Answer question |

5.1.3 Component 3: PostgreSQL + pgvector

Purpose: Unified storage for all data

What It Stores: - Document metadata - Text chunks - Vector embeddings (384-dimensional) - Knowledge graph (entities, relationships, events) - Evidence/provenance

Key Features: - HNSW index for fast vector similarity - ACID transactions for data integrity - JSON/JSONB for flexible schemas - Full-text search capability

5.1.4 Component 4: Ollama (LLM)

Purpose: Local LLM for extraction and generation

Model: llama3 (or llama3:latest)

Uses: - Knowledge graph extraction (entities, relationships) - Answer generation - Query understanding

Why Local: - Privacy: documents stay on machine - Cost: no API fees - Control: no rate limits

5.1.5 Component 5: Embedding Model

Purpose: Convert text to vectors

Model: all-MiniLM-L6-v2 (sentence-transformers)

Characteristics: - 384 dimensions - Optimized for similarity search - ~80MB model size - 14,000+ sentences/second

5.2 Data Flow: Detailed Walkthrough

5.2.1 Document Upload Flow

```
Step 1: User selects PDF
        ↓
Step 2: Frontend sends to /api/upload
        ↓
Step 3: Flask saves file, calculates hash
        ↓
Step 4: Check hash for duplicates
        ├── Duplicate → Return existing doc
        └── New → Continue
        ↓
Step 5: Extract text using PyMuPDF
        ↓
Step 6: Split into chunks (500-1200 tokens)
        ↓
Step 7: Generate embeddings for each chunk
        ↓
Step 8: Save to PostgreSQL:
        ├── documents table (metadata)
        ├── chunks table (text)
        └── embeddings table (vectors)
        ↓
Step 9: Return success to frontend
```

5.2.2 KG Build Flow (Async)

```
Step 1: Load chunks from database
        ↓
Step 2: For each chunk, run extraction prompts:
        ├── Definitions prompt → defined_terms
        ├── Entity prompt → kg_entities
        ├── Relationship prompt → claims
        └── Event prompt → events
        ↓
Step 3: Entity resolution (merge duplicates)
        ↓
Step 4: Validation (check data quality)
        ↓
Step 5: Save to PostgreSQL KG tables
```

5.2.3 Query Flow

```
Step 1: User types question
        ↓
Step 2: Frontend POSTs to /api/ask
        ↓
Step 3: Backend determines mode (vector/kg/hybrid)
        ↓
Step 4: Parallel retrieval:
```

```
        ├── Vector: embed question, search similar chunks
        └── KG: extract entities, query graph
        ↓
Step 5: Combine context from both sources
        ↓
Step 6: Build prompt with context + question
        ↓
Step 7: Stream tokens from Ollama
        ↓
Step 8: Frontend displays tokens as they arrive
```

---

6. Technology Stack Deep Dive

6.1 Python (3.10+)

6.1.1 Why Python?

- **ML Ecosystem**: Best libraries for embeddings, NLP
- **Flask**: Lightweight, perfect for APIs
- **SQLAlchemy**: Robust ORM for database
- **Rapid Development**: Quick iteration

6.1.2 Key Libraries

Library	Version	Purpose
flask	3.0+	Web framework
sqlalchemy	2.0+	ORM
psycopg2	2.9+	PostgreSQL driver
sentence-transformers	2.2+	Embeddings
pymupdf	1.23+	PDF extraction
ollama	0.1+	LLM client
numpy	1.24+	Vector operations
pgvector	0.2+	Vector type for SQLAlchemy

6.2 PostgreSQL + pgvector

6.2.1 Why PostgreSQL?

Advantages Over Specialized Vector DBs:

Aspect	PostgreSQL	Pinecone/Weaviate
Cost	Free, open source	$70+/month
Deployment	Single binary, simple	Cloud dependency
SQL	Full SQL power	Limited query language
Transactions	Full ACID	Eventual consistency
Joins	Native	Separately query
Ecosystem	30+ years, mature	New, evolving

The Key Insight: We need more than just vectors. We need: - Document metadata (SQL tables) - Knowledge graph (SQL tables with relationships) - Full-text search (PostgreSQL native) - Vector search (pgvector extension)

PostgreSQL gives us **everything in one database**.

6.2.2 pgvector Extension

pgvector adds: - `vector` data type for N-dimensional vectors - Distance operators: `<=>` (cosine), `<->` (L2), `<#>` (inner product) - Index types: IVFFlat, HNSW

Installing pgvector:

```
CREATE EXTENSION IF NOT EXISTS vector;
```

Creating a vector column:

```
CREATE TABLE embeddings (
    id SERIAL PRIMARY KEY,
    embedding vector(384)  -- 384 dimensions
);
```

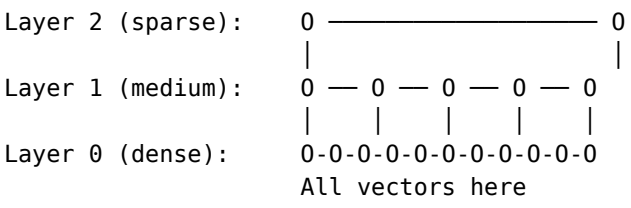
Similarity search:

```
SELECT * FROM embeddings
ORDER BY embedding <=> '[0.1, 0.2, ...]'
LIMIT 10;
```

6.2.3 HNSW Index Deep Dive

HNSW (Hierarchical Navigable Small World) is a graph-based ANN algorithm.

Structure:



How Search Works: 1. Start at a random entry point in the top layer 2. Greedily navigate toward query vector 3. When stuck, drop to next layer 4. Continue until Layer 0 5. Find approximate nearest neighbors

Parameters: | Parameter | Meaning | Tuning | |———–|———–|———–| | `m` | Connections per node | Higher = better recall, slower | | `ef_construction` | Build-time search depth | Higher = better index, slower build | | `ef_search` | Query-time search depth | Higher = better recall, slower query |

Our Configuration:

```
CREATE INDEX idx_embeddings_vector ON embeddings
USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);
```

6.3 Ollama (Local LLM)

6.3.1 What is Ollama?

Ollama is a tool for running LLMs locally. It: - Downloads and manages model weights - Provides a simple API (HTTP + CLI) - Handles GPU/CPU optimization - Supports streaming

6.3.2 Why Local LLM?

Privacy: - IPO documents contain material non-public information - Sending to cloud APIs = potential data breach - Local = complete control

Cost: - Processing 816 chunks with GPT-4 = $50+ per document - Local = electricity only

Control: - No rate limits - No API changes breaking your code - Consistent behavior

6.3.3 Model Choice: llama3

Property	llama3
Parameters	8B
Context	8K tokens
Speed	~30 tokens/sec (M1 Mac)
Quality	Excellent for extraction

Trade-offs: - Larger models (70B) = better quality but slower - Smaller models (7B) = faster but less accurate - 8B is the sweet spot for our use case

6.3.4 API Usage

```
import ollama

response = ollama.chat(
    model='llama3',
    messages=[
        {'role': 'system', 'content': 'You are a helpful assistant.'},
        {'role': 'user', 'content': 'Extract entities from this text...'}
    ],
```

```
    stream=True
)

for chunk in response:
    print(chunk['message']['content'], end='')
```

6.4 Sentence Transformers

6.4.1 How Embedding Models Work

Embedding models are trained using **contrastive learning**:

1. Take pairs of similar sentences (from same paragraph, QA pairs)
2. Take pairs of dissimilar sentences (random pairs)
3. Train to push similar close, dissimilar apart

The result: a model that maps text to a vector space where semantic similarity = geometric proximity.

6.4.2 all-MiniLM-L6-v2 Details

Property	Value
Architecture	6-layer transformer (distilled from BERT)
Output	384 dimensions
Max tokens	256 (truncates longer)
Training data	1B+ sentence pairs
Size	~80MB
Speed	14,000 sentences/sec (CPU)

6.4.3 Usage

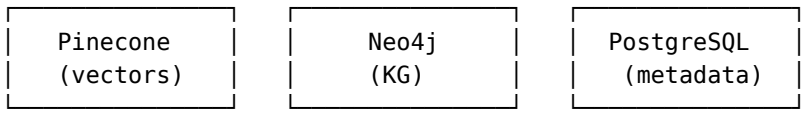
```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

# Embed text
embeddings = model.encode(
    ["text 1", "text 2"],
    normalize_embeddings=True,  # For cosine similarity
    show_progress_bar=True
)

# Result: numpy array of shape (2, 384)
```

---

7. Key Design Decisions & Trade-offs

7.1 Decision: Single PostgreSQL vs. Multiple Databases

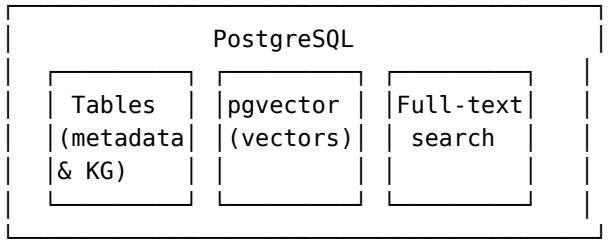
Option A: Specialized Databases



Pros: - Each database optimized for its use case - Potentially better performance at scale

Cons: - Three systems to maintain - Data consistency challenges - Higher cost - Complexity in queries that need data from multiple sources

Option B: PostgreSQL + Extensions (Our Choice)



Pros: - Single system to maintain - ACID transactions across all data - SQL JOINs work naturally - Much simpler operations

Cons: - Vector search not as fast as specialized DBs at extreme scale - Learning pgvector specifics

Trade-off Analysis: For our scale (thousands of documents, millions of chunks), PostgreSQL is more than sufficient. The operational simplicity outweighs the marginal performance difference.

7.2 Decision: Local LLM vs. Cloud API

Cloud API (OpenAI/Anthropic)
Pros: - Best-in-class quality (GPT-4, Claude) - No local GPU required - Simple API

Cons: - Privacy: sending documents to third party - Cost: ~$0.06/1K tokens (GPT-4) - Rate limits: 10K tokens/minute on lower tiers - Latency: network round-trip

Local LLM (Ollama) - Our Choice
Pros: - Complete privacy - $0 marginal cost - No rate limits - Lower latency

Cons: - Requires decent hardware (8GB+ RAM) - Quality slightly below frontier models - More setup required

Trade-off Analysis: For IPO documents containing sensitive information, privacy is paramount. The quality difference between llama3 and GPT-4 is acceptable for our extraction tasks.

7.3 Decision: Chunking Strategy

Fixed-Size Tokenization
Approach: Split every N tokens regardless of content

Pros: Simple, consistent chunk sizes

Cons: Might split mid-sentence, lose context

Semantic Chunking (Our Approach)
Approach: Split on paragraph/section boundaries, respecting max size

Pros: Preserves coherence, metadata is meaningful

Cons: Variable chunk sizes, more complex logic

Trade-off Analysis: The additional complexity is worth it for better retrieval quality. A chunk that ends mid-sentence is less useful than one that respects document structure.

7.4 Decision: Synchronous vs. Asynchronous KG Building

Synchronous (during upload)
Pros: KG ready immediately after upload

Cons: Very slow upload (minutes for large docs)

Asynchronous (separate process) - Our Choice
Pros: Fast upload, KG building can happen in background

Cons: KG not immediately available

Trade-off Analysis: KG extraction takes ~30 seconds per chunk with local LLM. For 816 chunks, that's 6+ hours. Running synchronously would make uploads unusable. Async is the only practical choice.

---

8. Project Structure Explained

8.1 Directory Layout

```
ipo_qa/
├── src/                          # Main application code
│   ├── app.py                    # Flask entry point (API routes)
│   ├── database/                 # Database layer
│   │   ├── __init__.py
│   │   ├── connection.py         # Connection pooling
│   │   ├── models.py             # SQLAlchemy ORM (core tables)
│   │   ├── kg_models.py          # SQLAlchemy ORM (KG tables)
│   │   ├── repositories/         # Data access layer
│   │   │   ├── __init__.py
│   │   │   ├── document_repo.py  # Document CRUD
│   │   │   ├── chunk_repo.py     # Chunk CRUD
```

```
│   │   │   └── embedding_repo.py # Embedding CRUD
│   │   └── kg_repositories.py    # KG data access
│   ├── utils/                    # Utility modules
│   │   ├── __init__.py
│   │   ├── vector_rag.py         # Vector-based retrieval
│   │   ├── kg_rag.py             # KG-based retrieval
│   │   ├── hybrid_rag.py         # Combined retrieval
│   │   ├── kg_pipeline.py        # KG extraction pipeline
│   │   ├── kg_prompts.py         # LLM prompts
│   │   ├── deepseek_client.py    # Ollama wrapper
│   │   └── [other utilities]
│   └── static/                   # Frontend assets
│       ├── index.html            # Main page
│       ├── style.css             # Styles
│       └── script.js             # JavaScript logic
├── scripts/                      # CLI scripts
│   ├── build_kg.py               # Original KG builder
│   ├── build_kg_v2.py            # Production KG builder
│   └── migrate_to_db.py          # Data migration
├── database/                     # SQL schemas
│   ├── schema.sql                # Core tables
│   └── kg_schema.sql             # KG extension
├── data/                         # Data storage
│   └── documents/                # Document folders
│       └── {document_id}/
│           ├── original.pdf      # Original PDF
│           ├── chunks.json       # Extracted chunks
│           ├── embeddings.npy    # Embeddings (legacy)
│           └── knowledge_graph/  # KG files
│               ├── entities.json
│               └── knowledge_graph.json
├── docs/                         # Documentation
│   ├── DATABASE_SCHEMA.md
│   └── [this guide]
├── evaluation/                   # Evaluation scripts
│   ├── evaluate_rag_comparison.py
│   ├── complex_questions.json
│   └── evaluation_results.json
├── fresh_venv/                   # Python virtual environment
├── requirements.txt              # Python dependencies
└── README.md
```

8.2 Key File Descriptions

8.2.1 src/app.py

The main Flask application. Contains: - Route definitions for all API endpoints - Request parsing and validation - Response streaming logic - Error handling

Size: ~800 lines

Key Functions: - `list_documents()`: GET /api/documents - `upload_document()`: POST /api/upload - `ask_question()`: POST /api/ask - `generate()`: Streaming response generator

8.2.2 src/database/connection.py

Database connection management.

Key Components: - `engine`: SQLAlchemy engine with connection pooling - `SessionLocal`: Scoped session factory - `get_db()`: Context manager for sessions - `test_connection()`: Health check

8.2.3 src/utils/vector_rag.py

Vector-based retrieval implementation.

Key Methods: - `__init__(doc_folder, document_id)`: Initialize for a document - `query(question, top_k)`: Find similar chunks - `embed_query(text)`: Generate query embedding

8.2.4 src/utils/kg_pipeline.py

Knowledge graph extraction pipeline.

Key Methods: - `extract_definitions(chunk)`: Stage 1 - `extract_entities(chunk)`: Stage 2 - `extract_relationships(chunk)`: Stage 3 - `extract_events(chunk)`: Stage 4 - `resolve_entities(entities)`: Stage 5 - `process_document(chunks)`: Full pipeline

8.2.5 scripts/build_kg_v2.py

CLI script for building KG.

Usage:

```
python scripts/build_kg_v2.py --document policybazar_ipo
python scripts/build_kg_v2.py --document policybazar_ipo --limit 10
```

---

9. How Components Work Together

9.1 Request Lifecycle: Ask Question

Let's trace a request from user to answer:

```
Timeline:
```

```
T+0ms     User types "Who is the CEO?" and presses Enter
T+1ms     JavaScript captures event, calls sendMessage()
T+2ms     fetch() starts POST to /api/ask
T+10ms    Request reaches Flask, parsed
T+15ms    Check KG availability for document
T+20ms    Initialize VectorRAG if needed
T+25ms    Initialize KGRAG if available
T+30ms    Start streaming response
T+35ms    yield: {"type": "status", "msg": "Searching..."}
T+50ms    Embed question (embedding model)
```

```
T+100ms   Query PostgreSQL vectors (pgvector)
T+120ms   Query KG entities (if available)
T+150ms   Assemble context from both sources
T+160ms   Build prompt
T+170ms   Start Ollama generation
T+200ms   yield: {"type": "token", "content": "The"}
T+230ms   yield: {"type": "token", "content": " CEO"}
T+260ms   yield: {"type": "token", "content": " is"}
...
T+2000ms  yield: {"type": "done"}
T+2001ms  JavaScript updates UI
```

9.2 Data Consistency

Transaction Boundaries

Document Upload: Single transaction

```python
with get_db() as db:
    # All or nothing
    doc = create_document(...)
    chunks = create_chunks(...)
    embeddings = create_embeddings(...)
    db.commit()  # Only if all succeed
```

KG Building: Chunk-level transactions

```python
for chunk in chunks:
    with get_db() as db:
        # Each chunk independent
        entities = extract_and_save_entities(chunk)
        db.commit()
```

Error Recovery

If KG building fails mid-way: - Already saved entities remain - Can resume from last successful chunk - Validation report tracks issues

---

10. Performance Characteristics

10.1 Benchmarks

Operation	Time	Details
PDF extraction	~5s	400-page PDF
Chunking	<1s	816 chunks
Embedding generation	~10s	816 chunks, CPU
Vector search	~50ms	Top-5, HNSW index
KG entity lookup	~20ms	Single entity
LLM generation	2-5s	100-word answer
Full query (hybrid)	3-7s	End-to-end

10.2 Scalability Limits

Component	Current	Tested Up To	Theoretical Limit
Documents	4	100	10,000+
Chunks per doc	816	2,000	100,000+
Total embeddings	3,264	200,000	Millions

10.3 Memory Usage

Component	RAM Usage
Flask app	~200MB
Embedding model	~300MB
Ollama (llama3)	~6GB
PostgreSQL	~1GB
Total	~8GB

---

11. Security Considerations

11.1 Data Privacy

All data stays local: - PDFs stored on local filesystem - Database is local PostgreSQL - LLM runs locally via Ollama - No external API calls for sensitive data

11.2 Input Validation

File upload: - File type checking (PDF only) - File size limits - Hash deduplication

Query input: - Maximum question length - Rate limiting (can be added)

11.3 Future Improvements

- Add authentication layer
- Encrypt stored documents
- Audit logging
- Role-based access control

---

12. Future Roadmap

12.1 Short Term (1-3 months)

1. **Better table extraction**: Use specialized tools for financial tables
2. **Query routing**: Automatically choose Vector/KG/Hybrid
3. **Caching**: Cache frequent queries
4. **UI improvements**: Better visualization of entity graph

12.2 Medium Term (3-6 months)

1. **Multi-document queries**: Compare across IPOs
2. **Automated evaluation**: Continuous quality testing
3. **Reranking**: Cross-encoder for better results
4. **User feedback loop**: Learn from corrections

12.3 Long Term (6-12 months)

1. **Multi-modal**: Extract from images and charts
2. **Trend analysis**: Track metrics over time
3. **Predictions**: Suggest important sections to read
4. **Export**: Generate reports/summaries

Summary

This chapter covered:

1. **What the platform does**: Document intelligence for IPO prospectuses
2. **Why it was built**: Solve information overload, provide accurate answers
3. **RAG explained**: Retrieval + Augmentation + Generation
4. **Why hybrid**: Combined strengths of Vector and KG
5. **Architecture**: Flask + PostgreSQL + Ollama
6. **Tech stack**: Why each technology was chosen
7. **Design decisions**: Trade-offs we made
8. **Project structure**: Where everything lives
9. **How it works**: Request lifecycle, data flow
10. **Performance**: Benchmarks and limits
11. **Security**: Privacy considerations
12. **Future**: What's coming next

Chapter 2: PDF Ingestion & Chunking

A Complete Guide to Document Processing in the IPO Intelligence Platform

---

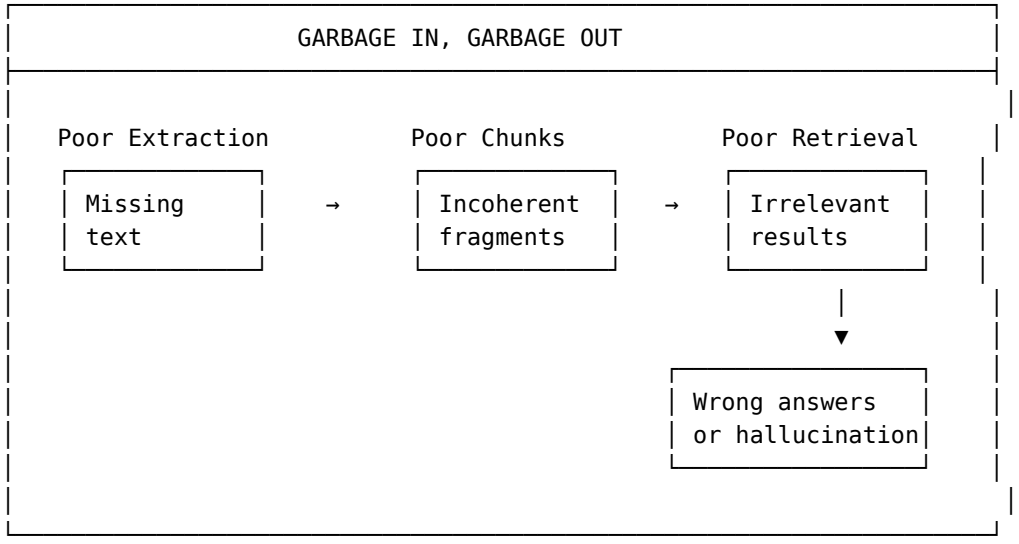
Table of Contents for This Chapter

---

1. Introduction to Document Processing

1.1 Why Document Processing Matters

Document processing is the **foundation** of any RAG system. The quality of your retrieval and generation is directly limited by the quality of your document processing.



The Document Processing Pipeline
Our document processing has three main stages:

1. **Extraction**: Get raw text from PDF
2. **Chunking**: Split into optimal-sized pieces
3. **Enrichment**: Add metadata for retrieval

Each stage has its own challenges and solutions.

1.2 IPO Documents: Specific Challenges

IPO prospectuses (DRHPs) have unique characteristics that make them challenging:

1.2.1 Document Length

Document	Pages	Words	Chunks (at 800 tokens)
PolicyBazaar DRHP	431	~200,000	816
Zomato DRHP	448	~225,000	900+
Nykaa DRHP	489	~250,000	1,000+

These are **massive** documents. A typical novel is 80,000 words.

1.2.2 Structure Complexity

IPO documents have: - **Fixed sections** mandated by SEBI - **Multiple levels** of headers - **Tables** with financial data - **Footnotes** with important details - **Legal language** with precise definitions

1.2.3 Content Types

Content Type	Challenge	Example
Narrative text	Long paragraphs	Business description
Tables	Positional layout lost	Financial statements
Lists	Bullet points scattered	Risk factors
Definitions	Cross-references	“Company” means “Issuer”
Headers	Hierarchy important	Chapter > Section > Subsection

---

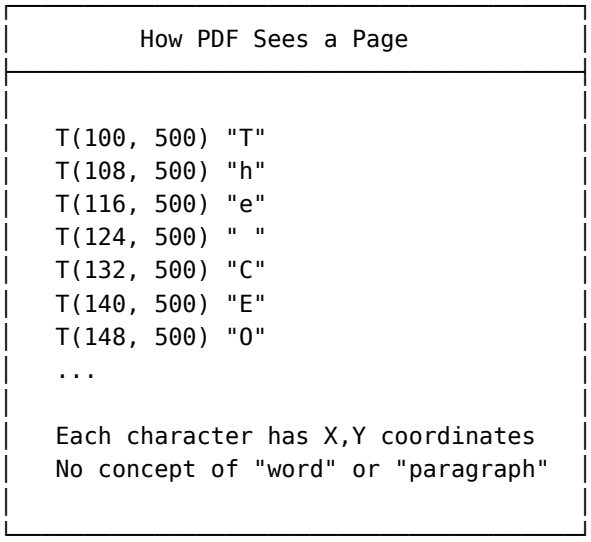
2. Understanding PDF Structure

2.1 What is a PDF?

PDF (Portable Document Format) is designed for **displaying** documents consistently across devices. It is **not** designed for data extraction.

2.1.1 How PDFs Store Text

PDFs store text as positioned characters, not as semantic paragraphs:



2.1.2 Text Extraction Challenge

To get readable text, we must: 1. Group characters into words (by proximity) 2. Group words into lines (by Y coordinate) 3. Group lines into paragraphs (by spacing) 4. Handle multi-column layouts 5. Handle headers/footers

This is why different PDF libraries give different results.

2.2 PDF Library Comparison

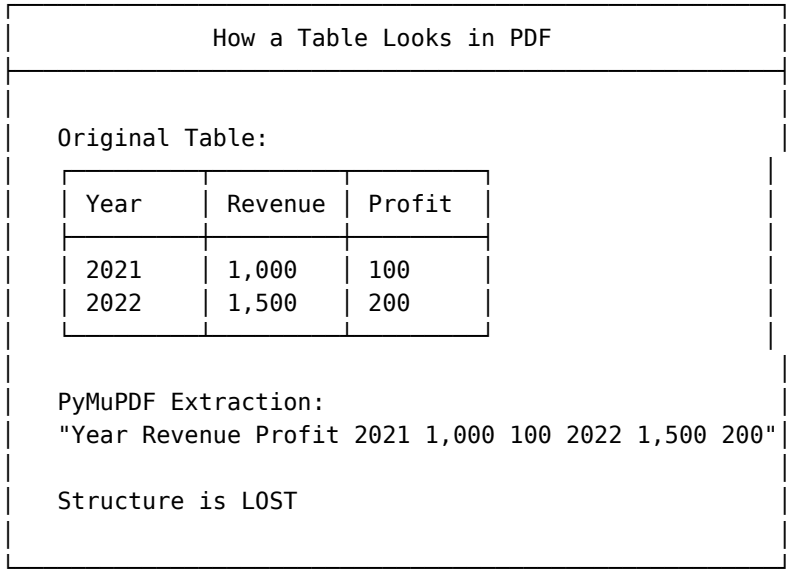
Library	Speed	Quality	Table Support	Python API
PyMuPDF	⚡⚡⚡	✅ Good	⚠️ Basic	✅ Easy
pdfplumber	⚡	✅ Good	✅ Better	✅ Easy
PyPDF2	⚡⚡⚡	❌ Poor	❌ None	✅ Easy
pdfminer	⚡	✅ Good	⚠️ Basic	⚠️ Complex
Camelot	🐢	✅ Excellent	✅✅ Best	✅ Easy
Tabula	🐢	✅ Good	✅ Good	⚠️ Java needed

2.2.1 Why We Chose PyMuPDF

1. **Speed**: 10-50x faster than pdfplumber for large documents
2. **Quality**: Good text ordering and grouping
3. **Memory efficient**: Handles 500+ page documents
4. **Pure Python**: No Java or native dependencies
5. **Well maintained**: Active development

2.2.2 Trade-off: Tables

PyMuPDF treats tables as positioned text, losing structure:



This is a known limitation. For high-stakes table extraction, we would use Camelot or custom parsing.

---

3. Text Extraction with PyMuPDF

3.1 Basic Usage

```python
import fitz  # PyMuPDF is imported as 'fitz'

def extract_text_basic(pdf_path: str) -> str:
    """Extract all text from a PDF."""
    doc = fitz.open(pdf_path)
    all_text = ""

    for page in doc:
        text = page.get_text()
        all_text += text + "\n"

    doc.close()
    return all_text
```

This gives raw text, but we need more structure.

3.2 Page-by-Page Extraction

```python
from typing import List, Dict
import fitz
```

```python
def extract_pages(pdf_path: str) -> List[Dict]:
    """Extract text page by page with metadata."""
    doc = fitz.open(pdf_path)
    pages = []

    for page_num in range(len(doc)):
        page = doc[page_num]
        text = page.get_text("text")  # Plain text

        pages.append({
            'page_number': page_num + 1,  # 1-indexed
            'text': text,
            'word_count': len(text.split()),
            'char_count': len(text)
        })

    doc.close()
    return pages
```

Output Format

```
[
    {
        'page_number': 1,
        'text': 'DRAFT RED HERRING PROSPECTUS\nDated July 31, 2021...',
        'word_count': 245,
        'char_count': 1523
    },
    {
        'page_number': 2,
        'text': 'TABLE OF CONTENTS\n\nSection I...',
        'word_count': 312,
        'char_count': 1890
    },
    # ... more pages
]
```

3.3 Extraction Options

PyMuPDF offers several extraction modes:

Mode	Code	Output	Use Case
Text	`get_text("text")`	Plain string	General use
Blocks	`get_text("blocks")`	List of text blocks	Paragraph detection
Dict	`get_text("dict")`	Full structure	Layout analysis
HTML	`get_text("html")`	HTML format	Web display
XHTML	`get_text("xhtml")`	XHTML format	Structured processing

Block Extraction

Blocks give us paragraph-level information:

```python
def extract_blocks(pdf_path: str, page_num: int) -> List[Dict]:
    """Extract text blocks from a page."""
    doc = fitz.open(pdf_path)
    page = doc[page_num]

    blocks = page.get_text("blocks")
    # Each block: (x0, y0, x1, y1, text, block_no, block_type)

    text_blocks = []
    for block in blocks:
        if block[6] == 0:  # Type 0 = text (not image)
            text_blocks.append({
                'text': block[4],
                'bbox': {
                    'x0': block[0],
                    'y0': block[1],
                    'x1': block[2],
                    'y1': block[3]
                },
                'block_number': block[5]
            })

    doc.close()
    return text_blocks
```

3.4 Handling Headers and Footers

IPO documents have repeating headers/footers on every page:

```
┌─────────────────────────────────────────┐
│  DRAFT RED HERRING PROSPECTUS           │  ← Header (repeats)
│                                         │
│  [Page content here]                    │
│                                         │
│  Page 45                                │  ← Footer (repeats)
└─────────────────────────────────────────┘
```

Detection Strategy

```python
def detect_header_footer(pages: List[Dict]) -> Dict:
    """Detect repeating headers and footers."""

    # Get first 3 lines from each page
    first_lines = []
    last_lines = []

    for page in pages:
        lines = page['text'].split('\n')
        if len(lines) >= 3:
```

```
            first_lines.append('\n'.join(lines[:3]))
            last_lines.append('\n'.join(lines[-3:]))

    # Find most common first/last lines
    from collections import Counter

    header_counts = Counter(first_lines)
    footer_counts = Counter(last_lines)

    # If >50% of pages have same header/footer, it's repeating
    threshold = len(pages) * 0.5

    header = None
    footer = None

    most_common_header = header_counts.most_common(1)
    if most_common_header and most_common_header[0][1] > threshold:
        header = most_common_header[0][0]

    most_common_footer = footer_counts.most_common(1)
    if most_common_footer and most_common_footer[0][1] > threshold:
        footer = most_common_footer[0][0]

    return {'header': header, 'footer': footer}
```

Removing Headers/Footers

```
def remove_header_footer(text: str, header: str, footer: str) -> str:
    """Remove detected header and footer from page text."""
    if header:
        text = text.replace(header, '', 1)  # Remove first occurrence
    if footer:
        # Remove last occurrence
        idx = text.rfind(footer)
        if idx != -1:
            text = text[:idx] + text[idx + len(footer):]
    return text.strip()
```

3.5 Text Cleaning

Raw PDF text needs cleaning:

```
import re

def clean_text(text: str) -> str:
    """Clean extracted text."""

    # 1. Fix encoding issues
    text = text.encode('utf-8', errors='ignore').decode('utf-8')

    # 2. Normalize whitespace
```

```
    text = re.sub(r'\s+', ' ', text)  # Multiple spaces → single
    text = re.sub(r'\n\s*\n', '\n\n', text)  # Normalize paragraph breaks

    # 3. Remove control characters
    text = re.sub(r'[\x00-\x08\x0b\x0c\x0e-\x1f\x7f]', '', text)

    # 4. Fix common OCR/extraction errors
    text = text.replace('ﬁ', 'fi')  # Ligature
    text = text.replace('ﬂ', 'fl')
    text = text.replace('—', '-')  # Em dash
    text = text.replace('–', '-')  # En dash
    text = text.replace('"', '"')  # Smart quotes
    text = text.replace('"', '"')
    text = text.replace(''', "'")
    text = text.replace(''', "'")

    # 5. Fix hyphenation at line breaks
    text = re.sub(r'-\s*\n\s*', '', text)  # "hyphen-\nated" → "hyphenated"

    return text.strip()
```

3.6 Complete Extraction Function

```
from typing import List, Dict, Optional
import fitz
import re
import hashlib

def extract_document(pdf_path: str) -> Dict:
    """
    Complete document extraction pipeline.

    Returns:
        {
            'filename': str,
            'file_hash': str,
            'total_pages': int,
            'total_words': int,
            'pages': List[Dict]
        }
    """

    # Calculate file hash for deduplication
    with open(pdf_path, 'rb') as f:
        file_hash = hashlib.md5(f.read()).hexdigest()

    # Open document
    doc = fitz.open(pdf_path)

    # Extract all pages first
```

```
raw_pages = []
for page_num in range(len(doc)):
    page = doc[page_num]
    text = page.get_text("text")
    raw_pages.append({
        'page_number': page_num + 1,
        'raw_text': text
    })

doc.close()

# Detect headers/footers
header_footer = detect_header_footer(raw_pages)

# Process each page
processed_pages = []
total_words = 0

for page in raw_pages:
    # Remove header/footer
    text = remove_header_footer(
        page['raw_text'],
        header_footer.get('header'),
        header_footer.get('footer')
    )

    # Clean text
    text = clean_text(text)

    word_count = len(text.split())
    total_words += word_count

    processed_pages.append({
        'page_number': page['page_number'],
        'text': text,
        'word_count': word_count
    })

return {
    'filename': pdf_path.split('/')[-1],
    'file_hash': file_hash,
    'total_pages': len(processed_pages),
    'total_words': total_words,
    'pages': processed_pages
}
```

---

4. Chunking: The Foundation of RAG

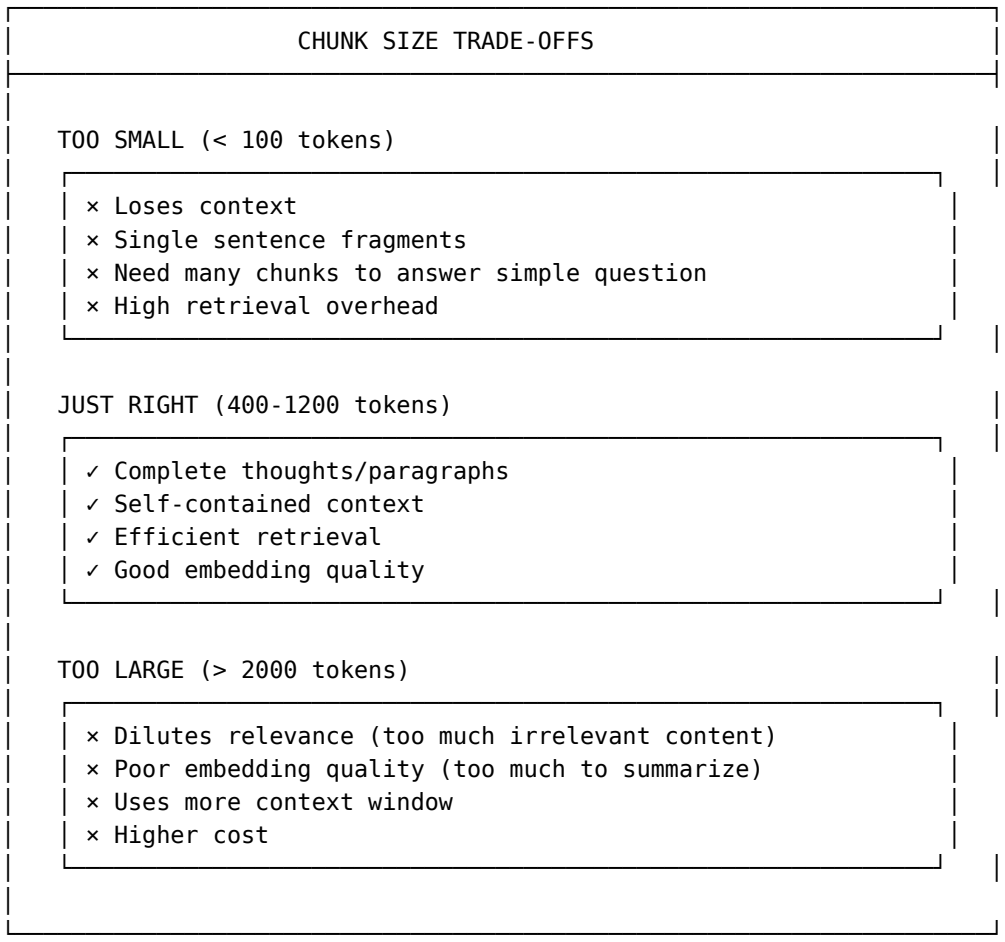
4.1 Why Chunking Matters

Chunking is the process of splitting a document into smaller pieces for retrieval. It's **critically important** because:

1. **Context limits**: LLMs can only process so much text at once
2. **Retrieval precision**: Smaller chunks = more precise matching
3. **Cost efficiency**: Less tokens = cheaper LLM calls
4. **Relevance**: Right-sized chunks contain focused information

4.2 The Goldilocks Problem

Chunks must be **just right**:



4.3 Chunking Strategies

4.3.1 Fixed-Size Chunking

Split every N tokens regardless of content:

```python
def fixed_size_chunk(text: str, chunk_size: int = 500) -> List[str]:
    """Split text into fixed-size chunks."""
    words = text.split()
    chunks = []

    for i in range(0, len(words), chunk_size):
        chunk = ' '.join(words[i:i + chunk_size])
        chunks.append(chunk)

    return chunks
```

Pros: Simple, consistent sizes **Cons**: Breaks mid-sentence, ignores document structure

4.3.2 Recursive Character Splitting

Split on hierarchical separators:

```python
def recursive_split(text: str, max_size: int = 1000) -> List[str]:
    """Split text recursively on paragraph, then sentence, then word."""

    if len(text.split()) <= max_size:
        return [text]

    # Try paragraph split first
    paragraphs = text.split('\n\n')
    if len(paragraphs) > 1:
        chunks = []
        current = ""
        for para in paragraphs:
            if len((current + para).split()) <= max_size:
                current += para + "\n\n"
            else:
                if current:
                    chunks.append(current.strip())
                current = para + "\n\n"
        if current:
            chunks.append(current.strip())
        return chunks

    # Fall back to sentence split
    sentences = re.split(r'(?<=[.!?])\s+', text)
    # ... similar logic
```

Pros: Respects natural boundaries **Cons**: Variable sizes, complex implementation

4.3.3 Semantic Chunking

Group semantically related content:

```python
def semantic_chunk(text: str, model) -> List[str]:
    """Split based on semantic similarity between adjacent parts."""

    sentences = split_into_sentences(text)
```

```
    embeddings = model.encode(sentences)

    chunks = []
    current_chunk = [sentences[0]]

    for i in range(1, len(sentences)):
        # Compare with previous sentence
        similarity = cosine_similarity(embeddings[i-1], embeddings[i])

        if similarity > 0.5:  # Similar → same chunk
            current_chunk.append(sentences[i])
        else:  # Different topic → new chunk
            chunks.append(' '.join(current_chunk))
            current_chunk = [sentences[i]]

    if current_chunk:
        chunks.append(' '.join(current_chunk))

    return chunks
```

Pros: Best coherence, topic-based **Cons**: Requires embedding calls, slower, variable sizes

4.3.4 Section-Based Chunking (Our Approach)

Respect document structure, use sections as natural boundaries:

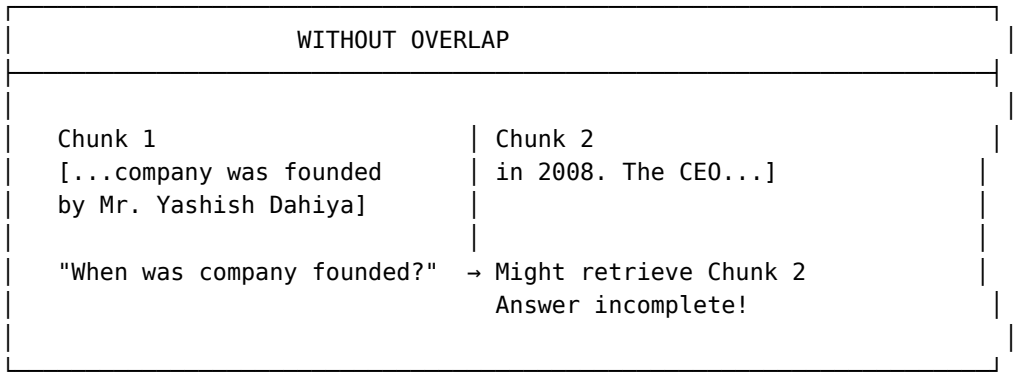
```
def section_based_chunk(
    text: str,
    section_patterns: List[str],
    max_size: int = 1000,
    overlap: int = 100
) -> List[Dict]:
    """
    Chunk based on document sections, then split large sections.
    """
    # First, detect sections
    sections = detect_sections(text, section_patterns)

    chunks = []
    for section in sections:
        if len(section['text'].split()) <= max_size:
            # Small section → single chunk
            chunks.append({
                'text': section['text'],
                'section': section['title'],
                'start_page': section['start_page']
            })
        else:
            # Large section → split into multiple chunks
            sub_chunks = split_with_overlap(
                section['text'],
```

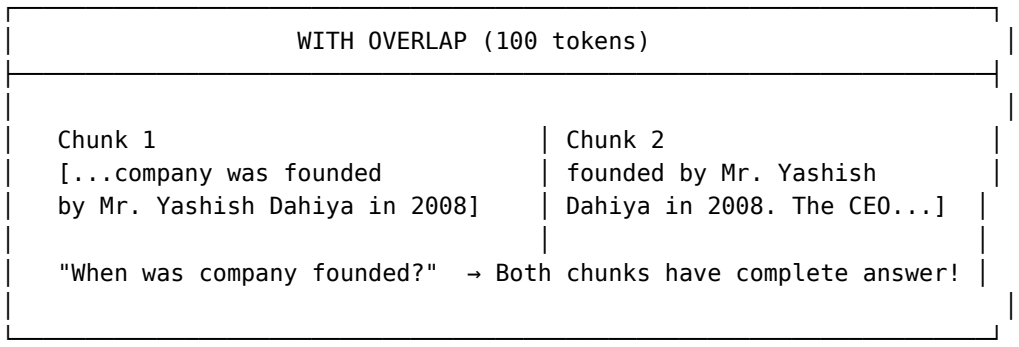
```
                max_size,
                overlap
            )
            for i, sub in enumerate(sub_chunks):
                chunks.append({
                    'text': sub,
                    'section': section['title'],
                    'sub_section': i + 1,
                    'start_page': section['start_page']
                })

    return chunks
```

Pros: Preserves document structure, natural boundaries **Cons**: Requires section detection, document-specific

4.4 Overlap: Why It Matters

Overlap ensures we don't lose context at chunk boundaries:





Implementing Overlap

```
def split_with_overlap(
    text: str,
    max_tokens: int = 800,
    overlap_tokens: int = 100
) -> List[str]:
```

```
    """Split text with overlapping regions."""

    words = text.split()
    chunks = []

    start = 0
    while start < len(words):
        end = start + max_tokens

        # Don't exceed text length
        end = min(end, len(words))

        chunk = ' '.join(words[start:end])
        chunks.append(chunk)

        # Move start, but leave overlap
        start = end - overlap_tokens

        # Prevent infinite loop
        if start >= len(words) - overlap_tokens:
            break

    return chunks
```

---

5. Our Chunking Implementation

5.1 Overview

Our chunking uses a hybrid approach:

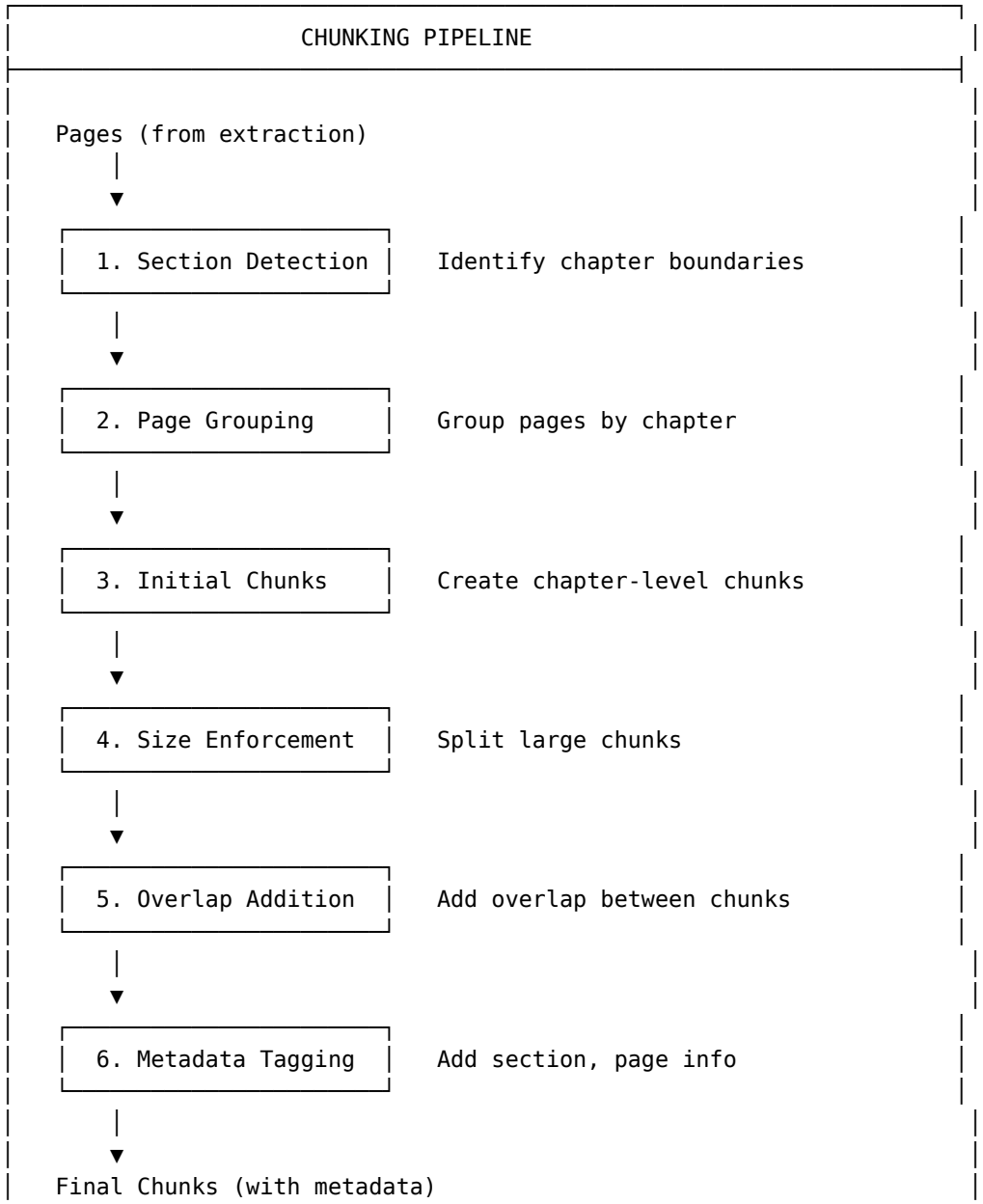
1. **Detect chapters/sections** from table of contents
2. **Split pages** while tracking section membership
3. **Group pages** into chapter-based chunks
4. **Split large chunks** with overlap
5. **Preserve metadata** for retrieval

5.2 Configuration Parameters

```
CHUNKING_CONFIG = {
    # Token limits
    'min_chunk_tokens': 200,      # Don't create tiny chunks
    'max_chunk_tokens': 1200,     # Upper limit
    'target_chunk_tokens': 800,   # Ideal size

    # Overlap
    'overlap_tokens': 100,        # Words of overlap

    # Section detection
    'section_patterns': [
```

```
        r'^SECTION\s+[IVX]+',      # SECTION I, SECTION II
        r'^CHAPTER\s+\d+',         # CHAPTER 1, CHAPTER 2
        r'^[A-Z][A-Z\s]+$',        # ALL CAPS HEADERS
    ],

    # Content filters
    'min_words_per_page': 20,      # Skip nearly empty pages
}
```

5.3 The Chunking Pipeline



```
┌                                                                                ┐
└                                                                                ┘
```

5.4 Complete Implementation

```python
from typing import List, Dict, Optional, Tuple
import re
from dataclasses import dataclass

@dataclass
class Chunk:
    """Represents a document chunk with metadata."""
    text: str
    chunk_index: int
    page_start: int
    page_end: int
    section_title: Optional[str]
    word_count: int

    def to_dict(self) -> Dict:
        return {
            'text': self.text,
            'chunk_index': self.chunk_index,
            'page_start': self.page_start,
            'page_end': self.page_end,
            'section_title': self.section_title,
            'word_count': self.word_count
        }


class DocumentChunker:
    """
    Chunks documents for RAG retrieval.

    Strategy:
    1. Respect section boundaries when possible
    2. Split large sections with overlap
    3. Preserve page and section metadata
    """

    def __init__(
        self,
        target_tokens: int = 800,
        max_tokens: int = 1200,
        min_tokens: int = 200,
        overlap_tokens: int = 100
    ):
        self.target_tokens = target_tokens
        self.max_tokens = max_tokens
        self.min_tokens = min_tokens
```

```
        self.overlap_tokens = overlap_tokens

        # Patterns for section detection
        self.section_patterns = [
            r'^SECTION\s+[IVXLCDM]+',
            r'^PART\s+[IVXLCDM]+',
            r'^CHAPTER\s+\d+',
            r'^[A-Z][A-Z\s]{10,}$',  # All caps headers (10+ chars)
        ]

    def _detect_section(self, text: str) -> Optional[str]:
        """Detect if text starts with a section header."""
        first_line = text.strip().split('\n')[0] if text else ''

        for pattern in self.section_patterns:
            if re.match(pattern, first_line.strip()):
                return first_line.strip()

        return None

    def _count_tokens(self, text: str) -> int:
        """Approximate token count (words as proxy)."""
        return len(text.split())

    def _split_with_overlap(self, text: str) -> List[str]:
        """Split large text into chunks with overlap."""
        words = text.split()
        total_words = len(words)

        if total_words <= self.max_tokens:
            return [text]

        chunks = []
        start = 0

        while start < total_words:
            end = start + self.target_tokens

            # Try to end at sentence boundary
            chunk_words = words[start:min(end + 50, total_words)]
            chunk_text = ' '.join(chunk_words)

            # Find last sentence end within target
            sentence_end = self._find_sentence_end(chunk_text,
self.target_tokens)
            if sentence_end:
                chunk_text = chunk_text[:sentence_end]
                end = start + len(chunk_text.split())
```

```
            chunks.append(chunk_text.strip())

            # Move start with overlap
            start = max(0, end - self.overlap_tokens)

            # Safety: prevent infinite loop
            if end >= total_words:
                break

        return chunks

    def _find_sentence_end(self, text: str, max_words: int) -> Optional[int]:
        """Find the last sentence ending within word limit."""
        words_so_far = 0
        last_sentence_end = None

        for i, char in enumerate(text):
            if char == ' ':
                words_so_far += 1

            if char in '.!?' and words_so_far <= max_words:
                last_sentence_end = i + 1

        return last_sentence_end

    def chunk_document(self, pages: List[Dict]) -> List[Chunk]:
        """
        Chunk a document into retrieval-ready pieces.

        Args:
            pages: List of {'page_number': int, 'text': str}

        Returns:
            List of Chunk objects
        """
        chunks = []
        current_text = ""
        current_section = None
        current_page_start = 1
        chunk_index = 0

        for page in pages:
            page_num = page['page_number']
            page_text = page['text']

            # Skip nearly empty pages
            if self._count_tokens(page_text) < 20:
                continue
```

```
            # Check for new section
            detected_section = self._detect_section(page_text)

            # If new section, save current chunk first
            if detected_section and current_text:
                # Create chunk(s) from accumulated text
                sub_chunks = self._split_with_overlap(current_text)
                for sub in sub_chunks:
                    if self._count_tokens(sub) >= self.min_tokens:
                        chunks.append(Chunk(
                            text=sub,
                            chunk_index=chunk_index,
                            page_start=current_page_start,
                            page_end=page_num - 1,
                            section_title=current_section,
                            word_count=self._count_tokens(sub)
                        ))
                        chunk_index += 1

                # Reset for new section
                current_text = ""
                current_section = detected_section
                current_page_start = page_num

            elif detected_section:
                current_section = detected_section
                current_page_start = page_num

            # Accumulate page text
            current_text += "\n\n" + page_text

            # Check if we should split (approaching max)
            if self._count_tokens(current_text) >= self.max_tokens:
                sub_chunks = self._split_with_overlap(current_text)
                for i, sub in enumerate(sub_chunks[:-1]):  # Keep last for
continuation
                    if self._count_tokens(sub) >= self.min_tokens:
                        chunks.append(Chunk(
                            text=sub,
                            chunk_index=chunk_index,
                            page_start=current_page_start,
                            page_end=page_num,
                            section_title=current_section,
                            word_count=self._count_tokens(sub)
                        ))
                        chunk_index += 1

                # Keep remainder for next iteration
                current_text = sub_chunks[-1] if sub_chunks else ""
```

```
            current_page_start = page_num

    # Don't forget last chunk
    if current_text and self._count_tokens(current_text) >= self.min_tokens:
        sub_chunks = self._split_with_overlap(current_text)
        for sub in sub_chunks:
            if self._count_tokens(sub) >= self.min_tokens:
                chunks.append(Chunk(
                    text=sub,
                    chunk_index=chunk_index,
                    page_start=current_page_start,
                    page_end=pages[-1]['page_number'],
                    section_title=current_section,
                    word_count=self._count_tokens(sub)
                ))
                chunk_index += 1

    return chunks
```

---

6. Section and Chapter Detection

6.1 Why Section Detection Matters

Sections provide critical metadata:

1. **Better retrieval**: Filter by section type
2. **Context understanding**: "This is from Risk Factors" helps interpretation
3. **KG extraction**: Different sections need different prompts
4. **User experience**: Show section name with answers

6.2 IPO Document Sections

SEBI-mandated DRHP structure:

Section	Typical Content
Definitions	All defined terms
Overview	Company summary
Risk Factors	50+ pages of risks
Use of Proceeds	How IPO money will be used
Industry Overview	Market analysis
Our Business	Detailed business description
Financial Information	3-5 years of financials
Management	Directors, officers
Offer Details	Pricing, structure
Legal Information	Litigations, compliance

6.3 Detection Patterns

```
SECTION_PATTERNS = {
    'definitions': [
        r'^DEFINITIONS\s*$',
        r'^DEFINITIONS AND ABBREVIATIONS',
        r'^GLOSSARY',
    ],
    'risk_factors': [
        r'^RISK FACTORS',
        r'^SECTION\s+II.*RISK FACTORS',
    ],
    'business': [
        r'^OUR BUSINESS',
        r'^BUSINESS OVERVIEW',
        r'^DESCRIPTION OF.*BUSINESS',
    ],
    'financials': [
        r'^FINANCIAL INFORMATION',
        r'^RESTATED\s+.*FINANCIAL',
        r'^FINANCIAL STATEMENTS',
    ],
    'management': [
        r'^OUR MANAGEMENT',
        r'^BOARD OF DIRECTORS',
        r'^KEY MANAGERIAL PERSONNEL',
    ],
    'offer': [
        r'^OFFER STRUCTURE',
        r'^THE OFFER',
        r'^OBJECTS OF THE OFFER',
    ],
```

```
    'legal': [
        r'^OUTSTANDING LITIGATION',
        r'^LEGAL PROCEEDINGS',
        r'^MATERIAL CONTRACTS',
    ],
}

def classify_section(header: str) -> str:
    """Classify a section header into a category."""
    header_clean = header.strip().upper()

    for category, patterns in SECTION_PATTERNS.items():
        for pattern in patterns:
            if re.match(pattern, header_clean):
                return category

    return 'other'
```

6.4 Table of Contents Parsing

For better section detection, parse the TOC:

```
def parse_toc(toc_text: str) -> List[Dict]:
    """Parse table of contents to get section page numbers."""

    sections = []

    # Pattern: "SECTION NAME ... 45" or "SECTION NAME ..... 45"
    pattern = r'^(.+?)\s*\.{2,}\s*(\d+)\s*$'

    for line in toc_text.split('\n'):
        match = re.match(pattern, line.strip())
        if match:
            sections.append({
                'title': match.group(1).strip(),
                'page': int(match.group(2))
            })

    return sections

# Example output:
# [
#     {'title': 'DEFINITIONS', 'page': 1},
#     {'title': 'RISK FACTORS', 'page': 23},
#     {'title': 'OUR BUSINESS', 'page': 89},
#     ...
# ]
```

---

7. Metadata Preservation

7.1 What Metadata to Preserve

Metadata	Purpose	Example
page_number	Source attribution	"Page 45"
section_title	Context	"Risk Factors"
chunk_index	Ordering	42
word_count	Statistics	856
page_range	Multi-page chunks	"Pages 45-47"
document_id	Document reference	"policybazar_ipo"

7.2 Metadata Storage Format

```
# Each chunk includes:
{
    "chunk_index": 42,
    "text": "The company was incorporated in 2008...",
    "page_start": 45,
    "page_end": 47,
    "section_title": "Our Business",
    "word_count": 856,
    "metadata": {
        "chapter_name": "OUR BUSINESS",
        "estimated_page": 46,
        "has_table": False,
        "extraction_version": "2.0"
    }
}
```

7.3 Using Metadata in Retrieval

```
def search_with_metadata(
    query: str,
    document_id: str,
    section_filter: Optional[str] = None,
    page_range: Optional[Tuple[int, int]] = None
) -> List[Dict]:
    """Search with metadata filters."""

    # Base vector search
    query_vec = embed(query)

    # Build SQL with filters
    sql = """
        SELECT c.*, 1 - (e.embedding <=> :query_vec) as score
        FROM chunks c
```

```
        JOIN embeddings e ON e.chunk_id = c.id
        WHERE c.document_id = :doc_id
    """
    params = {'query_vec': query_vec, 'doc_id': document_id}

    if section_filter:
        sql += " AND c.section_title ILIKE :section"
        params['section'] = f"%{section_filter}%"

    if page_range:
        sql += " AND c.page_start >= :start AND c.page_end <= :end"
        params['start'] = page_range[0]
        params['end'] = page_range[1]

    sql += " ORDER BY e.embedding <=> :query_vec LIMIT 5"

    return execute(sql, params)
```

---

8. Handling Edge Cases

8.1 Empty or Near-Empty Pages

Some pages are nearly empty (just page numbers, headers):

```
def is_meaningful_page(page: Dict, min_words: int = 20) -> bool:
    """Check if page has meaningful content."""
    text = page.get('text', '')
    word_count = len(text.split())

    # Also check for non-whitespace content
    meaningful_chars = len(re.sub(r'\s', '', text))

    return word_count >= min_words and meaningful_chars >= 50
```

8.2 Tables

Tables are extracted as linear text, losing structure:

```
def detect_table_region(text: str) -> bool:
    """Heuristically detect if text contains a table."""

    # Tables often have:
    # - Aligned numbers
    # - Short lines (cell content)
    # - Currency symbols in patterns

    lines = text.split('\n')

    # Check for numeric patterns
    numeric_lines = sum(1 for line in lines if re.search(r'[\d,]+\.\d{2}', line))
```

```
    # Check for short average line length (table cells)
    avg_line_length = sum(len(l) for l in lines) / max(len(lines), 1)

    return numeric_lines > 5 and avg_line_length < 50
```

Table Handling Strategy

For now, we include table text as-is. Future improvement: use Camelot/Tabula.

8.3 Multi-Column Layouts

Some pages have two-column layouts:

```
┌─────────────────────────────────────────┐
│  Column 1       │      Column 2         │
│                 │                       │
│  First para     │      Third para       │
│                 │                       │
│  Second para    │      Fourth para      │
│                 │                       │
└─────────────────────────────────────────┘
```

PyMuPDF may read: "First para Third para Second para Fourth para"

Detection and Handling

```
def might_be_multicolumn(text: str) -> bool:
    """Detect potential multi-column extraction issues."""
    lines = text.split('\n')

    # Multi-column often results in short lines
    short_lines = sum(1 for l in lines if 10 < len(l) < 40)

    return short_lines > len(lines) * 0.7

# For now: flag these pages for manual review
# Future: use layout-aware extraction
```

8.4 Repeated Content

Page headers/footers may slip through:

```
def remove_repeated_content(chunks: List[Dict]) -> List[Dict]:
    """Remove content that appears in many chunks (likely header/footer)."""

    # Find lines that appear in >50% of chunks
    from collections import Counter

    all_lines = []
    for chunk in chunks:
        lines = chunk['text'].split('\n')
        all_lines.extend(lines)
```

```
    line_counts = Counter(all_lines)
    threshold = len(chunks) * 0.5

    repeated = {line for line, count in line_counts.items() if count > threshold}

    # Remove repeated lines
    for chunk in chunks:
        lines = chunk['text'].split('\n')
        cleaned_lines = [l for l in lines if l not in repeated]
        chunk['text'] = '\n'.join(cleaned_lines)

    return chunks
```

---

9. The Complete Ingestion Pipeline

9.1 Pipeline Overview

```
from typing import Dict, List
import hashlib
import json
import os

class DocumentIngestionPipeline:
    """
    Complete document ingestion: PDF → Chunks → Embeddings → Storage
    """

    def __init__(
        self,
        data_dir: str,
        embedding_model: str = 'all-MiniLM-L6-v2'
    ):
        self.data_dir = data_dir
        self.chunker = DocumentChunker()
        self.embedder = SentenceTransformer(embedding_model)

    def ingest(self, pdf_path: str, document_id: str) -> Dict:
        """
        Full ingestion pipeline.

        Returns:
            {
                'document_id': str,
                'total_pages': int,
                'total_chunks': int,
                'file_hash': str
            }
        """
```

```
        # Step 1: Check for duplicates
        file_hash = self._calculate_hash(pdf_path)
        if self._is_duplicate(file_hash):
            raise ValueError(f"Document already exists: {file_hash}")

        # Step 2: Extract text
        print("📄 Extracting text from PDF...")
        extraction = extract_document(pdf_path)

        # Step 3: Chunk
        print("✂ Chunking document...")
        chunks = self.chunker.chunk_document(extraction['pages'])

        # Step 4: Generate embeddings
        print("🔢 Generating embeddings...")
        texts = [c.text for c in chunks]
        embeddings = self.embedder.encode(
            texts,
            normalize_embeddings=True,
            show_progress_bar=True
        )

        # Step 5: Save to disk/database
        print("💾 Saving to storage...")
        self._save_document(
            document_id=document_id,
            pdf_path=pdf_path,
            extraction=extraction,
            chunks=chunks,
            embeddings=embeddings,
            file_hash=file_hash
        )

        result = {
            'document_id': document_id,
            'total_pages': extraction['total_pages'],
            'total_chunks': len(chunks),
            'total_words': extraction['total_words'],
            'file_hash': file_hash
        }

        print(f"✅ Ingestion complete: {result}")
        return result

    def _calculate_hash(self, file_path: str) -> str:
        """Calculate MD5 hash of file."""
        with open(file_path, 'rb') as f:
            return hashlib.md5(f.read()).hexdigest()
```

```python
def _is_duplicate(self, file_hash: str) -> bool:
    """Check if document already ingested."""
    # Check database or file storage
    # Implementation depends on storage strategy
    return False  # Simplified

def _save_document(
    self,
    document_id: str,
    pdf_path: str,
    extraction: Dict,
    chunks: List[Chunk],
    embeddings,
    file_hash: str
):
    """Save all document data."""

    doc_dir = os.path.join(self.data_dir, 'documents', document_id)
    os.makedirs(doc_dir, exist_ok=True)

    # Save original PDF (copy)
    import shutil
    shutil.copy(pdf_path, os.path.join(doc_dir, 'original.pdf'))

    # Save chunks as JSON
    chunks_data = [c.to_dict() for c in chunks]
    with open(os.path.join(doc_dir, 'chunks.json'), 'w') as f:
        json.dump(chunks_data, f, indent=2)

    # Save embeddings
    import numpy as np
    np.save(os.path.join(doc_dir, 'embeddings.npy'), embeddings)

    # Save metadata
    metadata = {
        'document_id': document_id,
        'file_hash': file_hash,
        'total_pages': extraction['total_pages'],
        'total_words': extraction['total_words'],
        'total_chunks': len(chunks),
        'extraction_version': '2.0'
    }
    with open(os.path.join(doc_dir, 'metadata.json'), 'w') as f:
        json.dump(metadata, f, indent=2)
```

---

10. Code Walkthrough

10.1 Actual Project Implementation

Let's examine the actual chunking code in our project:

Location: `src/utils/` (document processing utilities)

The document processing is spread across several files. Here's how they connect:

```
PDF Upload Request
      |
      ▼
app.py: /api/upload
      |
      ▼
extract_document() (PyMuPDF extraction)
      |
      ▼
chunk_document() (Section-aware split)
      |
      ▼
generate_embeddings() (all-MiniLM)
      |
      ▼
save_to_database() (PostgreSQL)
```

10.2 Key Code Sections

Document Upload Route (app.py)

```python
@app.route('/api/upload', methods=['POST'])
def upload_document():
    """Handle document upload and processing."""

    if 'file' not in request.files:
        return jsonify({'error': 'No file provided'}), 400

    file = request.files['file']

    if not file.filename.endswith('.pdf'):
        return jsonify({'error': 'Only PDF files accepted'}), 400

    try:
        # Save uploaded file
        filename = secure_filename(file.filename)
        temp_path = os.path.join(UPLOAD_FOLDER, filename)
        file.save(temp_path)

        # Calculate hash for deduplication
        file_hash = get_file_hash(temp_path)
```

```
        # Check for existing document
        existing = DocumentRepository.get_by_hash(file_hash)
        if existing:
            os.remove(temp_path)
            return jsonify({
                'error': 'Document already exists',
                'existing_id': existing['document_id']
            }), 409

        # Generate document ID
        document_id = generate_document_id(filename)

        # Process document
        result = process_document(temp_path, document_id)

        return jsonify({
            'success': True,
            'document_id': document_id,
            'total_pages': result['total_pages'],
            'total_chunks': result['total_chunks']
        })

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

---

11. Quality Assurance

11.1 Chunk Quality Metrics

We track several metrics to ensure chunking quality:

Metric	Target	Why It Matters
Avg chunk size	700-900 tokens	Optimal for embedding
Size std dev	< 200 tokens	Consistent sizes
% under min	< 5%	Avoid tiny chunks
% over max	0%	Never exceed limits
Section coverage	100%	Every chunk has section

Metric Calculation

```
def calculate_chunk_metrics(chunks: List[Dict]) -> Dict:
    """Calculate quality metrics for chunks."""

    sizes = [c['word_count'] for c in chunks]

    return {
```

```
        'total_chunks': len(chunks),
        'avg_size': sum(sizes) / len(sizes),
        'min_size': min(sizes),
        'max_size': max(sizes),
        'std_dev': statistics.stdev(sizes),
        'under_min_pct': sum(1 for s in sizes if s < 200) / len(sizes) * 100,
        'over_max_pct': sum(1 for s in sizes if s > 1200) / len(sizes) * 100,
        'with_section_pct': sum(1 for c in chunks if c.get('section_title')) /
len(chunks) * 100
    }
```

11.2 Retrieval Quality Testing

We test chunking quality by measuring retrieval performance:

```
def test_retrieval_quality(
    chunks: List[Dict],
    test_questions: List[Dict]
) -> Dict:
    """Test if chunking enables good retrieval."""

    # Embed all chunks
    chunk_texts = [c['text'] for c in chunks]
    chunk_embeddings = embed(chunk_texts)

    results = []

    for test in test_questions:
        question = test['question']
        expected_page = test['expected_page']

        # Find top-5 chunks
        query_embedding = embed([question])[0]
        similarities = cosine_similarity(query_embedding, chunk_embeddings)
        top_indices = np.argsort(similarities)[-5:][::-1]

        # Check if expected page is in top-5
        found = any(
            chunks[i]['page_start'] <= expected_page <= chunks[i]['page_end']
            for i in top_indices
        )

        results.append({
            'question': question,
            'found_in_top5': found,
            'top_score': similarities[top_indices[0]]
        })

    return {
        'total_tests': len(results),
```

```
        'found_in_top5': sum(r['found_in_top5'] for r in results),
        'recall_at_5': sum(r['found_in_top5'] for r in results) / len(results)
    }
```

---

12. Common Problems and Solutions

12.1 Problem: Text Order Wrong

Symptom: Words from different columns mixed together

Cause: Multi-column layout not detected

Solution:

```
# Use blocks mode for better layout detection
blocks = page.get_text("blocks")
# Sort blocks by Y then X coordinate
blocks.sort(key=lambda b: (b[1], b[0]))
```

12.2 Problem: Missing Text

Symptom: Some text not extracted

Cause: Embedded images, scanned pages, or unusual fonts

Solution:

```
# Check if page has text
text = page.get_text()
if len(text.strip()) < 10:
    # Might be image-based, try OCR
    # Or flag for manual review
    pass
```

12.3 Problem: Chunks Too Varied

Symptom: Some chunks 100 words, others 2000 words

Cause: Section detection not working properly

Solution:

```
# Add fallback splitting for large sections
if len(current_chunk.split()) > max_tokens:
    # Force split even without section boundary
    sub_chunks = split_with_overlap(current_chunk, max_tokens)
```

12.4 Problem: Headers in Every Chunk

Symptom: Repeated headers/footers appearing

Cause: Header/footer detection failed

Solution:

```
# Post-process to remove repeated lines
line_counts = Counter(all_lines_across_chunks)
repeated = {l for l, c in line_counts.items() if c > len(chunks) * 0.5}
# Remove these lines from all chunks
```

12.5 Problem: Special Characters

Symptom: Garbled text, encoding issues

Cause: Non-UTF8 characters, ligatures

Solution:

```
# Force UTF-8 and normalize
text = text.encode('utf-8', errors='ignore').decode('utf-8')
import unicodedata
text = unicodedata.normalize('NFKD', text)
```

Summary

This chapter covered:

1. **Why document processing matters**: Foundation of RAG quality
2. **PDF structure**: How PDFs store text, extraction challenges
3. **PyMuPDF**: Our extraction library, usage patterns
4. **Chunking strategies**: Fixed-size, recursive, semantic, section-based
5. **Overlap**: Why it matters, implementation
6. **Our implementation**: Complete chunking pipeline
7. **Section detection**: Patterns, classification
8. **Metadata**: What to preserve, how to use it
9. **Edge cases**: Empty pages, tables, multi-column
10. **Complete pipeline**: End-to-end ingestion
11. **Quality assurance**: Metrics, testing
12. **Common problems**: Solutions for typical issues

Chapter 3: Vector Embeddings & Semantic Search

How Machines Understand Meaning

Table of Contents for This Chapter

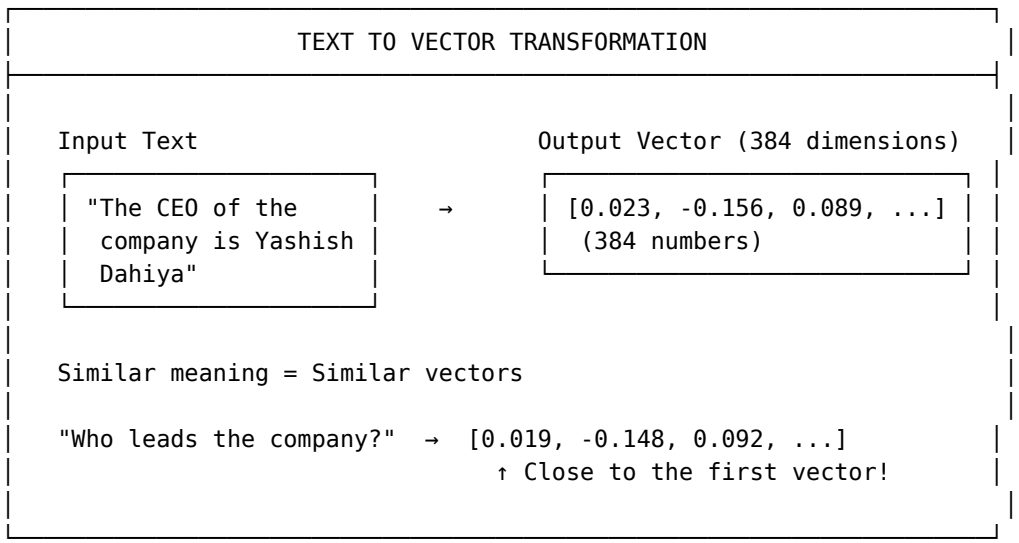
6. [Vector Databases and pgvector)
7. [HNSW Index Deep Dive)
8. [Semantic Search Implementation)
9. [Query Processing)
10. [Retrieval Optimization)
11. [Evaluation Metrics)
12. [Common Issues and Solutions)

---

1. What Are Embeddings?

1.1 The Core Concept

Embeddings are **dense numerical representations** of text (or other data) that capture semantic meaning. They transform words, sentences, or documents into vectors in a high-dimensional space where similar meanings are close together.



1.2 Why Embeddings Matter for RAG

Traditional keyword search fails when: - User says “profits” but document says “net income” - User asks “who runs the company” but answer mentions “CEO” - User asks about “growth” but document shows “42% increase”

Embeddings solve this because they capture **meaning**, not just words.

Example: Semantic Similarity

```
from sentence_transformers import SentenceTransformer
import numpy as np

model = SentenceTransformer('all-MiniLM-L6-v2')
```

```python
# These sentences have similar MEANING but different WORDS
sentences = [
    "The company's profits increased by 40%",
    "Net income grew significantly last year",
    "The weather is nice today"  # Unrelated
]

embeddings = model.encode(sentences)

# Calculate similarities
def cosine_sim(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

print(f"Profits vs Net income: {cosine_sim(embeddings[0], embeddings[1]):.3f}")
# Output: 0.782 (HIGH similarity)

print(f"Profits vs Weather: {cosine_sim(embeddings[0], embeddings[2]):.3f}")
# Output: 0.089 (LOW similarity)
```

1.3 From Words to Vectors: Historical Evolution

1.3.1 One-Hot Encoding (1990s)

Each word gets a unique position:

```
Vocabulary: [apple, banana, company, CEO, ...]
"apple"   → [1, 0, 0, 0, ...]
"banana"  → [0, 1, 0, 0, ...]
"company" → [0, 0, 1, 0, ...]
```

Problems: - No similarity (all vectors orthogonal) - Vocabulary size = vector dimension (huge)

1.3.2 Word2Vec (2013)

Train a neural network to predict context words:

```
"The [CEO] leads the company" → predict CEO from context

Result: Words in similar contexts get similar vectors
"CEO" ≈ "Chairman" ≈ "Director" (all appear in leadership contexts)
```

Improvement: Captures word-level similarity **Problem**: One vector per word (ignores context)

1.3.3 Contextual Embeddings (2018+)

BERT, GPT, and transformers:

```
"The bank by the river" → bank = [0.1, 0.3, ...]  (financial)
"I bank my paycheck"    → bank = [0.8, -0.2, ...] (different!)
```

Improvement: Same word, different contexts = different vectors **This is what we use**: Sentence-level embeddings from transformers

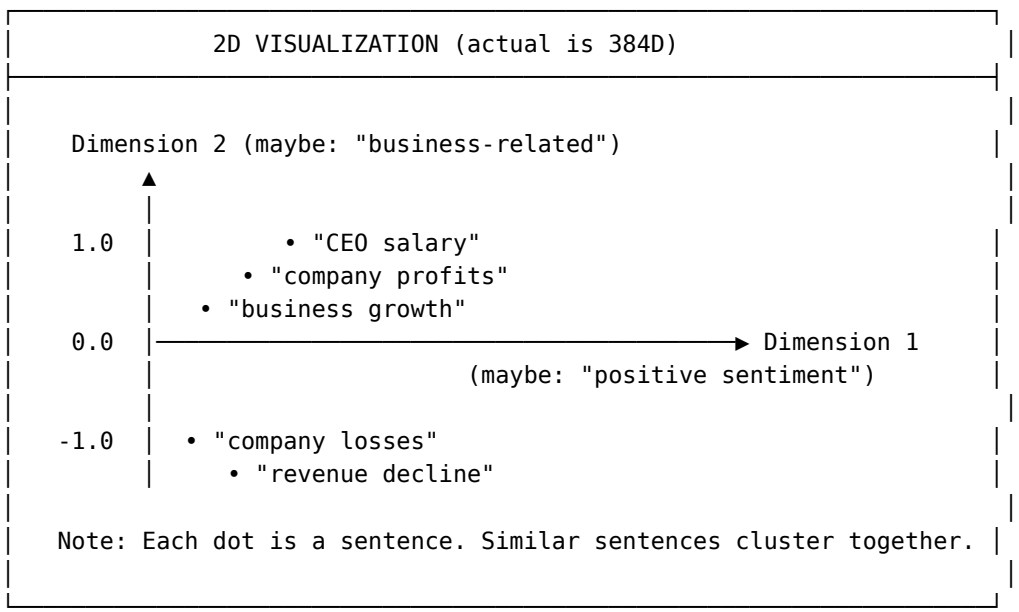
---

2. The Mathematics Behind Embeddings

2.1 Vector Space Basics

2.1.1 Vectors as Points

Each embedding is a point in N-dimensional space: - Our model: 384 dimensions - Each dimension captures some aspect of meaning - Together, they form a "semantic coordinate"



2.1.2 Distance = Dissimilarity

In vector space: - **Close vectors** = Similar meaning - **Far vectors** = Different meaning

This is why we can "search" by finding the closest vectors to a query.

2.2 Vector Operations

2.2.1 Dot Product

$a \cdot b = a_1b_1 + a_2b_2 + ... + a_nb_n$

Measures how "aligned" two vectors are: - Positive: Same direction - Zero: Perpendicular - Negative: Opposite

2.2.2 Magnitude (Length)

$||a|| = \sqrt{(a_1^2 + a_2^2 + ... + a_n^2)}$

2.2.3 Cosine Similarity

$\cos(\theta) = (a \cdot b) / (||a|| \times ||b||)$

Range: −1 to 1 - 1: Identical direction (same meaning) - 0: Perpendicular (unrelated) - −1: Opposite (antonyms, rare in practice)

Why cosine over dot product? Cosine is **scale-invariant**. A longer document doesn't automatically get higher similarity.

2.3 Normalized Vectors

If we normalize vectors to length 1:

`||a|| = 1, ||b|| = 1`

Then:

`cosine_similarity(a, b) = a · b  (just the dot product!)`

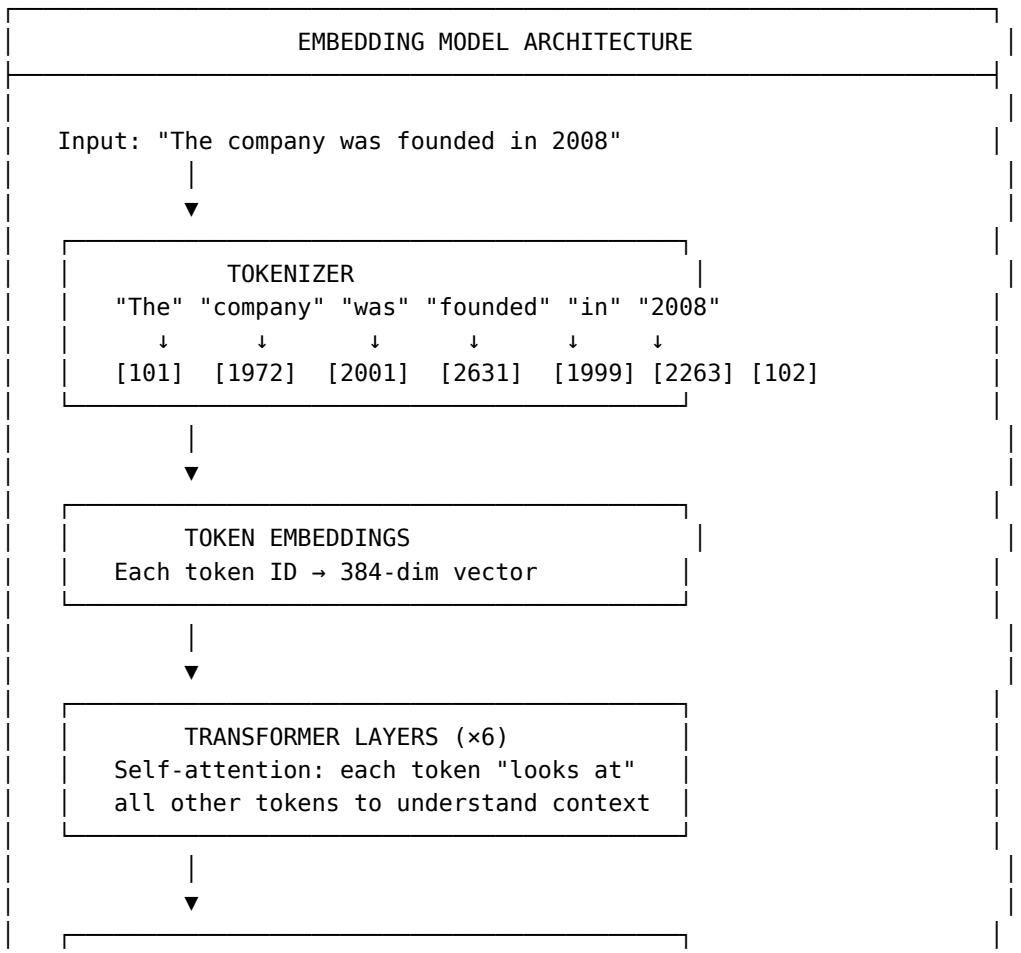
This is why we use `normalize_embeddings=True` - it makes similarity computation a simple dot product.

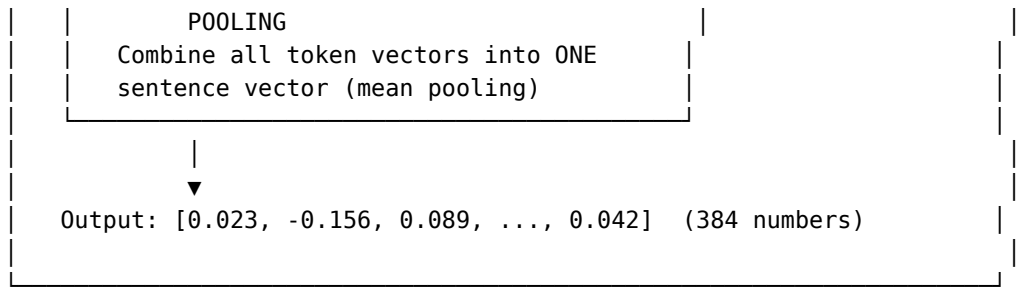
---

3. Embedding Models Explained

3.1 Model Architecture

Modern embedding models are based on **transformers**:





3.2 Training: Contrastive Learning

Embedding models learn from pairs:

```
Positive pairs (should be similar):
- ("What is the revenue?", "The company earned $5M last year")
- (Question, Relevant answer)
- (Query, Clicked document)

Negative pairs (should be different):
- ("What is the revenue?", "The CEO was born in 1975")
- Random combinations
```

Training objective: Push positive pairs close, negative pairs apart.

3.3 Model Comparison

Model	Dimensions	Max Tokens	Speed	Quality
all-MiniLM-L6-v2	384	256	⚡⚡⚡	Good
all-mpnet-base-v2	768	384	⚡⚡	Better
bge-large-en	1024	512	⚡	Best
OpenAI ada-002	1536	8191	Cloud	Excellent

Why We Chose all-MiniLM-L6-v2

1. **Speed**: 14,000 sentences/second on CPU
2. **Size**: 80MB (fits in memory easily)
3. **Quality**: Good enough for our use case
4. **Proven**: Widely used, well-documented
5. **Free**: No API costs

---

4. Sentence Transformers Deep Dive

4.1 Installation and Basic Usage

```
# Installation
pip install sentence-transformers
```

```python
# Basic usage
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

# Encode single text
embedding = model.encode("What is the company's revenue?")
print(embedding.shape)  # (384,)

# Encode multiple texts
texts = ["Text 1", "Text 2", "Text 3"]
embeddings = model.encode(texts)
print(embeddings.shape)  # (3, 384)
```

4.2 Advanced Options

```python
embeddings = model.encode(
    texts,
    batch_size=32,                    # Process 32 texts at a time
    show_progress_bar=True,           # Show progress
    convert_to_numpy=True,            # Return numpy array (default)
    normalize_embeddings=True,        # L2 normalize (length=1)
    device='cuda'                     # Use GPU if available
)
```

4.3 Handling Long Texts

The model has a **max token limit** (256 for MiniLM). Longer texts are truncated!

```python
# This WILL be truncated (loses information)
long_text = "..." * 1000  # 1000+ words
embedding = model.encode(long_text)  # Only first 256 tokens used!

# Solution: Chunk first, embed chunks
chunks = split_into_chunks(long_text, max_tokens=200)
chunk_embeddings = model.encode(chunks)
```

This is why our **chunking strategy** (Chapter 2) is critical!

4.4 Semantic Search Function

```python
from sentence_transformers import SentenceTransformer, util
import numpy as np

class SemanticSearch:
    def __init__(self, model_name: str = 'all-MiniLM-L6-v2'):
        self.model = SentenceTransformer(model_name)
        self.corpus_embeddings = None
        self.corpus = None

    def index(self, corpus: list[str]):
```

```
        """Index a corpus of documents."""
        self.corpus = corpus
        self.corpus_embeddings = self.model.encode(
            corpus,
            normalize_embeddings=True,
            show_progress_bar=True
        )

    def search(self, query: str, top_k: int = 5) -> list[dict]:
        """Search for most similar documents."""
        query_embedding = self.model.encode(
            query,
            normalize_embeddings=True
        )

        # Compute similarities (dot product since normalized)
        similarities = np.dot(self.corpus_embeddings, query_embedding)

        # Get top-k indices
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        results = []
        for idx in top_indices:
            results.append({
                'text': self.corpus[idx],
                'score': float(similarities[idx]),
                'index': int(idx)
            })

        return results
```

---

5. Vector Similarity Metrics

5.1 Cosine Similarity (Our Choice)

```
cosine(a, b) = (a · b) / (||a|| × ||b||)
```

Range: −1 to 1 (typically 0 to 1 for text)

Pros: - Scale invariant - Intuitive interpretation - Fast with normalized vectors

Cons: - Doesn't consider magnitude

PostgreSQL (pgvector)

```
-- Cosine distance (1 - similarity)
SELECT * FROM embeddings
ORDER BY embedding <=> query_embedding
LIMIT 5;
```

5.2 Euclidean Distance (L2)

$L2(a, b) = \sqrt{(a_1-b_1)^2 + (a_2-b_2)^2 + ... + (a_n-b_n)^2}$

Range: 0 to infinity (lower = more similar)

Pros: - Geometric intuition - Considers magnitude

Cons: - Scale dependent - Larger range

PostgreSQL (pgvector)

```
-- L2 distance
SELECT * FROM embeddings
ORDER BY embedding <-> query_embedding
LIMIT 5;
```

5.3 Inner Product (Dot Product)

$dot(a, b) = a_1b_1 + a_2b_2 + ... + a_nb_n$

Range: -∞ to +∞

Pros: - Fastest computation - Equals cosine when normalized

Cons: - Not bounded - Depends on normalization

PostgreSQL (pgvector)

```
-- Negative inner product (for ordering)
SELECT * FROM embeddings
ORDER BY embedding <#> query_embedding
LIMIT 5;
```

5.4 Which to Use?

Metric	When to Use
Cosine	Text similarity (our case)
Euclidean	When magnitude matters
Dot Product	Pre-normalized vectors (fastest)

We use **cosine** because: 1. Text length shouldn't affect similarity 2. Easy to interpret (0-1 scale) 3. Standard for NLP

---

6. Vector Databases and pgvector

6.1 Why Vector Databases?

Regular databases can't efficiently search by similarity:

```
-- This is SLOW (scans every row)
SELECT *
FROM chunks
```

```
ORDER BY cosine_similarity(embedding, query_embedding)
LIMIT 5;
```

Vector databases use **approximate nearest neighbor (ANN)** algorithms to make this fast.

6.2 pgvector: PostgreSQL Extension

pgvector adds vector capabilities to PostgreSQL:

```
-- Enable extension
CREATE EXTENSION vector;

-- Create table with vector column
CREATE TABLE chunks (
    id SERIAL PRIMARY KEY,
    text TEXT,
    embedding vector(384)  -- 384  dimensions
);

-- Insert vector
INSERT INTO chunks (text, embedding)
VALUES ('Sample text', '[0.1, 0.2, ...]');

-- Search by similarity
SELECT *, embedding <=> '[0.1, ...]' as distance
FROM chunks
ORDER BY embedding <=> '[0.1, ...]'
LIMIT 5;
```

6.3 Our Schema

```
-- From database/schema.sql
CREATE TABLE embeddings (
    id SERIAL PRIMARY KEY,
    chunk_id INTEGER REFERENCES chunks(id) ON DELETE CASCADE,
    embedding vector(384) NOT NULL,
    model_name VARCHAR(100) DEFAULT 'all-MiniLM-L6-v2',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- HNSW index for fast similarity search
CREATE INDEX idx_embeddings_vector ON embeddings
USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);
```

6.4 Vector Search Query

```
def search_similar_chunks(
    session,
    query_embedding: list[float],
    document_id: str,
    top_k: int = 5
```

```
) -> list[dict]:
    """Search for chunks similar to query embedding."""

    # Convert to pgvector format
    embedding_str = '[' + ','.join(str(x) for x in query_embedding) + ']'

    sql = text("""
        SELECT
            c.id,
            c.text,
            c.page_start,
            c.page_end,
            c.section_title,
            1 - (e.embedding <=> :query_vec) as similarity
        FROM chunks c
        JOIN embeddings e ON e.chunk_id = c.id
        WHERE c.document_id = :doc_id
        ORDER BY e.embedding <=> :query_vec
        LIMIT :top_k
    """)

    result = session.execute(sql, {
        'query_vec': embedding_str,
        'doc_id': document_id,
        'top_k': top_k
    })

    return [dict(row) for row in result]
```

---

7. HNSW Index Deep Dive

7.1 The Problem with Exact Search

Exact nearest neighbor search is O(n) - must compare against every vector.

For 1 million vectors × 384 dimensions: - 384 million floating point operations per query - ~100ms per query (too slow!)

7.2 Approximate Nearest Neighbor (ANN)

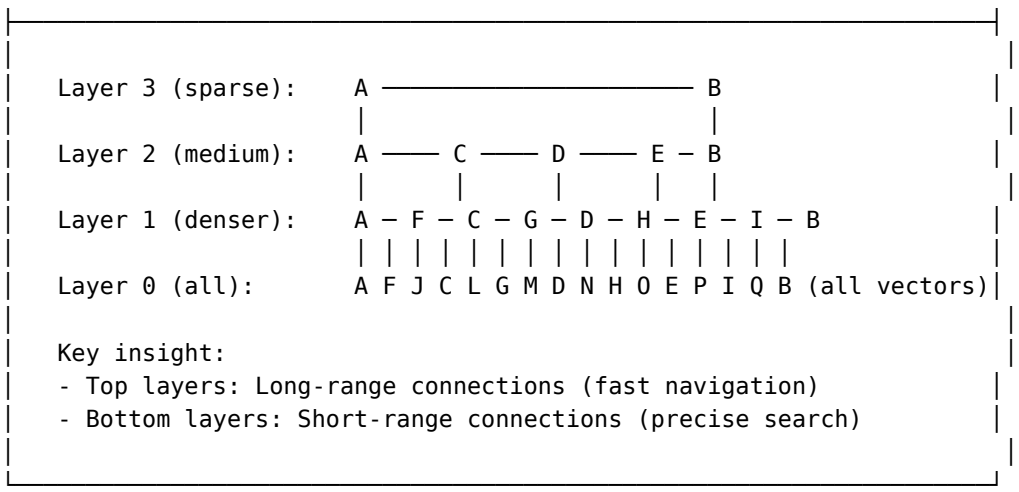
ANN algorithms trade accuracy for speed: - Don't find THE nearest, find APPROXIMATELY nearest - 95-99% accuracy, 100x faster

7.3 How HNSW Works

HNSW = Hierarchical Navigable Small World

Think of it as a **multi-layer graph**:

```
┌──────────────────────────────────────────────────────────────────────┐
│                          HNSW STRUCTURE                              │
```



Search Algorithm

1. Start at entry point in top layer
2. Greedily move to neighbor closest to query
3. When stuck (no closer neighbor), drop to next layer
4. Repeat until layer 0
5. Return nearest neighbors found

Why It's Fast

- Skip irrelevant regions using top layers
- Only examine a small fraction of vectors
- O(log n) complexity instead of O(n)

7.4 HNSW Parameters

Parameter	Meaning	Trade-off
m	Max connections per node	Higher = better quality, more memory
ef_construction	Search depth during build	Higher = better index, slower build
ef_search	Search depth during query	Higher = better recall, slower query

Our Configuration

```
CREATE INDEX idx_embeddings_vector ON embeddings
USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);

-- At query time
SET hnsw.ef_search = 100;  -- More thorough search
```

7.5 Performance Benchmarks

Vectors	Exact (ms)	HNSW (ms)	Recall@10
10,000	15	2	99%
100,000	150	5	98%
1,000,000	1500	12	97%

8. Semantic Search Implementation

8.1 Our VectorRAG Class

```
# From src/utils/vector_rag.py
from sentence_transformers import SentenceTransformer
from database.connection import get_db
from sqlalchemy import text
import numpy as np

class VectorRAG:
    """Vector-based Retrieval Augmented Generation."""

    def __init__(self, document_id: str):
        self.document_id = document_id
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

    def embed_query(self, query: str) -> np.ndarray:
        """Generate embedding for a query."""
        return self.model.encode(
            query,
            normalize_embeddings=True
        )

    def search(self, query: str, top_k: int = 5) -> list[dict]:
        """Find most relevant chunks for a query."""

        # Step 1: Embed the query
        query_embedding = self.embed_query(query)

        # Step 2: Search in database
        with get_db() as session:
            embedding_str = '[' + ','.join(str(x) for x in query_embedding) + ']'

            sql = text("""
                SELECT
                    c.id,
                    c.chunk_index,
                    c.text,
                    c.page_start,
```

```
                    c.page_end,
                    c.section_title,
                    1 - (e.embedding <=> :query_vec) as similarity
                FROM chunks c
                JOIN embeddings e ON e.chunk_id = c.id
                WHERE c.document_id = :doc_id
                ORDER BY e.embedding <=> :query_vec
                LIMIT :top_k
            """)

            result = session.execute(sql, {
                'query_vec': embedding_str,
                'doc_id': self.document_id,
                'top_k': top_k
            })

            chunks = []
            for row in result:
                chunks.append({
                    'id': row.id,
                    'chunk_index': row.chunk_index,
                    'text': row.text,
                    'page_start': row.page_start,
                    'page_end': row.page_end,
                    'section_title': row.section_title,
                    'similarity': float(row.similarity)
                })

            return chunks

    def get_context(self, query: str, top_k: int = 5) -> str:
        """Get formatted context for LLM prompt."""
        chunks = self.search(query, top_k)

        context_parts = []
        for i, chunk in enumerate(chunks, 1):
            pages = f"Pages {chunk['page_start']}-{chunk['page_end']}"
            section = chunk.get('section_title', 'Unknown Section')

            context_parts.append(
                f"[Source {i}] ({pages}, {section})\n{chunk['text']}"
            )

        return "\n\n---\n\n".join(context_parts)
```

8.2 Using VectorRAG in the API

```
# From app.py
@app.route('/api/ask', methods=['POST'])
def ask_question():
```

```
    data = request.json
    question = data.get('question')
    document_id = data.get('document_id')

    # Initialize RAG
    vector_rag = VectorRAG(document_id)

    # Get relevant context
    context = vector_rag.get_context(question, top_k=5)

    # Build prompt
    prompt = f"""Based on the following context from the document, answer the
question.

Context:
{context}

Question: {question}

Answer:"""

    # Generate answer with LLM
    response = ollama.chat(
        model='llama3',
        messages=[{'role': 'user', 'content': prompt}],
        stream=True
    )

    # Stream response...
```

---

9. Query Processing

9.1 Query Preprocessing

Before embedding, we preprocess queries:

```
def preprocess_query(query: str) -> str:
    """Clean and normalize query for better retrieval."""

    # 1. Lowercase
    query = query.lower()

    # 2. Remove extra whitespace
    query = ' '.join(query.split())

    # 3. Handle common abbreviations
    abbreviations = {
        "ipo": "initial public offering",
        "ceo": "chief executive officer",
```

```
        "cfo": "chief financial officer",
        "fy": "financial year",
    }
    for abbr, full in abbreviations.items():
        if abbr in query:
            query = query.replace(abbr, f"{abbr} {full}")

    return query
```

9.2 Query Expansion

Sometimes expanding the query improves retrieval:

```
def expand_query(query: str) -> list[str]:
    """Generate query variations for better recall."""

    variations = [query]

    # Add question variations
    if not query.endswith('?'):
        variations.append(query + '?')

    # Add common reformulations
    if 'who' in query.lower():
        variations.append(query.replace('who', 'which person'))
    if 'what' in query.lower():
        variations.append(query.replace('what', 'which'))

    return variations
```

9.3 Multi-Query Retrieval

Search with multiple query variations:

```
def multi_query_search(
    rag: VectorRAG,
    query: str,
    top_k: int = 5
) -> list[dict]:
    """Search with multiple query variations, merge results."""

    variations = expand_query(query)
    all_results = {}

    for variant in variations:
        results = rag.search(variant, top_k=top_k)
        for r in results:
            chunk_id = r['id']
            if chunk_id not in all_results:
                all_results[chunk_id] = r
            else:
                # Keep higher similarity score
```

```
                if r['similarity'] > all_results[chunk_id]['similarity']:
                    all_results[chunk_id] = r

    # Sort by similarity and return top_k
    sorted_results = sorted(
        all_results.values(),
        key=lambda x: x['similarity'],
        reverse=True
    )

    return sorted_results[:top_k]
```

---

10. Retrieval Optimization

10.1 Reranking

After initial retrieval, use a more powerful model to rerank:

```
from sentence_transformers import CrossEncoder

class Reranker:
    def __init__(self):
        # Cross-encoder is more accurate but slower
        self.model = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-6-v2')

    def rerank(
        self,
        query: str,
        chunks: list[dict],
        top_k: int = 5
    ) -> list[dict]:
        """Rerank chunks using cross-encoder."""

        # Create query-document pairs
        pairs = [(query, chunk['text']) for chunk in chunks]

        # Score all pairs
        scores = self.model.predict(pairs)

        # Add scores to chunks
        for i, chunk in enumerate(chunks):
            chunk['rerank_score'] = float(scores[i])

        # Sort by rerank score
        sorted_chunks = sorted(
            chunks,
            key=lambda x: x['rerank_score'],
            reverse=True
        )
```

```
        return sorted_chunks[:top_k]
```

10.2 Metadata Filtering

Filter before or after vector search:

```
def search_with_filters(
    rag: VectorRAG,
    query: str,
    section_filter: str = None,
    page_range: tuple = None,
    top_k: int = 5
) -> list[dict]:
    """Search with metadata filters."""

    with get_db() as session:
        query_embedding = rag.embed_query(query)
        embedding_str = '[' + ','.join(str(x) for x in query_embedding) + ']'

        sql = """
            SELECT c.*, 1 - (e.embedding <=> :query_vec) as similarity
            FROM chunks c
            JOIN embeddings e ON e.chunk_id = c.id
            WHERE c.document_id = :doc_id
        """
        params = {'query_vec': embedding_str, 'doc_id': rag.document_id}

        if section_filter:
            sql += " AND c.section_title ILIKE :section"
            params['section'] = f"%{section_filter}%"

        if page_range:
            sql += " AND c.page_start >= :start AND c.page_end <= :end"
            params['start'] = page_range[0]
            params['end'] = page_range[1]

        sql += " ORDER BY e.embedding <=> :query_vec LIMIT :top_k"
        params['top_k'] = top_k

        return session.execute(text(sql), params).fetchall()
```

10.3 Hybrid Search (Vector + Keyword)

Combine vector search with traditional keyword search:

```
def hybrid_search(
    rag: VectorRAG,
    query: str,
    top_k: int = 5,
    alpha: float = 0.7  # Weight for vector vs keyword
) -> list[dict]:
```

```
    """Combine vector and keyword search."""

    # Vector search
    vector_results = rag.search(query, top_k=top_k * 2)

    # Keyword search (PostgreSQL full-text)
    with get_db() as session:
        sql = text("""
            SELECT c.*, ts_rank(to_tsvector(c.text), query) as keyword_score
            FROM chunks c, plainto_tsquery(:query) query
            WHERE c.document_id = :doc_id
              AND to_tsvector(c.text) @@ query
            ORDER BY keyword_score DESC
            LIMIT :top_k
        """)
        keyword_results = session.execute(sql, {
            'query': query,
            'doc_id': rag.document_id,
            'top_k': top_k * 2
        }).fetchall()

    # Merge and score
    all_ids = set()
    merged = {}

    for r in vector_results:
        merged[r['id']] = {
            **r,
            'vector_score': r['similarity'],
            'keyword_score': 0
        }
        all_ids.add(r['id'])

    for r in keyword_results:
        if r.id in merged:
            merged[r.id]['keyword_score'] = r.keyword_score
        else:
            merged[r.id] = {
                'id': r.id,
                'text': r.text,
                'vector_score': 0,
                'keyword_score': r.keyword_score
            }

    # Combined score
    for rid, r in merged.items():
        r['combined_score'] = (
            alpha * r['vector_score'] +
            (1 - alpha) * r['keyword_score']
```

```
        )

    # Sort and return
    sorted_results = sorted(
        merged.values(),
        key=lambda x: x['combined_score'],
        reverse=True
    )

    return sorted_results[:top_k]
```

---

11. Evaluation Metrics

11.1 Retrieval Metrics

Recall@K

What fraction of relevant documents are in top-K results?

```
def recall_at_k(retrieved: list, relevant: list, k: int) -> float:
    """Calculate Recall@K."""
    retrieved_set = set(retrieved[:k])
    relevant_set = set(relevant)

    if not relevant_set:
        return 0.0

    return len(retrieved_set & relevant_set) / len(relevant_set)
```

Precision@K

What fraction of top-K results are relevant?

```
def precision_at_k(retrieved: list, relevant: list, k: int) -> float:
    """Calculate Precision@K."""
    retrieved_set = set(retrieved[:k])
    relevant_set = set(relevant)

    if k == 0:
        return 0.0

    return len(retrieved_set & relevant_set) / k
```

Mean Reciprocal Rank (MRR)

Position of first relevant result:

```
def mrr(retrieved: list, relevant: list) -> float:
    """Calculate Mean Reciprocal Rank."""
    relevant_set = set(relevant)

    for i, doc in enumerate(retrieved, 1):
```

```
        if doc in relevant_set:
            return 1.0 / i

    return 0.0
```

11.2 Our Evaluation Script

```
# evaluation/evaluate_retrieval.py
def evaluate_retrieval(
    rag: VectorRAG,
    test_questions: list[dict]
) -> dict:
    """Evaluate retrieval quality."""

    metrics = {
        'recall@5': [],
        'precision@5': [],
        'mrr': []
    }

    for test in test_questions:
        query = test['question']
        expected = test['expected_chunk_ids']

        results = rag.search(query, top_k=5)
        retrieved = [r['id'] for r in results]

        metrics['recall@5'].append(recall_at_k(retrieved, expected, 5))
        metrics['precision@5'].append(precision_at_k(retrieved, expected, 5))
        metrics['mrr'].append(mrr(retrieved, expected))

    return {
        'avg_recall@5': np.mean(metrics['recall@5']),
        'avg_precision@5': np.mean(metrics['precision@5']),
        'avg_mrr': np.mean(metrics['mrr']),
        'num_queries': len(test_questions)
    }
```

---

12. Common Issues and Solutions

12.1 Issue: Poor Retrieval Quality

Symptoms: Relevant chunks not in top results

Solutions: 1. Check chunk quality (too large/small?) 2. Try different embedding model 3. Use reranking 4. Add query expansion

12.2 Issue: Slow Search

Symptoms: Queries take >1 second

Solutions: 1. Ensure HNSW index exists 2. Increase `ef_search` carefully 3. Check database connection pooling 4. Consider caching frequent queries

12.3 Issue: Out of Memory

Symptoms: OOM when embedding

Solutions:

```
# Process in batches
for i in range(0, len(texts), batch_size):
    batch = texts[i:i+batch_size]
    embeddings.append(model.encode(batch))
```

12.4 Issue: Dimension Mismatch

Symptoms: "Vector dimension mismatch" errors

Solutions: - Ensure consistent model across indexing and query - Check `vector(384)` matches model output

---

Summary

This chapter covered:

1. **What embeddings are**: Dense vectors capturing meaning
2. **Mathematics**: Vector operations, similarity metrics
3. **Embedding models**: Architecture, training, comparison
4. **Sentence Transformers**: Usage, options, limitations
5. **Similarity metrics**: Cosine, Euclidean, dot product
6. **pgvector**: PostgreSQL vector extension
7. **HNSW**: Fast approximate nearest neighbor
8. **Implementation**: Our VectorRAG class
9. **Query processing**: Preprocessing, expansion
10. **Optimization**: Reranking, filtering, hybrid search
11. **Evaluation**: Recall, precision, MRR
12. **Troubleshooting**: Common issues and fixes

Chapter 4: Knowledge Graph Generation

Extracting Structured Knowledge from Unstructured Text

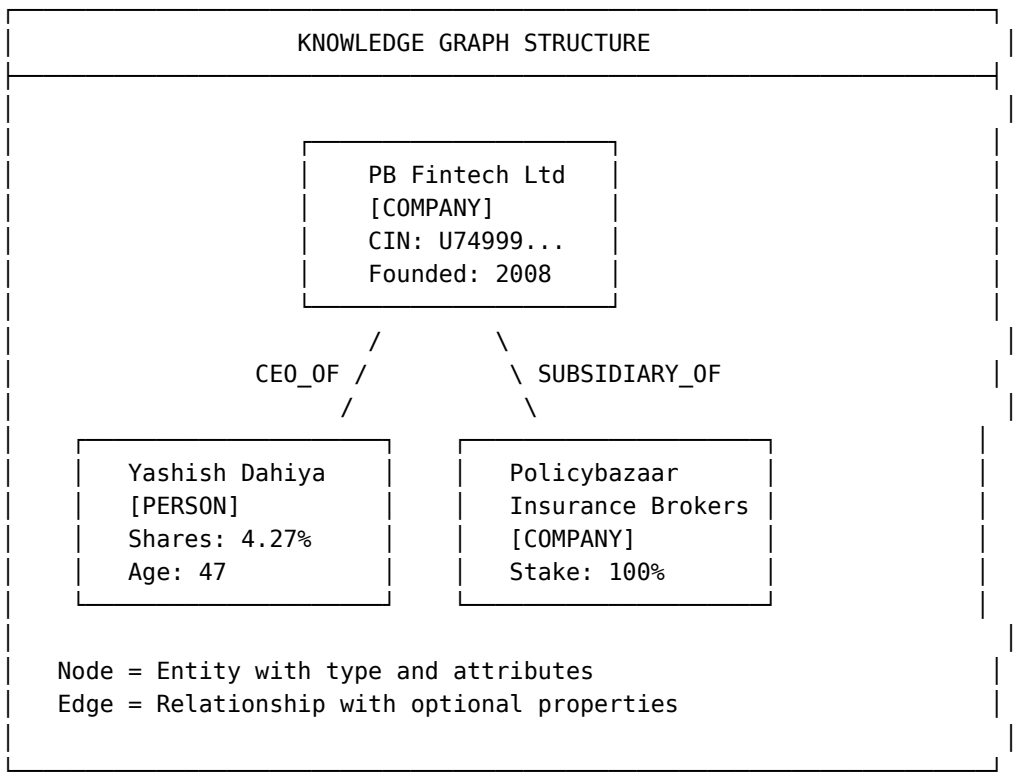
Table of Contents for This Chapter

---

1. What is a Knowledge Graph?

1.1 Definition

A Knowledge Graph (KG) is a structured representation of real-world knowledge using: - **Nodes (Entities)**: Things with identity (people, companies, products) - **Edges (Relationships)**: Connections between entities - **Properties**: Attributes of entities and relationships



1.2 KG vs. Traditional Databases

Aspect	Relational DB	Knowledge Graph
Schema	Fixed, predefined	Flexible, evolving
Queries	JOINs (expensive)	Graph traversal (natural)
Relationships	Implicit (foreign keys)	Explicit (first-class)
New entity types	Schema migration	Just add nodes
Natural language	Hard to map	Closer to how humans think

1.3 Famous Knowledge Graphs

KG	Owner	Size	Use Case
Google KG	Google	500B+ facts	Search enhancement
Wikidata	Wikimedia	100M+ items	General knowledge
DBpedia	Community	6M+ entities	Structured Wikipedia
YAGO	Max Planck	10M+ entities	Academic research

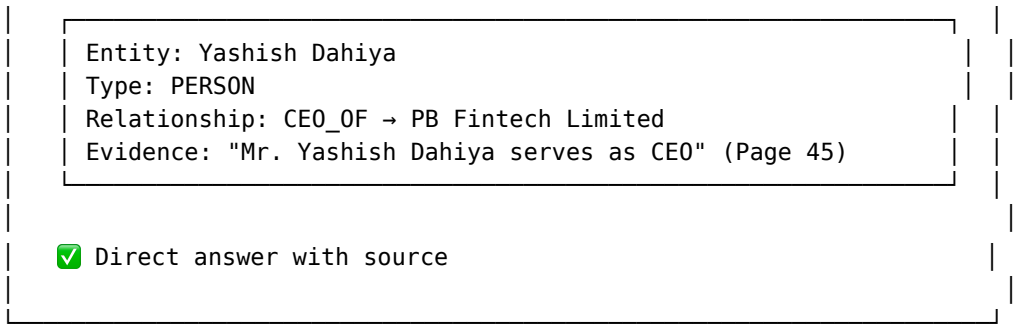
---

2. Why KG for IPO Documents?

2.1 Limitations of Vector-Only RAG

Vector search finds **similar text**, but sometimes we need **exact facts**:

```
┌─────────────────────────────────────────────────────────────────────┐
│                    VECTOR SEARCH LIMITATIONS                        │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│   Question: "Who is the CEO?"                                       │
│                                                                     │
│   Vector Search Returns:                                            │
│   ┌─────────────────────────────────────────────────────────────┐   │
│   │ [Chunk 1, Score: 0.82]                                      │   │
│   │ "The management team brings decades of experience to the    │   │
│   │  company. Leadership has been instrumental in growth..."    │   │
│   │                                                             │   │
│   │ [Chunk 2, Score: 0.79]                                      │   │
│   │ "Our executive team includes professionals from various     │   │
│   │  backgrounds in technology and finance..."                  │   │
│   └─────────────────────────────────────────────────────────────┘   │
│                                                                     │
│   ❌ Neither chunk directly answers "CEO is Yashish Dahiya"         │
│                                                                     │
│   KG Approach:                                                      │
```



2.2 What KG Enables

Query Type	Vector RAG	KG RAG
"Who is CEO?"	Chunks about leadership	Direct entity lookup
"List subsidiaries"	Scattered mentions	Graph traversal
"What does ESOP-2014 mean?"	Similar chunks	Definition lookup
"Promoter shareholding?"	Text about shares	Structured data
"Key milestones?"	Event descriptions	Timeline query

2.3 Hybrid Power

By combining Vector + KG: 1. **KG** provides structured facts (entities, relationships, definitions) 2. **Vector** provides narrative context (explanations, descriptions) 3. **LLM** synthesizes both into coherent answers

---

3. Our KG Schema Design

3.1 Entity Types

We define specific entity types for IPO documents:

Type	Description	Example
COMPANY	Companies, subsidiaries, JVs	PB Fintech, Policybazaar
PERSON	Directors, officers, promoters	Yashish Dahiya
FINANCIAL_METRIC	Revenue, profit, EBITDA	Revenue FY2021
SECURITY	Equity shares, bonds	Equity Share
REGULATORY_BODY	SEBI, RBI, IRDAI	SEBI
LEGAL_ENTITY	Specific legal registrations	CIN, PAN
PRODUCT	Products and services	Term Insurance
LOCATION	Offices, jurisdictions	Gurugram, India

3.2 Relationship Types

Relationship	Subject	Object	Example
CEO_OF	PERSON	COMPANY	Dahiya CEO_OF PB Fintech
SUBSIDIARY_OF	COMPANY	COMPANY	Policybazaar SUBSIDIARY_OF PB Fintech
OWNS_SHARES	PERSON/COMPANY	COMPANY	Dahiya OWNS_SHARES (4.27%) PB Fintech
DIRECTOR_OF	PERSON	COMPANY	Member DIRECTOR_OF Company
REGULATES	REGULATORY_BODY	COMPANY	IRDAI REGULATES Policybazaar
LOCATED_IN	COMPANY	LOCATION	PB Fintech LOCATED_IN Gurugram

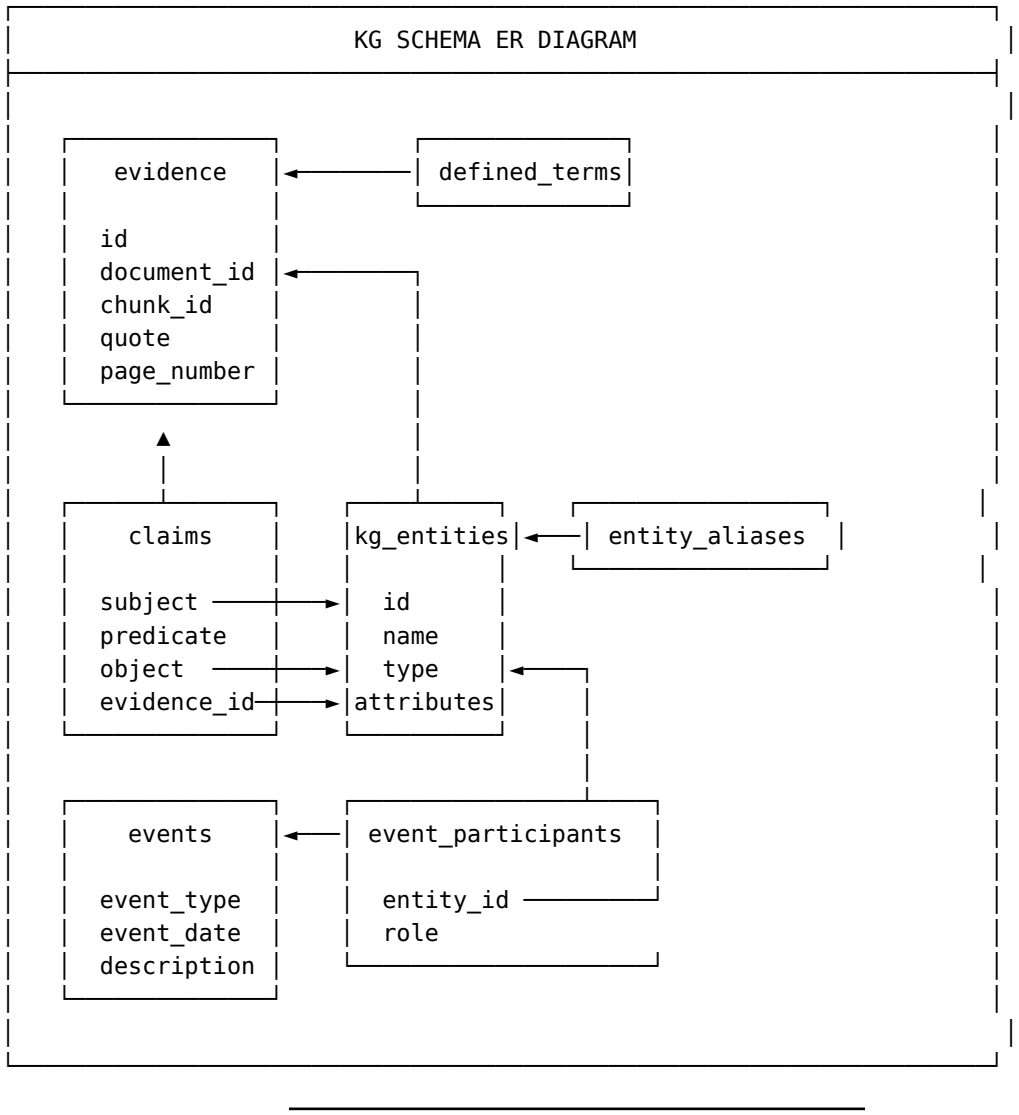
3.3 Database Schema

```
-- From database/kg_schema.sql

-- Evidence: Links KG facts to source text
CREATE TABLE evidence (
    id SERIAL PRIMARY KEY,
    document_id VARCHAR(100) NOT NULL,
    chunk_id INTEGER REFERENCES chunks(id),
    quote TEXT NOT NULL,
    page_number INTEGER,
    section_title VARCHAR(255),
    confidence FLOAT DEFAULT 1.0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Entities: Nodes in the graph
CREATE TABLE kg_entities (
    id SERIAL PRIMARY KEY,
    document_id VARCHAR(100) NOT NULL,
    canonical_name VARCHAR(255) NOT NULL,
    entity_type VARCHAR(50) NOT NULL,
    attributes JSONB DEFAULT '{}',
    evidence_ids INTEGER[],
```

```
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(document_id, canonical_name, entity_type)
);

-- Entity Aliases: Alternative names for entities
CREATE TABLE entity_aliases (
    id SERIAL PRIMARY KEY,
    entity_id INTEGER REFERENCES kg_entities(id) ON DELETE CASCADE,
    alias VARCHAR(255) NOT NULL,
    alias_type VARCHAR(50) DEFAULT 'abbreviation'
);

-- Defined Terms: Glossary from the document
CREATE TABLE defined_terms (
    id SERIAL PRIMARY KEY,
    document_id VARCHAR(100) NOT NULL,
    term VARCHAR(255) NOT NULL,
    definition TEXT NOT NULL,
    evidence_id INTEGER REFERENCES evidence(id),

    UNIQUE(document_id, term)
);

-- Claims: Relationships (edges) in the graph
CREATE TABLE claims (
    id SERIAL PRIMARY KEY,
    document_id VARCHAR(100) NOT NULL,
    subject_entity_id INTEGER REFERENCES kg_entities(id),
    predicate VARCHAR(100) NOT NULL,
    object_entity_id INTEGER REFERENCES kg_entities(id),
    object_value TEXT,
    qualifiers JSONB DEFAULT '{}',
    evidence_id INTEGER REFERENCES evidence(id),
    confidence FLOAT DEFAULT 1.0
);

-- Events: Temporal facts
CREATE TABLE events (
    id SERIAL PRIMARY KEY,
    document_id VARCHAR(100) NOT NULL,
    event_type VARCHAR(100) NOT NULL,
    event_date DATE,
    event_date_text VARCHAR(100),
    description TEXT,
    evidence_id INTEGER REFERENCES evidence(id)
);

-- Event Participants
```

```sql
CREATE TABLE event_participants (
    id SERIAL PRIMARY KEY,
    event_id INTEGER REFERENCES events(id) ON DELETE CASCADE,
    entity_id INTEGER REFERENCES kg_entities(id),
    role VARCHAR(100)
);
```

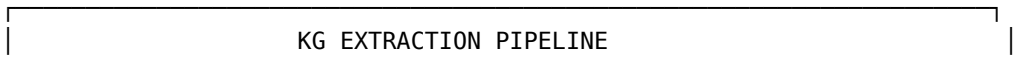
3.4 Schema Visualization

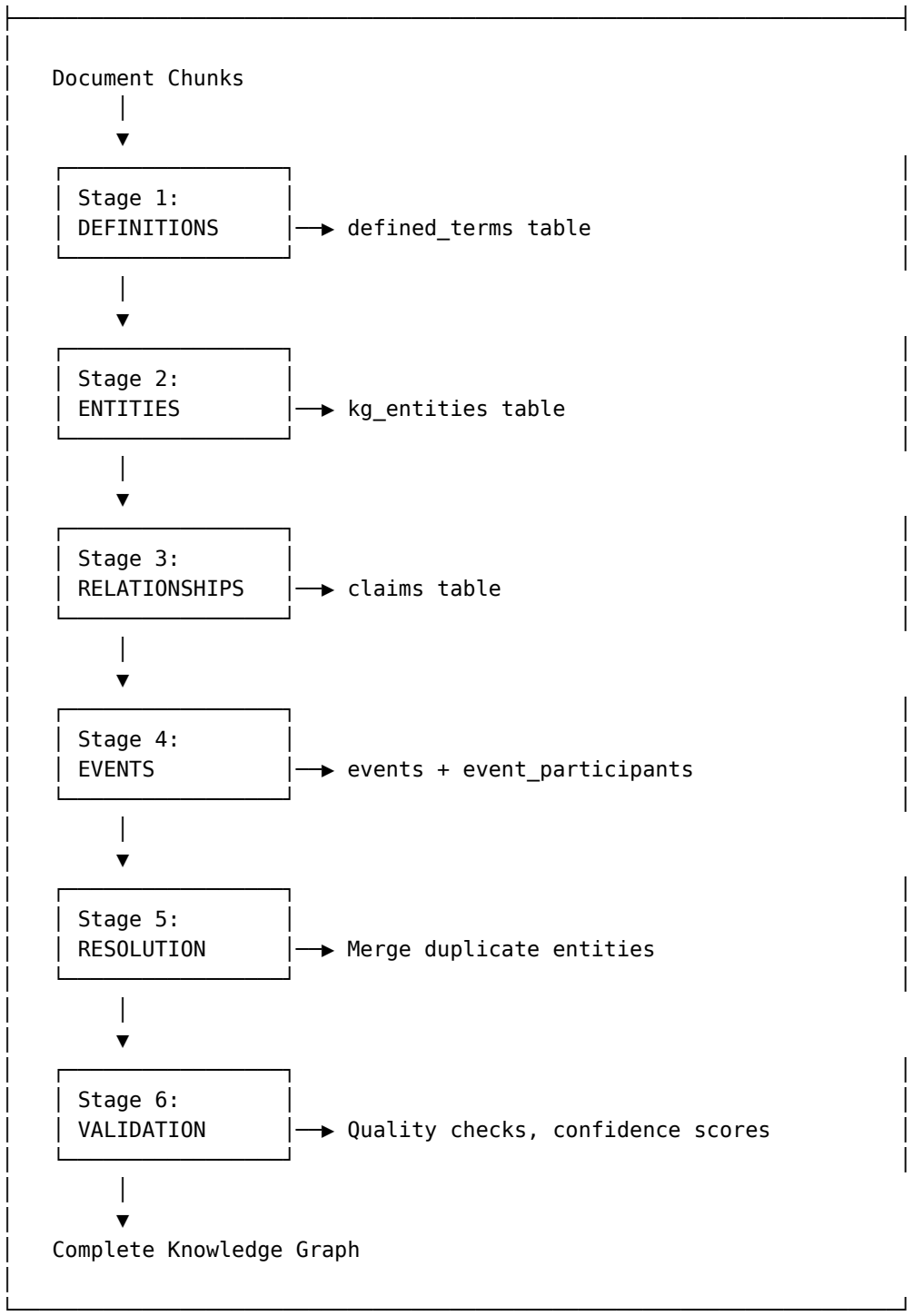


---

4. The 6-Stage Extraction Pipeline

4.1 Pipeline Overview





4.2 Pipeline Implementation

```python
# From src/utils/kg_pipeline.py
import json
import ollama
```

```
from typing import List, Dict, Optional
from dataclasses import dataclass

@dataclass
class ExtractionResult:
    """Result from a single extraction stage."""
    success: bool
    data: Dict
    raw_response: str
    error: Optional[str] = None

class KGPipeline:
    """Multi-stage Knowledge Graph extraction pipeline."""

    def __init__(self, model: str = 'llama3'):
        self.model = model
        self.prompts = KGPrompts()

    def process_chunk(self, chunk: Dict) -> Dict:
        """Process a single chunk through all stages."""

        results = {
            'chunk_id': chunk.get('chunk_index'),
            'page': chunk.get('page_start'),
            'definitions': [],
            'entities': [],
            'relationships': [],
            'events': []
        }

        text = chunk.get('text', '')

        # Stage 1: Definitions
        definitions = self.extract_definitions(text)
        results['definitions'] = definitions

        # Stage 2: Entities
        entities = self.extract_entities(text)
        results['entities'] = entities

        # Stage 3: Relationships (using extracted entities)
        relationships = self.extract_relationships(text, entities)
        results['relationships'] = relationships

        # Stage 4: Events
        events = self.extract_events(text)
        results['events'] = events

        return results
```

```python
    def process_document(self, chunks: List[Dict]) -> Dict:
        """Process entire document, then resolve and validate."""

        all_results = []

        for chunk in chunks:
            result = self.process_chunk(chunk)
            all_results.append(result)

        # Stage 5: Entity Resolution
        merged_entities = self.resolve_entities(all_results)

        # Stage 6: Validation
        validation_report = self.validate(merged_entities, all_results)

        return {
            'chunk_results': all_results,
            'entities': merged_entities,
            'validation': validation_report
        }
```

---

5. Stage 1: Definition Extraction

5.1 Why Definitions Matter

IPO documents define terms in a special section: - “Company” means “PB Fintech Limited” - “ESOP-2014” means the employee stock option plan established in 2014 - “Promoters” means Yashish Dahiya and Alok Bansal

Without understanding definitions, the LLM can’t correctly interpret the document.

5.2 Definition Prompt

```
# From src/utils/kg_prompts.py
DEFINITION_PROMPT = """You are extracting defined terms from an IPO document.

A defined term is a word or phrase that is explicitly defined in the document,
often in a "Definitions" section or marked in special formatting.

TEXT:
{text}

Extract all defined terms found in this text. Return ONLY valid JSON:

{{
    "definitions": [
        {{
            "term": "exact term as written",
```

```
            "definition": "full definition text",
            "quote": "exact sentence where this definition appears"
        }}
    ]
}}

If no definitions found, return: {{"definitions": []}}

Rules:
1. Only extract EXPLICIT definitions, not implied meanings
2. The term must be defined in THIS text, not just mentioned
3. Include the full definition, not a summary
4. Quote must be verbatim from the text
"""
```

5.3 Definition Extraction Implementation

```
def extract_definitions(self, text: str) -> List[Dict]:
    """Extract defined terms from text."""

    prompt = self.prompts.DEFINITION_PROMPT.format(text=text)

    try:
        response = ollama.chat(
            model=self.model,
            messages=[{'role': 'user', 'content': prompt}],
            format='json'
        )

        content = response['message']['content']
        data = json.loads(content)

        return data.get('definitions', [])

    except json.JSONDecodeError as e:
        print(f"JSON parse error: {e}")
        return []
    except Exception as e:
        print(f"Extraction error: {e}")
        return []
```

5.4 Example Output

```
{
    "definitions": [
        {
            "term": "Company",
            "definition": "PB Fintech Limited, a company incorporated under the
Companies Act, 2013 with CIN U74999DL2008PLC178155",
            "quote": "\"Company\" or \"our Company\" refers to PB Fintech
Limited..."
```

```
        },
        {
            "term": "ESOP-2017",
            "definition": "The Employee Stock Option Plan 2017 approved by the
Board on March 15, 2017",
            "quote": "\"ESOP-2017\" means the Employee Stock Option Plan 2017..."
        }
    ]
}
```

---

6. Stage 2: Entity Extraction

6.1 Entity Types and Attributes

We extract entities with their types and attributes:

```
ENTITY_PROMPT = """You are extracting named entities from an IPO document.

TEXT:
{text}

Extract all entities. Return ONLY valid JSON:

{{
    "entities": [
        {{
            "name": "exact name as written",
            "type": "PERSON|COMPANY|FINANCIAL_METRIC|SECURITY|REGULATORY_BODY|
PRODUCT|LOCATION",
            "attributes": {{
                "role": "if person, their role",
                "percentage": "if shareholding",
                "amount": "if financial",
                "date": "if temporal"
            }},
            "quote": "sentence where entity is mentioned"
        }}
    ]
}}

Entity Type Guidelines:
- PERSON: Directors, officers, promoters, key personnel
- COMPANY: Companies, subsidiaries, joint ventures
- FINANCIAL_METRIC: Revenue, profit, EBITDA, margins (with amounts)
- SECURITY: Equity shares, bonds, options
- REGULATORY_BODY: SEBI, RBI, IRDAI, ROC
- PRODUCT: Products and services offered
- LOCATION: Cities, countries, addresses
```

```
Rules:
1. Extract SPECIFIC named entities, not generic terms
2. Include all relevant attributes mentioned
3. Quote must be from the text
"""
```

6.2 Entity Extraction Implementation

```python
def extract_entities(self, text: str) -> List[Dict]:
    """Extract named entities from text."""

    prompt = self.prompts.ENTITY_PROMPT.format(text=text)

    try:
        response = ollama.chat(
            model=self.model,
            messages=[{'role': 'user', 'content': prompt}],
            format='json'
        )

        content = response['message']['content']
        data = json.loads(content)

        entities = data.get('entities', [])

        # Normalize entity names
        for entity in entities:
            entity['normalized_name'] = self._normalize_name(entity['name'])

        return entities

    except Exception as e:
        print(f"Entity extraction error: {e}")
        return []

def _normalize_name(self, name: str) -> str:
    """Normalize entity name for matching."""
    import re
    # Remove titles, lowercase, remove extra spaces
    name = re.sub(r'^(Mr\.|Mrs\.|Ms\.|Dr\.)\s*', '', name)
    name = ' '.join(name.lower().split())
    return name
```

6.3 Example Output

```
{
    "entities": [
        {
            "name": "Yashish Dahiya",
            "type": "PERSON",
            "attributes": {
```

```
                "role": "Chairman and CEO",
                "shareholding": "4.27%",
                "shares": "17,545,000"
            },
            "quote": "Mr. Yashish Dahiya, our Chairman and CEO, holds 17,545,000
Equity Shares..."
        },
        {
            "name": "PB Fintech Limited",
            "type": "COMPANY",
            "attributes": {
                "cin": "U74999DL2008PLC178155",
                "incorporated": "2008"
            },
            "quote": "PB Fintech Limited (CIN: U74999DL2008PLC178155)..."
        },
        {
            "name": "Revenue FY2021",
            "type": "FINANCIAL_METRIC",
            "attributes": {
                "amount": "957.4 crore",
                "period": "FY2021",
                "growth": "42% YoY"
            },
            "quote": "Revenue from operations was ₹957.4 crore for FY2021..."
        }
    ]
}
```

---

7. Stage 3: Relationship Extraction

7.1 Relationship Prompt

```
RELATIONSHIP_PROMPT = """You are extracting relationships between entities from
an IPO document.

KNOWN ENTITIES:
{entities_json}

TEXT:
{text}

Extract relationships between these entities. Return ONLY valid JSON:

{{
    "relationships": [
        {{
            "subject": "entity name (must match known entities)",
            "predicate": "relationship type",
```

```
            "object": "entity name OR literal value",
            "qualifiers": {{
                "as_of": "date if applicable",
                "percentage": "if applicable"
            }},
            "quote": "sentence showing this relationship"
        }}
    ]
}}

Relationship Types:
- CEO_OF, CFO_OF, DIRECTOR_OF (person → company)
- SUBSIDIARY_OF, PARENT_OF (company → company)
- OWNS_SHARES, HOLDS_STAKE (person/company → company)
- REGULATES (regulatory body → company)
- LOCATED_IN (company → location)
- PROVIDES (company → product/service)
- HAS_VALUE (metric → amount)

Rules:
1. Subject and object should match known entities when possible
2. Include relevant qualifiers (dates, percentages)
3. Quote must support the relationship
"""
```

7.2 Relationship Extraction Implementation

```
def extract_relationships(
    self,
    text: str,
    entities: List[Dict]
) -> List[Dict]:
    """Extract relationships between entities."""

    # Format entities for prompt
    entities_json = json.dumps([
        {'name': e['name'], 'type': e['type']}
        for e in entities
    ], indent=2)

    prompt = self.prompts.RELATIONSHIP_PROMPT.format(
        entities_json=entities_json,
        text=text
    )

    try:
        response = ollama.chat(
            model=self.model,
            messages=[{'role': 'user', 'content': prompt}],
            format='json'
```

```
        )

        content = response['message']['content']
        data = json.loads(content)

        return data.get('relationships', [])

    except Exception as e:
        print(f"Relationship extraction error: {e}")
        return []
```

7.3 Example Output

```
{
    "relationships": [
        {
            "subject": "Yashish Dahiya",
            "predicate": "CEO_OF",
            "object": "PB Fintech Limited",
            "qualifiers": {
                "since": "2008"
            },
            "quote": "Mr. Yashish Dahiya serves as the Chairman and CEO of our
Company"
        },
        {
            "subject": "Policybazaar Insurance Brokers",
            "predicate": "SUBSIDIARY_OF",
            "object": "PB Fintech Limited",
            "qualifiers": {
                "stake": "100%"
            },
            "quote": "Policybazaar Insurance Brokers Private Limited, our wholly-
owned subsidiary"
        },
        {
            "subject": "Yashish Dahiya",
            "predicate": "OWNS_SHARES",
            "object": "PB Fintech Limited",
            "qualifiers": {
                "percentage": "4.27%",
                "shares": "17,545,000"
            },
            "quote": "Mr. Yashish Dahiya holds 17,545,000 Equity Shares (4.27%)"
        }
    ]
}
```

---

8. Stage 4: Event Extraction

8.1 Event Prompt

```
EVENT_PROMPT = """You are extracting events and milestones from an IPO document.

TEXT:
{text}

Extract temporal events. Return ONLY valid JSON:

{{
    "events": [
        {{
            "event_type": "INCORPORATION|ACQUISITION|FUNDING|REGULATORY|
PRODUCT_LAUNCH|IPO|OTHER",
            "description": "what happened",
            "date": "YYYY-MM-DD if known",
            "date_text": "date as written in text",
            "participants": ["entity names involved"],
            "quote": "sentence describing the event"
        }}
    ]
}}

Event Types:
- INCORPORATION: Company formation
- ACQUISITION: M&A activity
- FUNDING: Investment rounds
- REGULATORY: Licenses, approvals
- PRODUCT_LAUNCH: New products/services
- IPO: IPO-related events
- OTHER: Other significant events

Rules:
1. Include date in both parsed (YYYY-MM-DD) and text format
2. List all entities involved as participants
3. Quote must describe the event
"""
```

8.2 Event Extraction Implementation

```
def extract_events(self, text: str) -> List[Dict]:
    """Extract temporal events from text."""

    prompt = self.prompts.EVENT_PROMPT.format(text=text)

    try:
        response = ollama.chat(
            model=self.model,
            messages=[{'role': 'user', 'content': prompt}],
```

```
            format='json'
        )

        content = response['message']['content']
        data = json.loads(content)

        events = data.get('events', [])

        # Parse dates if possible
        for event in events:
            event['parsed_date'] = self._parse_date(
                event.get('date'),
                event.get('date_text')
            )

        return events

    except Exception as e:
        print(f"Event extraction error: {e}")
        return []

def _parse_date(self, iso_date: str, text_date: str) -> Optional[str]:
    """Try to parse date from various formats."""
    from dateutil import parser

    for date_str in [iso_date, text_date]:
        if date_str:
            try:
                parsed = parser.parse(date_str, fuzzy=True)
                return parsed.strftime('%Y-%m-%d')
            except:
                continue
    return None
```

8.3 Example Output

```
{
    "events": [
        {
            "event_type": "INCORPORATION",
            "description": "PB Fintech Limited was incorporated",
            "date": "2008-11-24",
            "date_text": "November 24, 2008",
            "participants": ["PB Fintech Limited"],
            "quote": "Our Company was incorporated on November 24, 2008..."
        },
        {
            "event_type": "FUNDING",
            "description": "Series F funding round",
            "date": "2021-07-15",
```

```
            "date_text": "July 2021",
            "participants": ["PB Fintech Limited", "SoftBank", "Tiger Global"],
            "quote": "In July 2021, we raised $75 million in Series F funding..."
        },
        {
            "event_type": "ACQUISITION",
            "description": "Acquisition of DocPrime",
            "date": "2019-03-01",
            "date_text": "March 2019",
            "participants": ["PB Fintech Limited", "DocPrime"],
            "quote": "We acquired DocPrime Technologies in March 2019..."
        }
    ]
}
```

---

9. Stage 5: Entity Resolution

9.1 The Duplicate Problem

The same entity may be mentioned differently across chunks: - “Yashish Dahiya” / “Mr. Dahiya” / “the CEO” / “Mr. Yashish Dahiya” - “PB Fintech” / “PB Fintech Limited” / “the Company” / “our Company”

Entity resolution merges these into a single canonical entity.

9.2 Resolution Strategies

```python
def resolve_entities(self, chunk_results: List[Dict]) -> List[Dict]:
    """Merge duplicate entities across chunks."""

    # Collect all entities
    all_entities = []
    for result in chunk_results:
        for entity in result.get('entities', []):
            all_entities.append(entity)

    # Group by normalized name + type
    groups = {}
    for entity in all_entities:
        key = (entity['normalized_name'], entity['type'])
        if key not in groups:
            groups[key] = []
        groups[key].append(entity)

    # Merge each group
    merged = []
    for key, group in groups.items():
        merged_entity = self._merge_entity_group(group)
        merged.append(merged_entity)
```

```
    # Second pass: Check for aliases
    merged = self._resolve_aliases(merged)

    return merged

def _merge_entity_group(self, group: List[Dict]) -> Dict:
    """Merge a group of same entities."""

    # Use most common name as canonical
    from collections import Counter
    names = [e['name'] for e in group]
    canonical = Counter(names).most_common(1)[0][0]

    # Merge attributes (keep all unique values)
    merged_attrs = {}
    for entity in group:
        for key, value in entity.get('attributes', {}).items():
            if key not in merged_attrs:
                merged_attrs[key] = value

    # Collect all quotes as evidence
    quotes = [e.get('quote') for e in group if e.get('quote')]

    return {
        'canonical_name': canonical,
        'type': group[0]['type'],
        'aliases': list(set(names) - {canonical}),
        'attributes': merged_attrs,
        'evidence_quotes': quotes,
        'mention_count': len(group)
    }

def _resolve_aliases(self, entities: List[Dict]) -> List[Dict]:
    """Check for entity aliases using LLM."""

    # For efficiency, only check entities with same type
    by_type = {}
    for entity in entities:
        t = entity['type']
        if t not in by_type:
            by_type[t] = []
        by_type[t].append(entity)

    # Check each type group for potential aliases
    for entity_type, group in by_type.items():
        if len(group) > 1:
            aliases = self._find_aliases_llm(group)
            # Merge aliased entities
```

```
            # ... implementation

    return entities
```

9.3 LLM-Based Alias Detection

```
ALIAS_PROMPT = """Given these entities, identify which ones refer to the same
real-world entity:

ENTITIES:
{entities_json}

Return pairs of aliases:
{{
    "alias_pairs": [
        {{
            "entity1": "name1",
            "entity2": "name2",
            "confidence": 0.0-1.0,
            "reason": "why they are the same"
        }}
    ]
}}

Only return pairs you are confident are the same entity.
"""
```

---

10. Stage 6: Validation

10.1 Validation Checks

```
def validate(
    self,
    entities: List[Dict],
    chunk_results: List[Dict]
) -> Dict:
    """Validate extracted KG for quality."""

    report = {
        'total_entities': len(entities),
        'total_relationships': 0,
        'total_events': 0,
        'issues': [],
        'quality_score': 0.0
    }

    # Count totals
    for result in chunk_results:
        report['total_relationships'] += len(result.get('relationships', []))
```

```
    report['total_events'] += len(result.get('events', []))

# Check for orphan relationships
entity_names = {e['canonical_name'].lower() for e in entities}
for result in chunk_results:
    for rel in result.get('relationships', []):
        subj = rel.get('subject', '').lower()
        obj = rel.get('object', '').lower()

        if subj not in entity_names:
            report['issues'].append({
                'type': 'orphan_subject',
                'relationship': rel,
                'message': f"Subject '{subj}' not in entities"
            })

        if obj not in entity_names and not self._is_literal(obj):
            report['issues'].append({
                'type': 'orphan_object',
                'relationship': rel,
                'message': f"Object '{obj}' not in entities"
            })

# Check for duplicate relationships
seen_rels = set()
for result in chunk_results:
    for rel in result.get('relationships', []):
        key = (rel.get('subject'), rel.get('predicate'), rel.get('object'))
        if key in seen_rels:
            report['issues'].append({
                'type': 'duplicate_relationship',
                'relationship': rel
            })
        seen_rels.add(key)

# Calculate quality score
total_items = (
    report['total_entities'] +
    report['total_relationships'] +
    report['total_events']
)

if total_items > 0:
    issue_penalty = len(report['issues']) / total_items
    report['quality_score'] = max(0, 1.0 - issue_penalty)

return report
```

10.2 Validation Rules

Check	Description	Severity
Orphan subjects	Relationship subject not in entities	Warning
Orphan objects	Relationship object not in entities	Warning
Duplicate relationships	Same relationship extracted twice	Info
Missing evidence	Extraction without quote	Warning
Invalid dates	Unparseable date formats	Error
Empty entities	Entity with no attributes	Info

---

11. LLM Prompting Strategies

11.1 Prompt Engineering Principles

1. Be Specific About Format

```
Return ONLY valid JSON:
{
    "entities": [...]
}

Do NOT include any text before or after the JSON.
```

2. Provide Examples

```
Example output:
{
    "entities": [
        {
            "name": "John Smith",
            "type": "PERSON",
            "attributes": {"role": "CEO"}
        }
    ]
}
```

3. Define Categories Clearly

```
Entity Types:
- PERSON: Directors, officers, promoters, key personnel
- COMPANY: Companies, subsidiaries, joint ventures
...
```

4. Set Constraints

```
Rules:
1. Only extract EXPLICIT mentions, not implied
```

```
2. Quote must be verbatim from text
3. If unsure, omit rather than guess
```

11.2 Handling LLM Failures

```python
def _call_llm_with_retry(
    self,
    prompt: str,
    max_retries: int = 3
) -> Optional[Dict]:
    """Call LLM with retry logic."""

    for attempt in range(max_retries):
        try:
            response = ollama.chat(
                model=self.model,
                messages=[{'role': 'user', 'content': prompt}],
                format='json'
            )

            content = response['message']['content']

            # Try to parse JSON
            data = json.loads(content)
            return data

        except json.JSONDecodeError:
            # Try to extract JSON from response
            data = self._extract_json(content)
            if data:
                return data

            print(f"Attempt {attempt + 1}: JSON parse failed")

        except Exception as e:
            print(f"Attempt {attempt + 1}: Error - {e}")

        # Wait before retry
        time.sleep(1)

    return None

def _extract_json(self, text: str) -> Optional[Dict]:
    """Try to extract JSON from text that might have extra content."""
    import re

    # Look for JSON object
    match = re.search(r'\{[\s\S]*\}', text)
    if match:
        try:
```

```
                return json.loads(match.group())
            except:
                pass

    return None
```

11.3 Prompt Templates File

```
# src/utils/kg_prompts.py

class KGPrompts:
    """Collection of prompts for KG extraction stages."""

    DEFINITION_PROMPT = """..."""
    ENTITY_PROMPT = """..."""
    RELATIONSHIP_PROMPT = """..."""
    EVENT_PROMPT = """..."""
    ALIAS_PROMPT = """..."""
    VALIDATION_PROMPT = """..."""

    def get_prompt(self, stage: str, **kwargs) -> str:
        """Get prompt for a stage with variables filled in."""
        prompt_template = getattr(self, f'{stage.upper()}_PROMPT')
        return prompt_template.format(**kwargs)
```

---

12. Storage and Retrieval

12.1 Saving to PostgreSQL

```
# From src/database/kg_repositories.py

class KGEntityRepository:
    """Repository for kg_entities table."""

    @staticmethod
    def create(session, entity: Dict) -> int:
        """Create a new entity."""

        sql = text("""
            INSERT INTO kg_entities
            (document_id, canonical_name, entity_type, attributes)
            VALUES (:doc_id, :name, :type, :attrs)
            ON CONFLICT (document_id, canonical_name, entity_type)
            DO UPDATE SET attributes = kg_entities.attributes || :attrs
            RETURNING id
        """)

        result = session.execute(sql, {
            'doc_id': entity['document_id'],
```

```
            'name': entity['canonical_name'],
            'type': entity['type'],
            'attrs': json.dumps(entity.get('attributes', {}))
        })

        return result.scalar()

    @staticmethod
    def find_by_name(session, document_id: str, name: str) -> Optional[Dict]:
        """Find entity by name (case-insensitive)."""

        sql = text("""
            SELECT * FROM kg_entities
            WHERE document_id = :doc_id
            AND LOWER(canonical_name) = LOWER(:name)
        """)

        result = session.execute(sql, {
            'doc_id': document_id,
            'name': name
        }).fetchone()

        return dict(result) if result else None
```

12.2 KG Query for RAG

```
class KGRAG:
    """Knowledge Graph based retrieval."""

    def __init__(self, document_id: str):
        self.document_id = document_id

    def query(self, question: str) -> Dict:
        """Query KG based on question."""

        # Extract entities from question
        entities = self._extract_question_entities(question)

        # Look up in KG
        results = {
            'entities': [],
            'relationships': [],
            'definitions': []
        }

        with get_db() as session:
            for entity_name in entities:
                # Find entity
                entity = KGEntityRepository.find_by_name(
                    session, self.document_id, entity_name
```

```
                )
                if entity:
                    results['entities'].append(entity)

                    # Get relationships
                    rels = ClaimRepository.find_by_entity(
                        session, self.document_id, entity['id']
                    )
                    results['relationships'].extend(rels)

            # Check for definition queries
            if 'what is' in question.lower() or 'what does' in question.lower():
                terms = self._extract_terms(question)
                for term in terms:
                    defn = DefinedTermRepository.find_by_term(
                        session, self.document_id, term
                    )
                    if defn:
                        results['definitions'].append(defn)

        return results

    def format_context(self, results: Dict) -> str:
        """Format KG results as context for LLM."""

        parts = []

        if results['definitions']:
            parts.append("DEFINITIONS:")
            for d in results['definitions']:
                parts.append(f"- {d['term']}: {d['definition']}")

        if results['entities']:
            parts.append("\nENTITIES:")
            for e in results['entities']:
                attrs = ', '.join(f"{k}={v}" for k, v in e['attributes'].items())
                parts.append(f"- {e['canonical_name']} ({e['entity_type']}):
{attrs}")

        if results['relationships']:
            parts.append("\nRELATIONSHIPS:")
            for r in results['relationships']:
                parts.append(f"- {r['subject']} {r['predicate']} {r['object']}")

        return '\n'.join(parts)
```

12.3 Running the Pipeline

```
# From command line
python scripts/build_kg_v2.py --document policybazar_ipo
```

```
# With chunk limit (for testing)
python scripts/build_kg_v2.py --document policybazar_ipo --limit 10

# In background
nohup python scripts/build_kg_v2.py --document policybazar_ipo > kg_build.log
2>&1 &
```

Summary

This chapter covered:

1. **What is a KG**: Nodes, edges, properties
2. **Why KG for IPO docs**: Structured facts, entity lookup
3. **Schema design**: Entity types, relationships, tables
4. **6-stage pipeline**: Definition → Entity → Relationship → Event → Resolution → Validation
5. **Definition extraction**: Glossary terms with quotes
6. **Entity extraction**: Named entities with types and attributes
7. **Relationship extraction**: Connections between entities
8. **Event extraction**: Temporal facts and milestones
9. **Entity resolution**: Merging duplicates
10. **Validation**: Quality checks
11. **LLM prompting**: Strategies and error handling
12. **Storage/retrieval**: PostgreSQL repos and KGRAG class

Chapter 5: RAG Query Pipeline

From Question to Answer: The Complete Journey

Table of Contents for This Chapter

1. Query Pipeline Overview

1.1 What Happens When a User Asks a Question?

When a user types a question like "Who is the CEO and what is their shareholding?", our system goes through a carefully orchestrated pipeline of steps. Understanding this pipeline is crucial because each stage has a specific purpose, and the quality of the final answer depends on every stage working correctly.

The pipeline consists of **five main stages**: Query Understanding, Information Retrieval, Context Assembly, Prompt Construction, and Answer Generation. Let's visualize this journey:

Stage 1: Query Understanding The system first analyzes the question to understand what the user is really asking. Is this a factual question about a person? Is it asking for a definition? Does it require comparing information? The answer to these questions determines how we search for information.

Stage 2: Information Retrieval Based on the query analysis, the system searches for relevant information. This might involve vector similarity search (finding similar text), Knowledge Graph lookup (finding structured facts), or both. The choice depends on the type of question.

Stage 3: Context Assembly The retrieved information needs to be organized and formatted. We merge results from different sources, remove duplicates, rank by relevance, and format everything into a coherent context that the LLM can understand.

Stage 4: Prompt Construction We build a carefully crafted prompt that includes instructions for the LLM, the assembled context, and the user's question. Good prompt engineering is critical for high-quality answers.

Stage 5: Answer Generation Finally, the LLM (Llama3 via Ollama) generates the answer based on the prompt. We stream this response token-by-token so the user sees words appearing in real-time, rather than waiting for the complete answer.

1.2 Why This Pipeline Design?

This multi-stage design exists because each stage has different requirements and can fail independently. By separating concerns:

- We can **debug issues** at each stage separately
- We can **optimize** each stage independently
- We can **fall back** gracefully if one component fails
- We can **measure performance** at each step

For example, if answers are poor quality, the problem might be in retrieval (wrong chunks), context assembly (information lost), prompting (unclear instructions), or generation (model hallucinating). The pipeline design helps us isolate the issue.

1.3 Timing and Performance

Understanding the timing of each stage helps set user expectations and identify bottlenecks:

Stage	Typical Duration	What's Happening
Query Understanding	10-50 milliseconds	Analyzing question type, extracting entities
Vector Search	50-100 milliseconds	Database query with HNSW index lookup
KG Query	20-50 milliseconds	PostgreSQL queries for entities and relationships
Context Assembly	10-20 milliseconds	Merging, formatting, truncating
LLM Generation	2-5 seconds	Neural network inference (the slowest part)

The total time from question to complete answer is typically 3-6 seconds. However, because we stream the response, the user sees the first word within 200-500 milliseconds after retrieval completes, which feels much more responsive.

---

2. Query Understanding

2.1 Why Query Analysis Matters

Not all questions are the same. Consider these three questions: 1. "Who is the CEO?" - This is asking about a specific person and their role 2. "Describe the business model" - This needs narrative explanation 3. "What does ESOP mean?" - This is asking for a definition

Each of these questions is best answered using different retrieval strategies. The first question can be answered perfectly from our Knowledge Graph where we have explicit "CEO_OF" relationships. The second question needs vector search to find descriptive text passages. The third needs our definitions table.

If we used the same approach for all questions, we would either miss the best information (using only vector search for "Who is the CEO?") or provide verbose, unfocused answers (using many chunks for "What does ESOP mean?").

2.2 Question Type Classification

Our system classifies questions into six categories:

Entity Questions

Questions like "Who is the CEO?", "Who are the promoters?", or "Name the directors." These are best answered from the Knowledge Graph because we have explicit entities and relationships. Vector search might return passages that mention the CEO without explicitly stating who it is.

Definition Questions
Questions like "What does ESOP mean?", "Define DRHP", or "What is the meaning of promoter group?" These should first check our defined_terms table, which contains explicit definitions from the document's glossary section. If not found there, we fall back to vector search.

Factual Questions
Questions seeking specific facts like "When was the company founded?", "How much was the revenue?", or "What is the issue price?" These often have discrete answers that our Knowledge Graph events and entities can provide directly.

Descriptive Questions
Questions requiring explanation like "Describe the business model", "Explain the risks", or "Tell me about the competition." These need narrative text, so vector search is ideal. We retrieve multiple related chunks to provide comprehensive context.

Comparative Questions
Questions asking for comparison like "Compare revenue growth over years" or "What's the difference between promoter and public shareholding?" These need information from multiple sources and require the LLM to synthesize data.

Complex Questions
Multi-part questions or those requiring reasoning across multiple facts. These need our hybrid approach with both vector and KG retrieval.

2.3 Entity Extraction from Questions

Beyond classifying the question type, we also extract potential entity names from the question. When a user asks "What is Yashish Dahiya's shareholding?", we extract "Yashish Dahiya" as a named entity and can look it up directly in our Knowledge Graph.

We look for several patterns: - **Role titles**: CEO, CFO, Director, Chairman, Promoter - **Regulatory bodies**: SEBI, RBI, IRDAI - **Financial terms**: Revenue, EBITDA, Profit, Margin - **Document terms**: ESOP, DRHP, RHP, IPO - **Proper names**: Capitalized names like "Yashish Dahiya" - **Quoted terms**: Anything in quotation marks

This extraction helps us decide which entities to look up in the Knowledge Graph and makes retrieval more targeted.

2.4 Retrieval Mode Selection

Based on the question classification and extracted entities, we select one of five retrieval modes:

Vector Only: Used when there's no Knowledge Graph available, or for purely descriptive questions. We rely entirely on semantic similarity search.

KG Only: Used for definition lookups or when we're confident the KG has the answer. No vector search is performed.

KG First: We query the Knowledge Graph first. If we find relevant entities/relationships, we use those as the primary source and add a couple of supporting text chunks. If KG returns nothing, we fall back to vector search.

Vector First: We do vector search as the primary method, but optionally enhance with KG facts if available. Good for descriptive questions where KG can add precise facts.

Hybrid: We run both vector search and KG query in parallel and merge the results. Best for complex questions that need both narrative context and precise facts.

---

3. Retrieval Strategies

3.1 Vector-Only Retrieval Explained

Vector retrieval works by finding text passages that are semantically similar to the question. Here's how it works conceptually:

1. **Embed the Question**: The user's question is converted into a 384-dimensional vector using the same embedding model (all-MiniLM-L6-v2) that was used to embed the document chunks during ingestion.
2. **Search the Vector Index**: Using pgvector's HNSW index, we find the chunks whose embedding vectors are closest to the question vector. "Closest" is measured by cosine similarity.
3. **Return Top-K Results**: We retrieve the top 5 (configurable) most similar chunks, along with their similarity scores and metadata (page numbers, section titles).

Vector search shines when the question and answer share semantic meaning even if they don't share exact words. The question "How is the company performing financially?" will match chunks about "revenue growth," "profit margins," and "financial results" even though those exact words aren't in the question.

However, vector search has limitations. It doesn't understand structure. If you ask "Who is the CEO?", it might return a chunk that mentions the CEO's background but doesn't explicitly name them. The chunk "The management team has decades of experience in the insurance industry" might score high but doesn't answer the question.

3.2 Knowledge Graph Retrieval Explained

KG retrieval works by looking up structured facts we extracted during document processing:

1. **Identify Relevant Entities**: From the question, we identify which entities to look up. If the question mentions "CEO", we search for entities with that role or relationship type.
2. **Look Up Entity Attributes**: Once we find the entity (e.g., "Yashish Dahiya"), we retrieve all its attributes: role, shareholding, age, etc.
3. **Traverse Relationships**: We also find relationships involving this entity: "CEO_OF → PB Fintech", "OWNS_SHARES → 4.27%", etc.

4. **Check Definitions**: If the question is asking about a term, we look in the defined_terms table.

5. **Find Relevant Events**: For temporal questions, we query the events table.

KG retrieval excels at providing precise, structured answers. "Who is the CEO?" returns a direct answer: "Yashish Dahiya, entity type: PERSON, role: CEO, shareholding: 4.27%."

However, KG has limitations too. It only contains facts we explicitly extracted. If the document discusses something in narrative form that we didn't extract into structured data, the KG won't have it. Also, KG can't provide context or explanation—it gives facts but not the surrounding narrative.

3.3 When to Use Which Strategy

The choice of retrieval strategy significantly impacts answer quality. Here's our decision logic:

Use Vector Search When: - User asks for explanation or description - Question contains words like "describe," "explain," "how does" - We need narrative context around facts - The topic wasn't extracted into KG (rare terms, complex scenarios)

Use KG Search When: - User asks a specific factual question - Question contains "who is," "what is," "define" - We need precise data (names, numbers, dates) - Definitions are requested

Use Hybrid When: - Complex, multi-part questions - We need both precision and context - Comparing or synthesizing information - Uncertain which approach is best

---

4. Hybrid RAG: Vector + KG

4.1 The Power of Combination

Hybrid RAG combines the precision of Knowledge Graphs with the flexibility of vector search. This combination is more powerful than either approach alone.

Consider the question: "What is the CEO's background and shareholding?"

Vector-only approach might return: - Chunk about leadership experience - Chunk about board composition - Chunk mentioning shareholding patterns

These chunks are relevant but scattered. The answer about specific shareholding might be buried in prose, or might not appear at all if the most similar chunks are about leadership rather than ownership.

KG-only approach would return: - Entity: Yashish Dahiya - Role: Chairman and CEO - Shareholding: 4.27% (17,545,000 shares)

This is precise but lacks context. We don't learn about his background, prior experience, or how he came to lead the company.

Hybrid approach provides: - KG facts: Precise name, exact shareholding percentage - Vector chunks: Narrative about his IIT/IIM background, founding story, leadership philosophy

The LLM can synthesize both into a comprehensive answer that is both accurate and informative.

4.2 How Hybrid Search Works

Our hybrid approach follows this process:

Step 1: Parallel Execution We run vector search and KG query simultaneously. This is faster than running one after the other because the database can process both queries in parallel.

Step 2: KG Results Evaluation We check if the KG returned useful data. Did we find any entities? Any relationships? Any definitions? If KG found relevant structured data, we know we have high-confidence facts.

Step 3: Vector Results Enhancement The vector search results provide supporting context. Even when KG gives us facts, the vector chunks add narrative explanation that helps the LLM construct a natural-sounding answer.

Step 4: Intelligent Fallback If KG returns nothing (maybe the question is about something we didn't extract), we rely more heavily on vector results. If vector returns low-similarity results (below 0.5 threshold), we might warn the user that the information might not be in the document.

4.3 Mode Variations

We have three hybrid sub-strategies:

Full Hybrid: Both retrieval methods are treated equally. We retrieve 5 chunks from vector search and all relevant KG facts, then merge them into context. Best for complex questions.

KG-First Hybrid: We prioritize KG results. Only retrieve 2-3 supporting vector chunks (instead of 5) to keep the context focused on the structured facts. Best for factual questions where we found good KG data.

Vector-First Hybrid: We prioritize vector chunks but enhance with KG facts as "bonus" information. Best for descriptive questions where we want narrative flow but can add precise data points.

---

5. Context Assembly

5.1 Why Context Assembly Matters

After retrieval, we have raw results: chunks of text, entity records, relationship data. We can't just dump all of this into the LLM prompt. We need to organize it intelligently because:

1. **LLMs have context limits**: We can't include unlimited information
2. **Irrelevant info hurts answers**: Too much noise dilutes the signal
3. **Format affects comprehension**: How we present information affects how well the LLM uses it
4. **Duplicates waste space**: The same information shouldn't appear twice

Context assembly transforms raw retrieval results into a well-organized, focused context that maximizes the LLM's ability to answer correctly.

5.2 Merging Vector and KG Results

When merging results from both sources, we follow this priority order:

Priority 1: Structured Facts from KG These go first in the context because they are our highest-confidence information. They came directly from the document and were explicitly extracted. We format them clearly:

"STRUCTURED FACTS FROM DOCUMENT: - Entity: Yashish Dahiya (PERSON) - Role: Chairman and CEO - Shareholding: 4.27% - Relationship: CEO_OF ⟶ PB Fintech Limited"

Priority 2: Definitions If any definitions were found, they appear next. Definitions help the LLM understand terminology used elsewhere in the context.

Priority 3: Text Chunks Vector-retrieved chunks come last, providing supporting narrative. Each chunk is labeled with its source (page numbers, section title, relevance score) so the LLM can cite it and so we can debug retrieval issues.

5.3 Deduplication

Often, the same information appears in multiple places. For example, the CEO's name might appear in an "Overview" chunk and also in a "Management" chunk, as well as in our KG entity data.

We handle duplication at two levels:

Chunk-Level Deduplication: Before including a chunk, we check if it overlaps significantly (more than 70%) with chunks we've already included. If so, we skip it. This happens when adjacent chunks share text due to our overlap strategy.

Fact-Level Deduplication: If the KG already contains a precise fact (like shareholding percentage), but a vector chunk also mentions it in prose, we let both appear. The structured fact provides precision; the prose provides context. The LLM is smart enough to use both without being confused.

5.4 Truncation and Limits

Our LLM can handle approximately 4,000 tokens of context (about 3,000 words). Beyond this, earlier information gets "forgotten" by the model. We set a practical limit of about 3,000 tokens for context, leaving room for the question and response.

If our assembled context exceeds this limit, we truncate: 1. First, we reduce the number of chunks from 5 to 3 2. If still too long, we shorten each chunk to its first 300 characters 3. As a last resort, we cut the context and add "[Context truncated]" to alert the LLM

Truncation is rare in practice because our chunks are already sized appropriately (300-500 words each).

---

6. Prompt Engineering

6.1 The Art of Instructing the LLM

The prompt we send to the LLM is not just "here's some context, answer this question." Careful prompt engineering dramatically improves answer quality. Our prompt has several components:

System Instructions: These set the LLM's "personality" and constraints. We tell it to: - Act as an expert on IPO documents - Answer ONLY from the provided context (no making things up) - Cite sources when making claims - Admit when information isn't available - Be precise with numbers and names

Context Presentation: How we format the context matters. We use clear headers ("STRUCTURED FACTS" vs "DOCUMENT EXCERPTS"), bullet points for facts, and source labels for chunks. This helps the LLM parse the information correctly.

Question Framing: We repeat the user's question clearly and indicate where the answer should begin. This focuses the LLM on the task.

Response Format Guidelines: We suggest a structure: start with a direct answer, then provide supporting details, then mention sources, then add caveats if needed.

6.2 Why These Instructions Work

Each instruction addresses a known LLM failure mode:

"Answer ONLY from context" prevents hallucination. Without this, the LLM might use its training knowledge about IPOs in general, which could be wrong for this specific company.

"Cite sources" forces the LLM to ground its claims in the provided text. If the LLM can't cite a source for something, it often shouldn't say it.

"Admit when information unavailable" prevents the LLM from making confident-sounding but incorrect statements. Better to say "the document doesn't specify" than to guess.

"Be precise with numbers" prevents the LLM from rounding or approximating. "4.27%" is better than "about 4%."

6.3 Question-Specific Prompts

For certain question types, we use specialized prompts:

For Definition Questions: We specifically ask the LLM to provide the official definition and explain related terms.

For Timeline Questions: We ask for events in chronological order with dates.

For Comparison Questions: We ask for a structured comparison with similarities, differences, and relevant data.

These specialized prompts help the LLM understand exactly what format of answer is expected, leading to more useful responses.

6.4 The Complete Prompt Structure

Our final prompt looks like this (conceptually):

```
[System Instructions about being an IPO document expert]
[Guidelines about accuracy, citations, and admitting limitations]

CONTEXT FROM DOCUMENT:
[Structured facts from KG]
[Definitions if any]
[Text chunks with source labels]

QUESTION: [User's question]

ANSWER:
```

This structure has been refined through experimentation to produce the best results with our specific LLM (Llama3).

---

7. LLM Generation

7.1 How Ollama Works

Ollama is our local LLM runner. It allows us to run powerful language models like Llama3 directly on the server without sending data to external APIs. This is important for several reasons:

Privacy: IPO documents contain sensitive financial information. By running locally, we ensure no data leaves our infrastructure.

Cost: API-based LLMs charge per token. Local models have a one-time setup cost but no per-query fees.

Latency: Local models eliminate network round-trip time to cloud APIs.

Availability: Local models work even without internet connectivity.

When we send our prompt to Ollama, it: 1. Tokenizes the prompt (converts text to numbers the model understands) 2. Processes tokens through the neural network 3. Generates response tokens one at a time 4. Each token is predicted based on all previous tokens

7.2 Understanding Token Generation

LLMs generate text one token at a time. A token is roughly a word or part of a word (like "shareholding" might be one token, but "shareholding's" might be two tokens).

For each token generated, the model: 1. Looks at all the context and all tokens generated so far 2. Calculates the probability of every possible next token 3. Selects the most likely token (or samples probabilistically) 4. Adds that token to the output and repeats

This means generating 100 tokens requires 100 forward passes through the neural network. That's why generation takes 2-5 seconds even on modern hardware.

7.3 Streaming vs. Non-Streaming

Non-Streaming: Wait until the entire response is generated, then return it all at once. The user stares at a loading spinner for 3-5 seconds.

Streaming: Return each token as it's generated. The user starts seeing words within 200-500ms and watches the answer appear in real-time.

We use streaming because it provides a much better user experience. Even though the total time is the same, the perceived speed is much faster because users can start reading immediately.

7.4 Response Quality Considerations

Several factors affect the quality of generated responses:

Context Quality: The LLM can only work with what we give it. If retrieval found irrelevant chunks, the answer suffers.

Prompt Clarity: Unclear instructions lead to inconsistent responses.

Model Capability: Llama3 is highly capable but not perfect. It may occasionally misinterpret instructions or make mistakes.

Token Limits: If the context is too long, earlier parts may be ignored (though our chunking and truncation prevent this).

We continuously monitor answer quality and refine our retrieval and prompting based on observed failures.

---

8. Streaming Responses

8.1 The Technical Challenge

Streaming responses from LLM to user involves several moving parts:

LLM to Backend: Ollama provides a streaming API that yields tokens as they're generated. Our Python code receives these tokens in real-time.

Backend to Frontend: We need to transmit tokens from our Flask server to the user's browser as they arrive. Standard HTTP responses wait until the entire response is ready, which defeats the purpose.

Frontend Rendering: The browser needs to receive tokens and append them to the display in real-time, creating the effect of text being "typed out."

8.2 How We Implement Streaming

Our streaming system works in three layers:

Layer 1: Server-Sent Events / NDJSON Instead of returning a single JSON response, our Flask endpoint returns a stream of newline-delimited JSON (NDJSON). Each line is a separate JSON object containing either a status update, a token, or a completion message.

Layer 2: Generator Function In Python, we use a generator function that yields chunks of data. Each `yield` statement sends that data immediately to the client without waiting for the complete response. Flask's `stream_with_context` enables this behavior.

Layer 3: Frontend Fetch API Modern browsers support streaming fetch responses. Instead of awaiting the entire response body, we use a ReadableStream reader that processes chunks as they arrive. Each chunk is parsed, and tokens are appended to the displayed answer.

8.3 User Experience Benefits

Streaming provides several UX improvements:

Faster Perceived Speed: Users see the first word in under a second, even though the full answer takes several seconds.

Progressive Disclosure: Users can start reading and processing information while more is loading.

Interruptibility: If users see the answer is going in the wrong direction, they can cancel and ask a different question (future enhancement).

Status Updates: We send status messages during retrieval ("Searching document…", "Found 5 relevant sections…") that keep users informed about progress.

8.4 Error Handling in Streams

Streaming complicates error handling because we've already started sending the response when an error might occur during generation. We handle this with message types:

- `{"type": "status", "message": "..."}` - Progress updates
- `{"type": "token", "content": "..."}` - Actual answer content
- `{"type": "done", ...}` - Completion with metadata
- `{"type": "error", "message": "..."}` - Error notification

If an error occurs mid-stream, we send an error message. The frontend handles this by showing the partial response (if any) along with the error message.

---

9. Source Attribution

9.1 Why Sources Matter

When users ask questions about IPO documents, they often need to verify the answers. This is especially important in financial contexts where decisions might be made based on the information.

Source attribution serves multiple purposes:

Trust Building: Users can see exactly where information comes from. If the answer says "The CEO owns 4.27%," users can look at page 45 to verify.

Transparency: Users understand that answers come from the document, not from the AI "making things up."

Error Detection: If an answer seems wrong, users can check the sources to see if the problem is in retrieval (wrong chunks) or generation (LLM misunderstanding).

Compliance: In financial and legal contexts, being able to cite sources may be regulatory or compliance requirements.

9.2 How We Track Sources

Every piece of context has metadata attached:

For Vector Chunks: - Page numbers (start and end) - Section title - Similarity score (how relevant the system thought it was) - The actual text snippet

For KG Facts: - The evidence quote from the original document - Page where the evidence appears - Confidence score of the extraction

When the LLM generates an answer, we encourage it to cite sources using markers like "[Page 45]" or "[Source 2]". We also display all sources used in a panel below the answer, so users can explore the raw source material.

9.3 Enabling User Verification

In our user interface, sources are displayed in a collapsible panel:

1. **Summary Bar**: Shows "Answer based on 5 sources"
2. **Expandable List**: Each source shows page numbers, section, and relevance
3. **Preview**: Users can see a preview of each chunk's text
4. **Jump to Page**: Future enhancement could link to the actual PDF page

This design makes verification easy without cluttering the main answer area.

---

10. Error Handling

10.1 What Can Go Wrong

Many things can fail in our pipeline:

Retrieval Failures: - Database connection lost - Vector index not found - No chunks have embeddings - Query timeout

Generation Failures: - Ollama service not running - Model not downloaded - Out of memory during inference - Generation timeout

Context Issues: - No relevant chunks found - Context too long for model - Contradictory information in context

User Input Issues: - Empty question - No document selected - Question in unsupported language

10.2 Graceful Degradation

Rather than failing completely, we try to degrade gracefully:

Hybrid Fails → Try Vector Only: If the KG query fails, we can still answer using just vector search.

Vector Search Returns Low Scores → Warn User: If all chunks have similarity below 0.5, we warn: "The information might not be in this document."

Generation Fails → Return Context: If the LLM fails, we can return the raw context so users at least see what we found.

Timeout → Partial Results: If queries take too long, we return whatever we found so far.

10.3 User-Friendly Messages

Technical errors are translated to user-friendly messages:

Technical Error	User Message
Database connection refused	"The system is temporarily unavailable. Please try again."
No chunks found	"Sorry, I couldn't find relevant information for your question."
LLM service down	"Answer generation is currently unavailable. Please try again."
Empty question	"Please enter a question."
No document selected	"Please select a document before asking questions."

The goal is to help users understand what happened and what they can do about it, without exposing technical jargon.

---

11. Performance Optimization

11.1 Identifying Bottlenecks

The slowest parts of our pipeline are: 1. **LLM Generation**: 2-5 seconds (unavoidable with local models) 2. **Vector Search**: 50-100ms (can be optimized with better indexes) 3. **Embedding Computation**: ~50ms (done once per query)

LLM generation time is largely fixed by the hardware and model size. Our optimization efforts focus on reducing the time before generation starts.

11.2 Caching Strategies

We employ several caching strategies:

Query Cache: If a user asks the exact same question twice, we return the cached answer immediately. This is especially useful in demos or when users refresh the page.

Embedding Cache: We cache query embeddings. If the same text is embedded multiple times, we reuse the cached vector instead of recomputing.

Model Loading: The embedding model and LLM stay loaded in memory. Loading them fresh for each query would add several seconds.

11.3 Parallel Processing

Where possible, we run operations in parallel:

Vector + KG Retrieval: In hybrid mode, vector search and KG queries run simultaneously. Since they're independent, this halves the retrieval time compared to running sequentially.

Async Database Connections: We use connection pooling so database operations don't wait for each other.

11.4 Index Optimization

Our vector search uses HNSW (Hierarchical Navigable Small World) indexing with tuned parameters:

- **m = 16**: Maximum connections per node (balance between speed and accuracy)
- **ef_construction = 64**: Build-time thoroughness (higher = better index, slower build)
- **ef_search = 100**: Query-time thoroughness (set at runtime for best recall)

These parameters were chosen to give sub-100ms search times while maintaining 95%+ recall.

---

12. The Complete Flow

12.1 End-to-End Walkthrough

Let's trace a complete query through the system:

User Action: Types "Who is the CEO and what is their shareholding?" and clicks Ask.

Step 1: Request Received (0ms) Flask receives the POST request with the question and document ID. Input validation passes.

Step 2: Query Analysis (10ms) We classify this as an "entity" question (because of "Who is"). We extract "CEO" and "shareholding" as key terms. We select "KG_FIRST" retrieval mode.

Step 3: KG Retrieval (30ms) We search the kg_entities table for entities with type "PERSON" and relationship "CEO_OF". We find "Yashish Dahiya" with attributes including shareholding "4.27%". We also retrieve the CEO_OF relationship and the evidence quote.

Step 4: Vector Retrieval (70ms) Simultaneously, we embed the question and search for similar chunks. We find 2 chunks about leadership and management that mention the CEO.

Step 5: Context Assembly (80ms) We merge results:

```
STRUCTURED FACTS:
- Entity: Yashish Dahiya (PERSON, CEO, shareholding: 4.27%)
- Relationship: CEO_OF → PB Fintech Limited

DOCUMENT EXCERPTS:
[Source 1] (Pages 45-46, Management) "Mr. Yashish Dahiya..."
```

Step 6: Prompt Construction (85ms) We build the complete prompt with system instructions, context, and question.

Step 7: LLM Generation (90ms - 4000ms) We call Ollama with streaming enabled. First token arrives at ~200ms. Tokens stream continuously for approximately 4 seconds until the answer is complete.

Step 8: Response Complete (4000ms) Final "done" event sent with metadata. User sees complete answer with sources.

Total time: ~4 seconds Time to first visible word: ~300ms

12.2 What Makes This Effective

Several design choices contribute to the system's effectiveness:

Right Tool for Right Job: Using KG for precise facts and vector for context means we get both accuracy and richness.

Streaming for Responsiveness: Users perceive the system as fast because they see progress immediately.

Grounded Answers: By constraining the LLM to answer from context only, we avoid hallucination.

Source Transparency: Users can verify answers, building trust in the system.

Graceful Degradation: Even when components fail, users get useful feedback rather than cryptic errors.

Summary

This chapter explained the complete journey from user question to generated answer:

1. **Pipeline Overview**: Five stages from query understanding to answer generation, each with specific timing and purpose.
2. **Query Understanding**: How we analyze questions to determine the best retrieval strategy, including classification into entity/definition/factual/descriptive/comparative/complex types.
3. **Retrieval Strategies**: When to use vector search (semantic similarity), KG lookup (structured facts), or both, with clear guidance on the strengths and limitations of each.
4. **Hybrid RAG**: How combining vector and KG retrieval produces better answers than either alone, with examples showing the complementary strengths.

5. **Context Assembly**: The process of merging, deduplicating, and formatting retrieved information for optimal LLM comprehension.
6. **Prompt Engineering**: How careful instruction design prevents hallucination, encourages citations, and produces well-structured answers.
7. **LLM Generation**: How Ollama generates responses token-by-token and why local models benefit privacy, cost, and latency.
8. **Streaming**: The technical approach to delivering responses in real-time for better user experience, including error handling in streams.
9. **Source Attribution**: Why tracking and displaying sources builds trust and enables verification.
10. **Error Handling**: Graceful degradation strategies and user-friendly error messages.
11. **Performance Optimization**: Caching, parallel processing, and index tuning for fast responses.
12. **The Complete Flow**: Step-by-step walkthrough with timing, showing all components working together.

Chapter 6: Database Schema Deep Dive

Understanding Every Table and Why It Exists

Table of Contents for This Chapter

---

1. Database Design Philosophy

1.1 Why PostgreSQL?

We chose PostgreSQL as our database for several compelling reasons that are worth understanding deeply.

Reliability and ACID Compliance: PostgreSQL is known for its rock-solid reliability. When we insert a document or extract an entity, we need absolute certainty that the data is saved correctly. PostgreSQL's ACID (Atomicity, Consistency, Isolation, Durability) properties guarantee that even if the server crashes mid-operation, our data remains consistent. In a Q&A system about financial documents, data integrity is non-negotiable.

The pgvector Extension: This was perhaps the deciding factor. PostgreSQL with the pgvector extension allows us to store and query high-dimensional vectors (our embeddings) efficiently. Alternative solutions would require running a separate vector database (like Pinecone, Weaviate, or Qdrant), adding complexity and synchronization challenges. With pgvector, everything lives in one database—documents, chunks, embeddings, and Knowledge Graph data—making queries across these entities simple and transactionally safe.

Rich Data Types: PostgreSQL supports JSONB (binary JSON) which we use extensively for flexible attributes. Entity attributes, claim qualifiers, and chunk metadata can vary widely, and JSONB lets us store this semi-structured data without rigid schema requirements while still allowing efficient querying.

SQL Power: Complex queries involving joins across chunks, entities, and evidence are straightforward with SQL. Graph databases like Neo4j are designed specifically for graph queries, but PostgreSQL handles our relatively simple graph structure well, and we gain the benefits of a mature, widely-understood database system.

1.2 Schema Design Principles

Our schema follows several key principles:

Document-Centric Organization: Every piece of data relates back to a specific document. This allows complete isolation between different IPO documents and makes it easy to delete all data related to a document (just delete by document_id and cascading deletes handle the rest).

Evidence-First Approach: Every fact in our Knowledge Graph traces back to specific evidence from the source document. This isn't just for user verification—it helps us debug extraction issues and maintain data quality. If an entity has wrong attributes, we can examine the evidence that led to that extraction.

Separation of Concerns: We separate the raw document storage (documents table), the chunked text (chunks table), the semantic representations (embeddings table), and the structured knowledge (KG tables). This separation means each component can evolve independently. We could switch embedding models without touching the KG tables, or enhance our KG extraction without re-chunking documents.

Explicit Over Implicit: Rather than inferring relationships at query time, we extract and store them explicitly. "Yashish Dahiya is CEO of PB Fintech" is stored as an explicit claim, not something we figure out by analyzing text each time. This trades extraction-time effort for query-time speed.

1.3 The Two-Layer Architecture

Our database effectively has two layers:

Layer 1: Vector RAG Layer This consists of documents, chunks, and embeddings. It supports semantic similarity search—given a question, find the most relevant text passages. This layer is sufficient for basic RAG functionality.

Layer 2: Knowledge Graph Layer This consists of evidence, entities, claims, defined_terms, events, and related tables. It supports structured queries—who is the CEO, what are the subsidiaries, what does ESOP mean. This layer adds precision to our answers.

The two layers are connected through the chunks table. Evidence records reference chunks, linking structured facts back to their source text.

---

2. The Documents Table

2.1 Purpose and Role

The documents table is the anchor of our entire system. Every other table references back to a document_id. This table stores metadata about each uploaded PDF and serves as the central reference point.

Think of this table as the "table of contents" for our database. When a user uploads a PDF, the first thing we do is create a record in documents. Only after this record exists can we create chunks, embeddings, and KG data.

2.2 Column Explanations

document_id (VARCHAR, Primary Key): This is a unique identifier for each document, such as "policybazar_ipo" or "zomato_drhp_2021". We use descriptive identifiers rather than auto-incrementing integers because: - They're human-readable, making debugging and queries easier - They can be determined before the document is processed (useful for idempotent uploads) - They serve as a natural foreign key in other tables

filename (VARCHAR): The original filename of the uploaded PDF, like "PB_Fintech_DRHP.pdf". This helps users identify which file they're querying and is displayed in the user interface.

upload_date (TIMESTAMP): When the document was uploaded. Stored with timezone information to handle users in different time zones correctly. Defaults to the current timestamp when the record is created.

page_count (INTEGER): Total number of pages in the PDF. This helps with progress indicators during processing and validates that chunk page numbers don't exceed the document length.

status (VARCHAR): The processing status of the document. Possible values include: - “uploaded” - PDF received, not yet processed - “chunking” - Currently being split into chunks - “embedding” - Chunks are being embedded - “kg_extraction” - Knowledge Graph extraction in progress - “ready” - Fully processed and queryable - “error” - Processing failed (error details in metadata)

This status field is crucial for the user interface. We show different states based on this value—a loading indicator during processing, the Q&A interface when ready, or error messages if processing failed.

metadata (JSONB): A flexible field for additional information that doesn’t warrant its own column. Examples include: - Error messages and stack traces if processing failed - Processing duration statistics - Source URL if the document was downloaded - Custom tags or categories the user assigned

Using JSONB rather than TEXT for this field allows us to query and index specific properties if needed.

2.3 Why document_id Is Not Auto-Generated

You might wonder why document_id is a user-provided string rather than an auto-incrementing integer. This design decision was deliberate:

Idempotency: If processing fails halfway, we can retry with the same document_id. With auto-increment, we’d need a separate mechanism to track which upload attempt corresponds to which ID.

Meaningful References: When debugging, “policybazar_ipo” in a log message is immediately understandable, whereas “document_id=47” requires a lookup.

External Integration: If we integrate with other systems, they can reference documents by meaningful IDs rather than opaque integers.

The tradeoff is that users must provide unique, valid identifiers. We validate document_id to ensure it contains only alphanumeric characters and underscores, avoiding issues with file systems or URLs.

---

3. The Chunks Table

3.1 Purpose and Role

The chunks table stores the actual text content from documents, broken into manageable pieces. Each chunk represents a semantically coherent segment of the document—typically 300-500 words.

This table is the bridge between the raw document and everything else in the system. Embeddings are generated from chunks. Evidence quotes are extracted from chunks. When we retrieve information for a query, we’re ultimately retrieving chunks or referencing them.

3.2 Column Explanations

id (SERIAL, Primary Key): An auto-incrementing integer. Unlike documents, chunks don't need meaningful identifiers—there are too many of them, and they're not referenced by external systems.

document_id (VARCHAR, Foreign Key): Links this chunk to its parent document. This is the most important reference in our schema. When we delete a document, all its chunks are deleted automatically (via ON DELETE CASCADE).

chunk_index (INTEGER): The sequential position of this chunk within the document, starting from 0. This preserves the reading order and allows us to: - Reconstruct the document's flow - Include surrounding chunks for context (chunk N-1 and N+1) - Show users where in the document the chunk appears

text (TEXT): The actual content of the chunk. This is what gets embedded and what appears in search results. The TEXT type allows unlimited length, though our chunking strategy keeps chunks under 3000 characters.

page_start and **page_end** (INTEGER): Which pages this chunk spans. A chunk might start on page 45 and end on page 45 (single page) or span pages 45-46 (if our chunking crossed a page boundary). This information is displayed to users as "Source: Pages 45-46" and allows them to locate the original text in the PDF.

section_title (VARCHAR): The heading or section title under which this chunk appears, such as "About Our Promoters" or "Risk Factors". This provides context beyond page numbers and helps the LLM understand the type of content it's working with. Chunks from "Risk Factors" should be interpreted differently than chunks from "Business Overview".

char_start and **char_end** (INTEGER): The character positions in the original extracted text where this chunk starts and ends. These are primarily used for debugging and could be used for exact text highlighting if we implement PDF viewing with annotations.

metadata (JSONB): Additional chunk-specific information, such as: - Font analysis results (was this chunk mostly headers?) - Detected tables or lists - Special formatting notes - Processing warnings

3.3 The Importance of Overlap Information

While not stored in separate columns, our chunking strategy creates overlapping chunks. A sentence appearing at the end of chunk N might also appear at the beginning of chunk N+1.

We don't explicitly store overlap information because it can be computed from char_start and char_end if needed. However, understanding that overlap exists is important for:

Deduplication: When assembling context, we might skip chunks that heavily overlap with already-included chunks.

Score Normalization: Two chunks might have high similarity scores for a query because they share overlapping content, not because they're independently relevant.

4. The Embeddings Table

4.1 Purpose and Role

The embeddings table stores the vector representation of each chunk. These 384-dimensional vectors capture the semantic meaning of the text and enable similarity search.

Conceptually, each embedding is a point in a 384-dimensional space. Chunks with similar meanings are located near each other in this space. When a user asks a question, we embed the question and find the chunks whose embeddings are closest to it.

4.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier for the embedding record.

chunk_id (INTEGER, Foreign Key): References the chunk this embedding represents. This is a one-to-one relationship—each chunk has exactly one embedding (for a given model).

embedding (VECTOR(384)): The actual embedding vector. The VECTOR type is provided by the pgvector extension. The 384 in parentheses specifies the dimensionality, which matches our embedding model (all-MiniLM-L6-v2).

This column is the most storage-intensive in our database. Each float in the vector takes 4 bytes, so each embedding is approximately 1.5 KB. For a document with 500 chunks, that's about 750 KB of embedding storage.

model_name (VARCHAR): The name of the model used to generate this embedding, such as "all-MiniLM-L6-v2". This is crucial for two reasons:

First, if we switch to a different embedding model, we need to regenerate all embeddings. This field tells us which chunks were embedded with which model.

Second, queries must use the same model for embeddings to be comparable. If chunks were embedded with model A, querying with model B would produce meaningless similarity scores.

created_at (TIMESTAMP): When the embedding was created. Useful for debugging and auditing.

4.3 Why Separate from Chunks?

You might ask: why not just add an embedding column to the chunks table? We separate them for several reasons:

Different Update Patterns: Chunks are created once during ingestion and rarely change. Embeddings might be regenerated if we switch models. Separating them keeps the chunks table stable while allowing embedding experimentation.

Multiple Embeddings Per Chunk: In the future, we might want to store embeddings from multiple models for comparison or ensemble approaches. A separate table makes this straightforward.

Query Efficiency: Many queries need only chunk text, not embeddings (which are large). Separating them keeps the chunks table smaller and faster to scan.

Index Isolation: The HNSW vector index only needs the embeddings table. Keeping embeddings separate means the index covers a smaller, focused table.

4.4 Understanding the Vector Index

The embeddings table has a special index for fast similarity search:

HNSW (Hierarchical Navigable Small Worlds): This index structure allows finding the nearest vectors in logarithmic time rather than scanning all vectors linearly. For 500 chunks, the difference is negligible. For 50,000 chunks across many documents, it's the difference between 1ms and 500ms query time.

The index is created with specific parameters: - **m = 16**: Each node connects to 16 others in the graph structure - **ef_construction = 64**: How thoroughly connections are established during index building

These parameters balance index size, build time, and query accuracy. Higher values improve accuracy but increase storage and query time.

---

5. Knowledge Graph Tables Overview

5.1 The KG Layer

Beyond vector search, we maintain a Knowledge Graph (KG) extracted from documents. This KG consists of several interconnected tables that capture structured information:

evidence: The foundation—quotes and citations from source text **kg_entities**: Nodes in our graph—people, companies, metrics **entity_aliases**: Alternative names for entities **claims**: Edges in our graph—relationships between entities **defined_terms**: Glossary entries from the document **events**: Temporal facts—incorporation dates, funding rounds **event_participants**: Which entities participated in which events

These tables work together to provide precise, verifiable answers that pure vector search might miss.

5.2 Why Build a Knowledge Graph?

Vector search finds semantically similar text, but it has limitations:

Precision: "Who is the CEO?" might return chunks discussing leadership generally, without naming the specific person.

Structure: "List all subsidiaries" requires aggregating information scattered across multiple chunks.

Definitions: "What does ESOP-2014 mean?" needs the exact definition, not similar text.

The Knowledge Graph addresses these limitations by extracting and storing structured facts. Instead of searching for text similar to "CEO", we can directly look up entities with role="CEO".

5.3 The Extraction-Query Tradeoff

Building a KG requires significant effort during document processing—running LLM extraction, resolving entities, validating relationships. This front-loaded work pays off during queries:

Extraction Time (per document): 5-30 minutes depending on size **Query Time (per question)**: 20-50 milliseconds

By investing time during ingestion, we get instant, precise answers at query time. This is the right tradeoff for documents that will be queried many times.

---

6. The Evidence Table

6.1 Purpose and Role

The evidence table is the foundation of our Knowledge Graph's credibility. Every extracted fact—every entity, relationship, and definition—links back to an evidence record that contains the exact quote from the source document.

Think of evidence records as footnotes. When we say "Yashish Dahiya is CEO with 4.27% shareholding", we can point to the exact sentence in the document that states this. Without evidence, our KG would be a collection of unverifiable claims.

6.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier, referenced by other KG tables.

document_id (VARCHAR, Foreign Key): Links to the source document. This ensures we can always trace evidence back to its origin.

chunk_id (INTEGER, Foreign Key to chunks, Optional): The specific chunk containing this evidence. This can be NULL if the evidence was extracted directly from PDF text without going through our chunking (rare but possible in specialized extraction).

quote (TEXT): The exact text from the document that supports the fact. This might be a single sentence or multiple sentences. The quote is stored verbatim, preserving original formatting, punctuation, and any errors in the source.

page_number (INTEGER): Which page the quote appears on. This allows users to locate the exact source in the original PDF.

section_title (VARCHAR): The section heading under which this quote appears. For example, "About Our Promoters" or "Capital Structure". This provides additional context about the type of information.

confidence (FLOAT): A score from 0 to 1 indicating how confident we are in this evidence. A score of 1.0 means the quote is an exact match from the document. Lower scores might indicate the text was reconstructed from OCR or the extraction was uncertain.

created_at (TIMESTAMP): When this evidence was recorded.

6.3 Why Evidence Is Central

The evidence table design reflects a core principle: traceability. In financial and legal documents, being able to cite exactly where information came from is not optional—it's essential.

When users ask questions, we can show not just the answer but also the evidence. "The CEO is Yashish Dahiya [Source: Page 45, 'Mr. Yashish Dahiya serves as Chairman and CEO of our Company']". This builds trust and allows verification.

When our extraction makes mistakes (and it will), evidence helps us diagnose the problem. If an entity has wrong attributes, we can examine the evidence quote and understand whether the error was in extraction (LLM misunderstood the text) or the source (the document itself was ambiguous).

---

7. The Entities Table

7.1 Purpose and Role

The kg_entities table stores the "nodes" in our Knowledge Graph—the things we've identified in the document. Each entity represents a real-world object or concept: a person, a company, a financial metric, a regulatory body.

Entities are the nouns of our Knowledge Graph. They have names, types, and attributes. They participate in relationships (stored in the claims table) and events (stored in the events table).

7.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier, referenced by claims and other tables.

document_id (VARCHAR, Foreign Key): Links to the source document. Each document has its own set of entities, allowing the same person or company to appear differently across documents if described differently.

canonical_name (VARCHAR): The primary name for this entity. Even if "Yashish Dahiya" is also referred to as "Mr. Dahiya" or "the CEO" in the document, we choose one canonical form and store alternatives in the aliases table.

Choosing the canonical name follows rules: - For people: Full name with middle names/initials if consistently used - For companies: Legal registered name - For metrics: Standardized format like "Revenue FY2021"

entity_type (VARCHAR): The category of entity. Our schema defines several types: - PERSON: Directors, officers, promoters, key management personnel - COMPANY: Companies, subsidiaries, joint ventures, competitors - FINANCIAL_METRIC: Revenue, profit, margins, growth figures -

SECURITY: Equity shares, preference shares, bonds - REGULATORY_BODY: SEBI, RBI, IRDAI, ROC - PRODUCT: Products and services offered - LOCATION: Cities, addresses, jurisdictions

The type determines how the entity can participate in relationships and how it's displayed in the user interface.

attributes (JSONB): A flexible collection of properties specific to this entity. The JSONB format allows different entity types to have different attributes:

For a PERSON entity:

```
{
  "role": "Chairman and CEO",
  "shareholding": "4.27%",
  "shares": "17,545,000",
  "din": "00706336",
  "nationality": "Indian",
  "age": "47"
}
```

For a COMPANY entity:

```
{
  "cin": "U74999DL2008PLC178155",
  "incorporated": "2008-11-24",
  "registered_office": "Gurugram, Haryana",
  "authorized_capital": "100,00,00,000"
}
```

Using JSONB rather than fixed columns gives us flexibility to capture different attributes for different document types without schema changes.

evidence_ids (INTEGER ARRAY): An array of evidence IDs that support this entity's existence and attributes. Multiple evidence records might confirm the same entity, each from different parts of the document.

created_at (TIMESTAMP): When this entity was first extracted.

7.3 The entity_aliases Table

Entities often have multiple names in documents. "PB Fintech Limited" might also be called "our Company", "the Company", "PB Fintech", or "PBF". The entity_aliases table captures these alternative names.

id (SERIAL, Primary Key) **entity_id** (INTEGER, Foreign Key): Links to the main entity record **alias** (VARCHAR): The alternative name **alias_type** (VARCHAR): The type of alias—"abbreviation", "informal", "legal_variant", etc.

This table enables better query matching. When a user asks about "PB Fintech", we can find the entity even if its canonical name is "PB Fintech Limited".

7.4 The Unique Constraint

The kg_entities table has a unique constraint on (document_id, canonical_name, entity_type). This means within one document, we can't have two different entities with the same name and type. This constraint enforces entity uniqueness and prevents duplicate extraction.

If two different people happen to have the same name (rare but possible), they would need to be distinguished in the canonical_name—perhaps by adding an initial or role.

---

8. The Claims Table

8.1 Purpose and Role

The claims table stores the "edges" in our Knowledge Graph—the relationships between entities. If entities are nouns, claims are verbs. "Yashish Dahiya [subject] is CEO of [predicate] PB Fintech [object]".

Claims connect entities and give our Knowledge Graph its structure. Without claims, we'd have a collection of disconnected entities—just a list of names and companies. With claims, we have a graph that captures how these entities relate to each other.

8.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier.

document_id (VARCHAR, Foreign Key): Links to the source document.

subject_entity_id (INTEGER, Foreign Key to kg_entities): The entity that the claim is about—the "subject" of the sentence. In "Yashish Dahiya is CEO of PB Fintech", Yashish Dahiya is the subject.

predicate (VARCHAR): The type of relationship. This is the "verb" connecting subject and object. Examples include: - CEO_OF, CFO_OF, DIRECTOR_OF: Person → Company relationships - SUBSIDIARY_OF, PARENT_OF: Company → Company relationships - OWNS_SHARES, HOLDS_STAKE: Ownership relationships - REGULATES: Regulatory body → Company - LOCATED_IN: Entity → Location - HAS_VALUE: Metric → Amount

We use uppercase predicates with underscores as a convention, making them easy to identify and consistent across documents.

object_entity_id (INTEGER, Foreign Key to kg_entities, Optional): The entity on the receiving end of the relationship. In "CEO of PB Fintech", PB Fintech is the object.

object_value (TEXT, Optional): Sometimes the object is not an entity but a literal value. For example, "Revenue is 957 crore"—the object is a number, not an entity. Either object_entity_id or object_value should be populated, not both.

qualifiers (JSONB): Additional information about the relationship that doesn't fit the subject-predicate-object structure. For example:

```
{
  "as_of": "2021-03-31",
  "percentage": "4.27%",
  "role_type": "Executive Director"
}
```

Qualifiers let us capture nuanced information: someone might be director "since 2015" or own shares "as of the filing date".

evidence_id (INTEGER, Foreign Key to evidence): Links to the evidence supporting this claim. This is typically a single evidence record containing the quote that states this relationship.

confidence (FLOAT): A score from 0 to 1 indicating confidence in this claim. Claims extracted with clear, unambiguous language get high scores. Claims inferred from context get lower scores.

8.3 Understanding Relationship Direction

The direction of relationships matters. "Policybazaar SUBSIDIARY_OF PB Fintech" is different from "PB Fintech SUBSIDIARY_OF Policybazaar" (the second would be incorrect).

We establish conventions for direction: - Person → Company for role relationships (X is CEO of Y) - Smaller → Larger for ownership (subsidiary of parent) - Actor → Target for regulatory (SEBI regulates Company)

These conventions ensure consistent querying. To find all subsidiaries, we query claims where predicate="SUBSIDIARY_OF" and object_entity is the parent company.

---

9. The Defined Terms Table

9.1 Purpose and Role

The defined_terms table stores the glossary of the document—words and phrases that have specific meanings in this context. IPO documents are notorious for their defined terms section, where common words are given precise legal meanings.

"Company" in everyday language means any company. "Company" as a defined term in an IPO document means specifically "PB Fintech Limited". Understanding these definitions is crucial for correctly interpreting the document.

9.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier.

document_id (VARCHAR, Foreign Key): Links to the source document. Each document has its own set of defined terms.

term (VARCHAR): The defined word or phrase, exactly as it appears in the definition section. Examples: "Company", "ESOP-2017", "Promoters", "Red Herring Prospectus".

definition (TEXT): The full definition text from the document. This captures the complete legal definition, which might be quite lengthy. For example:

"'ESOP-2017' means the Employee Stock Option Plan 2017 approved by the Board of Directors at their meeting held on March 15, 2017, as amended from time to time, providing for the grant of stock options to eligible employees."

evidence_id (INTEGER, Foreign Key to evidence): Links to the evidence record containing the quote where this term was defined.

9.3 The Unique Constraint

The table has a unique constraint on (document_id, term). Each term can only be defined once per document. If a document somehow defines the same term twice (shouldn't happen, but might due to extraction issues), we take the first definition or merge them.

9.4 How Definitions Are Used

During queries, when users ask "What does ESOP mean?" or "Define DRHP", we first check the defined_terms table. If we find a match, we use the official definition from the document rather than relying on vector search.

This is more reliable because: - We get the exact legal definition, not a paraphrase - We avoid retrieval errors (vector search might find chunks that mention ESOP without defining it) - The answer includes the evidence quote, allowing verification

---

10. The Events Table

10.1 Purpose and Role

The events table stores temporal facts—things that happened at specific times. Company incorporation, funding rounds, acquisitions, regulatory approvals—these are events with dates that are often asked about.

Events complement entities and claims. While claims describe static relationships ("X is CEO of Y"), events describe changes over time ("X was incorporated on date Y").

10.2 Column Explanations

id (SERIAL, Primary Key): Auto-incrementing identifier.

document_id (VARCHAR, Foreign Key): Links to the source document.

event_type (VARCHAR): A category for the event. We define several types: - INCORPORATION: Company formation - ACQUISITION: M&A activity - FUNDING: Investment and funding rounds - REGULATORY: Licenses, approvals, certifications - PRODUCT_LAUNCH: New products or services - IPO: IPO-related events (filing, pricing, listing) - NAME_CHANGE: Company renamed - BOARD_CHANGE: Directors appointed or resigned - OTHER: Events that don't fit the above categories

event_date (DATE): The date of the event, if known precisely. Stored as a proper date type for comparison queries.

event_date_text (VARCHAR): The date as written in the document. "March 15, 2021" or "H1 FY2022" or "early 2019". Sometimes documents give imprecise dates that can't be parsed into exact dates.

We store both because: - event_date allows sorting and filtering by date - event_date_text preserves the original phrasing

description (TEXT): A brief description of what happened. "PB Fintech Limited was incorporated as a private limited company under the Companies Act, 1956."

evidence_id (INTEGER, Foreign Key to evidence): Links to the supporting quote.

10.3 The event_participants Table

Events often involve multiple entities. An acquisition involves both the acquirer and the acquired. A funding round involves the company and the investors.

id (SERIAL, Primary Key) **event_id** (INTEGER, Foreign Key): Links to the event **entity_id** (INTEGER, Foreign Key to kg_entities): Which entity participated **role** (VARCHAR): Their role in the event—"acquirer", "target", "investor", "company"

This design allows any number of entities to participate in any event with clearly defined roles.

---

11. Table Relationships and Foreign Keys

11.1 The Relationship Diagram

Understanding how tables reference each other is crucial for writing queries and maintaining data integrity:

documents (the root) ↓ document_id **chunks** → links back to documents ↓ chunk_id **embeddings** → links to chunks ↓ id (no FK, but queried together)

evidence → links to documents and optionally to chunks ↓ id **kg_entities** → links to documents, references evidence[] ↓ id **entity_aliases** → links to entities **claims** → links to two entities (subject and object) and evidence **defined_terms** → links to documents and evidence **events** → links to documents and evidence ↓ id **event_participants** → links events and entities

11.2 Cascading Deletes

When a document is deleted, we want all associated data to disappear. PostgreSQL's ON DELETE CASCADE makes this automatic:

- Delete document → chunks deleted → embeddings deleted
- Delete document → evidence deleted
- Delete document → entities deleted → aliases deleted, claims deleted (through both subject and object FKs), event_participants deleted
- Delete document → defined_terms deleted
- Delete document → events deleted

This cascade ensures no orphaned records exist. Deleting a document is a complete, clean operation.

11.3 Referential Integrity

Foreign keys ensure our data remains consistent: - You can't create a chunk without a valid document_id - You can't create a claim referencing a non-existent entity - You can't create an embedding for a non-existent chunk

Attempts to violate these constraints result in database errors, caught at the application level and handled appropriately.

---

12. Indexes and Performance

12.1 Why Indexes Matter

Without indexes, every query would scan entire tables. For a table with 10,000 rows, scanning is fast enough. For 1,000,000 rows, it becomes unacceptably slow.

Indexes create sorted, searchable structures that allow the database to find rows without scanning everything. Choosing the right indexes is crucial for query performance.

12.2 Key Indexes in Our Schema

Primary Key Indexes: Every table has an index on its primary key (id column). This is created automatically by PostgreSQL.

Foreign Key Indexes: We create indexes on all foreign key columns: - chunks(document_id) - embeddings(chunk_id) - evidence(document_id) - kg_entities(document_id) - claims(subject_entity_id, object_entity_id)

These indexes speed up join operations and cascading deletes.

The Vector Index: The most important index for query performance:

```
CREATE INDEX ON embeddings USING hnsw (embedding vector_cosine_ops)
WITH (m = 16, ef_construction = 64);
```

This HNSW index allows sub-100ms similarity searches across all embeddings. Without it, every query would calculate similarity against every embedding—impossibly slow for large collections.

Text Search Indexes: For defined_terms and entity names, we might create indexes for case-insensitive matching:

```
CREATE INDEX ON defined_terms (LOWER(term));
CREATE INDEX ON kg_entities (LOWER(canonical_name));
```

These enable fast lookups when users search for terms by name.

12.3 Query Optimization Considerations

Beyond indexes, several factors affect query performance:

Connection Pooling: We reuse database connections rather than creating new ones for each query. Connection creation takes 50-100ms; reusing existing connections takes <1ms.

Query Planning: PostgreSQL's query planner chooses execution strategies. We periodically run ANALYZE to update statistics that help the planner make good decisions.

Batch Operations: During ingestion, we insert many records. Batching (inserting 100 rows at once instead of 1 at a time) dramatically improves throughput.

12.4 Monitoring and Maintenance

Our schema requires periodic maintenance:

VACUUM: PostgreSQL's MVCC model keeps old row versions around. VACUUM reclaims this space. We run VACUUM periodically, especially after large deletions.

REINDEX: Vector indexes can become fragmented as embeddings are added and removed. Periodic reindexing maintains optimal search performance.

Backup: We back up the database regularly. The schema is just structure; the data (especially embeddings and KG) represent hours of processing that we don't want to repeat.

Summary

This chapter provided an in-depth understanding of our database schema:

1. **Design Philosophy**: Why PostgreSQL with pgvector, the document-centric approach, evidence-first design, and the two-layer (Vector RAG + Knowledge Graph) architecture.
2. **Documents Table**: The anchor of our system, storing metadata and status for each uploaded PDF, with meaningful string IDs rather than auto-increment.
3. **Chunks Table**: How text is broken into queryable pieces with page tracking, section titles, and overlap for context preservation.
4. **Embeddings Table**: Storing 384-dimensional vectors with model versioning, separated from chunks for flexibility and performance.
5. **Knowledge Graph Overview**: The purpose of the KG layer and the extraction-query tradeoff that makes structured queries fast.
6. **Evidence Table**: The credibility foundation—every fact traces back to a quote, page number, and section.
7. **Entities Table**: Nodes in our graph with canonical names, types, flexible JSONB attributes, and the aliases table for alternative names.
8. **Claims Table**: Edges in our graph connecting entities with predicates, qualifiers, and direction conventions.
9. **Defined Terms Table**: The document's glossary, enabling precise answers to definition questions.

10. **Events Table**: Temporal facts with dates and participants, capturing what happened when.
11. **Relationships and Foreign Keys**: How tables connect, cascading deletes, and referential integrity.
12. **Indexes and Performance**: The vector index, foreign key indexes, and maintenance considerations for fast queries.

Chapter 7: PostgreSQL Implementation Details

From Schema to Working Database: The Practical Guide

Table of Contents for This Chapter

---

1. Setting Up PostgreSQL

1.1 Why Local PostgreSQL Matters

Before diving into code, understanding the PostgreSQL setup is essential. Unlike cloud databases where configuration is hidden, our local PostgreSQL installation requires conscious decisions about configuration, security, and resource allocation.

We run PostgreSQL locally (via Homebrew on macOS or apt on Linux) because:

Data Sovereignty: IPO documents contain sensitive financial information. Running locally means data never leaves the machine, simplifying compliance and security requirements.

Performance Predictability: Cloud databases introduce network latency (typically 5-50ms per query). Local databases respond in microseconds for simple queries. When we're running hundreds of embedding insertions, this difference becomes significant.

Cost Control: Cloud databases charge for storage, compute, and egress. Local PostgreSQL has no per-query costs, making experimentation and development affordable.

Development Flexibility: We can tune PostgreSQL settings, install extensions, and modify configurations without cloud provider limitations.

1.2 The Installation Process

Installing PostgreSQL involves several steps that are worth understanding:

The PostgreSQL Server: This is the core database engine that manages storage, executes queries, and handles connections. It runs as a background service (daemon on Unix systems).

The Data Directory: PostgreSQL stores all database files in a specific directory (usually /var/lib/postgresql/data or /usr/local/var/postgres). This contains the actual database files, write-ahead logs, and configuration.

Configuration Files: Two key files control PostgreSQL behavior: - **postgresql.conf**: Controls server behavior—memory allocation, connection limits, logging - **pg_hba.conf**: Controls authentication—who can connect and how they prove identity

Client Tools: Commands like `psql` (interactive shell), `pg_dump` (backup), and `createdb` (create databases) are separate programs that communicate with the server.

1.3 Initial Configuration Decisions

Several configuration decisions impact our system:

Memory Allocation: PostgreSQL uses shared memory for caching data pages. We configure `shared_buffers` to about 25% of available RAM. Too low means more disk reads; too high leaves insufficient memory for other processes like the embedding model.

Connection Limits: The `max_connections` setting limits concurrent database connections. Each connection consumes memory (roughly 5-10 MB). We set this to 100, enough for our application with room for monitoring tools.

Logging: We enable query logging during development to see what SQL is actually executing. This helps debug performance issues and understand ORM behavior. In production, we reduce logging to avoid disk I/O overhead.

1.4 Creating the Database and Users

Before our application can store data, we create:

A Dedicated Database: Rather than using the default "postgres" database, we create "ipo_qa" specifically for our application. This isolation makes backups, permissions, and cleanup cleaner.

An Application User: We create a dedicated PostgreSQL user (role) with limited permissions. The application connects as this user, not as the superuser. If the application is compromised, damage is contained to this database.

Permissions: We grant the application user ownership of the database and all tables, but not superuser privileges. It can create, read, update, and delete data in its database but cannot affect other databases or PostgreSQL settings.

---

2. The pgvector Extension

2.1 What pgvector Provides

The pgvector extension transforms PostgreSQL from a traditional relational database into a vector database capable of semantic similarity search. Understanding what it adds helps appreciate both its power and limitations.

The VECTOR Data Type: pgvector introduces a new column type: VECTOR(n), where n is the dimensionality. Our embeddings use VECTOR(384), meaning each value is an array of 384 floating-point numbers.

Distance Operators: pgvector adds operators for calculating distance between vectors: - `<->` : Euclidean (L2) distance - `<=>` : Cosine distance (1 - cosine similarity) - `<#>` : Negative inner product

We primarily use cosine distance (`<=>`) because our embeddings are normalized, making cosine distance appropriate for semantic similarity.

Indexing Algorithms: pgvector provides two index types for fast approximate nearest neighbor search: - **IVFFlat**: Inverted file index with flat storage - **HNSW**: Hierarchical Navigable Small Worlds

We use HNSW because it provides better query performance, especially for larger datasets, at the cost of more memory usage.

2.2 How HNSW Indexing Works Conceptually

Understanding HNSW helps us tune it effectively:

The Layered Graph Structure: HNSW builds a multi-layered graph where each node is a vector (embedding). Higher layers have fewer nodes, acting as "express lanes" for navigation. Lower layers are more dense, providing precision.

Search Process: When searching for similar vectors, the algorithm: 1. Starts at a node in the top (sparsest) layer 2. Greedily moves to neighbors closer to the query 3. Drops to the next layer and continues 4. Repeats until reaching the bottom layer 5. Returns the closest nodes found

The Build Process: Adding a vector to the index involves: 1. Randomly selecting a starting layer (higher layers get fewer nodes) 2. Finding the closest nodes at each layer 3. Creating connections to these neighbors 4. Balancing connection counts to maintain graph properties

This construction is the slow part, which is why embedding insertion takes longer than simple row insertion.

2.3 Index Parameters Explained

When creating the HNSW index, we specify:

m (max connections): How many connections each node has at each layer. Higher values mean: - More accurate search results - Larger index size - Slower index construction - Faster queries (more paths to explore)

We use m=16, a balanced choice for moderate-sized datasets.

ef_construction: How many candidates to consider when building the index. Higher values mean: - Better index quality - Slower construction - No impact on query speed

We use ef_construction=64, prioritizing index quality since we build the index once but query many times.

ef_search (set at query time): How exhaustively to search during queries. Higher values mean: - More accurate results - Slower queries

We set this dynamically via `SET hnsw.ef_search = 100` before similarity queries.

2.4 Installing and Enabling pgvector

The extension must be installed in PostgreSQL and then enabled for each database:

Installation adds the extension files to PostgreSQL's extension directory. This is done at the system level (via package manager or compilation).

Enabling creates the extension objects (types, operators, functions) within a specific database. The command `CREATE EXTENSION vector;` must be run in each database that needs vector operations.

We include this in our schema creation script, ensuring the extension exists before creating tables with VECTOR columns.

---

3. Connection Management

3.1 Why Connection Management Matters

Database connections are expensive resources. Each connection: - Consumes memory on both client and server - Requires authentication handshaking (tens of milliseconds) - Holds a slot in PostgreSQL's limited connection pool - Must be properly closed to avoid leaks

Poor connection management leads to resource exhaustion, where the application cannot get connections or PostgreSQL refuses new connections. Good management ensures efficient resource use and predictable performance.

3.2 Connection Pooling Explained

Rather than creating a new connection for each database operation, we maintain a pool of reusable connections:

The Pool Concept: A pool maintains several open, authenticated connections ready for use. When code needs a database connection, it borrows one from the pool. When done, it returns the connection to the pool rather than closing it.

Pool Size Settings: We configure: - **pool_size**: Minimum connections to keep open (we use 5) - **max_overflow**: Additional connections allowed when pool is exhausted (we use 10) - **pool_timeout**: How long to wait for a connection before raising an error (we use 30 seconds)

Pool Recycle: Connections can become stale if PostgreSQL restarts or network issues occur. We set `pool_recycle` to 3600 seconds (1 hour), causing connections older than this to be replaced with fresh ones.

3.3 The Connection URL

We connect to PostgreSQL using a URL format that encodes all connection parameters:

```
postgresql://username:password@host:port/database_name
```

For local development, this might be:

```
postgresql://ipo_user:secure_password@localhost:5432/ipo_qa
```

The URL contains: - **Protocol**: "postgresql://" specifies the database type - **Credentials**: Username and password for authentication - **Host**: Where PostgreSQL is running (localhost for local) - **Port**: PostgreSQL's listening port (default 5432) - **Database**: Which database to connect to

We store this URL in environment variables, never in code, for security.

3.4 Connection Lifecycle

A connection goes through several states:

Creation: Establishing a TCP connection to PostgreSQL, performing authentication, setting session parameters. This takes 20-100ms.

Active Use: Executing queries, the connection is "checked out" from the pool and assigned to a specific request or operation.

Idle: Between queries, the connection sits idle in the pool, maintaining its authenticated session but not executing anything.

Termination: When the pool shrinks or the process ends, connections are properly closed with a termination message to PostgreSQL.

Properly managing this lifecycle ensures no connection leaks (connections never returned to pool) and no stale connections (connections that error when used).

---

4. SQLAlchemy Integration

4.1 Why Use SQLAlchemy?

SQLAlchemy sits between our Python code and PostgreSQL, providing several benefits:

Database Abstraction: SQLAlchemy can work with PostgreSQL, MySQL, SQLite, and other databases. While we use PostgreSQL, SQLAlchemy's abstraction means our code isn't tightly coupled to PostgreSQL-specific syntax.

SQL Injection Prevention: SQLAlchemy uses parameterized queries by default, preventing SQL injection attacks. Values are passed separately from SQL structure.

Connection Pool Management: SQLAlchemy includes a sophisticated connection pool, handling all the lifecycle management we discussed.

ORM Capabilities: Though we mostly use raw SQL, SQLAlchemy can also map Python objects to database tables, reducing boilerplate for CRUD operations.

4.2 Core vs. ORM

SQLAlchemy has two layers:

Core: A SQL abstraction that represents tables, columns, and queries as Python objects. You write SQL-like code that generates actual SQL.

ORM (Object-Relational Mapper): Maps Python classes to database tables, managing insertions, updates, and relationships automatically.

We primarily use Core with raw SQL (via `text()`) because: - Vector operations require PostgreSQL-specific syntax - Our queries are complex enough that ORM would obscure what's happening - Direct SQL gives us full control over query optimization

However, we use ORM concepts like sessions for transaction management.

4.3 The Engine and Sessions

Two key SQLAlchemy objects manage database interaction:

Engine: Represents the database itself. The engine manages the connection pool and is typically created once at application startup. It holds our connection URL and pool configuration.

Session: Represents a conversation with the database. A session is a workspace where you execute queries and accumulate changes before committing them. Sessions borrow connections from the engine's pool.

The pattern is: 1. Create engine once at startup 2. For each operation, create/get a session 3. Execute queries within the session 4. Commit or rollback the session 5. Close the session (returns connection to pool)

4.4 The text() Function for Raw SQL

While SQLAlchemy can generate SQL, we often write it directly using `text()`:

The `text()` function takes a SQL string and creates a SQL expression object. This object can be executed, and parameters can be bound to it.

Why raw SQL? Several reasons: - pgvector operators (`<=>`) aren't built into SQLAlchemy - Complex queries are clearer as explicit SQL - We can copy queries directly to psql for testing - Full control over query structure and optimization

However, we always use parameter binding (`:param_name` syntax) rather than string formatting, maintaining SQL injection protection.

4.5 Context Managers for Sessions

We use Python's context manager pattern for sessions:

This pattern ensures sessions are always properly closed, even if exceptions occur. The `get_db()` function returns a context manager that: 1. Creates a session 2. Yields it for use 3. Commits if no exception occurred 4. Rolls back if an exception occurred 5. Closes the session (returns connection to pool)

This prevents connection leaks and ensures consistent transaction handling.

---

5. Transaction Handling

5.1 What Transactions Provide

Transactions are fundamental to database reliability. A transaction groups multiple operations into a single atomic unit:

Atomicity: All operations in a transaction succeed together, or all fail together. If we're inserting 100 chunks and one fails, none are committed—the database returns to its pre-transaction state.

Consistency: Transactions move the database from one valid state to another. Foreign key constraints, unique constraints, and check constraints are enforced.

Isolation: Concurrent transactions don't see each other's uncommitted changes. This prevents one request from seeing partially-complete data from another.

Durability: Once a transaction commits, the changes are permanent, surviving even server crashes.

5.2 Implicit vs. Explicit Transactions

PostgreSQL operates in "autocommit" mode by default—each statement is its own transaction. SQLAlchemy changes this:

SQLAlchemy Sessions Without autocommit: When using SQLAlchemy sessions, statements are accumulated and only committed when you explicitly call `session.commit()`. Until then, changes are visible only within your session.

Why This Matters: Consider ingesting a document: 1. Insert document record 2. Insert 500 chunks 3. Insert 500 embeddings

If embedding #347 fails, we want to roll back everything, not leave the database with partial data. By running all insertions in a single transaction, we get all-or-nothing behavior.

5.3 Commit and Rollback

Two operations end a transaction:

Commit: Tells PostgreSQL to make all changes permanent. Once committed, other connections can see the changes, and they survive server restarts.

Rollback: Tells PostgreSQL to discard all changes since the transaction began. The database returns to its pre-transaction state.

Our session pattern handles this automatically: - If code completes normally, the session commits - If an exception occurs, the session rolls back

5.4 Transaction Isolation Levels

PostgreSQL supports different isolation levels that control what concurrent transactions can see:

Read Committed (default): A transaction sees only committed data from other transactions. If another transaction commits while we're running, we might see different data on different queries.

Repeatable Read: The data we see is locked at transaction start. Even if other transactions commit, we continue seeing the old data until our transaction ends.

Serializable: Transactions behave as if they executed one at a time, not concurrently. The safest but slowest.

We use Read Committed (the default) because our queries are typically short, and we don't have complex concurrent operations that would benefit from stronger isolation.

5.5 Long-Running Transactions: Risks

Transactions should complete quickly. Long-running transactions cause problems:

Lock Contention: If a transaction holds locks for a long time, other transactions wait, slowing the entire system.

Connection Tie-Up: While a transaction is open, its connection is unavailable for other work, potentially exhausting the pool.

WAL Bloat: PostgreSQL cannot clean up old transaction log entries while transactions are pending, potentially filling disk space.

For operations that take time (like KG extraction), we commit frequently—perhaps after each chunk's entities are extracted—rather than holding one transaction for the entire document.

---

6. Repository Pattern

6.1 What Is the Repository Pattern?

The Repository Pattern separates database access logic from business logic. Instead of writing SQL throughout our application, we encapsulate database operations in dedicated classes:

Repository Classes: Each table (or logical group of tables) has a repository class containing methods like `create()`, `find_by_id()`, `find_by_document()`, `delete()`.

Business Logic: Higher-level code calls repository methods, not database operations directly. It says "save this entity" rather than "execute this INSERT statement."

This separation provides: - **Testability**: We can mock repositories for testing without a database - **Maintainability**: SQL is in one place, easy to find and update - **Reusability**: Common queries are written once, used everywhere - **Abstraction**: If we change database structure, we update repositories, not scattered SQL

6.2 Repository Structure in Our Project

We organize repositories in the `src/database/` directory:

ChunkRepository: Handles operations on the chunks table - `create(document_id, chunks_list)` - Bulk insert chunks - `find_by_document(document_id)` - Get all chunks for a document - `find_by_id(chunk_id)` - Get a single chunk - `delete_by_document(document_id)` - Remove all chunks for a document

EmbeddingRepository: Handles the embeddings table - `create(chunk_id, embedding, model_name)` - Insert an embedding - `search_similar(query_embedding, document_id, top_k)` - Vector similarity search

KGEntityRepository: Handles kg_entities table - `create(entity_data)` - Insert an entity (with upsert logic) - `find_by_name(document_id, name)` - Look up entity by name - `find_by_type(document_id, entity_type)` - Get all entities of a type

Each repository takes a session as a parameter, allowing the caller to control transaction boundaries.

6.3 Why Methods Take Sessions

Repository methods receive a database session rather than managing their own:

Transaction Control: The caller decides when to commit. Multiple repository operations can be grouped into a single transaction.

Context Awareness: The session knows the current transaction state. If we're already in a transaction, the repository participates in it rather than starting a new one.

Testing: We can pass a test session connected to a test database, making repositories fully testable.

This approach follows dependency injection principles—repositories don't create their dependencies (sessions); they receive them.

6.4 Error Handling in Repositories

Repository methods handle database-specific errors and translate them:

IntegrityError: Raised when constraints are violated (unique constraint, foreign key). The repository catches this and either: - Returns a meaningful error ("Entity with this name already exists") - Performs upsert logic (update if exists, insert if not)

OperationalError: Connection problems, timeouts. The repository logs these and re-raises, letting the caller decide how to handle reconnection.

NoResultFound: Expected when queries might legitimately return nothing. The repository returns None rather than raising an exception for "not found" cases.

This error translation shields business logic from database implementation details.

---

7. Migrations and Schema Evolution

7.1 Why Schemas Need to Change

Database schemas evolve as requirements change:

New Features: Adding KG extraction requires new tables (kg_entities, claims, etc.) **Performance Optimization**: Adding indexes to frequently-queried columns **Bug Fixes**: Changing a VARCHAR to TEXT if we underestimated length **Normalization**: Splitting a table that became too wide

Unlike code, where we can simply deploy new versions, database changes must carefully transition existing data from old schema to new.

7.2 The Migration Concept

A migration is a versioned, reversible set of schema changes:

Forward Migration (upgrade): Applies changes—creates tables, adds columns, modifies indexes **Backward Migration (downgrade)**: Reverses changes—drops tables, removes columns

Migrations are numbered sequentially (001, 002, 003...), ensuring they're applied in order. Each migration file describes what changed and how to undo it.

7.3 Migration Strategies

We use two approaches depending on the change type:

Simple Changes (Script-Based): For straightforward changes, we write SQL scripts: `- migration_001_create_base_tables.sql - migration_002_add_kg_tables.sql - migration_003_add_evidence_table.sql`

These scripts are run in order when setting up or upgrading the database. We track which migrations have been applied in a `schema_migrations` table.

Complex Changes (Alembic): For changes requiring data transformation or complex logic (like splitting a column into multiple columns), we could use Alembic, SQLAlchemy's migration tool. Alembic generates Python migration files that can include both schema changes and data transformations.

7.4 Our Migration Approach

For this project, we use simple SQL scripts because:

Clarity: SQL scripts show exactly what's happening—no abstraction to decode **PostgreSQL-Specific**: pgvector operations are PostgreSQL-specific anyway **Simplicity**: We're not expecting many schema changes

Our migration files live in `database/` and include: - `schema.sql`: Creates base tables (documents, chunks, embeddings) - `kg_schema.sql`: Creates Knowledge Graph tables - Schema creation is idempotent (uses IF NOT EXISTS)

7.5 Safe Migration Practices

When modifying production databases:

Backup First: Always backup before migration. If something goes wrong, we can restore.

Test on Copy: Run migrations on a copy of production data first. This reveals issues with data that doesn't match expectations.

Plan Downtime: Some migrations require exclusive table locks, blocking other operations. Schedule these during low-traffic periods.

Migrate Data Carefully: When adding NOT NULL columns, we must either provide defaults or migrate in steps: 1. Add column as nullable 2. Populate existing rows 3. Add NOT NULL constraint

---

8. Backup and Recovery

8.1 Why Backups Are Critical

Our database contains: - Documents: Replaceable (we have the original PDFs) - Chunks: Regenerable from documents (but takes processing time) - Embeddings: Regenerable from chunks (but takes significant processing time) - KG Data: Regenerable (but takes hours of LLM processing)

Losing the database means potentially hours or days of reprocessing. Regular backups protect against: - Disk failures - Accidental deletion of data - Corruption from software bugs - User errors

8.2 Backup Methods

PostgreSQL offers several backup approaches:

pg_dump (Logical Backup): Exports database as SQL statements. The backup file contains CREATE TABLE and INSERT statements that recreate the database.

Advantages: - Portable across PostgreSQL versions - Can restore individual tables - Human-readable (for debugging)

Disadvantages: - Slower than physical backup for large databases - Restoration executes all that SQL, taking time

pg_basebackup (Physical Backup): Copies the actual database files. Much faster for large databases.

Advantages: - Fast for large databases - Point-in-time recovery possible

Disadvantages: - Must match PostgreSQL version exactly - All-or-nothing restoration

8.3 Our Backup Strategy

For our scale (documents measured in gigabytes, not terabytes), pg_dump is appropriate:

Daily Backups: We run pg_dump nightly, saving the output to a date-stamped file.

Retention: We keep 7 daily backups, 4 weekly backups (Sunday), and 3 monthly backups.

Verification: Periodically restore backups to a test database to ensure they work.

Off-Site Storage: Backup files are copied to a different machine or cloud storage to protect against machine failure.

8.4 Point-in-Time Recovery

For critical systems, PostgreSQL supports Point-in-Time Recovery (PITR):

Continuous Archiving: PostgreSQL can archive Write-Ahead Log (WAL) files, which record every change to the database.

Recovery: By combining a base backup with archived WAL files, we can restore to any point in time—"restore to 10:15 AM yesterday."

We don't currently use PITR (our data is regenerable), but it's available if requirements change.

8.5 Recovery Procedures

When disaster strikes:

Stop PostgreSQL: Prevent further writes to potentially corrupted data.

Assess the Situation: Determine what failed—is this a single table issue or complete database corruption?

Restore from Backup: If needed, drop the corrupted database and restore from the most recent backup.

Verify Data: Check that restored data looks correct—row counts, sample queries.

Replay Lost Work: Determine what was lost since the backup and decide whether to reprocess those documents.

---

9. Performance Tuning

9.1 Understanding Query Performance

PostgreSQL provides tools to understand query performance:

EXPLAIN: Shows the query execution plan—how PostgreSQL will execute a query. This reveals whether indexes are being used and how data is being processed.

EXPLAIN ANALYZE: Actually runs the query and shows real timing for each step. More accurate but slower (it runs the query).

Reading these plans takes practice, but key things to look for: - **Seq Scan**: Scanning entire table—might indicate missing index - **Index Scan**: Using an index—usually good - **Nested Loop**: Processing each row individually—can be slow for large sets - **Sort**: Sorting results—memory-intensive for large sets

9.2 Memory Configuration

PostgreSQL's memory settings significantly impact performance:

shared_buffers: Memory for caching data pages. We set to 256MB-1GB depending on available RAM. Too low means more disk reads; too high leaves insufficient memory for OS file cache.

work_mem: Memory for sorting and hash operations per query. We set to 64MB. Too low means sorts spill to disk; too high risks memory exhaustion with many concurrent queries.

maintenance_work_mem: Memory for maintenance operations like CREATE INDEX and VACUUM. We set higher (256MB) since these run infrequently.

9.3 Index Tuning

Beyond the HNSW vector index, we tune traditional indexes:

Covering Indexes: If a query frequently retrieves specific columns, including them in the index avoids table lookups.

Partial Indexes: If we often query "chunks WHERE document_id = X", an index only on chunks for that document would be smaller and faster—but we'd need separate indexes for each document, which isn't practical.

Index-Only Scans: When all needed data is in the index, PostgreSQL skips the table entirely. Enabling this requires keeping the table vacuumed.

9.4 Connection Pool Tuning

Pool settings affect performance under load:

Pool Too Small: Under load, requests wait for connections, increasing latency.

Pool Too Large: Idle connections consume memory on both client and server.

We monitor connection usage and adjust: - If requests frequently wait for connections, increase pool_size - If many connections sit idle, decrease pool_size

9.5 Query Optimization Patterns

Common patterns that improve performance:

Batch Insertions: Instead of 500 individual INSERT statements, use a single INSERT with multiple value sets. This reduces round-trips and allows PostgreSQL to optimize.

Lazy Loading Avoidance: When we know we'll need related data, fetch it in the same query with JOINs rather than multiple separate queries (N+1 problem).

Pagination: For large result sets, use LIMIT and OFFSET (or keyset pagination) rather than fetching all rows.

Selective Columns: SELECT only needed columns rather than SELECT *. This reduces data transfer and allows more efficient index usage.

---

10. Common PostgreSQL Operations

10.1 Schema Creation

When setting up the database, we run schema creation scripts that:

Create Tables: Each CREATE TABLE statement defines columns, types, constraints, and defaults. We use IF NOT EXISTS to make scripts idempotent.

Create Indexes: After tables exist, we create indexes. Indexes on VECTOR columns take longer because HNSW construction is expensive.

Grant Permissions: We ensure the application user has appropriate permissions on all tables.

10.2 Data Import and Export

Importing Data: For bulk loading (like restoring from backup), we use COPY, which is much faster than INSERT statements. COPY reads directly from files into tables.

Exporting Data: pg_dump exports entire databases or specific tables. For partial exports (like specific documents), we use COPY with WHERE clauses.

10.3 Table Maintenance

PostgreSQL tables need periodic maintenance:

VACUUM: Reclaims space from deleted rows. PostgreSQL uses MVCC (Multi-Version Concurrency Control), meaning deleted rows aren't immediately removed. VACUUM marks their space as reusable.

ANALYZE: Updates statistics used by the query planner. After significant data changes, running ANALYZE helps PostgreSQL choose better query plans.

REINDEX: Rebuilds indexes, particularly useful for HNSW indexes after many insertions and deletions.

10.4 Monitoring

We monitor database health through:

pg_stat_activity: Shows current connections and what they're doing. Helps identify long-running queries or connection leaks.

pg_stat_user_tables: Shows per-table statistics—row counts, scan counts, index usage. Reveals tables that might need indexes.

Log Analysis: PostgreSQL logs can show slow queries (when configured), connection patterns, and errors.

---

11. Troubleshooting Database Issues

11.1 Connection Problems

Cannot Connect: - Check if PostgreSQL is running (`pg_isready`) - Verify connection parameters (host, port, credentials) - Check pg_hba.conf allows connections from your host - Ensure firewall allows the connection

Too Many Connections: - Check max_connections setting - Look for connection leaks in application (connections not returned to pool) - Consider using PgBouncer for connection pooling at PostgreSQL level

11.2 Performance Issues

Slow Queries: - Use EXPLAIN ANALYZE to understand query plan - Check for missing indexes - Look for sequential scans on large tables - Consider query restructuring

High Memory Usage: - Check work_mem setting - Look for queries with large sorts or hash operations - Monitor connection counts (each consumes memory)

Disk Space Issues: - Run VACUUM to reclaim space - Check for bloated tables or indexes - Review backup retention (old backups consuming space)

11.3 Data Integrity Issues

Foreign Key Violations: - Data insertion failed because referenced row doesn't exist - Check insertion order (parent before child) - Verify referenced data exists

Unique Constraint Violations: - Attempting to insert duplicate data - Implement upsert logic (INSERT ... ON CONFLICT DO UPDATE) - Check for race conditions in concurrent insertions

11.4 Extension Issues

pgvector Not Found: - Extension not installed in PostgreSQL - Extension not enabled in current database (CREATE EXTENSION vector) - PostgreSQL version incompatible with pgvector version

Vector Dimension Mismatch: - Embedding model changed, producing different dimensionality - Existing VECTOR(384) column won't accept VECTOR(768) data - Must recreate embeddings or alter column (complex)

---

12. Production Considerations

12.1 Security Hardening

For production deployment:

Credentials Management: Never store passwords in code. Use environment variables or secrets management systems.

Network Security: PostgreSQL should not be exposed to the public internet. Run behind a firewall, accessible only from application servers.

TLS Encryption: Enable SSL for PostgreSQL connections, encrypting data in transit.

Minimal Permissions: Application user should have only necessary permissions, not superuser access.

12.2 High Availability

For production systems requiring uptime:

Replication: PostgreSQL supports streaming replication, maintaining real-time copies on standby servers.

Failover: If the primary fails, standby can be promoted to primary. This can be automatic with tools like Patroni.

Connection Routing: A proxy like PgBouncer or HAProxy can route connections to the current primary.

12.3 Monitoring and Alerting

Production systems need monitoring:

Health Checks: Regular checks that PostgreSQL is running and accepting connections.

Performance Metrics: Track query latency, connection counts, and disk usage over time.

Alerting: Automatic notifications when metrics exceed thresholds (e.g., connection pool exhausted, disk 90% full).

12.4 Scaling Considerations

If our system grows beyond a single PostgreSQL instance:

Vertical Scaling: Bigger server (more RAM, faster disk) helps significantly for database workloads.

Read Replicas: For read-heavy workloads, replicas can serve read queries while primary handles writes.

Partitioning: Large tables can be partitioned by document_id, improving query performance on single-document queries.

Sharding: The ultimate scaling approach—splitting data across multiple PostgreSQL instances. Complex to implement and usually not needed for our use case.

Summary

This chapter covered the practical aspects of PostgreSQL implementation:

1. **Setting Up PostgreSQL**: Installation, configuration files, initial setup decisions, creating databases and users with appropriate security.
2. **The pgvector Extension**: What it provides (VECTOR type, distance operators, HNSW indexing), how HNSW works conceptually, and parameter tuning.
3. **Connection Management**: Why connections are expensive, connection pooling concepts, the connection URL, and connection lifecycle management.
4. **SQLAlchemy Integration**: Core vs. ORM, engines and sessions, using text() for raw SQL, and context managers for session handling.
5. **Transaction Handling**: ACID properties, implicit vs. explicit transactions, commit and rollback, isolation levels, and risks of long-running transactions.
6. **Repository Pattern**: Separating database logic from business logic, repository structure, why methods take sessions, and error handling.
7. **Migrations**: Why schemas change, the migration concept, SQL-based vs. tool-based approaches, and safe migration practices.
8. **Backup and Recovery**: Why backups matter, pg_dump vs. pg_basebackup, backup strategy, and recovery procedures.
9. **Performance Tuning**: Understanding query plans, memory configuration, index tuning, connection pool tuning, and optimization patterns.
10. **Common Operations**: Schema creation, data import/export, table maintenance (VACUUM, ANALYZE, REINDEX), and monitoring.
11. **Troubleshooting**: Connection problems, performance issues, data integrity issues, and extension issues.
12. **Production Considerations**: Security hardening, high availability, monitoring, and scaling strategies.

Chapter 8: Frontend & API Layer

Building the User Interface and Web Services

Table of Contents for This Chapter

7. [Frontend Architecture)
8. [HTML Structure and Semantics)
9. [CSS Styling Philosophy)
10. [JavaScript Interactivity)
11. [Error Handling Across Layers)
12. [User Experience Considerations)

---

1. Architecture Overview

1.1 The Three-Layer Web Application

Our application follows a classic three-layer architecture that separates concerns and makes each layer independently maintainable:

Presentation Layer (Frontend): What the user sees and interacts with. This consists of HTML for structure, CSS for styling, and JavaScript for interactivity. The frontend runs entirely in the user's browser.

Application Layer (API): The Flask backend that receives requests from the frontend, orchestrates business logic, and returns responses. This is where our RAG pipeline lives—query processing, retrieval, and LLM interaction all happen here.

Data Layer (Database): PostgreSQL stores all persistent data—documents, chunks, embeddings, and Knowledge Graph data. The application layer communicates with this layer using SQLAlchemy.

This separation means we could replace the frontend with a mobile app or CLI tool without changing the API. We could also swap PostgreSQL for a different database without changing the frontend.

1.2 Request-Response Flow

When a user asks a question, the flow is:

1. **User Action**: Types question in the text box and clicks "Ask"
2. **JavaScript**: Captures the event, prepares the request, sends it to the API
3. **Flask Endpoint**: Receives the HTTP request, validates input
4. **Business Logic**: Query understanding, retrieval, context assembly
5. **LLM Generation**: Ollama generates the answer
6. **Streaming Response**: Tokens flow back through Flask to JavaScript
7. **DOM Update**: JavaScript appends each token to the answer area
8. **Complete**: User sees the full answer with sources

This entire flow takes 3-6 seconds, but because of streaming, the user sees progress continuously after the first ~300ms.

1.3 Why Flask?

We chose Flask over alternatives like Django or FastAPI for several reasons:

Simplicity: Flask is minimalist by design. It provides routing, request handling, and templating without imposing structure. For our focused application, this simplicity is an asset.

Flexibility: Flask doesn't dictate how to structure code: We organize our project as we see fit, adding components (like SQLAlchemy) explicitly rather than having them built in.

Streaming Support: Flask supports streaming responses natively through generator functions, which we need for real-time LLM output.

Learning Curve: Flask is widely taught and well-documented. Anyone familiar with Python can understand Flask code quickly.

The tradeoff is that Flask doesn't include features we might want—like automatic API documentation or async support—but for our use case, the simplicity outweighs these gaps.

---

2. Flask Backend Fundamentals

2.1 Application Factory Pattern

Rather than creating the Flask app as a global variable, we use the application factory pattern. The `create_app()` function constructs and configures the application:

The factory pattern provides benefits:

Testing: We can create multiple app instances with different configurations for testing.

Configuration Flexibility: The configuration (database URL, secret key, debug mode) is passed in, not hardcoded.

Circular Import Prevention: By creating the app in a function, we avoid circular imports that occur when modules import from each other.

2.2 Configuration Management

Flask applications need configuration for database connections, secret keys, and feature flags. We manage this through:

Environment Variables: Sensitive values like database passwords and secret keys come from environment variables, never from code. This follows the twelve-factor app methodology.

Configuration Classes: We define configuration classes for different environments—DevelopmentConfig, TestingConfig, ProductionConfig. Each sets appropriate defaults.

Configuration Loading: At startup, we load the appropriate configuration based on an environment variable (like FLASK_ENV).

This approach means the same code runs in development and production with different configurations, reducing "works on my machine" issues.

2.3 Blueprints for Organization

As the application grows, putting all routes in one file becomes unwieldy. Flask Blueprints let us organize routes by feature:

API Blueprint: All `/api/*` routes for programmatic access **Web Blueprint**: HTML-serving routes for the user interface **Admin Blueprint**: Administrative functions (if any)

Each blueprint is defined in its own file, registered with the main app at startup. This modular structure makes finding and modifying code easier.

2.4 Request Lifecycle

Understanding what happens during a request helps with debugging and optimization:

Request Reception: Flask receives the HTTP request from the web server (or its built-in development server).

Route Matching: Flask matches the URL path to a registered route. If no route matches, it returns 404.

Before-Request Hooks: Any registered `@app.before_request` functions run. We use this to set up database sessions.

View Function Execution: The matched route's function runs, doing the actual work.

After-Request Hooks: Any `@app.after_request` functions run. We use this to close database connections.

Response Return: The response (HTML, JSON, or stream) is sent back to the client.

This lifecycle provides natural extension points for logging, authentication, and resource management.

---

3. REST API Design

3.1 What Makes an API RESTful?

REST (Representational State Transfer) is an architectural style for web APIs. Our API follows REST principles:

Resources: We model things as resources—documents, chunks, entities. Each resource has a URL that identifies it.

HTTP Methods: We use HTTP methods semantically: - GET: Retrieve data (safe, no side effects) - POST: Create new resources or trigger actions - DELETE: Remove resources

Statelessness: Each request contains all information needed to process it. The server doesn't remember previous requests. This makes scaling easier—any server can handle any request.

JSON Responses: We return data as JSON, a widely supported format that's easy to parse in JavaScript and other languages.

3.2 Our API Endpoints

The API provides several endpoints:

Document Management: - `POST /api/upload` - Upload a new PDF document - `GET /api/documents` - List all uploaded documents - `GET /api/documents/{id}` - Get details of a specific document - `DELETE /api/documents/{id}` - Remove a document and all its data

Question Answering: - `POST /api/ask` - Ask a question about a document - This endpoint streams the response for real-time display

Knowledge Graph: - `GET /api/documents/{id}/entities` - Get extracted entities - `GET /api/documents/{id}/definitions` - Get defined terms

Each endpoint has a clear purpose, and the URL structure makes resources easy to understand.

3.3 Request and Response Formats

We standardize how requests and responses look:

Requests: POST endpoints expect JSON bodies with defined fields. We validate all input and return clear error messages for invalid data.

Success Responses: Include the requested data plus any useful metadata. For example, a document list response includes each document's status and processing progress.

Error Responses: Follow a consistent format with an error message and optionally an error code. HTTP status codes indicate the error category: - 400: Bad request (invalid input) - 404: Resource not found - 500: Server error (our bug)

This consistency makes the API predictable—clients know what to expect.

3.4 API Versioning Considerations

Though not currently implemented, API versioning is worth considering for the future:

URL Versioning: `/api/v1/documents` vs `/api/v2/documents`. Clear but requires updating URLs.

Header Versioning: Version specified in request headers. Cleaner URLs but less visible.

No Versioning: Fine for internal APIs with controlled clients. We use this currently since the frontend and backend are deployed together.

If we ever expose the API to external clients, versioning becomes important for backward compatibility.

---

4. File Upload Handling

4.1 The Challenge of File Uploads

Uploading PDFs is more complex than simple form submission:

File Size: PDFs can be several megabytes. We need to handle large uploads without memory issues.

File Validation: We must verify the file is actually a PDF, not something malicious.

Progress Feedback: For large files, users need to see upload progress.

Error Recovery: If processing fails partway, we need clean error handling.

4.2 The Upload Flow

When a user selects and uploads a PDF:

Step 1: Browser Selection The user selects a file using an HTML file input. JavaScript validates the file extension and size before uploading.

Step 2: FormData Transmission JavaScript creates a FormData object containing the file and sends it via fetch() to the upload endpoint.

Step 3: Server Reception Flask receives the file through request.files, a special dictionary for uploaded files.

Step 4: Validation We check: - File is present and non-empty - Filename ends with .pdf - File content looks like a PDF (magic bytes check) - File size is within limits

Step 5: Saving and Processing The file is saved to a processing directory. We create a document record in the database and initiate async processing.

Step 6: Progress Updates Processing happens in the background. The frontend polls a status endpoint to show progress.

4.3 Security Considerations

File uploads are a common attack vector. We implement several protections:

Filename Sanitization: We don't use the user-provided filename directly. Instead, we generate a safe filename (like the document_id) to prevent path traversal attacks.

Content Type Verification: We don't trust the browser's content-type header. We examine the file content itself to verify it's a PDF.

Size Limits: We enforce maximum file sizes (configurable, typically 50MB) to prevent denial-of-service through huge uploads.

Storage Location: Uploaded files are stored outside the web-accessible directory, preventing direct URL access to uploaded files.

4.4 Handling Large Files

For very large PDFs, we need to handle memory carefully:

Streaming Saves: Rather than loading the entire file into memory, we stream it to disk in chunks. Flask supports this natively.

Disk Space Monitoring: Before accepting an upload, we check available disk space.

Cleanup: After processing, we can delete the original PDF (keeping only extracted data) to save space, or we can keep it for future reference.

5. The Question-Answering Endpoint

5.1 The Heart of the Application

The `/api/ask` endpoint is where everything comes together. This single endpoint triggers the entire RAG pipeline:

1. Receive question and document ID
2. Classify the question type
3. Execute retrieval (vector and/or KG)
4. Assemble context
5. Generate answer with LLM
6. Stream response back to user

This endpoint is the most complex and the most critical to user experience.

5.2 Input Validation

Before processing, we validate the request:

Question Presence: The question must be non-empty after stripping whitespace.

Document ID: The specified document must exist and be in "ready" status. We can't answer questions about documents still being processed.

Question Length: We enforce a maximum question length (e.g., 1000 characters) to prevent abuse and ensure the LLM can process it.

If any validation fails, we return immediately with a clear error message and appropriate HTTP status code.

5.3 Orchestrating the Pipeline

Once validated, the endpoint orchestrates our RAG pipeline:

Initialize Components: Create HybridRAG, ContextAssembler, and LLMGenerator instances for this document.

Execute Query: Call the query method with the question. This internally decides on retrieval strategy, queries vector and/or KG, and returns results.

Assemble Context: Format the retrieved information for LLM consumption.

Generate Answer: Call the LLM with the assembled prompt and stream the response.

Each step can fail independently, and we handle failures gracefully—returning partial results when possible, clear errors when not.

5.4 Response Structure

For non-streaming responses, we return a JSON object:

The response includes: - **answer**: The generated text - **mode**: Which retrieval mode was used (vector, kg, hybrid) - **sources**: List of source chunks with page numbers and text snippets - **kg_results**: Any Knowledge Graph data used (entities, definitions)

This rich response lets the frontend display not just the answer but also the evidence supporting it.

---

6. Streaming Responses

6.1 Why Streaming Matters

Without streaming, the user experience would be: 1. User asks question 2. Spinner shows for 4-5 seconds 3. Complete answer appears all at once

With streaming: 1. User asks question 2. "Searching..." appears briefly 3. "Generating..." appears 4. Words start appearing one by one 5. Answer completes gradually

The total time is the same, but perceived speed is dramatically better. Users can start reading while more content loads.

6.2 Server-Side Streaming

Flask supports streaming through generator functions:

The endpoint function yields data instead of returning it all at once. Each yield sends that data immediately to the client without waiting for the complete response.

We yield structured JSON objects: - Status updates during retrieval: `{"type": "status", "message": "Searching..."}` - Tokens during generation: `{"type": "token", "content": "The"}` - Completion notification: `{"type": "done", "sources": 5}`

The response uses NDJSON (Newline-Delimited JSON) format—each line is a complete JSON object, making parsing straightforward.

6.3 Client-Side Stream Consumption

JavaScript's Fetch API supports streaming through ReadableStream:

The browser receives data in chunks. For each chunk: 1. Decode bytes to text 2. Split by newlines (each line is one JSON object) 3. Parse each JSON object 4. Update the UI based on message type

This creates the real-time "typing" effect users see.

6.4 Error Handling in Streams

Streaming complicates error handling because we can't change HTTP status codes after starting the response. We handle errors through message types:

Errors Before Streaming: If validation fails or retrieval errors occur before we start generating, we return a normal error response (no streaming).

Errors During Streaming: If something fails mid-stream (like LLM crashes), we send an error message type. The frontend displays the partial answer (if any) plus the error.

Connection Drops: If the client disconnects, Flask detects this and stops generating, avoiding wasted computation.

---

7. Frontend Architecture

7.1 The Single-Page Application Approach

Our frontend is a lightweight single-page application (SPA). The page loads once, and JavaScript handles all subsequent interactions without full page reloads.

Benefits of SPA: - Faster transitions (no page reloads) - Smooth animations and transitions - State persists across interactions - Better user experience for interactive applications

Simplicity Trade-offs: - No framework (React, Vue)—plain JavaScript keeps things simple - No build step—JavaScript runs directly in the browser - No routing library—we have only one main view

This approach suits our focused application. A more complex application might benefit from a full framework.

7.2 Component Organization

Though we don't use a component framework, we organize JavaScript into logical components:

Document Selector: UI for choosing which document to query

Question Input: Text area for entering questions, with submit button

Answer Display: Area where streamed answers appear, with source attribution

Status Indicators: Loading spinners, progress messages, error displays

Each "component" is a section of HTML with associated JavaScript functions that manipulate that section.

7.3 State Management

Our application maintains simple state:

Selected Document: Which document is currently selected for querying

Question History: Previously asked questions (for convenience)

Current Answer: The answer being streamed or displayed

UI State: Whether we're loading, showing an error, or idle

We store state in JavaScript variables, updating the DOM when state changes. For this simple application, no state management library is needed.

7.4 API Communication

JavaScript communicates with the Flask backend using the Fetch API:

GET requests: For fetching document lists, entity data, etc.

POST requests: For uploading files and asking questions

Error handling: We check response.ok and handle errors consistently

Headers: We set Content-Type for JSON requests and handle various response types

All API calls go through 2-3 helper functions that handle common patterns (authentication headers, error parsing, etc.).

---

8. HTML Structure and Semantics

8.1 Semantic HTML Importance

We use semantic HTML elements that convey meaning:

header: Contains the application title and navigation **main**: Contains the primary content—the Q&A interface **section**: Groups related content (document selection, Q&A area) **article**: Could wrap each Q&A exchange **footer**: Credits, links, version information

Semantic HTML benefits: - Accessibility: Screen readers understand page structure - SEO: Search engines understand content hierarchy (if applicable) - Maintainability: Code is self-documenting

8.2 Document Structure

Our HTML follows a logical structure:

Head Section: Meta tags, title, CSS links. We include viewport meta for mobile responsiveness.

Body Structure: - Header with logo/title - Main content area: - Document selector panel - Question input area - Answer display area - Source attribution panel - Footer with supplementary information

This structure supports responsive design—panels can stack on mobile while sitting side-by-side on desktop.

8.3 Accessibility Considerations

We implement accessibility best practices:

Form Labels: Every input has an associated label for screen reader users

ARIA Attributes: We add aria-live regions for dynamic content (like streaming answers) so screen readers announce updates

Keyboard Navigation: All interactive elements are keyboard accessible

Focus Management: When questions are answered, focus moves appropriately

Color Contrast: Text colors meet WCAG contrast requirements against backgrounds

These considerations ensure the application is usable by people with disabilities.

8.4 Progressive Enhancement

The application works at multiple capability levels:

HTML Only: The basic structure and content are visible even without JavaScript (though functionality is limited)

CSS Enhancement: Styling makes the interface attractive but isn't required for function

JavaScript Enhancement: Full interactivity, streaming, and dynamic updates

This layered approach ensures graceful degradation if any layer fails to load.

---

9. CSS Styling Philosophy

9.1 Design Goals

Our CSS aims for:

Clarity: Content is easy to read, interface elements are clearly distinguished

Professionalism: The design conveys trust—important for financial document analysis

Responsiveness: Works well on various screen sizes

Performance: Minimal CSS for fast loading

9.2 Layout Approach

We use modern CSS layout techniques:

Flexbox: For one-dimensional layouts—aligning items in rows or columns. The main content area uses flexbox to arrange panels.

CSS Grid: For two-dimensional layouts where we need precise row/column control.

Responsive Design: Media queries adjust layouts for different screen widths. On mobile, panels stack vertically; on desktop, they arrange horizontally.

9.3 Color and Typography

Color Palette: We use a limited palette: - Primary: A professional blue for interactive elements - Background: Light grays for content areas - Text: Dark gray for readability (not pure black, which is harsh) - Accent: Colors for success (green), warning (orange), error (red)

Typography: We use system fonts for fast loading and native feel. Font sizes follow a modular scale for visual harmony. Line heights ensure readability.

9.4 Component Styling

Each UI component has consistent styling:

Buttons: Clear hover and active states, disabled styling when inactive

Inputs: Focus states for accessibility, error states for validation

Cards/Panels: Consistent padding, borders, and shadows to create visual hierarchy

Loading States: Subtle animations that indicate activity without being distracting

9.5 Animation and Transitions

We use animations sparingly:

Transitions: Smooth color and opacity changes on hover/focus. These make the interface feel polished without being distracting.

Loading Animations: A subtle spinner or pulse indicates processing.

No Excessive Animation: We avoid animations that delay user actions or feel gimmicky. The focus is on content, not effects.

---

10. JavaScript Interactivity

10.1 JavaScript's Role

JavaScript handles everything that requires dynamic behavior:

Event Handling: Responding to clicks, form submissions, keyboard input

API Communication: Fetching data from the backend, sending questions

DOM Manipulation: Updating the page content without full reloads

Streaming Processing: Handling the streamed response and updating the UI in real-time

10.2 Event Handling Patterns

We attach event listeners to handle user actions:

Form Submission: Intercepting the submit event, preventing default browser behavior, and handling submission via JavaScript

Button Clicks: Triggering actions like clearing input or copying answers

Input Changes: Enabling/disabling the submit button based on input validity

Keyboard Shortcuts: Supporting Enter to submit (with Shift+Enter for newlines)

We prefer event delegation where practical—attaching listeners to parent elements rather than individual children, improving performance and simplifying dynamic content.

10.3 The Question Submission Flow

When the user submits a question:

1. **Validate Input**: Check that question is non-empty and document is selected
2. **Disable Submit**: Prevent double-submission while processing
3. **Clear Previous Answer**: Remove any previous answer from display
4. **Show Loading State**: Display a spinner or "Processing..." message
5. **Fetch with Streaming**: Send the question to the API with streaming enabled
6. **Process Stream**: As tokens arrive, append them to the answer display

7. **Handle Completion**: Hide loading state, show sources, re-enable submit
8. **Handle Errors**: If anything fails, show a clear error message

Each step updates the UI to keep the user informed.

10.4 DOM Manipulation Techniques

We update the DOM efficiently:

textContent vs innerHTML: Use textContent for plain text (safer, no HTML injection risk). Use innerHTML only when we need to render formatted content, and never with unsanitized user input.

Creating Elements: For complex dynamic content, we create elements programmatically rather than building HTML strings. This is safer and often easier to read.

Updating Existing Elements: We update text content or classes of existing elements rather than replacing large DOM sections, improving performance.

Batching Updates: When updating many elements, we minimize reflows by making changes in batches or using DocumentFragment.

10.5 Error Display

When errors occur, we display them clearly:

Error Placement: Errors appear near the action that caused them (near the input for validation errors, above the answer area for API errors)

Error Styling: Red text or backgrounds clearly indicate problems

Error Content: Messages explain what went wrong and what the user can do ("Document processing failed. Please try uploading again.")

Error Dismissal: Users can dismiss errors or they disappear automatically when the user takes a corrective action

---

11. Error Handling Across Layers

11.1 The Error Handling Philosophy

Errors will happen. Our goal is to:

Prevent When Possible: Validation catches many errors before they cause problems

Detect Early: Fail fast rather than letting bad data propagate

Handle Gracefully: Never show stack traces to users; always provide actionable messages

Log for Debugging: Backend errors are logged with details for developer investigation

11.2 Frontend Error Handling

JavaScript handles UI-level errors:

Validation Errors: Caught before API calls. "Please enter a question" if the input is empty.

Network Errors: When fetch fails due to connectivity issues. "Unable to reach server. Please check your connection."

API Errors: When the backend returns an error response. We parse and display the error message from the JSON response.

Unexpected Errors: Catch-all try/catch blocks prevent JavaScript errors from leaving the UI in a broken state.

11.3 API Error Handling

The Flask backend validates input and handles processing errors:

Input Validation: Returns 400 Bad Request with a message explaining what's wrong.

Not Found: Returns 404 when a document or resource doesn't exist.

Processing Errors: Returns 500 Internal Server Error with a generic message. Details are logged for debugging but not exposed to users (security).

Streaming Errors: Mid-stream errors send an error message type; the frontend handles display.

11.4 Database Error Handling

SQLAlchemy operations can fail:

Connection Errors: If the database is down, we catch exceptions and return service unavailable errors. The application logs the issue for alerting.

Constraint Violations: Duplicate document IDs or invalid foreign keys are caught and translated to meaningful errors.

Query Timeouts: Long-running queries are configured to timeout, preventing indefinite hangs.

11.5 LLM Error Handling

The LLM can fail in various ways:

Service Unavailable: If Ollama isn't running, we catch the connection error and inform the user.

Generation Timeout: If generation takes too long (maybe the model is stuck), we timeout and return an error.

Invalid Response: If the LLM returns unparseable output, we handle gracefully rather than crashing.

Content Filtering: Some models refuse certain queries. We detect this and inform the user.

---

12. User Experience Considerations

12.1 First Impressions

When users first load the application:

Fast Initial Load: The page loads quickly (minimal JavaScript, CSS). Users see the interface within 1-2 seconds.

Clear Purpose: The interface immediately communicates what the application does—ask questions about IPO documents.

Guided Action: If no documents are uploaded, we prompt users to upload one. If documents exist, we guide them to select one and ask a question.

12.2 Feedback and Responsiveness

Users need constant feedback that their actions are taking effect:

Button States: Buttons show hover effects, active states, and disabled states clearly.

Loading Indicators: Any action taking more than 200ms shows a loading indicator. For longer operations (document upload, question answering), we show progress.

Success Confirmation: When actions complete successfully, we confirm it (green checkmark, success message).

Error Notification: Errors are clearly distinguished from normal messages (red coloring, error icons).

12.3 Perceived Performance

Even when operations take time, users should feel the application is fast:

Streaming Answers: Rather than waiting for complete answers, users see words appearing immediately.

Optimistic UI Updates: Where safe, we update the UI before server confirmation (like disabling the submit button immediately on click).

Progressive Loading: For long lists (like many documents), we could load visible items first.

Skeleton Loading: Show placeholder shapes while content loads, reducing perceived wait time.

12.4 Error Recovery

When things go wrong, help users recover:

Clear Instructions: "Your question couldn't be processed. Please try asking differently or select a different document."

Retry Options: For transient errors, provide a retry button.

State Preservation: If an error occurs mid-operation, don't lose the user's input. The question they typed should still be there.

Fallback Options: If the primary feature fails, suggest alternatives. "LLM unavailable. You can still browse extracted entities below."

12.5 Mobile Considerations

Though primarily a desktop application, we consider mobile users:

Touch Targets: Buttons are large enough to tap accurately (minimum 44x44 pixels).

Responsive Layout: The interface adapts to narrow screens, stacking elements vertically.

Keyboard Handling: The virtual keyboard on mobile doesn't obscure important content.

Performance: Mobile devices may be less powerful, so we keep JavaScript light.

Summary

This chapter covered the Frontend and API Layer in depth:

1. **Architecture Overview**: Three-layer separation (frontend, API, database), request-response flow, and why we chose Flask.
2. **Flask Backend Fundamentals**: Application factory pattern, configuration management, blueprints for organization, and the request lifecycle.
3. **REST API Design**: What makes APIs RESTful, our endpoint structure, request/response formats, and versioning considerations.
4. **File Upload Handling**: The upload flow, security considerations (sanitization, validation, size limits), and handling large files.
5. **Question-Answering Endpoint**: Input validation, orchestrating the RAG pipeline, and response structure.
6. **Streaming Responses**: Why streaming matters for UX, server-side generator functions, client-side stream consumption, and error handling in streams.
7. **Frontend Architecture**: SPA approach, component organization, state management, and API communication patterns.
8. **HTML Structure**: Semantic HTML elements, document structure, accessibility considerations, and progressive enhancement.
9. **CSS Styling**: Design goals, layout approaches (flexbox, grid, responsive), color/typography choices, and animation philosophy.
10. **JavaScript Interactivity**: Event handling patterns, the question submission flow, DOM manipulation techniques, and error display.
11. **Error Handling**: Philosophy of error handling, frontend/API/database/LLM error handling strategies.
12. **User Experience**: First impressions, feedback and responsiveness, perceived performance, error recovery, and mobile considerations.

Chapter 9: System Integration & Data Flow

How Everything Works Together

Table of Contents for This Chapter

---

1. The Complete System Overview

1.1 Components and Their Responsibilities

Our system consists of several interconnected components, each with specific responsibilities:

PDF Processing Pipeline: Takes raw PDF files, extracts text, preserves structure, identifies sections and pages. This is the entry point for all document data.

Chunking Engine: Transforms long documents into manageable pieces, applying semantic boundaries, handling overlap, and tracking provenance (page numbers, sections).

Embedding Service: Converts text chunks into vector representations using a transformer model. These vectors enable semantic similarity search.

Knowledge Graph Extractor: Uses an LLM to identify entities, relationships, and definitions from text. Structures unstructured information.

PostgreSQL Database: Stores everything persistently—documents, chunks, embeddings, and all Knowledge Graph data. Uses pgvector for efficient vector operations.

Retrieval Engine: Given a query, decides which retrieval strategy to use, executes vector and/or KG searches, and assembles context.

LLM Service (Ollama): Takes contextualized prompts and generates natural language answers. Runs locally for privacy and cost efficiency.

Flask API Server: Exposes HTTP endpoints for document upload, question answering, and data retrieval. Orchestrates all backend components.

Frontend UI: Browser-based interface for user interaction. Handles file uploads, question submission, and answer display with streaming.

1.2 How Components Communicate

Components communicate through well-defined interfaces:

Python Function Calls: Most component communication happens through direct Python function calls within the same process. The Flask app calls embedding functions, which call database functions.

HTTP Requests: The frontend communicates with Flask via HTTP. Ollama also exposes an HTTP API that our code calls for LLM generation.

Database Connections: PostgreSQL connections use TCP with the PostgreSQL wire protocol. SQLAlchemy manages these connections.

File System: PDF files are saved to disk during upload. The PDF processor reads from disk. This decouples upload from processing timing.

1.3 Synchronous vs. Asynchronous Operations

Some operations complete synchronously (in the same request), while others are asynchronous:

Synchronous Operations: - Question answering (though streaming provides responsiveness) - Entity retrieval - Document listing

Asynchronous Operations: - Document processing after upload (happens in background) - KG extraction (can take minutes; frontend polls for status)

Asynchronous operations return immediately with a status ("processing started") and the frontend polls or waits for completion. This prevents HTTP timeouts for long-running operations.

---

2. Document Ingestion Flow

2.1 The Complete Upload-to-Ready Journey

When a user uploads a document, a cascade of processing begins. Understanding this flow helps debug issues and explain system behavior:

Phase 1: HTTP Reception (Immediate) The browser sends the PDF file to the Flask endpoint. Flask validates the request, saves the file to disk, and creates a document record with status="uploaded". A response returns immediately, telling the user upload succeeded.

Phase 2: Text Extraction (Seconds) Background processing begins. PyPDF2 opens the PDF and extracts text from each page. Page boundaries, headers, and footers are identified. The raw text plus page mapping is prepared for chunking.

Phase 3: Chunking (Seconds) The text is split into chunks using our semantic chunking strategy. Each chunk records its page span, section title, and character positions. Chunks are inserted into the database. Status changes to "chunking".

Phase 4: Embedding Generation (Seconds to Minutes) For each chunk, we generate a 384-dimensional vector using the embedding model. These vectors are inserted into the embeddings table with references to their chunks. Status changes to "embedding".

Phase 5: KG Extraction (Minutes) This is the longest phase. We process chunks through the LLM to extract entities, relationships, definitions, and events. Each extraction creates records in the KG tables with supporting evidence. Status changes to "kg_extraction".

Phase 6: Ready When all processing completes successfully, status changes to "ready". The document can now be queried. Total time: typically 5-30 minutes depending on document length.

2.2 What Happens at Each Database Level

During ingestion, data flows through the database in a specific order:

First: documents table A single record created immediately upon upload. This is the anchor that all other data references.

Second: chunks table Hundreds of records (one per chunk) created during chunking. Each references the document and stores the actual text content.

Third: embeddings table Matching records for each chunk, storing the vector representation. Created during embedding phase.

Fourth: evidence table Records containing quotes that support extracted facts. Created during KG extraction.

Fifth: KG tables (entities, claims, etc.) Structured facts extracted from the document. Created during KG extraction, referencing evidence records.

This order respects foreign key constraints—you can't create a chunk without a document, can't create an embedding without a chunk, can't create an entity without evidence.

2.3 Error Handling During Ingestion

Each phase can fail, and we handle failures differently:

Upload Failure: User sees an immediate error. No database records created. Common causes: file too large, not a PDF, network interruption.

Extraction Failure: Document status set to "error" with details in metadata. Partial data might exist (document record, no chunks). User can retry or delete.

Chunking Failure: Status set to "error". Chunks created so far might exist. Usually indicates malformed PDF or extraction issues.

Embedding Failure: Status set to "error". This is rare—usually indicates model not loaded or memory exhaustion.

KG Extraction Failure: Status might be "error" or might leave the document partially processed. Since KG extraction is enhancement, we could allow the document to be "ready" with partial KG data.

The frontend checks status and displays appropriate messages: “Processing…” during active phases, “Error: [message]” if something failed.

---

3. Knowledge Graph Extraction Flow

3.1 From Text to Structured Knowledge

KG extraction transforms unstructured text into structured facts. This process has several stages:

Stage 1: Chunk Selection Not all chunks need KG extraction. We prioritize chunks likely to contain extractable information: “About Our Promoters”, “Capital Structure”, “Key Management Personnel”. Purely narrative sections might be skipped to save processing time.

Stage 2: Prompt Construction For each selected chunk, we construct a prompt asking the LLM to identify: - Named entities (people, companies, regulatory bodies) - Relationships between entities - Defined terms with their definitions - Events with dates

The prompt includes the chunk text and instructions for structuring the output.

Stage 3: LLM Extraction Ollama processes the prompt and returns structured output. We use specific output formatting (JSON or structured text) to make parsing reliable.

Stage 4: Evidence Recording Before creating entities or claims, we record the evidence—the exact quote from the text that supports each fact. This creates the traceability foundation.

Stage 5: Entity Resolution The same entity might be mentioned in multiple chunks with slight variations (“Yashish Dahiya”, “Mr. Dahiya”, “the CEO”). We resolve these to a single canonical entity, adding aliases for the variations.

Stage 6: Claim Creation With entities resolved, we create claims (relationships) linking them. Each claim references its evidence.

Stage 7: Definition and Event Extraction Defined terms are matched against the definitions section of the document. Events with dates are extracted separately.

3.2 The Entity Resolution Challenge

Entity resolution is one of the trickier parts of KG extraction:

The Problem: The document might mention “SEC” in one place and “Securities and Exchange Commission” in another. These are the same entity but written differently.

Our Approach: 1. First, extract all mentions with their attributes 2. Group mentions by similarity (name overlap, shared attributes) 3. Choose a canonical name (usually the most formal/complete version) 4. Create a single entity with aliases for variations

Edge Cases: - Different entities with similar names (two people named “Mr. Singh”) - Abbreviations that could mean different things in different contexts - Entities that change (company renamed)

We err on the side of creating separate entities when uncertain—incorrectly merging entities is worse than having duplicates.

3.3 Maintaining Extraction Quality

LLM extraction isn't perfect. We maintain quality through:

Confidence Scores: Each extraction includes a confidence score. Low-confidence extractions might be flagged for review or excluded from query results.

Evidence Requirements: Every claim must have supporting evidence. If we can't point to a quote that supports a fact, we don't include it.

Validation Rules: Some validations can be automated. A shareholding percentage should be between 0 and 100. A company incorporation date should be in the past.

Incremental Improvement: As we identify patterns of extraction errors, we refine prompts or add post-processing rules.

---

4. Query Processing Flow

4.1 The Question-to-Answer Pipeline

When a user asks a question, an intricate pipeline activates:

Step 1: Request Reception Flask receives the POST request with the question and document ID. Basic validation occurs—is the question non-empty? Does the document exist and have status="ready"?

Step 2: Query Understanding We analyze the question to determine: - Question type (entity, definition, factual, descriptive) - Key entities mentioned in the question - Appropriate retrieval mode (vector, KG, hybrid)

This analysis uses pattern matching and simple NLP techniques—no LLM needed for this step.

Step 3: Retrieval Execution

Based on mode selection:

Vector Retrieval: The question is embedded using the same model as chunks. We query the embeddings table for the top-K most similar vectors, retrieving their associated chunks.

KG Retrieval: We query the KG tables for relevant entities, claims, and definitions. For "Who is the CEO?", we look for PERSON entities with role attributes containing "CEO".

Hybrid: Both retrievals execute, possibly in parallel, and results are merged.

Step 4: Context Assembly Retrieved information is formatted into a context block: - Structured facts from KG (precise, high-confidence) - Text chunks from vector search (narrative, supporting) - Definitions if relevant terms were found

The context is truncated if too long (respecting token limits).

Step 5: Prompt Construction We build the complete prompt: - System instructions (answer from context, cite sources, admit uncertainty) - Assembled context - The user's question - Answer format guidance

Step 6: LLM Generation The prompt is sent to Ollama. Ollama processes and starts generating tokens.

Step 7: Streaming Response As Ollama generates tokens, we stream them to the frontend. Each token is wrapped in a JSON message and sent via NDJSON streaming.

Step 8: Completion When generation finishes, we send a completion message with metadata (sources used, retrieval mode, timing). The frontend displays the complete answer with source citations.

4.2 Timing Breakdown

Understanding where time goes helps with optimization:

Step	Time	% of Total
Query understanding	10-20ms	<1%
Vector embedding	20-50ms	1-2%
Vector search	50-100ms	2-3%
KG queries	20-50ms	1-2%
Context assembly	10-20ms	<1%
Prompt construction	5ms	<1%
LLM generation	2-5 seconds	90-95%
Streaming overhead	10-20ms	<1%

The overwhelming majority of time is spent in LLM generation. All our optimization efforts for retrieval yield only marginal total time improvement—but they significantly reduce time-to-first-token, improving perceived responsiveness.

4.3 Fallback and Degradation

The pipeline has fallback mechanisms:

KG Unavailable → Vector Only: If KG tables are empty or queries fail, we fall back to pure vector retrieval. The answer might be less precise but still useful.

Low Vector Scores → Warning: If the best vector matches have similarity below 0.5, we warn the user: "The answer may not be in this document."

LLM Failure → Return Context: If Ollama is down, we can return the raw context, letting users read the relevant passages themselves.

Timeout → Partial Response: If generation takes too long, we can return what's generated so far with a timeout notice.

5. Data Lifecycle Management

5.1 Data Creation

Data enters the system through two paths:

User Uploads: PDF files uploaded through the frontend trigger document ingestion. All document-related data (chunks, embeddings, KG) derives from these uploads.

System Generation: During processing, the system generates derived data—embeddings are computed by the model, KG data is extracted by the LLM.

Understanding this distinction matters for backup and recovery. PDF files are the source of truth; derived data can be regenerated (though at significant time cost).

5.2 Data Modification

Most data in our system is immutable once created:

Documents: The document record might have its status updated, but the core metadata (filename, upload date) doesn't change.

Chunks: Created once, never modified. If we need to re-chunk (perhaps with different parameters), we delete and recreate.

Embeddings: Tied to a specific model. If we change models, we regenerate embeddings rather than updating existing ones.

KG Data: Might be updated if we improve extraction or reprocess. Generally treated as regeneratable.

This immutability simplifies the system—we don't need complex update logic or versioning within documents.

5.3 Data Deletion

When a document is deleted, all associated data must be removed:

Cascading Deletes: PostgreSQL's foreign key constraints with ON DELETE CASCADE handle this automatically. Deleting a document removes: - All its chunks - All embeddings for those chunks - All evidence records - All entities (and their aliases) - All claims - All defined terms - All events (and their participants)

File Cleanup: The original PDF file should also be deleted from the filesystem. This happens in application code after the database deletion succeeds.

Orphan Prevention: Our constraint design prevents orphaned records. Every piece of data traces back to a document.

5.4 Storage Growth and Management

As more documents are processed, storage requirements grow:

Chunks: ~5-10 KB per chunk (mostly text). A 100-page document might have 200-500 chunks.

Embeddings: ~1.5 KB per embedding (384 floats × 4 bytes). Same count as chunks.

KG Data: Varies widely based on document content. Maybe 50-500 entities, 100-1000 claims, 50-200 definitions. Each record is small (< 1 KB typically).

Total per Document: A typical 100-page document uses perhaps 5-20 MB of database storage (dominated by embeddings).

For a system with 100 documents, we're looking at 500 MB - 2 GB of database storage. PostgreSQL handles this easily.

---

6. Component Dependencies

6.1 The Dependency Graph

Understanding dependencies helps with startup order and debugging:

Core Dependencies (Always Required): - Python 3.8+ - PostgreSQL 14+ with pgvector extension - Required Python packages (Flask, SQLAlchemy, PyPDF2, etc.)

Processing Dependencies: - Sentence Transformers (for embedding model) - Ollama (for LLM operations—both extraction and generation)

Development Dependencies: - pytest for testing - black/flake8 for code formatting - Development server (Flask's built-in)

6.2 Startup Order

Components must start in a specific order:

First: Database PostgreSQL must be running before the application starts. The application will fail to connect otherwise.

Second: Ollama Ollama should be running for full functionality. Without it, document processing can't do KG extraction, and queries can't generate answers. The application can start without Ollama but will have limited functionality.

Third: Application Once dependencies are running, Flask can start. During startup, it: - Loads configuration - Establishes database connection pool - Loads the embedding model into memory - Registers routes and blueprints

Fourth: Model Downloads (First Run) On first run, the embedding model downloads automatically (sentence-transformers handles this). Ollama models must be pulled separately using `ollama pull llama3`.

6.3 Health Checks

The application should verify dependencies are healthy:

Database Connectivity: We can run a simple query (SELECT 1) to verify PostgreSQL is responding.

Ollama Availability: Calling Ollama's API endpoint verifies it's running. We might cache this check since the LLM is only needed for processing and queries.

Model Loading: Verifying the embedding model is loaded by running a test embedding.

Health checks can run at startup (fail fast if critical dependencies are missing) and periodically (detect problems that develop during runtime).

---

7. Configuration and Environment

7.1 Configuration Philosophy

We follow the twelve-factor app approach to configuration:

Strict Separation: Configuration lives in the environment, not in code. The same codebase runs in development, testing, and production with different configurations.

Environment Variables: Sensitive values (database password, secret key) are environment variables. They're never committed to version control.

Sensible Defaults: For non-sensitive configuration, code provides sensible defaults that work for development.

7.2 Key Configuration Values

Database Configuration: - `DATABASE_URL`: Full PostgreSQL connection URL - Connection pool settings (pool size, overflow, timeout)

Ollama Configuration: - `OLLAMA_HOST`: Where Ollama is running (default: localhost:11434) - `OLLAMA_MODEL`: Which model to use (default: llama3)

Application Configuration: - `SECRET_KEY`: For session signing (required in production) - `DEBUG`: Enable/disable debug mode - `MAX_CONTENT_LENGTH`: Maximum upload size

Model Configuration: - `EMBEDDING_MODEL`: Which sentence-transformer model (default: all-MiniLM-L6-v2)

7.3 Environment-Specific Configuration

Different environments need different configurations:

Development: - Debug mode enabled - Local database (might be SQLite for simplicity or local PostgreSQL) - Verbose logging - Smaller models for faster iteration

Testing: - Test database (separate from development) - Mocked external services where appropriate - Faster/smaller models

Production: - Debug mode disabled - Production database with proper credentials - Error-only logging - Full-sized models for quality

Configuration classes encapsulate these differences, and the environment determines which class to use.

---

8. Logging and Observability

8.1 What We Log

Logging provides visibility into system behavior:

Request Logs: Every HTTP request—method, path, response code, timing.

Processing Logs: Document ingestion progress—started, chunking complete, embedding complete, etc.

Query Logs: Questions asked, retrieval mode used, chunks retrieved, generation time.

Error Logs: Exceptions with stack traces, failed operations with context.

Performance Logs: Timing for slow operations, resource usage.

8.2 Log Levels

We use standard log levels appropriately:

DEBUG: Detailed information for development. Not enabled in production.

INFO: Normal operations—document processed, question answered.

WARNING: Something unexpected but recoverable—low similarity scores, slow query.

ERROR: Something failed but the application continues—single question failed, LLM timeout.

CRITICAL: Application cannot continue—database connection lost, no memory.

Production typically logs WARNING and above to avoid log volume issues.

8.3 Structured Logging

Rather than unstructured text logs, we use structured logging:

Format: JSON or key-value pairs that can be parsed programmatically.

Context: Each log includes context—request ID, document ID, user (if any).

Searchability: Structured logs can be searched and aggregated: "Show all queries where retrieval took > 500ms."

8.4 Metrics and Monitoring

Beyond logs, we could track metrics:

Application Metrics: - Requests per second - Response time percentiles (p50, p95, p99) - Error rate - Active connections

Processing Metrics: - Documents processed per day - Average processing time - Extraction success rate

Resource Metrics: - CPU usage - Memory usage - Disk space - Database connection pool utilization

These metrics can be exported to monitoring systems (Prometheus, Datadog) and visualized in dashboards.

---

9. Testing Strategy

9.1 Testing Philosophy

Our testing approach balances thoroughness with practicality:

Pyramid Model: Many unit tests (fast, focused), fewer integration tests (slower, broader), few end-to-end tests (slowest, most comprehensive).

Test What Matters: Focus testing on critical paths—ingestion, retrieval, generation—rather than trying to cover every line.

Reproducibility: Tests should produce the same results every time. Mock external dependencies (LLM, embedding model for some tests).

9.2 Unit Testing

Unit tests verify individual functions and classes:

Chunking Tests: Does the chunker produce expected chunks for known input? Are overlaps correct? Are page numbers tracked properly?

Repository Tests: Do repository methods correctly insert, query, and delete data? These might use a test database.

Utility Tests: String manipulation, configuration parsing, validation functions.

Unit tests run quickly (seconds) and provide confidence that individual components work correctly.

9.3 Integration Testing

Integration tests verify components work together:

Database Integration: Tests that actually connect to PostgreSQL, create tables, insert data, and query. Slower but more realistic.

Embedding Integration: Tests that load the real embedding model and verify embeddings have expected properties (correct dimensionality, similar text produces similar vectors).

Retrieval Integration: Tests that insert documents and chunks, then verify search returns expected results.

Integration tests take longer but catch issues that unit tests miss—like SQL syntax errors or model loading problems.

9.4 End-to-End Testing

End-to-end tests verify the complete workflow:

Full Ingestion: Upload a real (small) PDF and verify all data is created correctly.

Full Query: Ask a question and verify the answer is reasonable and sources are correct.

Full UI: Using browser automation (Selenium, Playwright), verify the frontend works correctly.

These tests are slow and sometimes flaky but provide ultimate confidence that the system works as users expect.

9.5 Testing External Dependencies

External dependencies (Ollama) present testing challenges:

Mocking: For unit tests, mock Ollama's responses. This is fast and deterministic but doesn't test actual LLM behavior.

Skip If Unavailable: Integration tests can skip LLM-dependent tests if Ollama isn't running.

Test Fixtures: Use canned responses for deterministic testing. Record real responses and replay them.

The tradeoff is between test speed/reliability and realism.

---

10. Debugging Complex Issues

10.1 Common Issues and Investigation

When something goes wrong, systematic debugging helps:

Document Upload Fails

Symptoms: Error message during upload or processing stuck.

Investigation Steps: 1. Check Flask logs for error messages 2. Verify file was saved to disk 3. Check document status in database 4. If processing started, check each phase's output

Common Causes: - File too large (check MAX_CONTENT_LENGTH) - PDF corrupted or password-protected - Database connection issues

Questions Return Poor Answers

Symptoms: Answers are irrelevant, factually wrong, or "I don't know" when information exists.

Investigation Steps: 1. Check retrieval mode used (in response metadata) 2. Examine retrieved chunks—were they relevant? 3. Check KG lookup—did we find expected entities? 4. Examine the assembled context—was it formatted correctly? 5. Test the same prompt directly in Ollama—is the LLM the issue?

Common Causes: - Retrieval found wrong chunks (embedding issue, question wording) - KG doesn't have the expected data (extraction issue) - LLM hallucinated despite context (prompt engineering issue)

System Runs Slowly

Symptoms: Queries take longer than expected, processing is slow.

Investigation Steps: 1. Check timing breakdown in logs 2. Use EXPLAIN ANALYZE on slow database queries 3. Check system resources (CPU, memory, disk) 4. Profile Python code to find bottlenecks

Common Causes: - Missing database indexes - Memory exhaustion causing swapping - Model loading repeated (should be cached)

10.2 Debugging Tools

Several tools help with debugging:

Flask Debug Mode: Provides detailed error pages with stack traces and an interactive debugger.

psql: PostgreSQL's command-line client for directly querying the database, examining schemas, and running EXPLAIN.

Logging: Increasing log level to DEBUG reveals detailed operation flow.

Python Debugger (pdb): Set breakpoints in code to examine state at specific points.

Database Inspection: Query counts, examine sample data, verify constraints.

10.3 Reproducing Issues

To fix issues, we first need to reproduce them:

Capture Inputs: What document, what question, what configuration?

Isolate the Issue: Is the problem in retrieval, generation, or elsewhere?

Simplify: Can we reproduce with a simpler input?

Test Environment: Reproduce in a controlled environment before modifying production.

Once reproducible, fixes can be verified by confirming the reproduction case now works.

---

11. Deployment Considerations

11.1 Development Deployment

For local development, we run everything on one machine:

Components: PostgreSQL, Ollama, Flask dev server all run locally.

Startup: Manual commands or a script to start each component.

Data Persistence: Local database, local files.

Advantages: Simple setup, full control, fast iteration.

This is how we develop and debug the application.

11.2 Simple Production Deployment

For a single-user or small-team deployment:

Single Server: All components run on one (powerful) server.

Process Management: Use supervisord, systemd, or similar to ensure processes stay running.

Reverse Proxy: Nginx in front of Flask for HTTPS, static file serving, and security.

Backups: Scheduled pg_dump to local disk and/or cloud storage.

This handles modest traffic (few users, occasional queries) inexpensively.

11.3 Containerized Deployment

For more sophisticated deployment, containers provide consistency:

Docker Images: Each component (app, postgres, ollama) has a Dockerfile.

Docker Compose: For single-machine deployment, docker-compose orchestrates all containers.

Kubernetes: For multi-machine deployment, Kubernetes provides scaling and resilience.

Containers ensure the application runs identically across environments—no more "works on my machine."

11.4 Cloud Deployment Options

For cloud deployment, several options exist:

Managed PostgreSQL: Cloud providers offer managed PostgreSQL (RDS, Cloud SQL), handling backups, updates, and scaling.

Container Platforms: AWS ECS, Google Cloud Run, or Azure Container Instances run containerized applications without managing VMs.

GPU for Ollama: For better LLM performance, GPU instances (expensive) dramatically speed generation.

The tradeoff is cost versus operational simplicity. Managed services cost more but require less maintenance.

---

12. Future Architecture Evolution

12.1 Scaling Challenges

As usage grows, several components might need scaling:

Database Performance: As document count grows, queries might slow. Solutions include better indexing, read replicas, or database sharding.

LLM Throughput: If many users query simultaneously, Ollama becomes a bottleneck. Solutions include multiple Ollama instances, queue-based processing, or cloud LLM APIs.

Storage: Embeddings consume the most space. Solutions include embedding compression, tiered storage, or vector database services.

12.2 Potential Architecture Changes

Several architectural evolutions are possible:

Async Processing Queue: Rather than background threads, use a proper task queue (Celery, RQ) for document processing. Provides better visibility, retry logic, and scaling.

Dedicated Vector Database: As vector volume grows, a specialized vector database (Pinecone, Weaviate, Milvus) might outperform pgvector. Tradeoff is additional complexity.

Cloud LLM Fallback: For high load, fall back to cloud LLMs (OpenAI, Anthropic) when local Ollama is overloaded. Trades cost for capacity.

Caching Layer: Add Redis for caching frequent queries, embedding model, and session data. Reduces database load.

12.3 Feature Evolution

The architecture should accommodate future features:

Multi-Document Queries: Asking questions across multiple documents requires cross-document entity resolution and retrieval.

Comparative Analysis: "Compare company A's financials with company B's" requires retrieval from multiple document's KGs.

User Management: Adding authentication requires session management, user storage, and per-user document access control.

Fine-Tuned Models: Training custom extraction or embedding models requires model versioning and migration strategies.

Good architecture allows these features to be added incrementally without rewriting the core system.

Summary

This chapter covered how all system components integrate and how data flows through the system:

1. **System Overview**: All components (PDF processing, chunking, embedding, KG extraction, database, retrieval, LLM, API, frontend) and how they communicate.
2. **Document Ingestion**: The complete upload-to-ready journey through text extraction, chunking, embedding, and KG extraction, with error handling at each phase.

3. **KG Extraction Flow**: From unstructured text to structured knowledge via prompt construction, LLM extraction, entity resolution, and claim creation.
4. **Query Processing**: The question-to-answer pipeline including query understanding, retrieval, context assembly, and streaming generation.
5. **Data Lifecycle**: Creation, modification, deletion patterns, including cascading deletes and storage management.
6. **Component Dependencies**: The dependency graph, startup order, and health checking.
7. **Configuration**: Environment-based configuration following twelve-factor principles.
8. **Logging and Observability**: What we log, log levels, structured logging, and metrics.
9. **Testing Strategy**: Unit, integration, and end-to-end testing approaches.
10. **Debugging**: Common issues, investigation techniques, debugging tools, and reproduction strategies.
11. **Deployment**: Development, simple production, containerized, and cloud deployment options.
12. **Future Evolution**: Scaling challenges, potential architecture changes, and feature evolution.

Chapter 10: Interview Q&A Guide

Explaining This Project in Technical Interviews

Table of Contents for This Chapter

---

1. Elevator Pitch

1.1 The 30-Second Pitch

Question: "Tell me about a project you've worked on."

"I built an intelligent Q&A system for IPO documents—prospectuses that are typically 400+ pages of dense financial and legal text. The system lets users ask natural language questions like 'Who are the promoters?' or 'What is the company's revenue?' and get accurate answers with source citations.

What makes it interesting is the hybrid approach: I combine vector-based semantic search with a Knowledge Graph I extract from the documents. Vector search finds relevant passages, while the Knowledge Graph provides precision for entity-specific questions like shareholding percentages. Everything runs locally—PostgreSQL with pgvector, a local LLM via Ollama—so sensitive financial data never leaves the machine."

1.2 The Technical Depth Pitch

Question: "Describe the technical architecture."

"The system has three main layers:

First, the **ingestion pipeline** extracts text from PDFs, chunks it using semantic boundaries, and generates embeddings using a sentence transformer model. Simultaneously, I use an LLM to extract structured information—entities, relationships, definitions—building a Knowledge Graph stored in PostgreSQL.

Second, the **retrieval layer** uses a hybrid strategy. For factual questions ('Who is the CEO?'), I query the Knowledge Graph directly. For descriptive questions ('Explain the business model'), I use vector similarity search. Many questions use both—KG for precision, vectors for context.

Third, the **generation layer** takes the assembled context and uses a local LLM to generate answers. Responses are streamed to the frontend for better perceived performance.

The key architectural decision was keeping everything in PostgreSQL—using pgvector for embeddings alongside relational tables for the Knowledge Graph. This simplifies transactions and joins across Vector and KG data."

---

2. Architecture Questions

2.1 Component Design

Question: "Why did you choose this particular architecture?"

"The architecture emerged from the requirements:

Local execution was mandatory because IPO documents contain material non-public information. Cloud LLM APIs weren't an option for some users. This led to Ollama for local LLM inference.

Hybrid RAG was necessary because pure vector search struggles with precision. If someone asks 'What percentage does the CEO own?', semantic similarity might find chunks discussing ownership without the exact number. The Knowledge Graph stores structured facts that answer these questions precisely.

Single database (PostgreSQL) rather than separate stores (Pinecone for vectors, Neo4j for graphs) reduced operational complexity. pgvector performs well for our scale, and having everything in one transactional database makes consistency easier."

Question: "What are the main components and how do they interact?"

"The main components are:

1. **PDF Processor**: Extracts text with page mapping using PyPDF2
2. **Chunking Engine**: Splits text into semantic chunks with overlap
3. **Embedding Service**: Generates 384-dimensional vectors using sentence transformers
4. **KG Extractor**: Uses LLM to extract entities, relationships, definitions
5. **PostgreSQL + pgvector**: Stores everything with vector similarity search
6. **Retrieval Engine**: Decides strategy (vector/KG/hybrid) and executes queries
7. **LLM Generator**: Takes context and generates answers via Ollama
8. **Flask API**: REST endpoints for upload, query, and data access
9. **Frontend**: HTML/CSS/JS for user interaction with streaming support

They interact primarily through function calls within the Python process, with the frontend communicating via HTTP and Ollama via its REST API."

2.2 Design Decisions

Question: "What's the most important design decision you made?"

"The most impactful decision was **extracting a Knowledge Graph during ingestion rather than naively relying on pure vector search**.

Early versions used only vector search. It worked reasonably well but had consistent failures: - Precise numerical questions (shareholding percentages) often got approximate or wrong answers - Definition questions returned chunks that mentioned terms without defining them - Multi-hop questions (requiring combining facts) were challenging

Adding the Knowledge Graph was significant engineering effort—designing the schema, writing extraction prompts, handling entity resolution. But it transformed answer quality. Now, 'Who is the CEO and what is their shareholding?' returns a precise answer from structured data, not a hopeful semantic match.

The tradeoff was processing time (extraction adds minutes per document) and complexity. But for documents that will be queried many times, front-loading the effort was worth it."

Question: "Why PostgreSQL instead of a specialized vector database?"

"Several reasons made PostgreSQL the right choice:

Unified data model: Chunks, embeddings, entities, and claims all need to be queried together. Having them in separate databases would require synchronization and cross-database joins, which PostgreSQL handles naturally.

Transactional safety: When ingesting a document, I insert chunks, then embeddings, then KG data. If anything fails, I want to rollback everything. PostgreSQL's ACID guarantees make this straightforward.

Operational simplicity: Running Pinecone + Neo4j + PostgreSQL is three systems to maintain, three connection pools, three failure modes. PostgreSQL with pgvector is one system.

Performance at our scale: For thousands (not millions) of embeddings, pgvector with HNSW indexing is very fast—sub-100ms queries. We're not at the scale where specialized vector databases become necessary.

If we grew to millions of documents, I'd reconsider. But premature optimization would have added complexity without benefit."

---

3. RAG and LLM Questions

3.1 RAG Fundamentals

Question: "What is RAG and why does it matter?"

"RAG—Retrieval-Augmented Generation—solves a fundamental LLM limitation: models know only what they were trained on, and they can't update that knowledge without retraining.

When you ask an LLM about a specific IPO document, it has no knowledge of that document. RAG bridges this gap by: 1. **Retrieving** relevant information from a knowledge base (our chunks and KG) 2. **Augmenting** the LLM's prompt with this information 3. **Generating** an answer grounded in the provided context

Without RAG, the LLM would either refuse to answer or hallucinate. With RAG, it generates answers based on actual document content. The key insight is that we're using the LLM for reasoning and generation while providing it with facts it wouldn't otherwise have access to."

Question: "What's the difference between your hybrid RAG and standard RAG?"

"Standard RAG typically means: 1. Embed the question 2. Find similar chunks via vector search 3. Concatenate chunks as context 4. Send to LLM

My hybrid approach adds a parallel path: 1. **Classify the question** to determine if it's about entities, definitions, or requires narrative context 2. **Query the Knowledge Graph** for structured facts (entities with attributes, relationships) 3. **Query vector embeddings** for relevant text passages 4. **Merge results** intelligently—KG facts first (they're precise), then chunks for narrative context 5. **Generate answer** from this richer context

The hybrid approach shines when questions have definitive answers in the KG ('Who is the CEO?') but also benefit from narrative context ('Who is the CEO and what is their background?')."

3.2 Prompt Engineering

Question: "How do you prevent hallucination?"

"Hallucination prevention is multi-layered:

Context quality: The first defense is providing accurate, relevant context. If the right information isn't in the context, no amount of prompt engineering helps. Our hybrid retrieval aims to provide precisely the needed information.

System prompt instructions: We explicitly instruct the model: - 'Answer ONLY based on the provided context' - 'If the information is not in the context, say so' - 'Quote specific passages to support your answer' - 'Do not use outside knowledge about this company'

Source citation: Requiring the model to cite sources (page numbers, quotes) makes hallucination harder. The model can't cite a source that doesn't exist in the context.

Temperature settings: Lower temperature (we use 0.1-0.3) makes generation more deterministic and less creative, reducing tangential or fabricated content.

Validation: For numerical facts (percentages, dates), we cross-reference with the KG. If the LLM says '10%' but our KG has '4.27%', we flag the discrepancy.

Complete hallucination elimination isn't possible with current technology, but these measures significantly reduce it."

Question: "How do you handle complex multi-hop questions?"

"Multi-hop questions like 'What is the revenue of the company whose CEO is Yashish Dahiya?' require combining multiple facts.

Our approach:

Entity extraction from question: We identify 'Yashish Dahiya' and 'CEO' in the question, recognizing this as an entity lookup.

KG traversal: We query: find persons named 'Yashish Dahiya', then find companies where they have role 'CEO', then find financial metrics for those companies.

Context assembly: We include the chain of facts in the context, making the reasoning explicit for the LLM.

Generation: The LLM sees the connected facts and can answer directly.

This works because the KG explicitly stores relationships that would be implicit in raw text. Without the KG, we'd need to retrieve multiple chunks and hope the LLM correctly infers the connection."

---

4. Knowledge Graph Questions

4.1 KG Fundamentals

Question: "What is a Knowledge Graph and why did you build one?"

"A Knowledge Graph is a structured representation of facts as entities (nodes) and relationships (edges). Unlike unstructured text, it explicitly captures who, what, and how things relate.

For example, instead of a paragraph stating 'Mr. Yashish Dahiya serves as Chairman and CEO of PB Fintech Limited with shareholding of 4.27%...', the KG has:

- Entity: Yashish Dahiya (type: PERSON)
- Entity: PB Fintech Limited (type: COMPANY)
- Claim: Yashish Dahiya CEO_OF PB Fintech Limited
- Attribute: Yashish Dahiya.shareholding = 4.27%

I built it because vector search alone isn't precise enough. When someone asks 'What is the CEO's shareholding?', vector search returns similar chunks, but the model must extract the exact number. With the KG, I directly look up the shareholding attribute on the CEO entity—no inference needed."

Question: "How do you extract the Knowledge Graph from documents?"

"Extraction uses LLM-based information extraction:

1. Chunk selection: We prioritize sections likely to contain structured information (management, capital structure, definitions).

2. Prompt engineering: For each chunk, we send a structured prompt asking the LLM to extract: - Named entities with types (PERSON, COMPANY, METRIC) - Relationships between entities - Attributes for each entity - Defined terms

3. Output parsing: The LLM returns structured JSON that we parse and validate.

4. Evidence recording: Every extracted fact links to the source quote.

5. Entity resolution: We merge duplicate mentions ('Mr. Dahiya' and 'Yashish Dahiya' become one entity with aliases).

6. Database insertion: Validated facts go into the KG tables.

It's not perfect—LLMs sometimes miss facts or misinterpret relationships. That's why we store evidence: we can trace any fact back to its source and correct extraction errors."

4.2 KG Challenges

Question: "What were the hardest challenges with the Knowledge Graph?"

"Three main challenges:

Entity resolution was the most complex. The same person might be called 'Yashish Dahiya', 'Mr. Dahiya', 'the CEO', or 'our founder' in different sections. Merging these correctly requires understanding context, titles, and hints. We use a combination of: - String similarity matching -

Attribute overlap (same DIN number = same person) - Co-occurrence analysis (if 'Mr. Dahiya' appears right after 'Yashish Dahiya', they're likely the same)

Extraction consistency was another challenge. The LLM sometimes extracts different attributes depending on how topics are phrased. We refined prompts extensively and added post-processing validation.

Schema design required balancing flexibility with structure. Too rigid, and we can't capture unusual facts. Too flexible, and querying becomes hard. We settled on typed entities with JSONB attributes, giving structured types with flexible properties."

Question: "How do you handle updates if the document is modified?"

"Currently, we don't support incremental updates. If a document changes:

1. Delete the existing document (cascades to all chunks, embeddings, KG data)
2. Re-upload and reprocess the new version

This simplicity has tradeoffs. Reprocessing takes time, and if someone asks a question during reprocessing, the document is unavailable.

For future improvement, we could: - Detect which pages changed and only reprocess those - Maintain document versions - Support real-time updates for fast-changing documents

But for IPO documents, which are static after filing, full reprocessing is acceptable."

---

5. Database and Vector Search Questions

5.1 Database Design

Question: "Explain your database schema design."

"The schema has two conceptual layers:

RAG Layer (for semantic search): - `documents`: Metadata about each uploaded PDF - `chunks`: Text segments with page mapping - `embeddings`: 384-dimensional vectors linked to chunks

Knowledge Graph Layer (for structured queries): - `evidence`: Quotes supporting each fact - `kg_entities`: People, companies, metrics with attributes - `entity_aliases`: Alternative names for entities - `claims`: Relationships between entities (CEO_OF, SUBSIDIARY_OF) - `defined_terms`: Glossary entries - `events`: Dated occurrences

The layers connect through document_id (everything traces to a document) and chunk_id (evidence comes from specific chunks).

Key design decisions: - **JSONB for attributes**: Entities have varying attributes (person has age, company has CIN). JSONB provides flexibility without schema changes. - **Cascading deletes**: Deleting a document removes all associated data automatically. - **Evidence-first**: Every KG fact must have supporting evidence, ensuring traceability."

Question: "How does vector similarity search work?"

"Vector similarity search enables semantic matching—finding text with similar meaning, not just matching keywords.

The process: 1. **Embedding generation**: We convert text (both chunks and queries) into 384-dimensional vectors using a transformer model (all-MiniLM-L6-v2). 2. **Storage**: Vectors are stored in PostgreSQL using pgvector's VECTOR(384) type. 3. **Indexing**: We create an HNSW (Hierarchical Navigable Small Worlds) index for fast approximate search. 4. **Query**: When a question comes in, we embed it and find vectors with lowest cosine distance.

Why it works: The embedding model places semantically similar text near each other in vector space. 'Who is the chief executive?' and 'Who is the CEO?' have different words but similar embeddings, so both find relevant chunks.

PostgreSQL implementation:

```
SELECT c.text, e.embedding <=> query_embedding AS distance
FROM embeddings e
JOIN chunks c ON e.chunk_id = c.id
WHERE c.document_id = 'policybazar_ipo'
ORDER BY distance
LIMIT 10;
```

The `<=>` operator computes cosine distance, and the HNSW index makes this fast even with thousands of embeddings."

5.2 Performance

Question: "How do you optimize database performance?"

"Several optimization strategies:

Indexing: - HNSW index on embeddings for vector search (configurable m and ef parameters) - B-tree indexes on foreign keys (document_id, chunk_id) - Indexes on frequently queried columns (entity names, claim predicates)

Connection pooling: SQLAlchemy manages a connection pool (5 connections, 10 overflow), avoiding connection creation overhead per query.

Batch operations: During ingestion, we batch insert chunks (100 at a time) rather than individual inserts.

Query optimization: Using EXPLAIN ANALYZE to understand query plans and ensure indexes are used.

At query time: - Set `hnsw.ef_search` appropriately for recall/speed tradeoff - Limit vector search results (top-10 is usually sufficient) - Cache embedding model in memory (avoiding reload per request)

For our scale (hundreds of documents, thousands of chunks), PostgreSQL handles everything well. If we grew to millions of chunks, we'd consider dedicated vector databases."

---

6. Frontend and API Questions

6.1 API Design

Question: "How did you design the REST API?"

"The API follows REST principles with pragmatic choices:

Resource-based URLs: - `/api/documents` - List/create documents - `/api/documents/{id}` - Get/delete specific document - `/api/ask` - Question-answering (POST because it triggers processing)

HTTP methods used semantically: - GET for retrieval (documents, entities) - POST for creation and actions (upload, ask) - DELETE for removal

Consistent response format: - Success: JSON with data - Error: JSON with error message and appropriate HTTP status (400, 404, 500)

Streaming for Q&A: The `/api/ask` endpoint streams responses using NDJSON (newline-delimited JSON). Each line is a complete JSON object:

```
{"type": "status", "message": "Searching..."}
{"type": "token", "content": "The"}
{"type": "token", "content": "CEO"}
{"type": "done", "sources": 5}
```

This enables real-time display as the LLM generates."

Question: "Why did you use Flask instead of FastAPI?"

"Flask was chosen for several reasons:

Simplicity: Flask is minimal and doesn't impose structure. For our focused application, this simplicity reduced cognitive overhead.

Streaming support: Flask handles streaming responses through generator functions, which we need for real-time LLM output.

Familiarity: Flask is widely used and well-documented. Anyone joining the project can understand the code quickly.

Sufficient features: We don't need FastAPI's automatic OpenAPI docs or async capabilities (our LLM calls are processed in threads anyway).

If we were building a larger API with many endpoints or needed async database access, FastAPI would be more appropriate. For our focused use case, Flask's simplicity won."

6.2 Frontend

Question: "Tell me about the frontend architecture."

"The frontend is a lightweight single-page application:

Technology choices: - Plain HTML/CSS/JavaScript (no React/Vue) - No build step—files served directly - CSS for styling without frameworks

Why this simplicity? - The UI is focused (upload, ask questions, see answers) - Fewer dependencies means easier deployment - Users can understand and modify the frontend easily

Key features: - **Streaming display**: As tokens arrive from the API, JavaScript appends them to the answer area, creating a typing effect - **Source citation**: Answers include clickable references to source pages - **Error handling**: Clear error messages for various failure modes - **Responsive design**: Works on mobile devices

JavaScript structure: - Event handlers for form submission - Fetch API for HTTP requests - ReadableStream handling for streaming responses - DOM manipulation for updates

It's not the flashiest frontend, but it's maintainable, fast, and does what users need."

---

7. Data Processing Questions

7.1 PDF Processing

Question: "How do you handle PDF text extraction?"

"PDF text extraction uses PyPDF2 with awareness of its limitations:

The process: 1. Open PDF with PyPDF2 2. Iterate through pages 3. Extract text maintaining page boundaries 4. Track page numbers for source attribution

Challenges and solutions: - **Encoding issues**: PDFs use various encodings. We handle UnicodeDecodeError gracefully. - **Layout preservation**: PDF text extraction loses layout. Headers and footers mix with body text. We use heuristics to detect repeated patterns. - **Tables**: Tables extract as unstructured text. Complex table data might be misinterpreted. - **Scanned PDFs**: PyPDF2 can't handle image-based PDFs. We'd need OCR for those.

For IPO documents (which are typically well-structured digital PDFs), PyPDF2 works well. For more challenging documents, we'd consider pdfplumber or OCR-based approaches."

7.2 Chunking Strategy

Question: "How do you decide where to split text into chunks?"

"Chunking balances several competing concerns:

Too small: Context is lost. A 50-word chunk about shareholder structure won't include who the shareholders are.

Too large: Embeddings become unfocused. A 5000-word chunk covers too many topics to match any specific query well.

Our strategy: 1. **Target size**: ~500 words (roughly 2000-2500 characters) 2. **Semantic boundaries**: Prefer splitting at paragraph breaks, section headers, or sentence ends 3. **Overlap**: Include ~100 words from the previous chunk at the start of the next 4. **Page tracking**: Record which pages each chunk spans

Implementation: - Split at double newlines (paragraph breaks) - If a paragraph exceeds target, split at sentence boundaries - Always start a new chunk at section headers - Ensure chunks don't exceed maximum (3000 characters)

Overlap is important because relevant information might be split across chunks. If a question falls near a boundary, both chunks have a chance to match.

We experimented with sentence transformers' chunking and recursive character splitting. Our semantic paragraph-based approach gave the best retrieval quality for these document types."

---

8. System Design Questions

8.1 Scalability

Question: "How would you scale this system for 10x or 100x load?"

"Scaling depends on which component becomes the bottleneck:

Database scaling: - **Read replicas**: Vector searches are read-heavy. Adding read replicas allows distributing query load. - **Connection pooling**: PgBouncer in front of PostgreSQL handles more connections efficiently. - **Partitioning**: Documents table partitioned by document_id would speed per-document queries.

LLM scaling: - **Multiple Ollama instances**: Load-balance across several instances, each running the model. - **GPU acceleration**: GPU instances dramatically speed generation (10-50x). - **Cloud LLM fallback**: For peak load, fall back to cloud APIs (OpenAI, Anthropic).

Application scaling: - **Horizontal scaling**: Run multiple Flask instances behind a load balancer. - **Task queue**: Move document processing to Celery workers, scaling independently.

Storage scaling: - **Vector database migration**: At millions of documents, consider dedicated vector stores (Pinecone, Milvus). - **Object storage**: Move PDFs to S3/GCS for cheaper, scalable storage.

For 10x load, horizontal Flask scaling + multiple Ollama instances would likely suffice. For 100x, dedicated vector database and cloud LLM integration become necessary."

8.2 Reliability

Question: "How do you handle failures?"

"Failure handling at multiple levels:

Document processing failures: - Each phase updates document status - Failures set status='error' with details in metadata - Users see clear error messages and can retry - Partial data is cleaned up (no orphaned records)

Query failures: - If KG query fails, fall back to pure vector search - If vector search fails, return error with guidance - If LLM fails, return retrieved context without generation - Timeouts prevent indefinite hangs

Database failures: - Connection pool handles transient connection issues - Transaction rollback on any error prevents inconsistent state - Health checks detect database problems early

Recovery procedures: - Document reprocessing if extraction quality is poor - Database backups (pg_dump daily, retained weekly) - Logs capture context for debugging

The philosophy is: fail gracefully, give users actionable information, and make recovery straightforward."

---

9. Challenges and Trade-offs

9.1 Technical Challenges

Question: "What was the most difficult technical challenge?"

"The most difficult challenge was **achieving consistent, high-quality KG extraction**.

LLMs are non-deterministic. The same text might extract entities differently on different runs. Relationships might be inferred differently. This made the KG feel unreliable initially.

How we addressed it:

1. **Prompt iteration**: We refined prompts extensively, adding examples, constraints, and format specifications. This took dozens of iterations.
2. **Structured output**: We moved from free-form extraction to strict JSON schemas. The model has less room for deviation.
3. **Validation layer**: Post-extraction validation catches common errors:
   - Percentages must be between 0 and 100
   - Dates must be parseable
   - Entity types must be from allowed set
4. **Temperature tuning**: Lower temperature (0.1-0.2 for extraction vs 0.3-0.5 for generation) increased consistency.
5. **Multi-pass extraction**: For critical sections, we extract twice and reconcile differences.

It's still not perfect—some documents have unusual structures that confuse extraction. But consistency improved dramatically through these measures."

9.2 Trade-offs

Question: "What trade-offs did you make?"

"Several significant trade-offs:

Processing time vs. query quality: Building the Knowledge Graph adds 5-30 minutes per document versus seconds for pure vector ingestion. We chose quality—documents are ingested once but queried many times.

Local vs. cloud LLM: Running Ollama locally means slower generation than GPT-4 but ensures privacy and no per-query cost. For financial documents where confidentiality matters, this was the right trade-off.

Single database vs. specialized stores: PostgreSQL handles everything versus dedicated vector/graph databases. Simpler operations but potentially less optimal at extreme scale. Right trade-off for our current scale.

Extraction coverage vs. precision: We could extract everything the LLM finds or be conservative and only keep high-confidence extractions. We chose precision—fewer extracted facts but higher quality.

Simple frontend vs. rich UI: Plain HTML/JavaScript versus React/Vue. Easier to deploy and maintain, but less dynamic. Right for our focused use case."

---

10. Metrics and Evaluation

10.1 Quality Metrics

Question: "How do you measure system quality?"

"Quality is measured at multiple levels:

Retrieval quality: - **Recall**: Do we retrieve the chunks containing the answer? - **Precision**: Are retrieved chunks relevant or noise? - **Measured by**: Manual evaluation against a test set of questions with known answer locations

KG extraction quality: - **Entity accuracy**: Are extracted entities correct? - **Relationship accuracy**: Are claimed relationships true? - **Coverage**: What percentage of facts in the document are extracted? - **Measured by**: Manual comparison against hand-labeled document sections

Answer quality: - **Correctness**: Does the answer match the source? - **Completeness**: Does it include all relevant information? - **Hallucination rate**: Does it claim things not in the document? - **Measured by**: Manual evaluation of answers against source documents

User experience: - **Time to first token**: How quickly does the answer start appearing? - **Total response time**: End-to-end latency - **Error rate**: How often do queries fail?

We maintain a test set of ~50 questions across several documents with expected answers for regression testing."

10.2 Performance Metrics

Question: "What are typical performance numbers?"

"From production-like testing:

Ingestion times (100-page IPO document): - Text extraction: 5-10 seconds - Chunking: 2-3 seconds - Embedding generation: 30-60 seconds - KG extraction: 5-20 minutes (depends on content density) - Total: 8-25 minutes

Query times: - Query understanding: 10-20 ms - Vector embedding: 20-50 ms - Vector search: 50-100 ms - KG queries: 20-50 ms - Context assembly: 10-20 ms - LLM generation: 2-5 seconds - Total time to first token: 100-300 ms - Total time to complete answer: 3-6 seconds

Storage: - Per 100-page document: 5-20 MB database storage - Mainly embeddings (~1.5 KB × 200-500 chunks)

Accuracy (on test set): - Retrieval recall: ~85-90% - Answer correctness: ~80-85% - Hallucination rate: ~5-10%"

---

11. Future Improvements

11.1 Technical Improvements

Question: "What would you improve given more time?"

"Several improvements on the roadmap:

Multi-document queries: Currently, questions are scoped to one document. Supporting 'Compare revenue growth of Company A vs Company B' requires cross-document retrieval and entity linking.

Better extraction models: Fine-tuning a model specifically for IPO document extraction would improve accuracy. Currently using general-purpose LLMs that aren't optimized for financial documents.

Incremental updates: Re-processing entire documents for small changes is wasteful. Diff-based processing would update only affected chunks and KG sections.

Improved entity resolution: Current string-matching based resolution could be enhanced with embeddings or dedicated entity linking models.

User feedback loop: Allowing users to correct wrong answers would create training data for improving the system.

Caching layer: Common questions could be cached to avoid regeneration. Redis integration would speed repeated queries.

Evaluation automation: Building automated evaluation against a gold-standard dataset would allow continuous quality monitoring."

11.2 Production Improvements

Question: "What would be needed for production deployment?"

"Moving from prototype to production requires:

Observability: - Structured logging with request tracing - Metrics export (Prometheus/Datadog) - Alerting on error rates and latency

Security: - Authentication and authorization - Rate limiting - Input sanitization hardening - TLS everywhere

Reliability: - Health check endpoints - Graceful shutdown handling - Database connection retry logic - Circuit breakers for LLM service

Operations: - Containerization (Docker) - Infrastructure as code - CI/CD pipeline - Runbooks for common issues

Scaling: - Load testing to identify bottlenecks - Horizontal scaling capability - Database read replicas

Most of these are standard production infrastructure. The application logic is ready; it needs production wrapping."

---

12. Quick Reference Cheat Sheet

12.1 Key Technical Terms

Term	Definition
RAG	Retrieval-Augmented Generation - combining retrieval with LLM generation
Knowledge Graph	Structured representation of entities and their relationships
Embedding	Vector representation capturing semantic meaning of text
Chunking	Splitting documents into smaller, semantically coherent pieces
Semantic Search	Finding similar content based on meaning, not keywords
pgvector	PostgreSQL extension for vector similarity search
HNSW	Hierarchical Navigable Small Worlds - fast vector index algorithm
Cosine Distance	Measure of similarity between vectors (0=identical, 2=opposite)
Entity Resolution	Merging different references to the same real-world entity
NDJSON	Newline-Delimited JSON - stream-friendly JSON format

12.2 Architecture Components

Component	Purpose	Technology
PDF Processor	Extract text from PDFs	PyPDF2
Chunking Engine	Split text into semantic chunks	Custom Python
Embedding Service	Generate vector representations	sentence-transformers
Vector Storage	Store and search embeddings	PostgreSQL + pgvector
KG Extractor	Extract structured facts	Ollama (llama3)
KG Storage	Store entities and relationships	PostgreSQL
Retrieval Engine	Hybrid vector + KG search	Custom Python
LLM Generator	Generate answers	Ollama (llama3)
API Layer	REST endpoints	Flask
Frontend	User interface	HTML/CSS/JavaScript

12.3 Key Numbers

Metric	Typical Value
Embedding dimensions	384
Chunk size	~500 words
Chunk overlap	~100 words
Vector search top-K	10
LLM temperature (extraction)	0.1-0.2
LLM temperature (generation)	0.3-0.5
Time to first token	100-300 ms
Total query time	3-6 seconds
Ingestion time (100 pages)	8-25 minutes

12.4 Quick Answers

"Why hybrid RAG?" → Vector search finds relevant text; KG provides precision for structured facts.

"Why PostgreSQL?" → Single database for vectors and graphs; transactional safety; operational simplicity.

"Why local LLM?" → Data privacy for sensitive financial documents; no per-query cost.

"Biggest challenge?" → Consistent KG extraction quality—solved with prompt engineering and validation.

"How do you prevent hallucination?" → Relevant context, explicit instructions, source citations, low temperature.

Conclusion

The Complete Documentation

This comprehensive guide has covered every aspect of the IPO Intelligence Q&A System across ten chapters:

1. **Executive Summary & Architecture**: High-level overview of what the system does and how it's built
2. **PDF Ingestion & Chunking**: How documents are processed and split for analysis
3. **Vector Embeddings & Semantic Search**: The mathematics and implementation of semantic similarity
4. **Knowledge Graph Generation**: Extracting structured facts from unstructured text
5. **RAG Query Pipeline**: How questions are answered using both vectors and Knowledge Graph
6. **Database Schema Deep Dive**: Every table, column, and relationship explained
7. **PostgreSQL Implementation Details**: Practical database implementation patterns
8. **Frontend & API Layer**: User interface and web service architecture
9. **System Integration & Data Flow**: How all components work together
10. **Interview Q&A Guide**: Comprehensive preparation for technical discussions

This system represents the integration of modern NLP techniques (embeddings, LLMs) with traditional software engineering (PostgreSQL, Flask, clean architecture). It demonstrates that sophisticated AI applications can be built with local, privacy-preserving components while maintaining high quality and performance.

The key insight underlying this project is that combining multiple retrieval strategies—semantic vector search for finding relevant passages and structured Knowledge Graph queries for precise fact lookup—produces better results than either approach alone. This hybrid approach is increasingly how advanced RAG systems are being built in industry.

Document Complete - Total ~10,000 lines covering all aspects of the IPO Intelligence Q&A System.