Technical Design Document: Extraction Layer for Side-by-Side PDF Viewer

1. Overview

The extraction layer is a backend subsystem responsible for transforming raw PDFs into structured data used by the side-by-side PDF viewer. It powers text extraction, layout mapping, semantic chunking, embedding generation, and metadata storage. This document outlines the full engineering architecture, workflow, components, and integration points for the extraction layer.

2. System Goals and Requirements

Functional Requirements: • Convert PDFs into structured chunks containing text, page, bounding boxes, and metadata. • Normalize bounding boxes for consistent rendering in web viewers. • Generate embeddings for semantic search and retrieval. • Store structured outputs in the data lake, SQL database, and vector index. • Support incremental and idempotent processing. Non-Functional Requirements: • Scalable and parallelizable processing pipeline. • Fault tolerance and retry logic. • Efficient batch processing for embeddings. • Cloud storage integration (S3/Azure/GCS). • Low-latency metadata retrieval for the viewer.

3. Architecture Diagram (Textual Description)

The system follows a multi-layer architecture: User Upload → Ingestion API → Data Lake Storage → Extraction Worker → Metadata Outputs → SQL DB + Vector DB → Viewer Modules: • Ingestion Service – Stores original PDFs and creates extraction jobs. • Extraction Worker – Performs parsing, normalization, chunking, embedding. • Metadata Store – Parquet in data lake, plus relational index. • Vector Index – Embeddings for semantic retrieval. • Viewer – Consumes metadata and provides interactive experiences.

4. Extraction Layer Components

4.1 PDF Loader Loads PDFs from storage using PyMuPDF or similar. 4.2 Layout Extractor Extracts: • Text blocks • Bounding boxes • Page dimensions • Formatting markers 4.3 Normalization Module Converts PDF coordinates into normalized values aligned with browser coordinate systems. 4.4 Chunker Segments text into semantic chunks with identifiers. 4.5 Embedding Generator Produces embeddings in batch for semantic search. 4.6 Metadata Packager Packages all structured elements and writes to data lake, SQL DB, and vector DB.

5. Data Models

Chunk Metadata Schema: • chunk_id: string • doc_id: string • page: int • text: string • bbox_normalized: [float, float, float, float] • bbox_points: [float, float, float, float] • embedding: vector (optional at ingestion) • chunk_hash: string SQL Database Tables: • documents • chunks (chunk_id, doc_id, page, bbox, preview_text) Vector DB Metadata: • embedding • doc_id • chunk_id • bbox • page

6. Workflow and Data Flow

1. PDF Upload triggers job creation. 2. Extraction worker loads PDF from data lake. 3. Layout extractor parses blocks and coordinates. 4. Normalizer converts box coordinates. 5. Chunker creates semantic chunk units. 6. Embedding generator creates vector embeddings in batches. 7. Data is persisted: • Parquet files in data lake • SQL DB rows for quick lookups • Vector DB entries for search 8. Viewer fetches SQL metadata and renders PDF with highlight overlays.

7. Performance Considerations

• Parallelize processing on page or document level. • Batch embeddings to reduce API overhead. • Cache metadata in Redis for low-latency viewer loads. • Use chunk hashing to skip redundant processing. • Implement retry logic for embedding API failures. • Use Parquet for efficient metadata storage and analytics.

8. Failure Modes and Mitigation

Potential Issues: • Damaged PDFs • Missing text due to scanned pages • OCR failures • Embedding generation timeouts • Inconsistent bounding boxes Mitigations: • OCR fallback pipeline using Tesseract/Vision API. • Page-level retries. • Hash-based detection to resume failed jobs. • Dual storage of raw and normalized bounding boxes for debugging. • Validation suite with overlay previews.

9. Security Considerations

• Use signed URLs for PDF upload and retrieval. • Encrypt data lake buckets. • Restrict worker IAM permissions. • Sanitize text for unsafe characters. • Log access for audit compliance.

10. Deployment Approach

Deployment Options: • Dockerized processing worker. • Container orchestration: Kubernetes or AWS ECS. • Event-driven triggers via S3/Azure Blob events. • CI/CD for pipeline updates. • Autoscaling based on queue depth.

11. Testing Strategy

Unit Tests: • bbox normalization correctness • chunk segmentation logic • embedding vector lengths • metadata schema validation Integration Tests: • full PDF-to-chunk pipeline • vector DB search correctness Load Tests: • large document ingestion • parallel extraction performance

12. Conclusion

The extraction layer is the backbone of the intelligent PDF experience. It transforms raw PDFs into structured, searchable, interactive documents. By integrating layout analysis, normalization, semantic chunking, and embeddings, it enables the side-by-side viewer to deliver a powerful AI-augmented reading and annotation experience. This technical design ensures scalability, precision, and maintainability across large document processing workloads.